Annamalai University

Faculty of Engineering and Technology

Department of Computer Science and Engineering

M.E(Computer Science and Engineering)

II Semester

19 CSCSPC21 - ANALYSIS OF ALGORITHMS

Prepared By
Dr.R.Priya
Professor
Department of Computer Science and Engineering
Annamalai University

19 CSCSPC21	**ANALYSIS OF ALGORITHMS**	**L**	**T**	**P**	**C**
		3	**0**	**0**	**3**

COURSE OBJECTIVES:

- To introduce students the advanced methods of designing and analyzing algorithms.
- To enable the students to choose appropriate algorithms and use it for a specific problem.
- To familiarize students with basic paradigms and data structures used to solve advanced algorithmic problems.
- To understand different classes of problems concerning their computation difficulties.
- To introduce the students to get familiarity in recent developments in the area of algorithmic design.

Sorting: Review of various sorting algorithms, topological sorting, Graph: Definitions and Elementary Algorithms: Shortest path by BFS, Shortest path in edge-weighted case (Dijkstra's), depth-first search and computation of strongly connected components, emphasis on correctness proof of the algorithm and time/space analysis, example of amortized analysis.

Matroids: Introduction to greedy paradigm, algorithm to compute a maximum weight maximal independent set, Application to MST. Graph Matching: Algorithm to compute maximum matching. Characterization of maximum matching by augmenting paths, Edmond's Blossom algorithm to compute augmenting path.

Flow-Networks: Maxflow - mincut theorem, Ford-Fulkerson Method to compute maximum flow, Edmond-Karp maximum-flow algorithm. Matrix Computations: Strassen's algorithm and introduction to divide and conquer paradigm, inverse of a triangular matrix, relation between the time complexities of basic matrix operations, LUP-decomposition.

Shortest path in Graphs: Floyd- Warshall algorithm and introduction to dynamic programming paradigm. More examples of dynamic programming. Modulo Representation of integers/polynomials: Chinese Remainder theorem, Conversion between base-representation and modulo-representation.

Extension of polynomials. Application: Interpolation problem. Discrete Fourier Transform (DFT): In complex field, DFT in modulo ring. Fast Fourier Transform algorithm. Schonhage-Strassen Integer Multiplication algorithm.

Linear Programming: Geometry of the feasibility region and Simplex algorithm. NP-completeness: Examples, proof of NP-hardness and NP-Completeness. One or more of the following topics based on time and interest: Approximation algorithms, Randomized Algorithms, Interior Point Method, Advanced Number Theoretic Algorithm. Recent Trends in problem solving paradigms using recent searching and sorting techniques by applying recently proposed data structures.

REFERENCES:

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Strin "Introduction to Algorithms", Second Edition, MIT Press, MCGraw Hill, 2001.
2. Alfred V.Aho, John E. Hopcroft, Jeffrey d.Ullman, "The Design and Analysis of Computer Algorithms", Addison-Wesley Publishing Company, 1974.
3. Jon Kleinberg, Eva Tardos, "Algorithm Design" ,Pearson Addision Wesley, 2005.

COURSE OUTCOMES:

After completion of course, students would be able to:

- Analyze the complexity/performance of different algorithms.
- Determine the appropriate data structure for solving a particular set of problems.
- Categorize the different problems in various classes according to their complexity.
- Students should have an insight of recent activities in the field of the advanced data structure.

UNIT-1

SORTING:

Sorting Algorithm: It is an algorithm made up of a series of instructions that takes an array as input, and outputs a sorted array. There are many sorting algorithms, such as: Selection sort, Bubble Sort, Heap Sort, Quick sort, Radix sort, Counting sort, Bucket sort, Shell sort, Comb sort.

Popular Sorting algorithms

- Simple sorts
  - Insertion sort
  - Selection sort
- Efficient sorts
  - Merge sort
  - Heap sort
  - Quick sort
  - Shell sort
- Bubble sort and variants
  - Bubble sort
  - Comb sort
- Distribution sort
  - Counting sort
  - Bucket sort
  - Radix sort

Simple sorts

Two of the simplest sorts are insertion sort and selection sort, both of which are efficient on small data, due to low overhead, but not efficient on large data. Insertion sort is generally faster than selection sort in practice, due to fewer comparisons and good performance on almost-sorted data, and thus is preferred in practice, but selection sort uses fewer writes, and thus is used when write performance is a limiting factor.

Insertion sort

Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly sorted lists, and is often used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one. Shell sort is a variant of insertion sort that is more efficient for larger lists.

Selection sort

Selection sort is an in-place comparison sort. It has O(n^2) complexity, making it inefficient on large lists and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity and also has performance advantages over more complicated algorithms in certain situations. The algorithm finds the minimum value, swaps it with the value in the first position, and repeats these steps for the remainder of the list. It does no more than n swaps, and thus is useful where swapping is very expensive.

Efficient sorts

Practical general sorting algorithms are almost always based on an algorithm with average time complexity (and generally worst-case complexity) O(n log n), of which the most common are heap sort, merge sort and quick sort. Each has advantages and drawbacks, with the most significant being that simple implementation of merge sort uses O(n) additional space and simple implementation of quick sort has O(n^2) worst-case complexity.

Merge sort

Merge sort takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (i.e., 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list. Of the algorithms described here, this is the first that scales well to very large lists, because its worst-case running time is O(n log n). It is also easily applied to lists, not only arrays, as it only requires sequential access, not random access. However, it has additional O(n) space complexity and involves a large number of copies in simple implementations.

Heap sort

Heap sort is a much more efficient version of selection sort. It also works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a heap, a special type of binary tree. Once the data list has been made into a heap, the root node is guaranteed to be the largest (or smallest) element. When it is removed and placed at the end of the list, the heap is rearranged so the largest element remaining moves to the root. Using the heap, finding the next largest element takes O(log n) time, instead of O(n) for a linear scan as in simple selection sort. This allows Heap sort to run in O(n log n) time and this is also the worst case complexity.

Quick sort

Quick sort is a divide and conquer algorithm which relies on a partition operation: to partition an array, an element called a pivot is selected. All elements smaller than the pivots are moved before it and all greater elements are moved after it. This can be done efficiently in linear time and in-place. The lesser and greater sub-lists are then recursively sorted. This yields average time complexity of

O(*n* log *n*), with low overhead, and thus this is a popular algorithm. The most complex issue in quick sort is thus choosing a good pivot element, as consistently poor choices of pivots can result in drastically slower O(n^2) performance, but good choice of pivots yields O(*n* log *n*) performance, which is asymptotically optimal.

Shell sort

A Shell sort is different from bubble sort in that it moves elements to numerous swapping positions. It improves upon insertion sort by moving out of order elements more than one position at a time. The concept behind Shell sort is that insertion sort performs in time, where k is the greatest distance between two out-of-place elements. This means that generally, they perform in O(n^2), but for data that is mostly sorted, with only a few elements out of place, they perform faster.

The worst-case time complexity of Shell sort is an open problem and depends on the gap sequence used, with known complexities ranging from O(n^2) to O($n^{4/3}$) and Θ(n $\log^2$ n). This, combined with the fact that Shell sort is in-place, only needs a relatively small amount of code and does not require use of the call stack, makes it useful in situations where memory is at a premium, such as in embedded systems and operating system kernels.

Bubble sort and variants

Bubble sort and variants such as the shell sort and cocktail sort, are simple, highly-inefficient sorting algorithms. They are frequently seen in introductory texts due to ease of analysis, but they are rarely used in practice.

Bubble sort

A bubble sort is a sorting algorithm that continuously steps through a list, swapping items until they appear in the correct order. Bubble sort is a simple sorting algorithm. The algorithm starts at the beginning of the data set. It compares the first two elements and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. This algorithm's average time and worst-case performance is O(n^2), so it is rarely used to sort large, unordered data sets. Bubble sort can be used to sort a small number of items (where its asymptotic inefficiency is not a high penalty). Bubble sort can also be used efficiently on a list of any length that is nearly sorted (that is, the elements are not significantly out of place). For example, if any numbers of elements are out of place by only one position (e.g. 0123546789 and 1032547698), bubble sort's exchange will get them in order on the first pass, the second pass will find all elements in order, so the sort will take only 2*n* time.

Comb sort

Comb sort is a relatively simple sorting algorithm based on bubble sort. The basic idea is to eliminate turtles, or small values near the end of the list, since in a bubble sort these slow the sorting down tremendously. It accomplishes this by initially swapping elements that are a certain distance from one another in the array, rather than only swapping elements if they are adjacent to one another, and then shrinking the chosen distance until it is operating as a normal bubble sort. Thus, if Shell sort can be thought of as a generalized version of insertion sort that swaps elements spaced a certain distance away from one another, comb sort can be thought of as the same generalization applied to bubble sort.

Distribution sort

Distribution sort refers to any sorting algorithm where data is distributed from their input to multiple intermediate structures which are then gathered and placed on the output. For example, both bucket sort and flash sort are distribution based sorting algorithms. Distribution sorting algorithms can be used on a single processor, or they can be a distributed algorithm, where individual subsets are separately sorted on different processors, then combined. This allows external sorting of data too large to fit into a single computer's memory.

Counting sort

Counting sort is applicable when each input is known to belong to a particular set, *S*, of possibilities. The algorithm runs in O(|*S*| + *n*) time and O(|*S*|) memory where *n* is the length of the input. It works by creating an integer array of size |*S*| and using the *i*th bin to count the occurrences of the *i*th member of *S* in the input. Each input is then counted by incrementing the value of its corresponding bin. Afterward, the counting array is looped through to arrange all of the inputs in order. This sorting algorithm often cannot be used because *S* needs to be reasonably small for the algorithm to be efficient, but it is extremely fast and demonstrates great asymptotic behavior as *n* increases. It also can be modified to provide stable behavior.

Bucket sort

Bucket sort is a divide and conquer sorting algorithm that generalizes counting sort by partitioning an array into a finite number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. A bucket sort works best when the elements of the data set are evenly distributed across all buckets.

Radix sort

Radix sort is an algorithm that sorts numbers by processing individual digits. n numbers consisting of k digits each are sorted in O(n · k) time. Radix sort can process digits of each number either starting from the least significant digit (LSD) or starting from the most significant digit (MSD). The LSD algorithm first sorts the list by the least significant digit while preserving their relative order using a stable sort. Then it sorts them by the next digit and so on from the least significant to the most significant, ending up with a sorted list. While the LSD radix sort requires the use of a stable sort, the MSD radix sort algorithm does not (unless stable sorting is desired). In-place MSD radix sort is not stable. It is common for the counting sort algorithm to be used internally by the radix sort. A hybrid sorting approach, such as using insertion sort for small bins improves performance of radix sort significantly.

Explanation of Sorting Algorithms:

INSERTION SORT:

- Insertion sort is an in-place sorting algorithm.
- It uses no auxiliary data structures while sorting.
- It is inspired from the way in which we sort playing cards.

Insertion sort algorithm

```
for(i = 1 ; i < n ; i++)
{
key = A [ i ];
j = i - 1;
while(j >0&& A [ j ]> key)
{
A [ j+1] = A [ j ];
j--;
}
A [ j+1] = key;
}
```

Here,

- i = variable to traverse the array A
- key = variable to store the new number to be inserted into the sorted sub-array
- j = variable to traverse the sorted sub-array

Working of insertion sort:

Consider the following elements to be sorted in ascending order : 6, 2, 11, 7, 5

Insertion sort works as

Step 1	• It selects the second element (2). • It checks whether it is smaller than any of the elements before it. • Since 2 < 6, so it shifts 6 towards right and places 2 before it. • The resulting list is 2, 6, 11, 7, 5.
Step 2	• It selects the third element (11). • It checks whether it is smaller than any of the elements before it. • Since 11 > (2, 6), so no shifting takes place. • The resulting list remains the same.
Step 3	• It selects the fourth element (7). • It checks whether it is smaller than any of the elements before it. • Since 7 < 11, so it shifts 11 towards right and places 7 before it. • The resulting list is 2, 6, 7, 11, 5.
Step 4	• It selects the fifth element (5). • It checks whether it is smaller than any of the elements before it. • Since 5 < (6, 7, 11), so it shifts (6, 7, 11) towards right and places 5 before them. • The resulting list is 2, 5, 6, 7, 11.

As a result, sorted elements in ascending order are-2, 5, 6, 7, 11

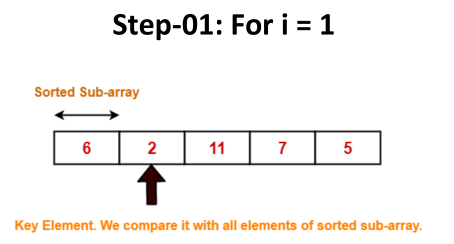
Insertion Sort Algorithm:

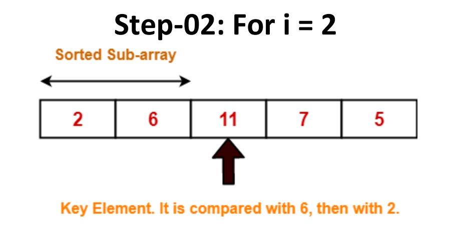
Let A be an array with n elements. The insertion sort algorithm used for sorting is as follows-

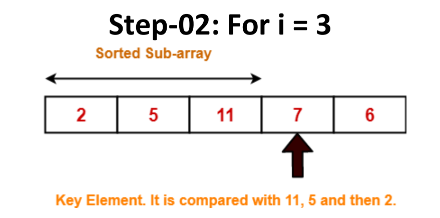
Insertion Sort Example:

Consider the following elements are to be sorted in ascending order: 6, 2, 11, 7, 5

The above insertion sort algorithm works as illustrated below

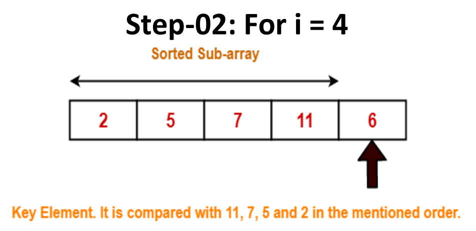

Step-01: For i = 1
Sorted Sub-array
6
2
11
7
5
Key Element. We compare it with all elements of sorted sub-array.



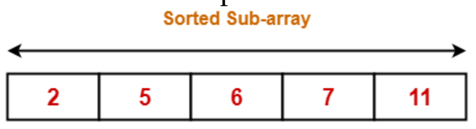


2	5	11	7	6	For j = 2; 11 > 7 so A[3] = 11
2	5	11	11	6	For j = 1; 5 < 7 so loop stops and A[2] = 7
2	5	7	11	6	After inner loop ends

Working of inner loop when i = 3


Step-02: For i = 4
Sorted Sub-array
2
5
7
11
6
Key Element. It is compared with 11, 7, 5 and 2 in the mentioned order.

Loop gets terminated as ‘i’ becomes 5. The state of array after the loops are finished



With each loop cycle,

- One element is placed at the correct location in the sorted sub-array until array A is completely sorted.

Time Complexity Analysis:

- Selection sort algorithm consists of two nested loops.
- Owing to the two nested loops, it has $O(n^2)$ time complexity.

Insertion Sort	Time Complexity
Best Case	N
Average Case	n^2
Worst Case	n^2

Space Complexity Analysis:

- Selection sort is an in-place algorithm.
- It performs all computation in the original array and no other array is used.
- Hence, the space complexity works out to be O(1).

SELECTION SORT:

- Selection sort is one of the easiest approaches to sorting.
- It is inspired from the way in which we sort things out in day to day life.
- It is an in-place sorting algorithm because it uses no auxiliary data structures while sorting.

Working of Selection Sort:

Consider the following elements are to be sorted in ascending order using selection sort-

6, 2, 11, 7, 5

Selection sort works as:

- It finds the first smallest element (2).
- It swaps it with the first element of the unordered list.
- It finds the second smallest element (5).
- It swaps it with the second element of the unordered list.
- Similarly, it continues to sort the given elements.

As a result, sorted elements in ascending order are-2, 5, 6, 7, 11

Selection Sort Algorithm:

Let A be an array with n elements. Then, selection sort algorithm used for sorting is as follows-

Selection sort algorithm	

for(i = 0 ; i < n-1 ; i++)
{
index = i;
for(j = i+1 ; j < n ; j++)
{
if(A[j]< A[index])
index = j;
}
temp = A[i];
A[i] = A[index];
A[index] = temp;
}
	Here,
	• i = variable to traverse the array A
	• index = variable to store the index of minimum element
	• j = variable to traverse the unsorted sub-array
	• temp = temporary variable used for swapping

Selection Sort Example:

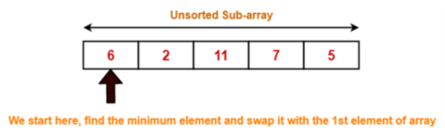
Consider the following elements to be sorted in an ascending order: 6, 2, 11, 7, 5

With each loop cycle,

- The minimum element in unsorted sub-array is selected.
- It is then placed at the correct location in the sorted sub-array until array A is completely sorted.

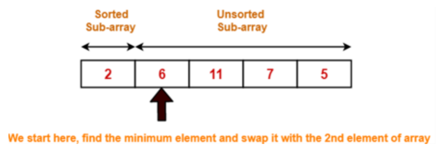
The above Selection sort algorithm works as illustrated below

Step-01: For i = 0



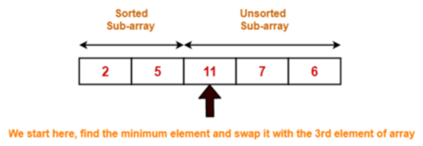
Unsorted Sub-array
6
2
11
7
5
We start here, find the minimum element and swap it with the 1st element of array

Step-02: For i = 1



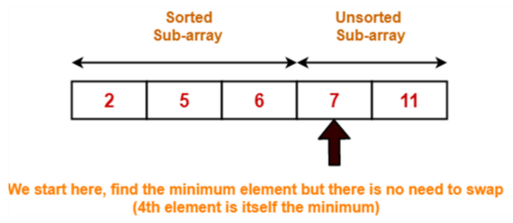
Sorted Sub-array
Unsorted Sub-array
2
6
11
7
5
We start here, find the minimum element and swap it with the 2nd element of array

Step-03: For i = 2



Sorted Sub-array
Unsorted Sub-array
2
5
11
7
6
We start here, find the minimum element and swap it with the 3rd element of array

Step-04: For i = 3

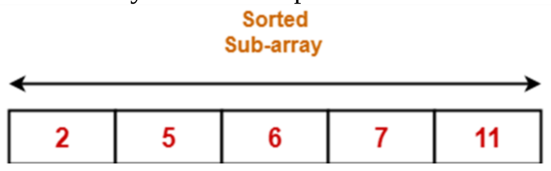


Sorted Sub-array
Unsorted Sub-array
2
5
6
7
11
We start here, find the minimum element but there is no need to swap (4th element is itself the minimum)

Step-05: For i = 3

Loop gets terminated as ‘i’ becomes 4.

The state of array after the loops are finished is as shown



Time Complexity Analysis:

- ➢ Selection sort algorithm consists of two nested loops.
- ➢ Owing to the two nested loops, it has O(n^2) time complexity.

Selection Sort	**Time Complexity**
Best Case	n^2
Average Case	n^2
Worst Case	n^2

Space Complexity Analysis:

- ➢ Selection sort is an in-place algorithm.
- ➢ It performs all computation in the original array and no other array is used.
- ➢ Hence, the space complexity works out to be O(1).

QUICK SORT:

- ➢ Quick Sort is a famous sorting algorithm.
- ➢ It sorts the given data items in ascending order.
- ➢ It uses the idea of divide and conquer approach.
- ➢ It follows a recursive algorithm.

Quick Sort Algorithm:

Consider

- a = Linear Array in memory
- beg = Lower bound of the sub array in question
- end = Upper bound of the sub array in question

Quick sort algorithm

```
Partition_Array(a , beg , end , loc)
1.  Begin
2.  Set left = beg , right = end , loc = beg
3.  Set done = false
4.  While(not done) do
5.  While((a[loc]<= a[right])and(loc ≠ right)) do
6.  Set right = right - 1
7.  end while
8.  if(loc = right) then
9.  Set done = true
10. else if(a[loc]> a[right]) then
11. Interchange a[loc] and a[right]
12. Set loc = right
13. end if
14. if(not done) then
15. While((a[loc]>= a[left])and(loc ≠ left)) do
16. Set left = left + 1
17. end while
18. if(loc = left) then
19. Set done = true
20. else if(a[loc]< a[left]) then
21. Interchange a[loc] and a[left]
22. Set loc = left
23. end if
24. end if
25. end while
26. End
```

Then, Quick Sort Algorithm is as follows-

Working of Quick Sort:

- Quick Sort follows a recursive algorithm.
- It divides the given array into two sections using a partitioning element called as pivot.

The division performed is such that-

- All the elements to the left side of pivot are smaller than pivot.
- All the elements to the right side of pivot are greater than pivot.

After dividing the array into two sections, the pivot is set at its correct position. Then, sub arrays are sorted separately by applying quick sort algorithm recursively.

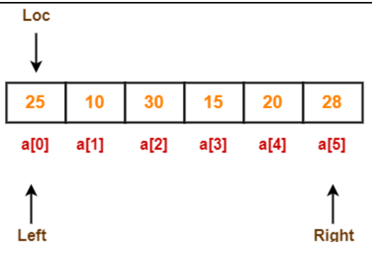
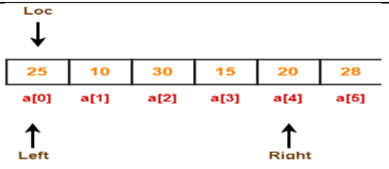
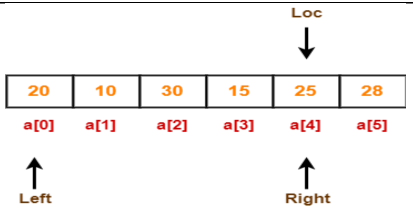
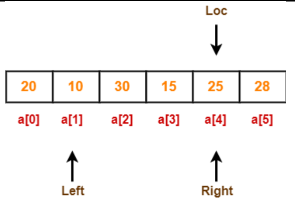
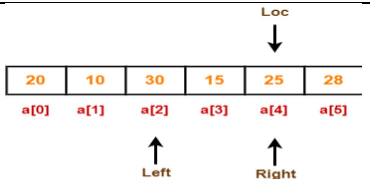
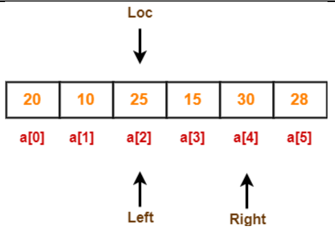
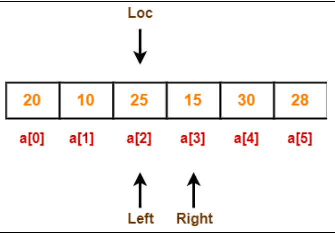
Quick Sort Example:

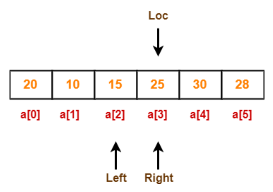
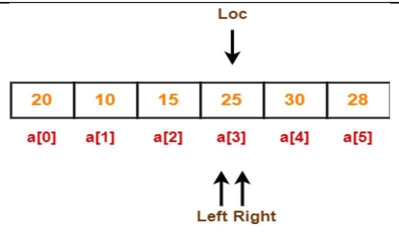
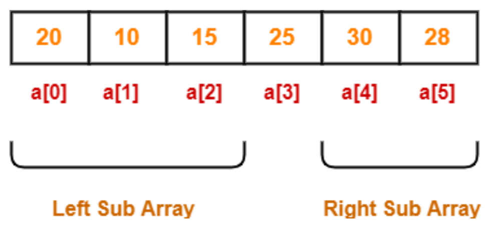
Consider the following array to be sorted in ascending order using quick sort algorithm-

25	10	30	15	20	28
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]

Quick Sort Example

Quick Sort Algorithm works in the following steps-

Step-01	Initially- • **Left** and **Loc** (pivot) points to the first element of the array. • **Right** points to the last element of the array. So to begin with, we set **loc** = 0, **left** = 0 and **right** = 5 as	 Loc 25 10 30 15 20 28 a[0] a[1] a[2] a[3] a[4] a[5] Left Right
Step-02	Since **loc** points at **left**, so algorithm starts from **right** and move towards left. As a[loc] < a[right], so algorithm moves **right** one position towards left as Now, **loc** = 0, **left** = 0 and **right** = 4.	 Loc 25 10 30 15 20 28 a[0] a[1] a[2] a[3] a[4] a[5] Left Right
Step-03	Since **loc** points at **left**, so algorithm starts from **right** and move towards left. As a[loc] > a[right], so algorithm swaps a[loc] and a[right] and **loc** points at **right** as Now, **loc** = 4, **left** = 0 and **right** = 4.	 Loc 20 10 30 15 25 28 a[0] a[1] a[2] a[3] a[4] a[5] Left Right
Step-04	Since **loc** points at **right**, so algorithm starts from **left** and move towards right. As a[loc] > a[left], so algorithm moves **left** one position towards right as- Now, **loc** = 4, **left** = 1 and **right** = 4.	 Loc 20 10 30 15 25 28 a[0] a[1] a[2] a[3] a[4] a[5] Left Right
Step-05	Since **loc** points at right, so algorithm starts from **left** and move towards right. As a[loc] > a[left], so algorithm moves **left** one position towards right as- Now, **loc** = 4, **left** = 2 and **right** = 4.	 Loc 20 10 30 15 25 28 a[0] a[1] a[2] a[3] a[4] a[5] Left Right
Step-06	Since **loc** points at **right**, so algorithm starts from **left** and move towards right. As a[loc] < a[left], so we algorithm swaps a[loc] and a[left] and **loc** points at **left** as- Now, **loc** = 2, **left** = 2 and **right** = 4.	 Loc 20 10 25 15 30 28 a[0] a[1] a[2] a[3] a[4] a[5] Left Right
Step-07	Since **loc** points at **left**, so algorithm starts from **right** and move towards left. As a[loc] < a[right], so algorithm moves **right** one position towards left as- Now, **loc** = 2, **left** = 2 and **right** = 3.	 Loc 20 10 25 15 30 28 a[0] a[1] a[2] a[3] a[4] a[5] Left Right

Step-08	Since **loc** points at **left**, so algorithm starts from **right** and move towards left. As a[loc] > a[right], so algorithm swaps a[loc] and a[right] and **loc** points at **right** as- Now, **loc** = 3, **left** = 2 and **right** = 3.	 Loc 20 10 15 25 30 28 a[0] a[1] a[2] a[3] a[4] a[5] Left Right
Step-09	Since **loc** points at **right**, so algorithm starts from **left** and move towards right. As a[loc] > a[left], so algorithm moves **left** one position towards right as- Now, **loc** = 3, **left** = 3 and **right** = 3.	 Loc 20 10 15 25 30 28 a[0] a[1] a[2] a[3] a[4] a[5] Left Right
	Now, • **loc**, **left** and **right** points at the same element. • This indicates the termination of procedure. • The pivot element 25 is placed in its final position. • All elements to the right side of element 25 are greater than it. • All elements to the left side of element 25 are smaller than it.	 20 10 15 25 30 28 a[0] a[1] a[2] a[3] a[4] a[5] Left Sub Array Right Sub Array

Now, quick sort algorithm is applied on the left and right sub arrays separately in the similar manner.

Quick Sort Analysis:

- To find the location of an element that splits the array into two parts, O(n) operations are required.
- This is because every element in the array is compared to the partitioning element.
- After the division, each section is examined separately.
- If the array is split approximately in half (which is not usually), then there will be $\log_2 n$ splits.
- Therefore, total comparisons required are $f(n) = n \times \log_2 n = O(n\log_2 n)$.
- Order of Quick Sort = $O(n\log_2 n)$

Worst Case:

- Quick Sort is sensitive to the order of input data.
- It gives the worst performance when elements are already in the ascending order.
- It then divides the array into sections of 1 and (n-1) elements in each call.
- Then, there are (n-1) divisions in all.
- Therefore, here total comparisons required are $f(n) = n \times (n-1) = O(n^2)$.
- Order of Quick Sort in worst case = $O(n^2)$

Advantages of Quick Sort:

The advantages of quick sort algorithm are-

- Quick Sort is an in-place sort, so it requires no temporary memory.
- Quick Sort is typically faster than other algorithms.(because its inner loop can be efficiently implemented on most architectures)
- Quick Sort tends to make excellent usage of the memory hierarchy like virtual memory or caches.
- Quick Sort can be easily parallelized due to its divide and conquer nature.

Disadvantages of Quick Sort:

The disadvantages of quick sort algorithm are-

- The worst case complexity of quick sort is $O(n^2)$.
- This complexity is worse than O(nlogn) worst case complexity of algorithms like merge sort, heap sort etc.
- It is not a stable sort i.e. the order of equal elements may not be preserved.

REVIEW OF VARIOUS SORTING ALGORITHMS:

Review of Sorting:

So far we have seen a number of algorithms for sorting a list of numbers in ascending order. Recall that an in-place sorting algorithm is one that uses no additional array storage (however, we allow Quick sort to be called in-place even though they need a stack of size O (log n) for keeping track of the recursion). A sorting algorithm is stable if duplicate elements remain in the same relative position after sorting.

Slow Algorithms: Include Bubble Sort, Insertion Sort and Selection Sort. These are all simple Θ (n 2) in-place sorting algorithms. Bubble Sort and Insertion Sort can be implemented as stable algorithms, but Selection Sort cannot (without significant modifications).

Merge sort:

Merge sort is a stable Θ(n log n) sorting algorithm. The downside is that Merge Sort is the only algorithm of the three that requires additional array storage, implying that it is not an in-place algorithm.

Quick sort:

Widely regarded as the fastest of the fast algorithms. This algorithm is O(n log n) in the expected case and O(n2) in the worst case. The probability that the algorithm takes asymptotically longer (assuming that the pivot is chosen randomly) is extremely small for large n. It is an (almost) in-place sorting algorithm but is not stable.

Heap sort:

Heap sort is based on a nice data structure, called a heap, which is a fast priority queue. Elements can be inserted into a heap in O(log n) time and the largest item can be extracted in O(log n) time. (It is also easy to set up a heap for extracting the smallest item.) If only we want to extract the k largest values, a heap can allow us to do this is O(n + k log n) time. It is an in-place algorithm, but it is not stable.

Analysis of different sorting techniques

Analysis based on important properties of different sorting techniques including their complexity, stability and memory constraints.

Time complexity Analysis:

We have discussed the best, average and worst case complexity of different sorting techniques with possible scenarios.

Comparison based sorting:

In comparison based sorting, elements of an array are compared with each other to find the sorted array.

Time and Space Complexity Comparison Table:

Sorting Algorithm	**Time Complexity**			**Space Complexity**
	Best Case	**Average Case**	**Worst Case**	**Worst Case**
Bubble Sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Selection Sort	$\Omega(N^2)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Insertion Sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Merge Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N \log N)$	$O(N)$
Heap Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N \log N)$	$O(1)$
Quick Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N^2)$	$O(N \log N)$
Radix Sort	$\Omega(N k)$	$\Theta(N k)$	$O(N k)$	$O(N + k)$
Count Sort	$\Omega(N + k)$	$\Theta(N + k)$	$O(N + k)$	$O(k)$
Bucket Sort	$\Omega(N + k)$	$\Theta(N + k)$	$O(N^2)$	$O(N)$

- **Bubble sort and Insertion sort:**
  Average and worst case time complexity: n^2
  Best case time complexity: n when array is already sorted.
  Worst case: when the array is reverse sorted.
- **Selection sort:**
  Best, average and worst case time complexity: n^2 which is independent of distribution of data.

- **Merge sort:**
  Best, average and worst case time complexity: nlogn which is independent of distribution of data.
- **Heap sort:**
  Best, average and worst case time complexity: nlogn which is independent of distribution of data.
- **Quick sort :**
  It is a divide and conquer approach with recurrence relation:
  $T(n) = T(k) + T(n-k-1) + cn$
  Worst case: when the array is sorted or reverse sorted, the partition algorithm divides the array in two sub-arrays with 0 and n-1 elements. Therefore,
  $T(n) =T(0) + T(n-1) +cn$
  Solving this we get, **$T(n) = O(n^2)$**
  Best case and Average case: On an average, the partition algorithm divides the array in two sub-arrays with equal size. Therefore,$T(n) =2T(n/2) + cn$

  Solving this we get, **$T(n) = O(n \log n)$**

Non-comparison based sorting:
In non-comparison based sorting, elements of array are not compared with each other to find the sorted array.

- **Radix sort:**
  Best, average and worst case time complexity: nk where k is the maximum number of digits in elements of array.
- **Count sort:**
  Best, average and worst case time complexity: n+k where k is the size of count array.
- **Bucket sort:**
  Best and average time complexity: n+k where k is the number of buckets.
  Worst case time complexity: n^2 if all elements belong to same bucket.

In-place/Outplace technique:
A sorting technique is in-place if it does not use any extra memory to sort the array. Among the comparison based techniques discussed, only merge sort is out placed technique as it requires an extra array to merge the sorted sub-arrays.
Among the non-comparison based techniques discussed, all are out placed techniques. Counting sort uses a counting array and bucket sort uses a hash table for sorting the array.

Online/Offline technique:
A sorting technique is considered online if it can accept new data while the procedure is ongoing i.e. complete data is not required to start the sorting operation.
Among the comparison based techniques discussed, only Insertion Sort qualifies for this because of the underlying algorithm it uses i.e. it processes the array (not just elements) from left to right and if new elements are added to the right, it doesn't impact the ongoing operation.

Stable/Unstable technique:

A sorting technique is stable if it does not change the order of elements with the same value. Out of comparison based techniques, bubble sort, insertion sort and merge sort are stable techniques. Selection sort is unstable as it may change the order of elements with the same value. For example, consider the array 4, 4, 1, 3.

In the first iteration, the minimum element found is 1 and it is swapped with 4 at 0th position. Therefore, the order of 4 with respect to 4 at the 1st position will change. Similarly, quick sort and heap sort are also unstable.

Out of non-comparison based techniques, Counting sort and Bucket sort are stable sorting techniques whereas radix sort stability depends on the underlying algorithm used for sorting.

Analysis of sorting techniques:

- When the array is almost sorted, insertion sort can be preferred.
- When order of input is not known, merge sort is preferred as it has worst case time complexity of (n log n) and it is stable as well.
- When the array is sorted, insertion and bubble sort gives complexity of n but quick sort gives complexity of n^2.
- Insertion sorting algorithm will take the least time when all elements of input array are identical. Insertion sort will have the complexity of n when the input array is already sorted.

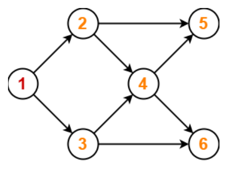
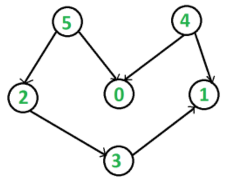
TOPOLOGICAL SORTING:

Topological Sorting or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex ‘u’ to vertex ‘v’, ‘u’ comes before ‘v’ in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

It is important to note that

- Topological Sorting is possible if and only if the graph is a **Directed Acyclic Graph**.
- There may exist multiple different topological orderings for a given directed acyclic graph.

Topological Sort Example:

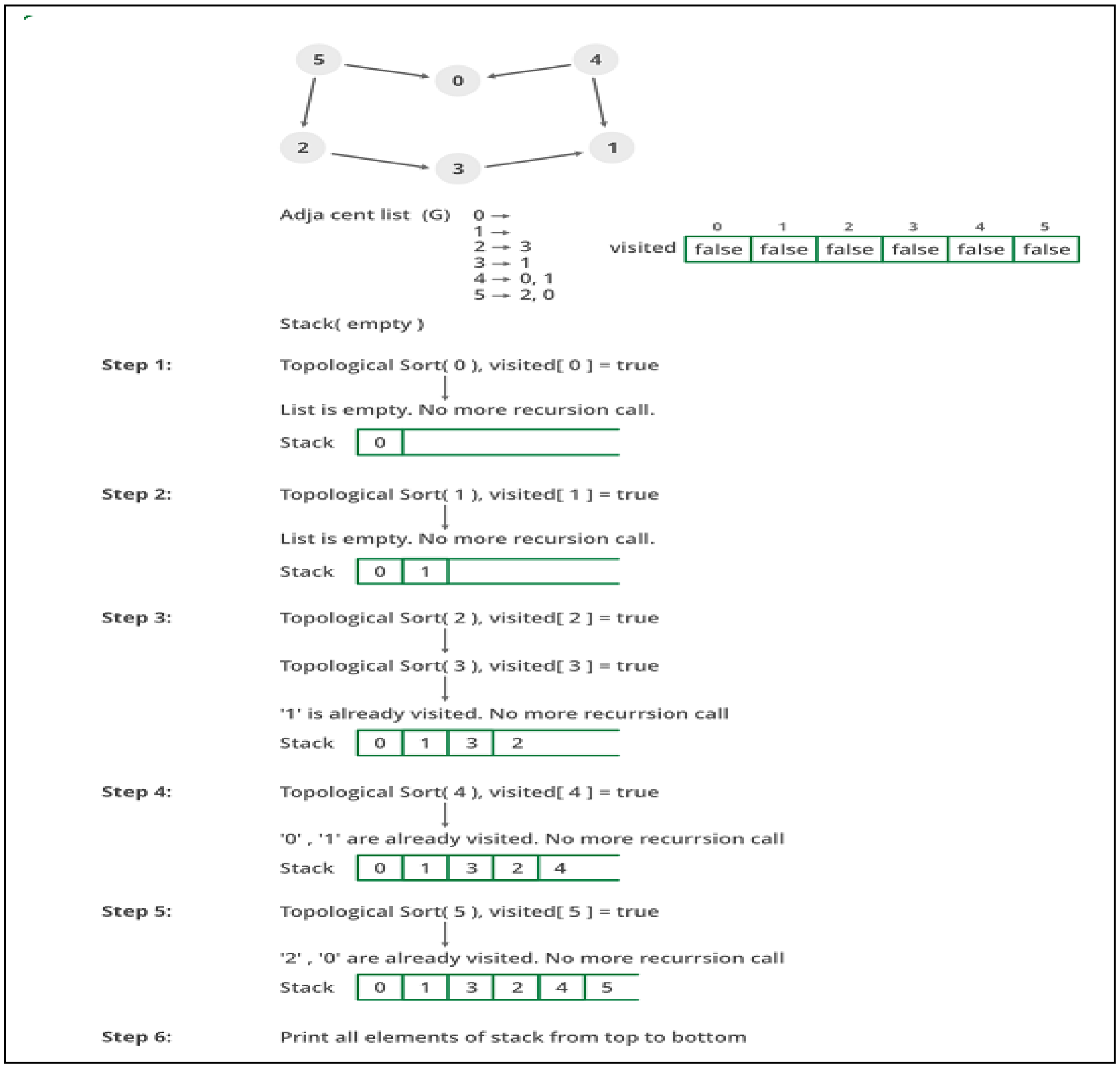
Consider the following directed acyclic graph-  Topological Sort Example For this graph, following 4 different topological orderings are possible- • 1 2 3 4 5 6 • 1 2 3 4 6 5 • 1 3 2 4 5 6 • 1 3 2 4 6 5	For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges). 

Topological Sorting Vs Depth First Search Traversal (DFS):

In DFS, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex ‘5’ should be printed before vertex ‘0’, but unlike DFS, the vertex ‘4’ should also be printed before vertex ‘0’. So topological sorting is different from DFS. For example, a DFS of the shown graph is “5 2 3 1 0 4”, but it is not a topological sorting.

Algorithm to find Topological Sorting:

To modify DFS is used to find Topological Sorting of a graph. In DFS, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don’t print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, and then push it to a stack. Finally, print contents of stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack. Below image is an illustration of this approach.



```
Adja cent list  (G)   0 →
                      1 →
                      2 → 3
                      3 → 1
                      4 → 0, 1
                      5 → 2, 0
```

	0	1	2	3	4	5
visited	false	false	false	false	false	false

Stack(empty)

Step 1: Topological Sort(0), visited[0] = true
↓
List is empty. No more recursion call.
Stack | 0 |

Step 2: Topological Sort(1), visited[1] = true
↓
List is empty. No more recursion call.
Stack | 0 | 1 |

Step 3: Topological Sort(2), visited[2] = true
↓
Topological Sort(3), visited[3] = true
↓
'1' is already visited. No more recurrsion call
Stack | 0 | 1 | 3 | 2 |

Step 4: Topological Sort(4), visited[4] = true
↓
'0' , '1' are already visited. No more recurrsion call
Stack | 0 | 1 | 3 | 2 | 4 |

Step 5: Topological Sort(5), visited[5] = true
↓
'2' , '0' are already visited. No more recurrsion call
Stack | 0 | 1 | 3 | 2 | 4 | 5 |

Step 6: Print all elements of stack from top to bottom

Applications of Topological Sort:

Few important applications of topological sort are

- Scheduling jobs from the given dependencies among jobs
- Instruction Scheduling
- Determining the order of compilation tasks to perform in make files
- Data Serialization

GRAPH: DEFINITIONS AND ELEMENTARY ALGORITHMS:

A graph is an abstract notation used to represent the connection between pairs of objects.

Graph is a data structure that consists of the following two components:

1**.** A finite set of vertices also called as nodes.

2**.** A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

A graph consists of

- **Vertices** − Interconnected objects in a graph are called vertices. Vertices are also known as nodes.
- **Edges** − Edges are the links that connect the vertices.

There are two types of graphs −

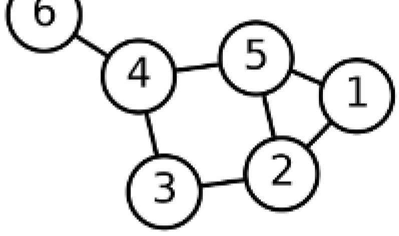
- **Directed graph** − In a directed graph, edges have direction, i.e., edges go from one vertex to another.
- **Undirected graph** − In an undirected graph, edges have no direction.

Graphs are used to represent many real-life applications:

- Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network.
- Graphs are also used in social networks like linkedin, Facebook. For example, in Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender and locale.

Graph Definitions

A **graph** *G* consists of two types of elements: **vertices** and **edges**. Each edge has two **endpoints**, which belong to the vertex set. We say that the edge **connects** (or joins) these two vertices. The **vertex set** of *G* is denoted *V(G)*, or just *V* if there is no ambiguity. An edge between vertices *u* and *v* is written as {*u*, *v*}. The **edge set** of *G* is denoted *E(G)*, or just *E* if there is no ambiguity. The graph in this picture has the vertex set V = {1, 2, 3, 4, 5, 6}. The edge set E = {{1, 2}, {1, 5}, {2, 3}, {2, 5}, {3, 4}, {4, 5}, {4, 6}}.



Self Loop: A **self-loop** is an edge whose end points are a single vertex. **Multiple edges** are two or more edges that join the same two vertices.

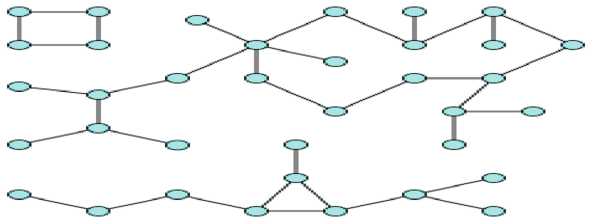
Multi Graph: A graph is called **simple** if it has no self-loops and no multiple edges, and a **multi graph** if it does have multiple edges.

Degree: The **degree** of a vertex *v* is the number of edges that connect to *v*.

Path: **Path** in a graph *G = (V, E)* is a sequence of vertices v_1, v_2, …, v_k, with the property that there are edges between v_i and v_{i+1}. We say that the path goes from v_1 to v_k. The sequence 6, 4, 5, 1, 2 is a path from 6 to 2 in the graph above. A path is **simple** if its vertices are all different.

Cycle: A **cycle** is a path v_1, v_2, …, v_k for which $k > 2$, the first $k - 1$ vertices are all different and $v_1 = v_k$. The sequence 4, 5, 2, 3, 4 is a cycle in the graph above.A graph is **connected** if for every pair of vertices *u* and *v*, there is a path from *u* to *v*.If there is a path connecting *u* and *v*, the **distance** between these vertices is defined as the minimal number of edges on a path from *u* to *v*.

Connected Component: A **connected component** is a sub-graph of maximum size, in which every pair of vertices are connected by a path. Here is a graph with three connected components.

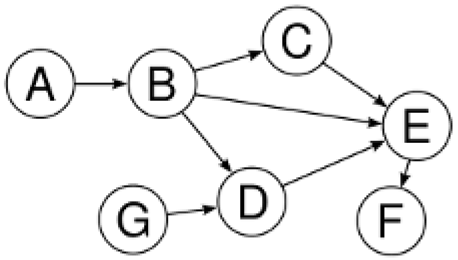


Trees: A tree **is a connected simple acyclic graph. A vertex with degree 1 in a tree is called a** leaf**.**

Directed graphs:

A directed graph **or** digraph **G = (V, E) consists of a vertex set V and an edge set of** ordered pairs **E of elements in the vertex set.**

Here is a simple acyclic digraph (often called a **DAG**, “directed acyclic graph”) with seven vertices and eight edges.

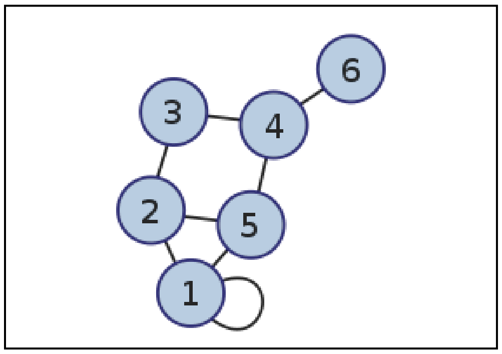


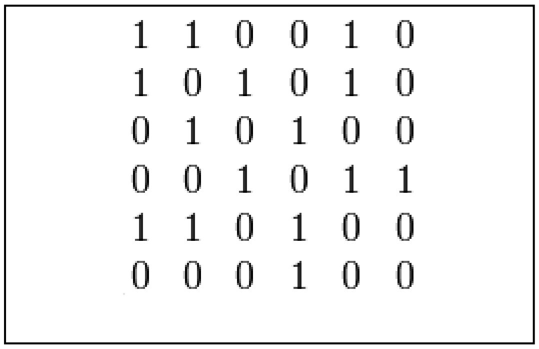
Adjacency matrix:

An adjacency matrix is a |*V*|x|*V*|-matrix of integers, representing a graph *G = (V, E)*.

- The vertices are number from 1 to |*V*|.
- The number at position (*i*, *j*) indicates the number of edges from *i* to *j*.

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w. Here is an undirected graph and its symmetric adjacency matrix.





$$\begin{matrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{matrix}$$

The adjacency matrix representation is best suited for **dense graphs**, graphs in which the number of edges is close to the maximal.

In a **sparse graph**, an adjacency matrix will have a large **memory overhead** and **finding all neighbors** of a vertex will be **costly**.

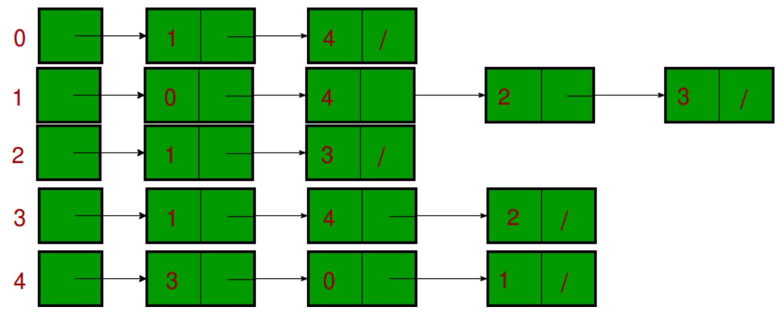
Pros*:* Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).

Cons*:* Consumes more space O(V^2). Even if the graph is sparse (contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.

Adjacency list:

The adjacency list graph data structure is well suited for sparse graphs. It consists of an array of size |*V*|, where position *k* in the array contains a list of all neighbors to *k*.

- Note that the “neighbor list” doesn’t have to be an actual list. It could be any data structure representing a set of vertices. Hash tables, arrays or linked lists are common choices.
- An array of lists is used. Size of the array is equal to the number of vertices. Let the array be array [ ]. An entry array[i] represents the list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is adjacency list representation of the above graph.



There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

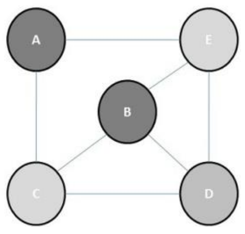
Graph Coloring

Graph coloring is a method to assign colors to the vertices of a graph so that no two adjacent vertices have the same color. Some graph coloring problems are −

- **Vertex coloring** − A way of coloring the vertices of a graph so that no two adjacent vertices share the same color.
- **Edge Coloring** − It is the method of assigning a color to each edge so that no two adjacent edges have the same color.
- **Face coloring** − It assigns a color to each face or region of a planar graph so that no two faces that share a common boundary have the same color.

Chromatic Number

Chromatic number is the minimum number of colors required to color a graph. For example, the chromatic number of the following graph is 3.



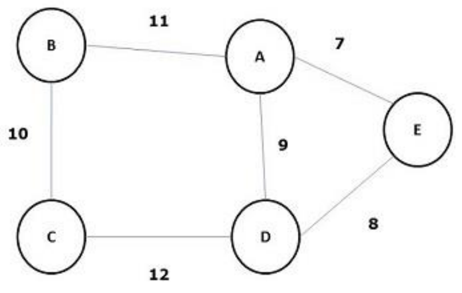
The concept of graph coloring is applied in preparing timetables, mobile radio frequency assignment, Suduku game, register allocation and coloring of maps.

Steps for graph coloring

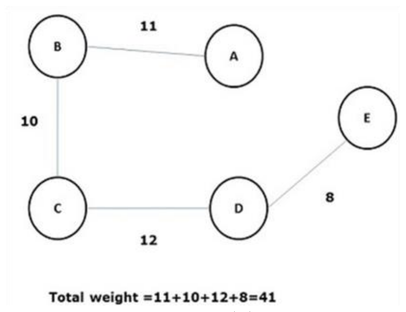
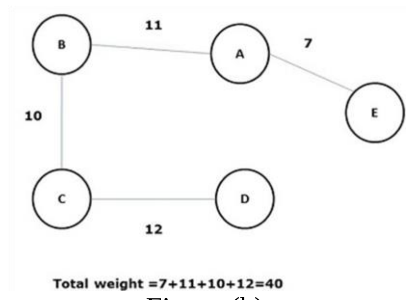
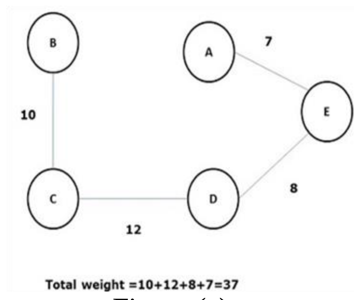
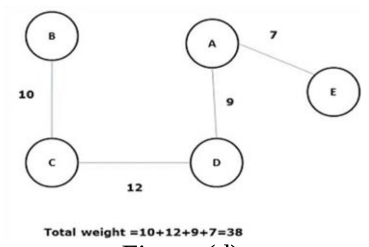
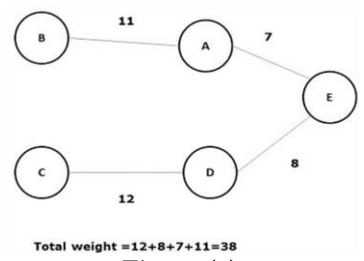
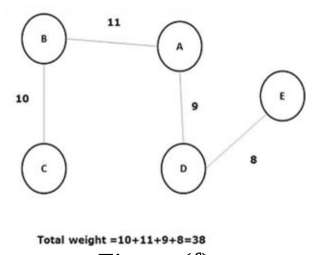
- Set the initial value of each processor in the n-dimensional array to 1.
- Now to assign a particular color to a vertex, determine whether that color is already assigned to the adjacent vertices or not.
- If a processor detects same color in the adjacent vertices, it sets its value in the array to 0.
- After making n^2 comparisons, if any element of the array is 1, then it is a valid coloring.

Minimal Spanning Tree

A spanning tree whose sum of weight (or length) of all its edges is less than all other possible spanning tree of graph G is known as a **minimal spanning tree** or **minimum cost spanning** tree. The following figure shows a weighted connected graph.



Some possible spanning trees of the above graph are shown below:

 Figure (a)	 Figure (b)	 Figure (c)
 Figure (d)	 Figure (e)	 Figure (f)

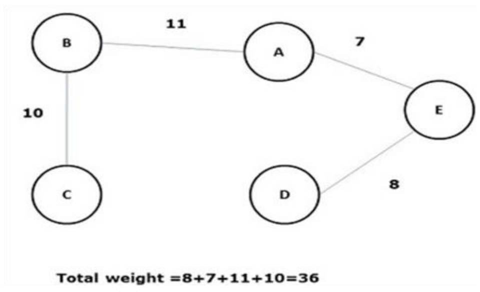


Figure (g) Minimum Spanning Tree

Among all the above spanning trees, figure (g) is the minimum spanning tree. The concept of minimum cost spanning tree is applied in travelling salesman problem, designing electronic circuits, designing efficient networks and designing efficient routing algorithms. To implement the minimum cost-spanning tree, the following two methods are used:

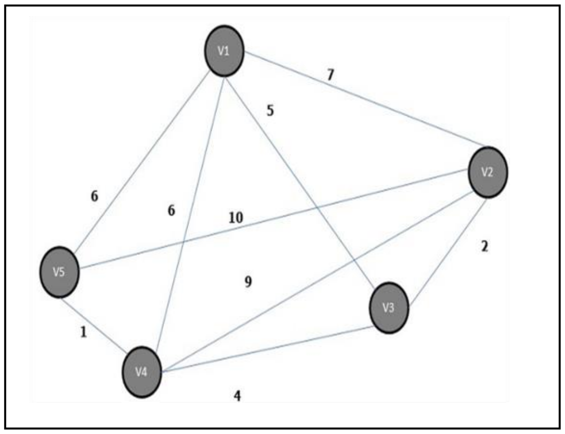
- Prim’s Algorithm
- Kruskal’s Algorithm

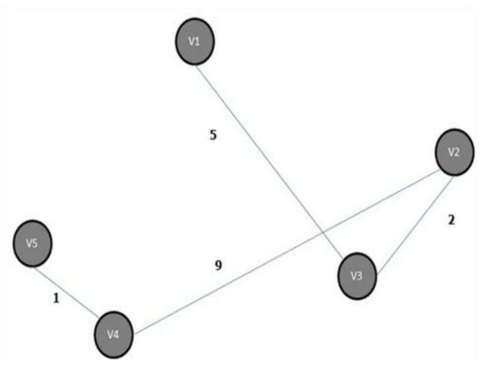
Prim's Algorithm

Prim’s algorithm is a greedy algorithm, which helps us find the minimum spanning tree for a weighted undirected graph. It selects a vertex first and finds an edge with the lowest weight incident on that vertex.

Steps for Prim’s Algorithm:

1. Select any vertex, say v_1 of Graph G.
2. Select an edge, say e_1 of G such that $e_1 = v_1 v_2$ and $v_1 \neq v_2$ and e_1 has minimum weight among the edges incident on v_1 in graph G.
3. Now, following step 2, select the minimum weighted edge incident on v_2.
4. Continue this till n–1 edges have been chosen. Here **n** is the number of vertices.





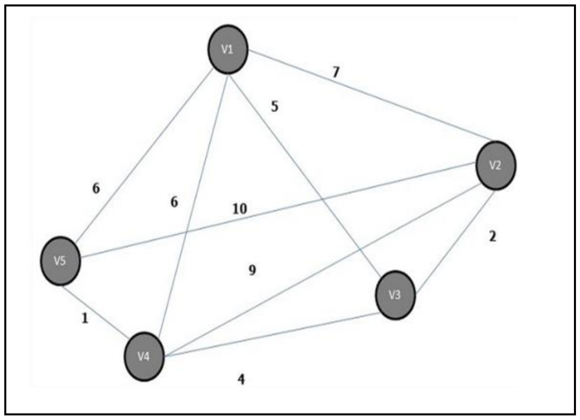
The minimum spanning tree is

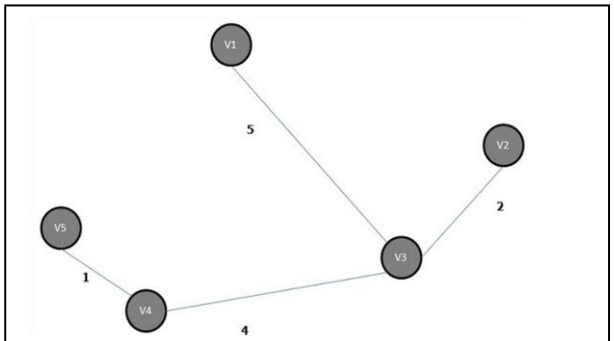
Kruskal's Algorithm

Kruskal’s algorithm is a greedy algorithm, which helps us find the minimum spanning tree for a connected weighted graph, adding increasing cost arcs at each step. It is a minimum-spanning-tree algorithm that finds an edge of the least possible weight that connects any two trees in the forest.

Steps of Kruskal’s Algorithm

- Select an edge of minimum weight; say e_1 of Graph G and e_1 is not a loop.
- Select the next minimum weighted edge connected to e_1.
- Continue this till n-1 edges have been chosen. Here n is the number of vertices.





The minimum spanning tree of the given graph

SHORTEST PATH ALGORITHM

Shortest Path algorithm is a method of finding the least cost path from the source node(S) to the destination node (D). Here, we will discuss Moore's algorithm, also known as Breadth First Search Algorithm.

Moore's algorithm

1. Label the source vertex, S and label it **i** and set **i=0**.
2. Find all unlabeled vertices adjacent to the vertex labeled **i**. If no vertices are connected to the vertex, S, then vertex, D, is not connected to S. If there are vertices connected to S, label them **i+1**.
3. If D is labeled, then go to step 4, else go to step 2 to increase i=i+1.
4. Stop after the length of the shortest path is found.

SHORTEST PATH BY BFS:

SEARCH ALGORITHMS -BREADTH-FIRST SEARCH

Breadth-first search (BFS) also visits all vertices that belong to the same component as *v*. However, the vertices are visited in distance order: the algorithm first visits *v*, then all neighbors of *v*, then their neighbors, and so on.

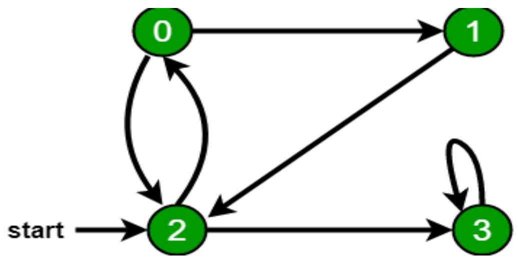
```
AlgorithmBFS(G, v)
   Q ← new empty FIFO queue
   Mark v as visited.
Q.enqueue(v)
while Q is not empty
a ← Q.dequeue()
      // Perform some operation on a.
```

```
for all unvisited neighbors x of a
        Mark x as visited.
Q.enqueue(x)
```

Before running the algorithm, all |V| vertices must be marked as not visited.

Breadth First Search or BFS for a Graph

For a graph Breadth First Traversal (or Search) is similar to Breadth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex. For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.



Time complexity

The time complexity of BFS can be computed as the total number of iterations performed by thefor loop.Let E' be the set of all edges in the connected component visited by the algorithm. For each edge {*u*, *v*} in E' the algorithm makes two for loop iteration steps: one time when the algorithm visits the neighbors of *u*, and one time when it visits the neighbors of *v*.Hence, the time complexity is Θ(|V| + |E'|).

Applications of Breadth-first Search (BFS):

1. **Shortest Path and Minimum Spanning Tree for unweighted graph** In an unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using the minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.
2. **Peer to Peer Networks.** In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.
3. **Crawlers in Search Engines:** Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be

used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of the built tree can be limited.

4. **Social Networking Websites:** In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.
5. **GPS Navigation systems:** Breadth First Search is used to find all neighboring locations.
6. **Broadcasting in Network:** In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
7. **Garbage Collection:** Breadth First Search is used in copying garbage collection using Cheney's algorithm. Breadth First Search is preferred over Depth First Search because of better locality of reference.
8. **Cycle detection in undirected graph:** In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. We can use BFS to detect cycle in a directed graph also.
9. **Ford–Fulkerson algorithm** In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to $O(VE^2)$.
10. **To test if a graph is Bipartite** We can either use Breadth First or Depth First Traversal.
11. **Path Finding** We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.
12. **Finding all nodes within one connected component:** We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node. Many algorithms like Prim's Minimum Spanning Tree and Dijkstra's Single Source Shortest Path use structure similar to Breadth First Search.

SHORTEST PATH IN EDGE-WEIGHTED CASE (DIJKSTRA'S):

DIJKSTRA'S ALGORITHM

Dijkstra's algorithm computes the shortest path from a vertex s, the source, to all other vertices. The graph must have non-negative edge costs.

Dijkstra's algorithm is applicable for:

- ➢ Both directed and undirected graphs
- ➢ All edges must have nonnegative weights.
- ➢ Graph must be connected

The algorithm returns two arrays:

- dist[k] holds the length of a shortest path from s to k,
- prev[k] holds the previous vertex in a shortest path from s to k.

```
AlgorithmDijkstra(G, s)
for each vertex v in G
dist[v] ← ∞
```

```
prev[v] ← undefined
dist[s] ← 0
   Q ← the set of all nodes in G
while Q is not empty
u ← vertex in Q with smallest distance in dist[]
      Remove u from Q.
ifdist[u] = ∞
break
for each neighbor v of u
alt ← dist[u] + dist_between(u, v)
if alt <dist[v]
dist[v] ← alt
prev[v] ← u
returndist[], prev[]
```

Given a graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph. Dijkstra’s algorithm is very similar to Prim’s algorithm for minimum spanning tree. Like Prim’s MST, we generate a *SPT (shortest path tree)* with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, and other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source. Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the givengraph.

Algorithm

Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

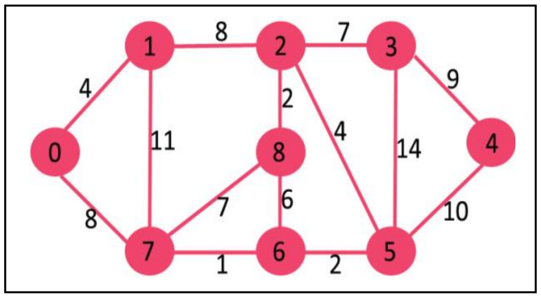
1. Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
2. While *sptSet* doesn’t include all vertices

   a) Pick a vertex u which is not there in *sptSet* and has minimum distance value.

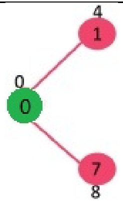
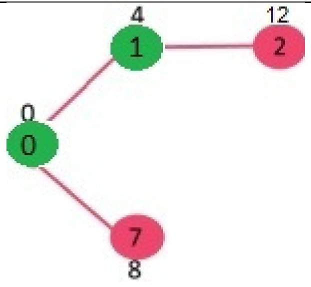
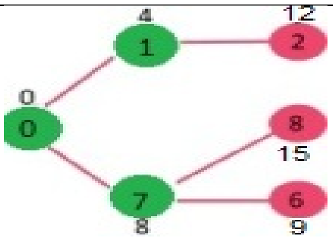
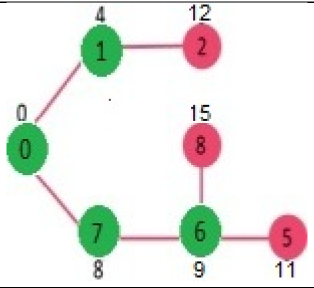
   b) Include u to *sptSet*.

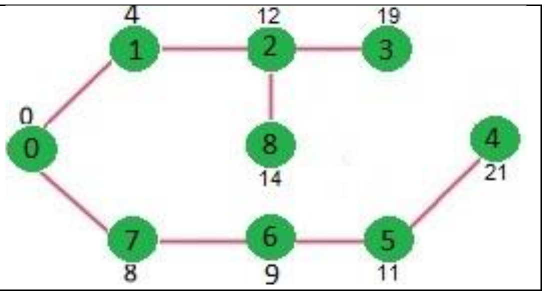
   c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Let us understand with the following example:



The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8.

Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.	
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.	
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).	
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.	

We repeat the above steps until *sptSet* does include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).	 0 0 1 4 2 12 3 19 8 14 4 21 7 8 6 9 5 11

➢ **Time complexity**

To compute the time complexity we can use the same type of argument as for BFS. The main difference is that we need to account for the cost of adding, updating and finding the minimum distances in the queue. If we implement the queue with a heap, all of these operations can be performed in *O*(log |V|) time. This gives the time complexity *O*((|E| + |V|)log|V|).

➢ **Worst case time complexity:** Θ(E+V log V)

➢ **Average case time complexity**: Θ(E+V log V)

➢ **Best case time complexity:** Θ(E+V log V)

➢ **Space complexity:** Θ(V)

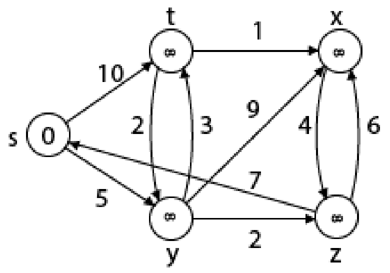
Real Time Applications of Dijkstra's Algorithm:

1. It is used in Google Maps
2. It is used in finding Shortest Path.
3. It is used in geographical Maps
4. To find locations of Map this refers to vertices of graph.
5. Distance between the locations refers to edges.
6. It is used in IP routing to find Open shortest Path First.
7. It is used in the telephone network. In a telephone network the lines have bandwidth, BW. We want to route the phone call via the highest BW.
8. Flight: A travel agent requests software for making an agenda of flights for clients. The agent has access to a data base with all airports and flights. Besides the flight number, origin airport and destination, the flights have departure and arrival time. Specifically the agent wants to determine the earliest arrival time for the destination given an origin airport and start time.
9. File Server: We want to designate a file server in a local area network. Now, we consider that most of time transmitting files from one computer to another computer is the connect time. So we want to minimize the number of "hops" from the file server to every other computer on the network.

Disadvantage of Dijkstra's Algorithm:

1. It does a blind search, so wastes a lot of time while processing.
2. It can't handle negative edges.
3. It leads to the acyclic graph and most often cannot obtain the right shortest path.
4. We need to keep track of vertices that have been visited.

Example for Dijkstra's algorithm:



Solution:

Step1: Q =[s, t, x, y, z]

We scanned vertices one by one and find out its adjacent. Calculate the distance of each adjacent to the source vertices.

We make a stack, which contains those vertices which are selected after computation of shortest distance.

Firstly we take's' in stack M (which is a source)

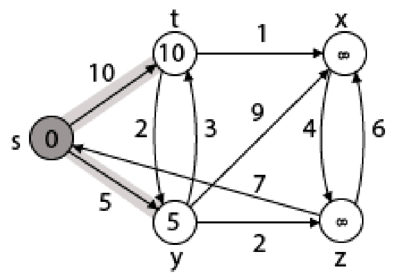
M = [S] Q = [t, x, y, z]

Step 2: Now find the adjacent of s that are t and y.

Adj [s] → t, y [Here s is u and t and y are v]

Case - (i) s → t
d [v] > d [u] + w [u, v]
d [t] > d [s] + w [s, t]
∞ > 0 + 10 [false condition]
Then **d [t] ← 10**
π [t] ← 5
Adj [s] ← t, y

Case - (ii) s→ y
d [v] > d [u] + w [u, v]
d [y] > d [s] + w [s, y]
∞ > 0 + 5 [false condition]
∞ > 5
Then **d [y] ← 5**
π [y] ← 5

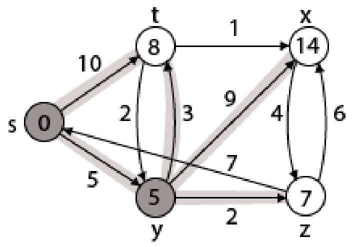


By comparing case (i) and case (ii)

Adj [s] → t = 10, y = 5

y is shortest

y is assigned in 5 = [s, y]



Step 3: Now find the adjacent of y that is t, x, z.

Adj [y] → t, x, z   [Here y is u and t, x, z are v]

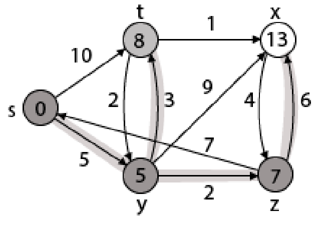
Case - (i) y →t	**Case - (ii)** y → x	**Case - (iii)** y → z
d [v] > d [u] + w [u, v]	d [v] > d [u] + w [u, v]	d [v] > d [u] + w [u, v]
d [t] > d [y] + w [y, t]	d [x] > d [y] + w [y, x]	d [z] > d [y] + w [y, z]
10 > 5 + 3	∞ > 5 + 9	∞ > 5 + 2
10 > 8	∞ > 14	∞ > 7
Then d [t] ← 8	Then d [x] ← 14	Then d [z] ← 7
π [t] ← y	π [x] ← 14	π [z] ← y

By comparing case (i), case (ii) and case (iii)

Adj [y] → x = 14, t = 8, z =7

z is shortest

z is assigned in 7 = [s, z]

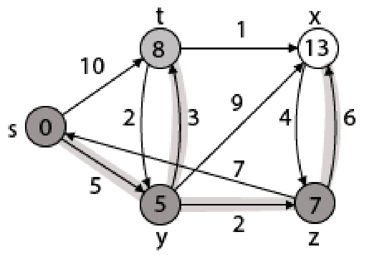


Step - 4 Now we will find adj [z] that are s, x

Adj [z] → [x, s]   [Here z is u and s and x are v]

Case - (i) z → x	**Case - (ii)** z → s
d [v] > d [u] + w [u, v]	d [v] > d [u] + w [u, v]
d [x] > d [z] + w [z, x]	d [s] > d [z] + w [z, s]
14 > 7 + 6	0 > 7 + 7
14 > 13	0 > 14
Then d [x] ← 13	∴ This condition does not satisfy so it will be discarded.
π [x] ← z	

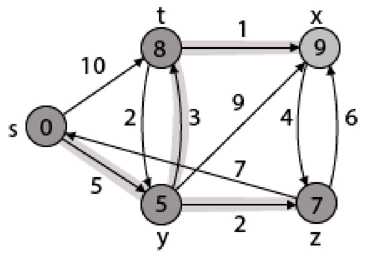
Now we have x = 13.



Step 5: Now we will find Adj [t]

Adj [t] → [x, y] [Here t is u and x and y are v]

Case - (i) t → x	**Case - (ii)** t → y
d [v] > d [u] + w [u, v]	d [v] > d [u] + w [u, v]
d [x] > d [t] + w [t, x]	d [y] > d [t] + w [t, y]
13 > 8 + 1	5 > 10
13 > 9	∴ This condition does not satisfy so it will be discarded.
Then d [x] ← 9	
π [x] ← t	

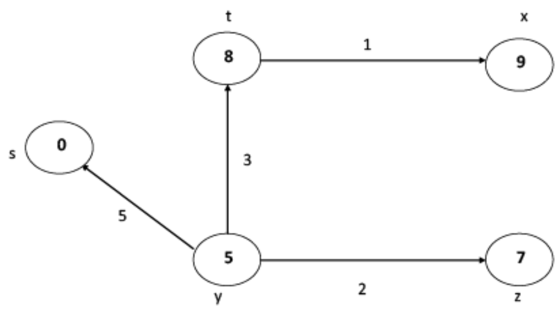


Thus we get all shortest path vertex as

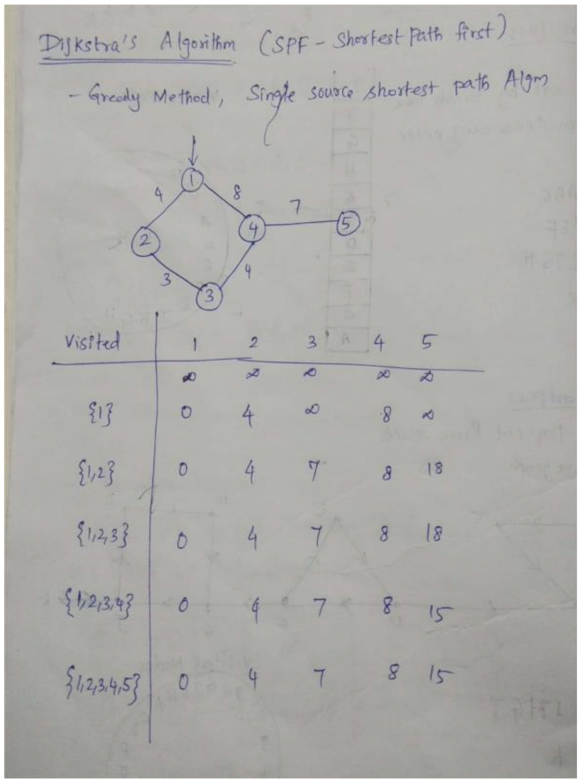
Weight from s to y is 5
Weight from s to z is 7
Weight from s to t is 8
Weight from s to x is 9

These are the shortest distance from the source's' in the given graph.

Final Shortest path is:



Example 2:

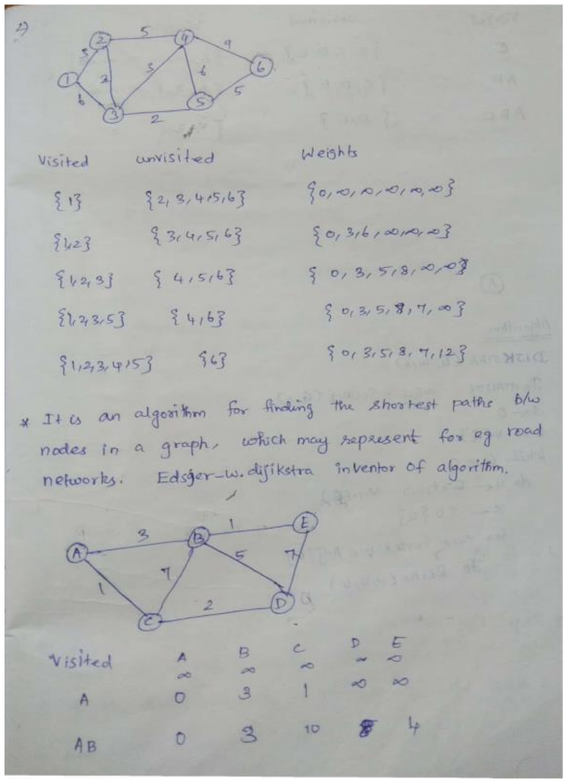


Dijkstra's Algorithm (SPF - Shortest Path first)

- Greedy Method, Single source shortest path Algm

Visited	1	2	3	4	5
	∞	∞	∞	∞	∞
{1}	0	4	∞	8	∞
{1,2}	0	4	7	8	18
{1,2,3}	0	4	7	8	18
{1,2,3,4}	0	4	7	8	15
{1,2,3,4,5}	0	4	7	8	15

Page 1



2)

Visited	unvisited	Weights
{1}	{2, 3, 4, 5, 6}	{0, ∞, ∞, ∞, ∞, ∞}
{1,2}	{3, 4, 5, 6}	{0, 3, 6, ∞, ∞, ∞}
{1,2,3}	{4, 5, 6}	{0, 3, 5, 8, ∞, ∞}
{1,2,3,5}	{4, 6}	{0, 3, 5, 8, 7, ∞}
{1,2,3,4,5}	{6}	{0, 3, 5, 8, 7, 12}

* It is an algorithm for finding the shortest paths b/w nodes in a graph, which may represent for eg road networks. Edsger-W. dijikstra inventor of algorithm.

Visited	A	B	C	D	E
	∞	∞	∞	∞	∞
A	0	3	1	∞	∞
AB	0	3	10	8	4

Page 2

DEPTH-FIRST SEARCH:

Depth-first search (DFS) is an algorithm that visits all edges in a graph G that belong to the same connected component as a vertex v.

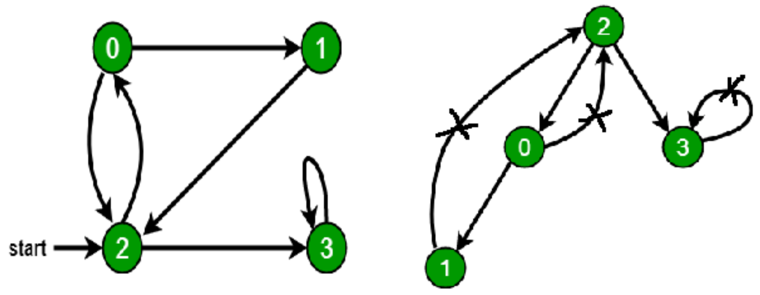
```
Algorithm DFS(G, v)
if v is already visited
return
    Mark v as visited.
    // Perform some operation on v.
for all neighbors x of v
DFS(G, x)
```

Before running the algorithm, all |V| vertices must be marked as not visited.

Depth First Search or DFS for a Graph

For a graph Depth First Traversal (or Search) is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Depth First Traversal of the following graph is 2, 0, 1, 3.



Time complexity

Let E' be the set of all edges in the connected component visited by the algorithm. The algorithm makes two calls to DFS for each edge {*u*, *v*} in E': one time when the algorithm visits the neighbors of *u*, and one time when it visits the neighbors of *v*. Hence, the time complexity of the algorithm is Θ(|V| + |E'|).

Applications of Depth First Search (DFS)

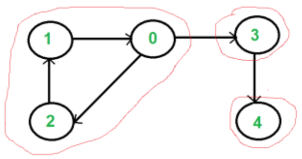
1. For a weighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.
2. **Detecting cycle in a graph**

A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges.

3. **Path Finding**
   We can specialize the DFS algorithm to find a path between two given vertices u and z.
   i) Call DFS(G, u) with u as the start vertex.
   ii) Use a stack S to keep track of the path between the start vertex and the current vertex.
   iii) As soon as destination vertex z is encountered, return the path as the contents of the stack
4. **Topological Sorting**
   Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when re-computing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in make files, data serialization and resolving symbol dependencies in linkers.
5. **To test if a graph is bipartite**
   We can augment either BFS or DFS when we first discover a new vertex, color it opposited its parents and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black!
6. **Finding Strongly Connected Components of a graph** A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex.
7. **Solving puzzles with only one solution**, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

STRONGLY CONNECTED COMPONENTS:

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (**SCC**) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph.



To find all strongly connected components in O(V+E) time using Kosaraju's algorithm.

Following is detailed Kosaraju's algorithm.
Create an empty stack 'S' and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack. In the above graph, if we start DFS from vertex 0, we get vertices in stack as 1, 2, 4, 3, 0.

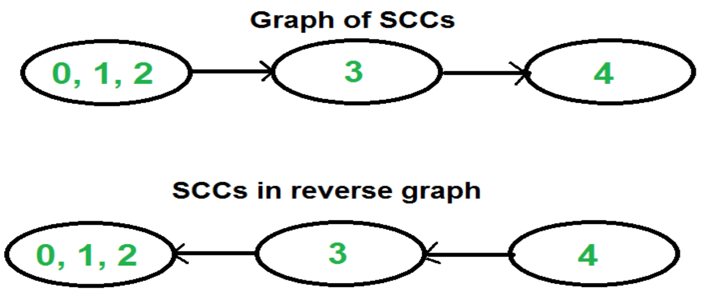
1. Reverse directions of all arcs to obtain the transpose graph.
2. One by one pop a vertex from S while S is not empty. Let the popped vertex be 'v'. Take v as source and do DFS (call DFSUtil(v)). The DFS starting from v prints strongly connected component of v. In the above example, we process vertices in order 0, 3, 4, 2, 1 (One by one popped from stack)

How does this work?

The above algorithm is DFS based. It does DFS two times. DFS of a graph produces a single tree if all vertices are reachable from the DFS starting point. Otherwise DFS produces a forest. So DFS of a graph with only one SCC always produces a tree. The important point to note is DFS may produce a tree or a forest when there are more than one SCCs depending upon the chosen starting point.

- For example, in the above diagram, if we start DFS from vertices 0 or 1 or 2, we get a tree as output. And if we start from 3 or 4, we get a forest.
- To find and print all SCCs, we would want to start DFS from vertex 4 (which is a sink vertex), then move to 3 which is sink in the remaining set (set excluding 4) and finally any of the remaining vertices (0, 1, 2). So how do we find this sequence of picking vertices as starting points of DFS? Unfortunately, there is no direct way for getting this sequence.
- However, if we do a DFS of graph and store vertices according to their finish times, we make sure that the finish time of a vertex that connects to other SCCs (other than its own SCC), will always be greater than finish time of vertices in the other SCC. For example, in DFS of above example graph, finish time of 0 is always greater than 3 and 4 (irrespective of the sequence of vertices considered for DFS).
- And finish time of 3 is always greater than 4. DFS doesn't guarantee about other vertices, for example finish times of 1 and 2 may be smaller or greater than 3 and 4 depending upon the sequence of vertices considered for DFS. So to use this property, we do DFS traversal of complete graph and push every finished vertex to a stack. In stack, 3 always appears after 4 and 0 appear after both 3 and 4.
- In the next step, we reverse the graph. Consider the graph of SCCs. In the reversed graph, the edges that connect two components are reversed. So the SCC {0, 1, 2} becomes sink and the SCC {4} becomes source.

As discussed above, in stack, we always have 0 before 3 and 4. So if we do a DFS of the reversed graph using sequence of vertices in stack, we process vertices from sink to source (in reversed graph). That is what we wanted to achieve and that is all needed to print SCCs one by one.



EMPHASIS ON CORRECTNESS PROOF OF THE ALGORITHM AND TIME/SPACE ANALYSIS:

When designing a completely new algorithm, a very thorough analysis of its correctness and efficiency is needed. The last thing we would want is our solution not being adequate for a problem it was designed to solve in the first place.

Mathematical Induction

Mathematical induction (MI) is an essential tool for proving the statement that proves an algorithm's correctness. The general idea of MI is to prove that a statement is true for every natural number n.

This means we have to go through 3 steps:

1. **Induction Hypothesis**: Define the rule we want to prove for every n, let's call the rule F(n)
2. **Induction Base**: Proving the rule is valid for an initial value, or rather a starting point - this is often proven by solving the Induction Hypothesis F(n) for n=1 or whatever initial value is appropriate
3. **Induction Step**: Proving that if we know that F(n) is true, we can step one step forward and assume F(n+1) is correct.

Basic Example:

Problem: If we define S(n) as the sum of the first n natural numbers, for example S(3) = 3+2+1, prove that the following formula can be applied to any n: S(n)=(n+1)∗n2S(n)=(n+1)∗n2

Let's trace our steps:

Induction Hypothesis: S(n) defined with the formula above

Induction Base: In this step we have to prove that S(1) = 1: S(1)=(1+1)∗12=22=1S(1)=(1+1)∗12=22=1

Induction Step: In this step we need to prove that if the formula applies to S(n), it also applies to S(n+1) as follows: S(n+1)=(n+1+1)∗(n+1)2=(n+2)∗(n+1)2S(n+1)=(n+1+1)∗(n+1)2=(n+2)∗(n+1)2

> This is known as an implication (a=>b), which just means that we have to prove b is correct providing we know a is correct.
>
> S(n+1)=S(n)+(n+1)=(n+1)∗n2+(n+1)=n2+n+2n+22S(n+1)=S(n)+(n+1)=(n+1)∗n2+(n+1)=n2+n+2n+22
>
> =n2+3n+22=(n+2)∗(n+1)2=n2+3n+22=(n+2)∗(n+1)2

Note that S(n+1) = S(n) + (n+1) just means we are recursively calculating the sum. Example with literals: S(3) = S(2) + 3= S(1) + 2 + 3 = 1 + 2 + 3 = 6.

In theoretical computer science, correctness of an algorithm is asserted when it is said that the algorithm is correct with respect to a specification. A termination proof is a type of mathematical proof that plays a critical role in formal verification because total correctness of an algorithm depends on termination.

Steps for proving by induction Description

The proof consists of two steps:

The basis (base case): prove that the statement holds for the first natural number n. Usually, n = 0 or n = 1.

The inductive step: prove that, if the statement holds for some natural number n, then the statement holds for n + 1.

- **Correctness –Aim:** Proving the correctness of algorithms
  - Loop Invariants
  - Mathematical Induction
- **Time Complexity – Aim**: Determining the cost of recursive algorithms
  - Recursion reminder
  - Expressing recursive algorithms as recurrences
  - Applying the Master Theorem

Correctness of an Algorithm:

Simply, an algorithm is correct if for any valid input it produces the result required by the algorithm's specification. For example, a sorting function: sort (int[ ] in)

- We specify that for a valid integer array as input
- The sort function will sort the input integer array into ascending numerical order
- The result that is returned is a correctly sorted integer array, for any valid array of integers

Testing:

• Testing is a critical skill for a developer

• Provides no guarantees of correctness

– Depends on generation of good test-cases

– Generally these follow sensible heuristics

• Test correct values, e.g. sort ({1, 2 ,3})

• Test incorrect values, e.g. sort ({'a', 'b', 'c'})

• Test boundaries, e.g. sort ({0, 1, ..., MAX_VALUE})

TIME COMPLEXITY

Progress and Running Time

- Invariants not only tell us something about the progress of an algorithm: – e.g. For a sorting algorithm
  - So far, all items are sorted up to some n [progress]
- They can tell us about running time or cost – e.g. For a sorting algorithm
  - The worst case performance will be O(n2) [running time]
- Complexity for iterative algorithms is mostly an exercise in counting operations, then stating the complexity.

AMORTIZED ANALYSIS:

It is defined as analyze a sequence of operations on a data structure. An amortized analysis is any strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive.

Goal: Show that although some individual operations may be expensive, *on average* the cost per operation is small.

- Average in this context does not mean that we're averaging over a distribution of inputs.
- No probability is involved.
- An amortized analysis guarantees the average performance of each operation in the worst case.

Types of Amortized Analysis:

Three common amortization arguments:

1. Aggregate analysis
2. Accounting method
3. Potential method

Examples:

- stack with multi pop operation
- binary counter
- dynamic tables

Aggregate analysis

The aggregate method, though simple, lacks the precision of the other two methods. In particular, the accounting and potential methods allow a specific amortized cost to be allocated to each operation.

Stack operations

- PUSH(S, x) – Θ(1) – pushes object x onto the stack
- POP(S) – Θ(1) – pops and returns the top object of S
- MULTIPOP(S,k) – Θ(min{|S|,k}) – pops the top k items from the stack (or until empty) and returns the last item:

```
while S is not empty and k > 0 do
   x ← POP(S)
   k ← k −1
return x
```

- Suppose we wish to analyze the running time for a sequence of n PUSH, POP and MULTIPOP operations, starting with an empty stack. Considering individual operations without amortization, we would say that a MULTIPOP operation could take Θ(|S|) time and |S| could be as large as n −1. So in the hypothetical worst case, a single operation could take Θ(n) time and n such operations strung together would take Θ ¡ n 2 ¢ time.

- However, a little more thought reveals that such a string of operations is not possible. While a single POP could take Θ(n) time, it would have to be preceded by Θ(n) PUSH operations, which are cheap. Taken together, the Θ(n) PUSH operations and the one MULTIPOP operation take Θ(n)· Θ(1)+ 1 ·Θ(n) = Θ(n) time thus, each operation in the sequence takes Θ(1) time on average.
- In general, if there occur r MULTIPOP operations with arguments k1,...,kr, then there must also occur at least k1+···+kr PUSH operations, so that there are enough items in the stack to pop. (To simplify notation, we assume that k is never chosen larger than |S|.) Thus, in a string of n operations, the total cost of all non-O(1) operations is bounded above by O(n), so the total cost of all operations is O(n)·O(1)+ O(n) = O(n). Thus, the amortized running time per operation is O(1).

Running time of MULTIPOP:

- Linear in # of POP operations.
- Let each PUSH/POP cost1.
- # of iterations of **while** loop is min*(s, k),* where *s* = # of objects on stack.
- Therefore, total cost = min*(s,k)*.

Sequences of *n* PUSH, POP, MULTIPOP operations:

- Worst-case cost of MULTIPOP is *O(n)*.
- Have *n* operations.
- Therefore, worst-case cost of sequence is $O(n^2)$.

Observation
Each object can be popped only once per time that it's pushed.

- Have ≤ *n* PUSHes⇒≤ *n* POPs, including those in MULTIPOP.
- Therefore, total cost =*O(n)*.
- Average over the *n* operations ⇒*O(*1*)* per operation on average. Again, notice no probability.
- Showed *worst-case O(n)* cost for sequence.
- Therefore, *O(*1*)* per operation on average. This technique is called ***aggregate analysis***.

Accounting method (or) Banker's Method:
Charge i^th operation a fictitious amortized cost ĉi , where $1 pays for 1 unit of work (i.e., time).

- This fee is consumed to perform the operation.
- Any amount not immediately consumed is stored in the bank for use by subsequent operations.
- Thus, the total amortized costs provide an upper bound on the total true costs. Assign different charges to different operations.

- The **accounting method assigns** a different cost to each type of operation.
- The **estimated cost** of an operation may be greater or less than its actual cost; correspondingly, the surplus of one operation can be used to pay the debt of other operations. In symbols, given an operation whose actual cost is c, we assign an amortized (estimated) cost cb.
- The **amortized costs** must satisfy the condition that, for any sequence of n operations with actual costs c1,..., cn and amortized costs cb1,..., cbn, we have

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$$

- The bank balance must not go negative! We must ensure that for all n.
- As long as this condition is met, we know that the amortized cost provides an upper bound on the actual cost of any sequence of operations. The difference between the above sums is the total surplus or "credit" stored in the data structure and must at all times be nonnegative. In this way, the accounting model is like a debit account.

Example:
Perhaps you have bought pre-stamped envelopes at the post office before. In doing so, you pay up-front for both the envelopes and the postage. Then, when it comes time to send a letter, no additional charge is required. This accounting can be seen as an amortization of the cost of sending a letter:

Operation	**Actual cost ci**	**Amortized cost ĉi**
Buy an envelope	5¢	49¢
Mail a letter	44¢	0¢

Obviously, for any valid sequence of operations, the amortized cost is at least as high as the actual cost. However, the amortized cost is easier to keep track of—it's one fewer item on your balance sheet.
For the stack the amortized costs as follows:

Operation	**Actual cost ci**	**Amortized cost ĉi**
PUSH	1	2
POP	1	0
MULTIPOP	min{|S|,k}	0

When an object is pushed to the stack, it comes endowed with enough credit to pay not only for the operation of pushing it onto the stack, but also for whatever operation will eventually remove it from the stack, be that a POP, a MULTIPOP, or no operation at all.

Intuition: When pushing an object, pay $2.

- $1 pays for the PUSH.
- $1 is prepayment for it being popped by either POP or MULTIPOP.
- Since each object has $1, which is credit, the credits can never go negative.

- Therefore, total amortized cost, = *O(n)*, is an upper bound on total actual cost.

Potential method (or) Physicist's method

The potential method is similar in spirit to the accounting method. Rather than assigning a credit to each element of the data structure, the potential method assigns a credit to the entire data structure as a whole. This makes sense when the total credit stored in the data structure is a function only of its state and not of the sequence of operations used to arrive at that state. We think of the credit as a “potential energy” (or just “potential”) for the data structure.

- The strategy is to define a potential function Φ which maps a state D to a scalar-valued potential Φ(D).
- Given a sequence of n operations with actual costs c1,..., cn, which transform the data structure from its initial state D0 through states D1,...,Dn, we define heuristic costs

$$\widehat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

- This rule can be seen as the conservation of energy: it says that the surplus (or deficit) cost cbi − ci caused by a given operation must be equal to the change in potential of the data structure caused by that operation.
- An operation which increases the potential of the system is assigned a positive heuristic cost, whereas an operation which decreases the potential of the system is assigned a negative heuristic cost.
- Summing the heuristic costs of all n operations, we find

$$\begin{aligned}\sum_{i=1}^{n} \widehat{c}_i &= \sum_{i=1}^{n} c_i + \Phi(D_i) - \Phi(D_{i-1}) \qquad \text{(a telescoping sum)}\\ &= \left(\sum_{i=1}^{n} c_i\right) + \Phi(D_n) - \Phi(D_0).\end{aligned}$$

Where D_i= data structure after *i* th operation,
D= initial data structure,
c_i= actual cost of *i* th operation,
ĉi = amortized cost of *i* th operation.

- Thus, the total credit stored in the data structure is Φ(Dn) − Φ(D0). This quantity must remain nonnegative at all times in order to ensure that the amortized cost provides an upper bound on the actual cost of any sequence of operations.
- (Any function Φ which satisfies this property can be used as the potential function.) One often chooses the potential function Φ so that Φ(D0) = 0; then one must check that Φ remains nonnegative at all times.

Like the accounting method, but think of the credit as *potential* stored with the entire data structure.

- Accounting method stores credit with specific objects.
- Potential method stores potential in the data structure as a whole.

- Can release potential to pay for future operations.
- Most flexible of the amortized analysis methods.

Stack example:

We define the potential of a stack S to be Φ(S) = |S|, the number of elements in the stack. An empty stack has zero potential, and clearly Φ is always nonnegative, so Φ is an admissible potential function. Then the heuristic costs of the stack operations are as follows:

- A PUSH operation increases the size of S by 1, and has actual cost cPUSH = 1. Thus, the amortized cost of a PUSH operation is 2.
- A POP operation decreases the size of S by 1, and has actual cost cPOP = 1. Thus, the amortized cost of a POP operation is 0.
- The operation MULTIPOP(S,k) decreases the size of S by min{|S|,k}, and has actual cost cMULTIPOP = min{|S|,k}. Thus, the amortized cost of a MULTIPOP operation is 0.

Thus, the amortized costs for this application of the potential method are the same as those we came up with using the accounting method

Operation	**Actual cost ci**	**Amortized cost ĉi**
PUSH	1	2
POP	1	0
MULTIPOP	min{|S|,k}	0

- Amortized costs can provide a clean abstraction of data-structure performance.
- Any of the analysis methods can be used when an amortized analysis is called for, but each method has some situations where it is arguably the simplest.
- Different schemes may work for assigning amortized costs in the accounting method, or potentials in the potential method, sometimes yielding radically different bounds.

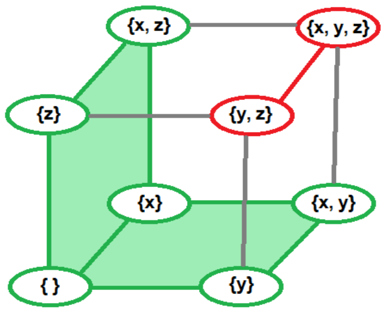
UNIT-2

MATROIDS:

Matroid is a structure that abstracts and generalizes the notion of linear independence in vector spaces. Matroid is a pair ⟨X,I⟩⟨X,I⟩ where XX is called ground set and II is set of all *independent* subsets of XX. In other words matroid ⟨X,I⟩⟨X,I⟩ gives a classification for each subset of XX to be either *independent* or *dependent* (included in II or not included in II).Of course, we are not speaking about arbitrary classifications. These 3 properties must hold for any matroid:

1. Empty set is independent.
2. Any subset of independent set is independent.
3. If independent set AA has smaller size than independent set BB, there exist at least one element in BB that can be added into AA without loss of independency.

These are *axiomatic properties* of matroid. To prove that we are dealing with matroid, we generally have to prove these three properties. For example, explicit presentation of matroid on ground set {x,y,z}{x,y,z} which considers {y,z}{y,z} and {x,y,z}{x,y,z} to be dependent and marked red. Other sets are independent and marked green in the below diagram.



Examples

There are matroids of different types. There are some examples:

- **Uniform matroid.** Matroid that considers subset SS independent if size of SS is not greater than some constant kk (|S|≤k|S|≤k). Simplest one, this matroid does not really distinguish elements of ground set in any form, it only cares about number of taken elements. All subsets of size kk are bases for this matroid, all subsets of size (k+1)(k+1) are circuits for this matroid. We can also define some specific cases of uniform matroid.

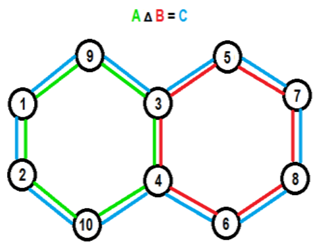
- **Trivial matroid**. k=0 k=0. Only empty set is considered independent, any element of ground set is considered dependent (any combination of ground set elements is also considered dependent as a consequence).

- **Complete matroid**. k=|X| k=|X|. All subsets are considered independent including complete ground set itself.

- **Linear (algebra) matroid**. Ground set consists of vectors of some vector space. Set of vectors is considered independent if it is linearly independent (no vector can be expressed as linear combination of other vectors from that set). This is the matroid from which whole matroid theory originates from. Linear bases of vector set are bases of matroid. Any circuit of this matroid is set of vectors, where each vector can be expressed as combination of all other vectors, but this combination involves all other vectors in circuit.

- **Colorful matroid.** Ground set consists of colored elements. Each element has exactly one color. Set of elements is independent if no pair of included elements share a color. Rank of a set is amount of different colors included into a set. Bases of this matroid are sets that have exactly one element of each color. Circuits of this matroid are all possible pairs of elements of the same color.

- **Graphic matroid.** This matroid is defined on edges of some undirected graph. Set of edges is independent if it does not contain a cycle. This type of matroids is the greatest one to show some visual examples, because it can include dependent subsets of a large size and can be represented on a picture at the same time. If graph is connected then any basis of this graph is just a spanning tree of this graph. If graph is not connected then basis is a forest of spanning trees that include one spanning tree for each connected component. Circuits are simple loops of this graph. Independence oracle for this matroid type can be implemented with DFS, BFS (start from each vertex in graph and check that no edge connect a vertex with already visited one) or DSU (keep connected components, start with disjoint vertices, join by all edges and ensure that each edge connected different components upon addition). Here is an example of circuit combinations property in graphic matroid:



- **Truncated matroid.** We can limit rank of any matroid by some number kk without breaking matroid properties. For example, basis of truncated colorful matroid is set of elements that include no more than kk different colors and all colors are unique. Basis of truncated graphic matroid is acyclic set of edges that leaves at least (n−k)(n−k) connected components in the graph (where nn is amount if vertices in a graph). This is possible because third matroid property does not only refer to bases of matroid, but to any independent set in matroid and when all independent sets with sizes greater than kk are simultaneously removed, independent sets of size kk become new bases and for any lesser independent set can still find elements from each basis that can be added.

- **Matroid on a subset of ground set.** We can limit ground set of matroid to its subset without breaking matroid properties. This is possible because rules of dependence does not rely on specific elements being in ground set. If we remove an edge from a graph, we will still have a valid graph. If we remove an element from set (of vectors or colored elements) we will still get a valid set of some element of the same type and rules will preserve. Now, we can also define rank of subset in matroid as a rank of matroid on a ground set limited to this subset.

- **Expanded matroid. Direct matroid sum.** We can consider two matroids as one big matroid without any difficulties if elements of ground set of first matroid does not affect independence, neither intersect with elements of ground set of second matroid and vice versa. If we consider two graphic matroids on two connected graphs, we can unite their graphs together resulting in graph with two connected components, but it is clear that including some edges in one component have no effect on other component. This is called *direct matroid sum*. Formally,

  M1=⟨X1,I1⟩M1=⟨X1,I1⟩,

  M2=⟨X2,I2⟩M2=⟨X2,I2⟩,

  M1+M2=⟨X1∪X2,I1×I2⟩M1+M2=⟨X1∪X2,I1×I2⟩,

  Where ×× means cartesian product of two sets. We can unite as many matroids of as many different types without restrictions.

INTRODUCTION TO GREEDY PARADIGM:

Greedy Algorithm:

In Greedy Algorithm a set of resources are recursively divided based on the maximum, immediate availability of that resource at any given stage of execution. To solve a problem based on the greedy approach, there are two stages

1. Scanning the list of items
2. Optimization.

These stages are covered parallelly, on course of division of the array.

To understand the greedy approach, we need to have a working knowledge of recursion and context switching. This helps us to understand how to trace the code.

Two conditions define the greedy paradigm

- Each stepwise solution must structure a problem towards its **best**-accepted solution.
- It is sufficient if the **structuring** of the problem can **halt** in a finite number of greedy steps.

History of Greedy Algorithms:

Here is an important landmark of greedy algorithms:

- Greedy algorithms were conceptualized for many graph walk algorithms in the 1950s.
- EsdgerDjikstra conceptualized the algorithm to generate minimal spanning trees. He aimed to shorten the span of routes within the Dutch capital, Amsterdam.
- In the same decade, Prim and Kruskal achieved optimization strategies that were based on minimizing path costs along weighed routes.
- In the '70s, American researchers, Cormen, Rivest and Stein proposed a recursive substructuring of greedy solutions in their classical introduction to algorithms book.

- The greedy paradigm was registered as a different type of optimization strategy in the NIST records in 2005.
- Till date, protocols that run the web, such as the open-shortest-path-first (OSPF) and many other network packet switching protocols use the greedy strategy to minimize time spent on a network.

Components of Greedy Algorithm:

Greedy algorithms have the following five components:

- **A candidate set** − A solution is created from this set.
- **A selection function** − Used to choose the best candidate to be added to the solution.
- **A feasibility function** − Used to determine whether a candidate can be used to contribute to the solution.
- **An objective function** − Used to assign a value to a solution or a partial solution.
- **A solution function** − Used to indicate whether a complete solution has been reached.

Characteristics of the Greedy Approach:

The important characteristics of a greedy method are:

- There is an ordered list of resources, with costs or value attributions. These quantify constraints on a system.
- We will take the maximum quantity of resources in the time a constraint applies.
- For example, in an activity scheduling problem, the resource costs are in hours, and the activities need to be performed in serial order.

Here are the reasons for using the greedy approach:

- The greedy approach has a few tradeoffs, which may make it suitable for optimization.
- One prominent reason is to achieve the most feasible solution immediately. In the activity selection problem, if more activities can be done before finishing the current activity, these activities can be performed within the same time.
- Another reason is to divide a problem recursively based on a condition, with no need to combine all the solutions.
- In the activity selection problem, the "recursive division" step is achieved by scanning a list of items **only once** and considering certain activities.

How to solve the activity selection problem?

In the activity scheduling example, there is a "start" and "finish" time for every activity. Each Activity is indexed by a number for reference. There are two activity categories.

1. **Considered activity**: is the activity, which is the reference from which the ability to do more than one remaining activity is analyzed.
2. **Remaining activities:** activities at one or more indexes ahead of the considered activity.

The total duration gives the cost of performing the activity. That is (finish - start) gives us the durational as the **cost** of an activity. The **greedy extent** is the number of remaining activities we can perform in the time of a considered activity.

Architecture of the Greedy approach:

STEP 1: Scan the list of activity costs, starting with index 0 as the considered Index.

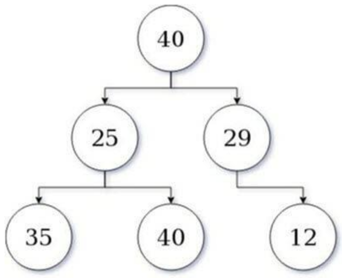
STEP 2: When more activities can be finished by the time, the considered activity finishes, start searching for one or more remaining activities.

STEP 3: If there are no more remaining activities, the current remaining activity becomes the next considered activity. Repeat step 1 and step 2, with the new considered activity. If there are no remaining activities left, go to step 4.

STEP 4: Return the union of considered indices. These are the activity indices that will be used to maximize throughput.

Disadvantages of Greedy Algorithms:

- It is not suitable for problems where a solution is required for every sub-problem like sorting.
- In such problems, the greedy strategy can be wrong; in the worst case even lead to a non-optimal solution.
- Therefore the disadvantage of greedy algorithms is using not knowing what lies ahead of the current greedy state. Below is a depiction of the disadvantage of the greedy approach.



In the greedy scan shown here as a tree (higher value higher greed), an algorithm state at value: 40, is likely to take 29 as the next value. Further, its quest ends at 12. This amounts to a value of 41.
However, if the algorithm took a sub-optimal path or adopted a conquering strategy then 25 would be followed by 40, and the overall cost improvement would be 65, which is valued 24 points higher as a suboptimal decision.

Examples of Greedy Algorithms:

Most networking algorithms use the greedy approach. Here is a list of few of them:

- Prim's Minimal Spanning Tree Algorithm
- Travelling Salesman Problem
- Graph - Map Coloring
- Kruskal's Minimal Spanning Tree Algorithm
- Dijkstra's Minimal Spanning Tree Algorithm
- Graph - Vertex Cover
- Knapsack Problem
- Job Scheduling Problem

ALGORITHM TO COMPUTE A MAXIMUM WEIGHT MAXIMAL INDEPENDENT SET:

The Greedy Approach and Divide and conquer algorithms are used to compute a maximum weight maximal independent set.

Greedy Algorithm: The **maximum** (weighted) **independent set** (MIS(MWIS)) is one of the most important optimization problems. In several **heuristic** methods for optimization problems, the **greedy** strategy is the most natural and simplest one. For MIS, two simple **greedy algorithms** have been investigated. One is called GMIN, which selects a vertex of minimum degree, removes it and its neighbors from the graph and iterates this process on the remaining graph until no vertex remains. (the set of selected vertices is an independent set). The other is called GMAX, which deletes a vertex of maximum degree until no edge remains (the set of remaining vertices is an independent set).

Divide and Conquer:

Divide and Conquer is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using following three steps.

1. **Divide**: Break the given problem into sub-problems of same type.
2. **Conquer**: Recursively solve these sub-problems
3. **Combine**: Appropriately combine the answers

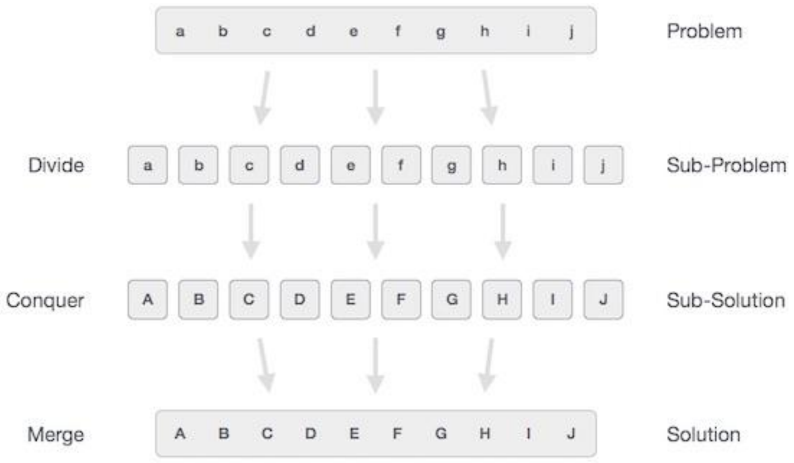
Divide and Conquer algorithm:

```
DAC(a, i, j)
{
if(small(a, i, j))
return(Solution(a, i, j))
else
    m = divide(a, i, j)          // f1(n)
    b = DAC(a, i, mid)           // T(n/2)
    c = DAC(a, mid+1, j)         // T(n/2)
    d = combine(b, c)            // f2(n)
return(d)
}
```

Divide/Break: This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.

Conquer/Solve: This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.

Merge/Combine: When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively conquers and merge steps works so close that they appear as one. Below diagram indicates this:



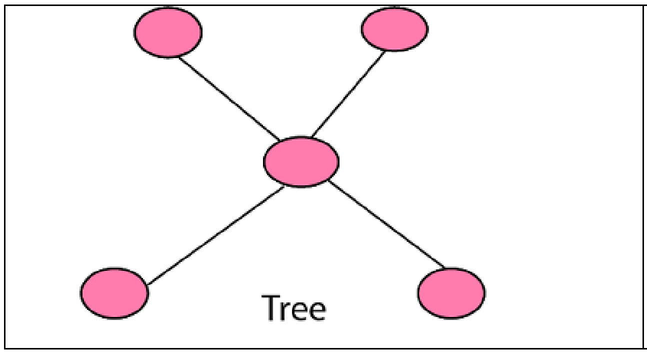
The following computer algorithms are based on **divide-and-conquer** programming approach

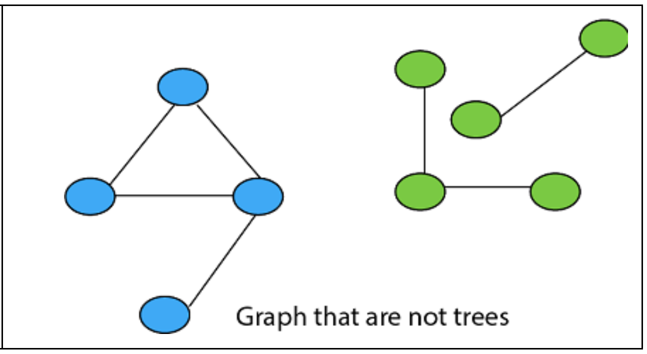
1. **Binary Search** is a searching algorithm. In each step, the algorithm compares the input element x with the value of the middle element in array. If the values match, return the index of the middle. Otherwise, if x is less than the middle element, then the algorithm recurs for left side of middle element, else recurs for the right side of the middle element.
2. **Quick sort** is a sorting algorithm. The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot and all greater elements move to right side. Finally, the algorithm recursively sorts the sub-arrays on left and right of pivot element.
3. **Merge Sort** is also a sorting algorithm. The algorithm divides the array in two halves, recursively sorts them and finally merges the two sorted halves.
4. **Closest Pair of Points** The problem is to find the closest pair of points in a set of points in x-y plane. The problem can be solved in O(n^2) time by calculating distances of every pair of points and comparing the distances to find the minimum. The Divide and Conquer algorithm solves the problem in O(nLogn) time.
5. **Strassen's Algorithm** is an efficient algorithm to multiply two matrices. A simple method to multiply two matrices need 3 nested loops and is O(n^3). Strassen's algorithm multiplies two matrices in O(n^2.8974) time.
6. **Cooley–Tukey Fast Fourier Transform (FFT) algorithm** is the most common algorithm for FFT. It is a divide and conquer algorithm which works in O(nlogn) time.
7. **Karatsuba algorithm for fast multiplication** it does multiplication of two *n*-digit numbers and at most it takes $3n^{\log_2 3} \approx 3n^{1.585}$ single-digit multiplications in general (and exactly $n^{\log_2 3}$ when *n* is a power of 2). It is therefore faster than the classical algorithm, which requires n^2 single-digit products. If $n = 2^{10} = 1024$, in particular, the exact counts are $3^{10} = 59,049$ and $(2^{10})^2 = 1,048,576$, respectively.

APPLICATION TO MST:

Tree: A tree is a graph with the following properties:

1. The graph is connected (can go from anywhere to anywhere)
2. There are no cyclic (Acyclic)

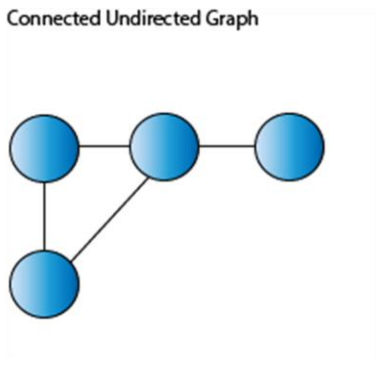




Spanning Tree:

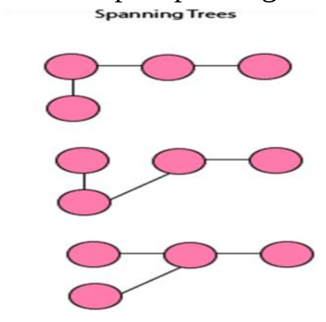
Given a connected undirected graph, a spanning tree of that graph is a sub-graph that is a tree and joined all vertices. A single graph can have many spanning trees.

For Example:



Connected Undirected Graph

For the corresponding connected graph, There can be multiple spanning Trees like



Properties of Spanning Tree:

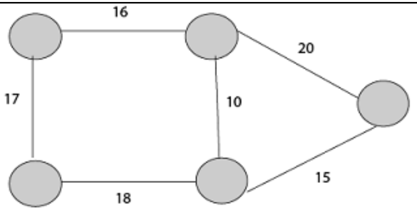
1. There may be several minimum spanning trees of the same weight having the minimum number of edges.
2. If all the edge weights of a given graph are the same, then every spanning tree of that graph is minimum.
3. If each edge has a distinct weight, then there will be only one, unique minimum spanning tree.
4. A connected graph G can have more than one spanning trees.
5. A disconnected graph can't have to span the tree, or it can't span all the vertices.

6. Spanning Tree does not contain cycles.
7. Spanning Tree has **(n-1) edges** where n is the number of vertices.

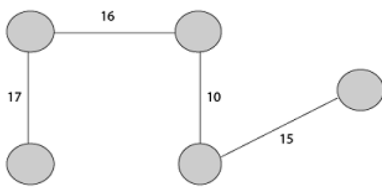
Addition of even one single edge results in the spanning tree losing its property of **Acyclicity** and elimination of one single edge results in its losing the property of connectivity.

Minimum Spanning Tree:

Minimum Spanning Tree is a Spanning Tree which has minimum total cost. If we have a linked undirected graph with a weight (or cost) combine with each edge. Then the cost of spanning tree would be the sum of the cost of its edges.



Connected , Undirected Graph



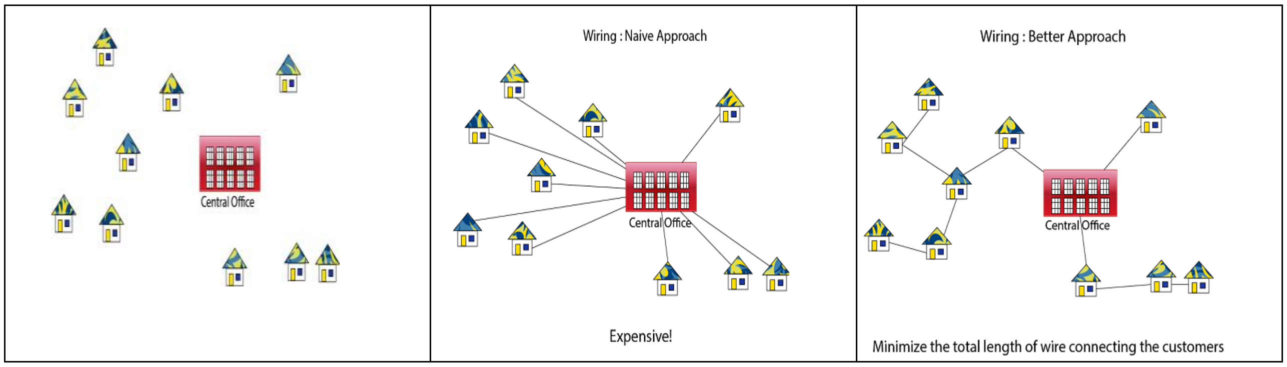
Minimum Cost Spanning Tree
Total Cost = 17+16+10+15=58

Application of Minimum Spanning Tree:

1. Consider n stations are to be linked using a communication network and laying of communication links between any two stations involves a cost. The ideal solution would be to extract a sub-graph termed as minimum cost spanning tree.
2. Suppose we want to construct highways or railroads spanning several cities then we can use the concept of minimum spanning trees.
3. Designing Local Area Networks.
4. Laying pipelines connecting offshore drilling sites, refineries and consumer markets.
5. Suppose we want to apply a set of houses with
   - Electric Power
   - Water
   - Telephone lines
   - Sewage lines

   To reduce cost, we can connect houses with minimum cost spanning trees.

Example: Problem laying Telephone Wire



Methods of Minimum Spanning Tree

There are two methods to find Minimum Spanning Tree

1. Kruskal's Algorithm
2. Prim's Algorithm

Kruskal's Algorithm:

This is an algorithm to construct a Minimum Spanning Tree for a connected weighted graph. It is a Greedy Algorithm. If the graph is not linked, then it finds a Minimum Spanning Tree.

Steps for finding MST using Kruskal's Algorithm:

1. Arrange the edge of G in order of increasing weight.
2. Starting only with the vertices of G and proceeding sequentially add each edge which does not result in a cycle, until (n - 1) edges are used.
3. EXIT.

MST- KRUSKAL (G, w)

```
1. A ← ∅
2. for each vertex v ∈ V [G]
3. do MAKE - SET (v)
4. sort the edges of E into non decreasing order by weight w
5. for each edge (u, v) ∈ E, taken in non decreasing order by weight
6. do if FIND-SET (μ) ≠ if FIND-SET (v)
7. then A ← A ∪ {(u, v)}
8. UNION (u, v)
9. return A
```

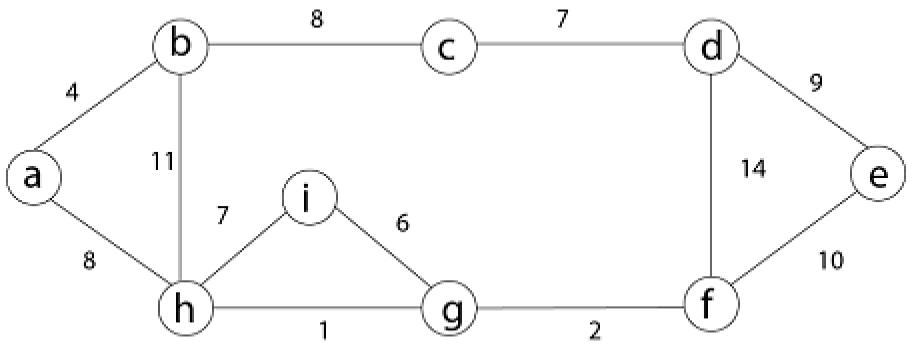
Analysis:

Where E is the number of edges in the graph and V is the number of vertices, Kruskal's Algorithm can be shown to run in O(E log E) time, or simply, O(E log V) time, all with simple data structures. These running times are equivalent because:

- E is at most V^2 and log V^2= 2 x log V is O (log V).

- If we ignore isolated vertices, V ≤ 2 E, so log V is O (log E).Thus the total time is :O (E log E) = O (E log V).

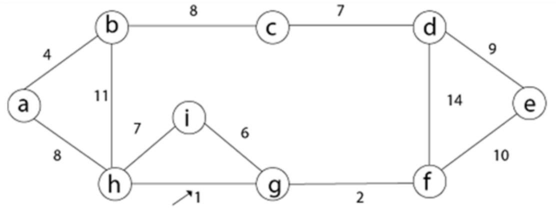
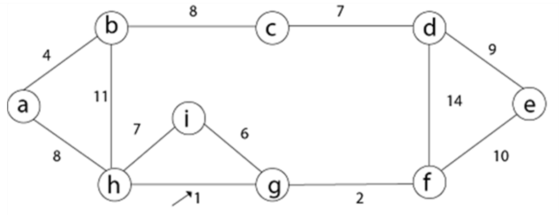
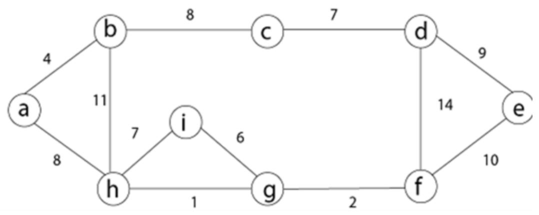
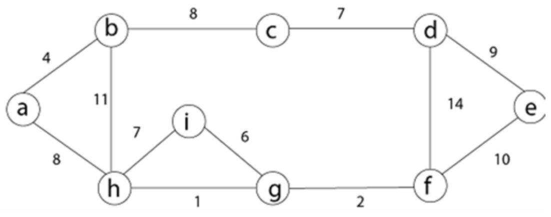
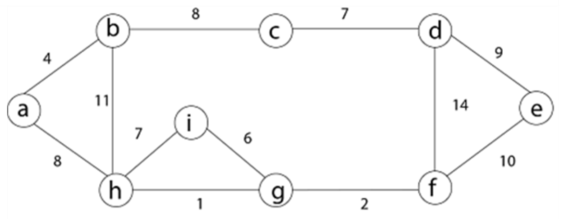
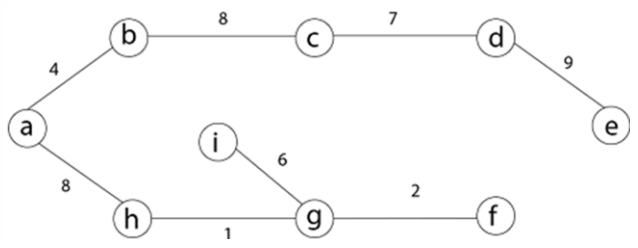
For Example: Find the Minimum Spanning Tree of the following graph using Kruskal's algorithm.



Solution: First we initialize the set A to the empty set and create |v| trees, one containing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight.
There are 9 vertices and 12 edges. So MST formed (9-1) = 8 edges

Weight	Source	Destination
1	H	g
2	G	F
4	A	B
6	I	G
7	H	I
7	C	D
8	B	C
8	A	H
9	D	E
10	E	F
11	B	H
14	D	F

Now, check for each edge (u, v) whether the endpoints u and v belong to the same tree. If they do then the edge (u, v) cannot be supplementary. Otherwise, the two vertices belong to different trees, and the edge (u, v) is added to A and the vertices in two trees are merged in by union procedure.

Steps to find Minimum Spanning Tree using Kruskal's algorithm	
Step1: So, first take (h, g) edge 	**Step 2:** then (g, f) edge. 
Step 3: then (a, b) and (i, g) edges are considered, and the forest becomes 	**Step 4:** Now, edge (h, i). Both h and i vertices are in the same set. Thus it creates a cycle. So this edge is discarded. Then edge (c, d), (b, c), (a, h), (d, e), (e, f) are considered, and the forest becomes. 
Step 5: In (e, f) edge both endpoints e and f exist in the same tree so discarded this edge. Then (b, h) edge, it also creates a cycle.	**Step 6:** After that edge (d, f) and the final spanning tree is shown as in dark lines. 
Step 7: This step will be required Minimum Spanning Tree because it contains all the 9 vertices and (9 - 1) = 8 edges e → f, b → h, d → f [cycle will be formed]  Minimum Cost MST	

Prim's Algorithm

It is a greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices:

- Contain vertices already included in MST.
- Contain vertices not yet included.

At every step, it considers all the edges and picks the minimum weight edge. After picking the edge, it moves the other endpoint of edge to set containing MST.

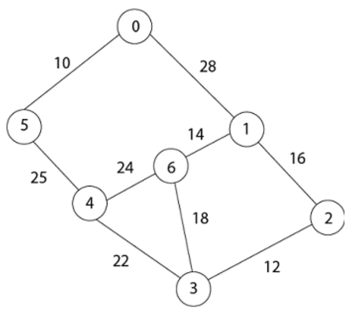
Steps for finding MST using Prim's Algorithm:

1. Create MST set that keeps track of vertices already included in MST.
2. Assign key values to all vertices in the input graph. Initialize all key values as INFINITE (∞). Assign key values like 0 for the first vertex so that it is picked first.
3. While MST set doesn't include all vertices.
    a. Pick vertex u which is not is MST set and has minimum key value. Include 'u' to MST set.
    b. Update the key value of all adjacent vertices of u. To update, iterate through all adjacent vertices. For every adjacent vertex v, if the weight of edge u.v less than the previous key value of v, update key value as a weight of u.v.

MST-PRIM (G, w, r)

```
1. for each u ∈ V [G]
2. do key [u] ← ∞
3. π [u] ← NIL
4. key [r] ← 0
5. Q ← V [G]
6. While Q ?∅
7. do u ← EXTRACT - MIN (Q)
8. for each v ∈Adj [u]
9. do if v ∈ Q and w (u, v) < key [v]
10. then π [v] ← u
11. key [v] ← w (u, v)
```

Example: Generate minimum cost spanning tree for the following graph using Prim's algorithm.

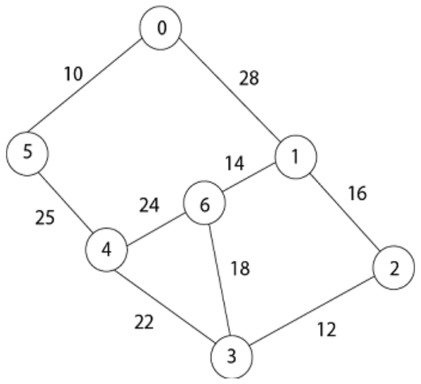


Solution: In Prim's algorithm, first we initialize the priority Queue Q. to contain all the vertices and the key of each vertex to ∞ except for the root, whose key is set to 0. Suppose 0 vertex is the root, i.e., r. By EXTRACT - MIN (Q) procure, now u = r and Adj [u] = {5, 1}.Removing u from set Q and adds it to set V - Q of vertices in the tree. Now, update the key and π fields of every vertex v adjacent to u but not in a tree.

Vertex	0	1	2	3	4	5	6
Key Value	0	∞	∞	∞	∞	∞	∞
Parent	NIL	NIL	NIL	NIL	NIL	NIL	NIL

Taking 0 as starting vertex
Root = 0
Adj [0] = 5, 1
Parent, π [5] = 0 and π [1] = 0
Key [5] = ∞ and key [1] = ∞
w [0, 5) = 10 and w (0,1) = 28
w (u, v) < key [5] , w (u, v) < key [1]
Key [5] = 10 and key [1] = 28
So update key value of 5 and 1 is:

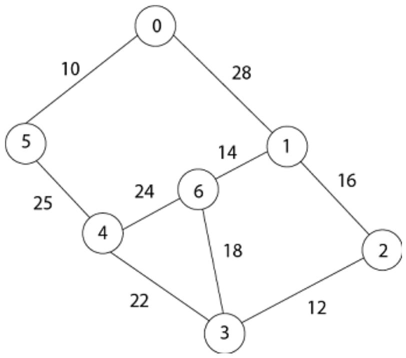
Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	∞	∞	10	∞
Parent	NIL	0	NIL	NIL	NIL	0	NIL



Now by EXTRACT_MIN (Q) Removes 5 because key [5] = 10 which is minimum so u = 5.
Adj [5] = {0, 4} a
nd 0 is already in heap
Taking 4, key [4] = ∞ π [4] = 5
(u, v) < key [v] then key [4] = 25
w (5,4) = 25

w (5,4) < key [4]
date key value and parent of 4.

Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	∞	25	10	∞
Parent	NIL	0	NIL	NIL	5	0	NIL



Now remove 4 because key [4] = 25 which is minimum, so u =4

Adj [4] = {6, 3}
Key [3] = ∞ key [6] = ∞
w (4,3) = 22 w (4,6) = 24
w (u, v) < key [v] w (u, v) < key [v]
w (4,3) < key [3] w (4,6) < key [6]

Update key value of key [3] as 22 and key [6] as 24.

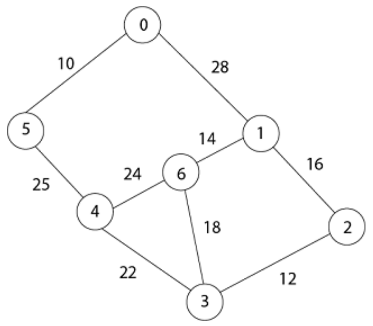
And the parent of 3, 6 as 4.

π[3]= 4 π[6]= 4

Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	22	25	10	∞
Parent	NIL	0	NIL	4	5	0	NIL

u = EXTRACT_MIN (3, 6) [key [3] < key [6]]
u = 3 i.e. 22 < 24

Now remove 3 because key [3] = 22 is minimum so u =3.



Adj [3] = {4, 6, 2}
4 is already in heap
4 ≠ Q key [6] = 24 now becomes key [6] = 18
Key [2] = ∞ key [6] = 24
w (3, 2) = 12 w (3, 6) = 18
w (3, 2) < key [2] w (3, 6) < key [6]

Now in Q, key [2] = 12, key [6] = 18, key [1] = 28 and parent of 2 and 6 is 3.

π [2] = 3 π[6]=3

Now by EXTRACT_MIN (Q) Removes 2, because key [2] = 12 is minimum.

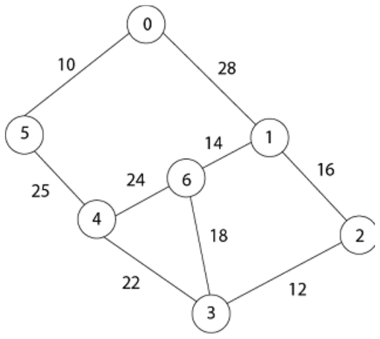
Vertex	0	1	2	3	4	5	6
Key Value	0	28	12	22	25	10	18
Parent	NIL	0	3	4	5	0	3

u = EXTRACT_MIN (2, 6)
u = 2 [key [2] < key [6]]
12 < 18
Now the root is 2
Adj [2] = {3, 1}
3 is already in a heap
Taking 1, key [1] = 28
w (2,1) = 16
w (2,1) < key [1]

So update key value of key [1] as 16 and its parent as 2.

π[1]= 2

Vertex	0	1	2	3	4	5	6
Key Value	0	16	12	22	25	10	18
Parent	NIL	2	3	4	5	0	3



Now by EXTRACT_MIN (Q) Removes 1 because key [1] = 16 is minimum.
Adj [1] = {0, 6, 2}
0 and 2 are already in heap.
Taking 6, key [6] = 18
w [1, 6] = 14
w [1, 6] < key [6]

Update key value of 6 as 14 and its parent as 1.

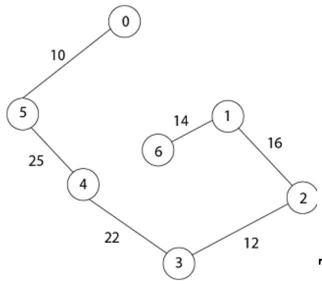
Π [6] = 1

Vertex	0	1	2	3	4	5	6
Key Value	0	16	12	22	25	10	14
Parent	NIL	2	3	4	5	0	1

Now all the vertices have been spanned, Using above the table we get Minimum Spanning Tree.

0 → 5 → 4 → 3 → 2 → 1 → 6
[Because Π [5] = 0, Π [4] = 5, Π [3] = 4, Π [2] = 3, Π [1] =2, Π [6] =1]

Thus the final spanning Tree is



Total Cost = 10 + 25 + 22 + 12 + 16 + 14 = 99

GRAPH MATCHING:

Graph matching is the problem of finding a similarity between graphs. Graphs are commonly used to encode structural information in many fields, including computer vision and pattern recognition, and graph matching is an important tool in these areas. In these areas it is commonly assumed that the comparison is between the *data graph* and the *model graph*.

The case of **exact graph matching** is known as the graph isomorphism problem. The problem of exact matching of a graph to a part of another graph is called subgraph isomorphism problem.

The **inexact graph matching** refers to matching problems when exact matching is impossible, e.g., when the numbers of vertices in the two graphs are different. In this case it is required to find the best possible match. For example, in image recognition applications, the results of image segmentation in image processing typically produces data graphs with the numbers of vertices much larger than in the model graphs data expected to match against. In the case of attributed graphs, even if the numbers of vertices and edges are the same, the matching still may be only inexact.

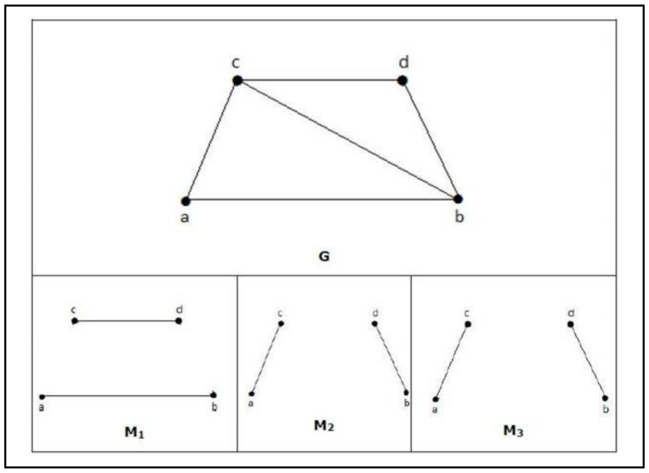
Two categories of search methods are the ones based on identification of possible and impossible pairings of vertices between the two graphs and methods which formulate graph matching as an optimization problem. Graph edit distance is one of similarity measures suggested for graph matching. The class of algorithms is called error-tolerant graph matching.

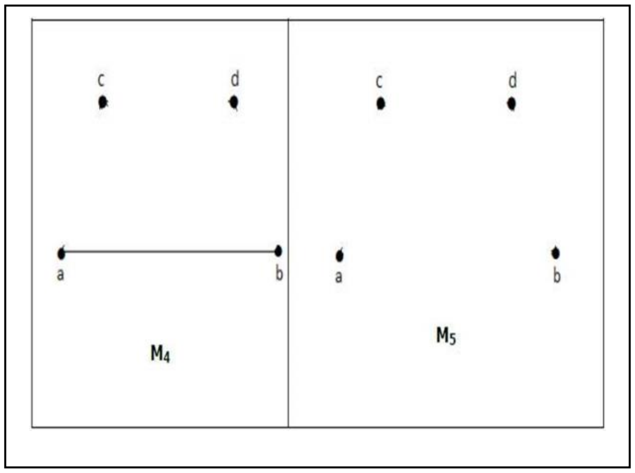
Definition

A matching graph is a sub-graph of a graph where there are no edges adjacent to each other. Simply, there should not be any common vertex between any two edges.

Matching

Let 'G' = (V, E) be a graph. A subgraph is called a matching M(G), **if each vertex of G is incident with at most one edge in M,** i.e.,deg(V) $\leq 1 \ \forall \ V \in G$. Which means in the matching graph M(G), the vertices should have a degree of 1 or 0, where the edges should be incident from the graph G.





Notation − M(G) **The Example:**

In a matching,

ifdeg(V) = 1, then (V) is said to be matched

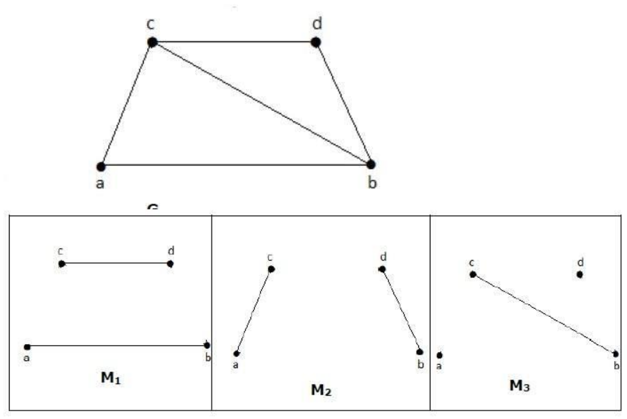
ifdeg(V) = 0, then (V) is not matched.

In a matching, no two edges are adjacent. It is because if any two edges are adjacent, then the degree of the vertex which is joining those two edges will have a degree of 2 which violates the matching rule.

Maximal Matching

A matching M of graph 'G' is said to be maximal **if no other edges of 'G' can be added to M**.

Example



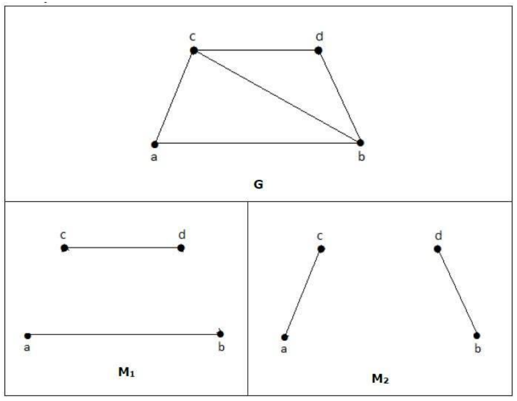
M_1, M_2, M_3 from the above graph are the maximal matching of G.

Maximum Matching

It is also known as largest maximal matching. Maximum matching is defined as the maximal matching with maximum number of edges.

The number of edges in the maximum matching of 'G' is called its **matching number**.

Maximum Matching –Example



For a graph given in the above example, M_1 and M_2 are the maximum matching of 'G' and its matching number is 2. Hence by using the graph G, we can form only the sub-graphs with only 2 edges maximum. Hence we have the matching number as two.

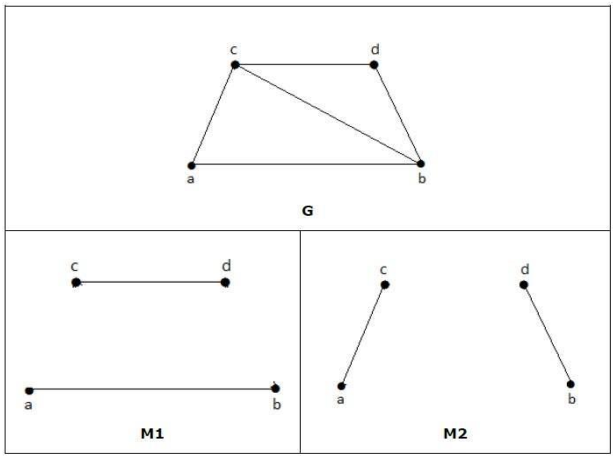
Perfect Matching

A matching (M) of graph (G) is said to be a perfect match, **if every vertex of graph g (G) is incident to exactly one edge of the matching (M),** i.e.,deg(V) = 1 ∀ V

The degree of each and every vertex in the subgraph should have a **degree** of 1.

Perfect Matching - Example

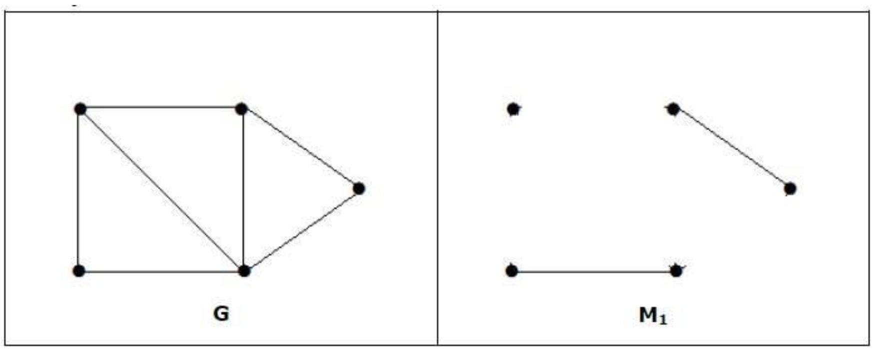
In the following graphs, M_1 and M_2 are examples of perfect matching of G.



Note − Every perfect matching of graph is also a maximum matching of graph, because there is no chance of adding one more edge in a perfect matching graph.

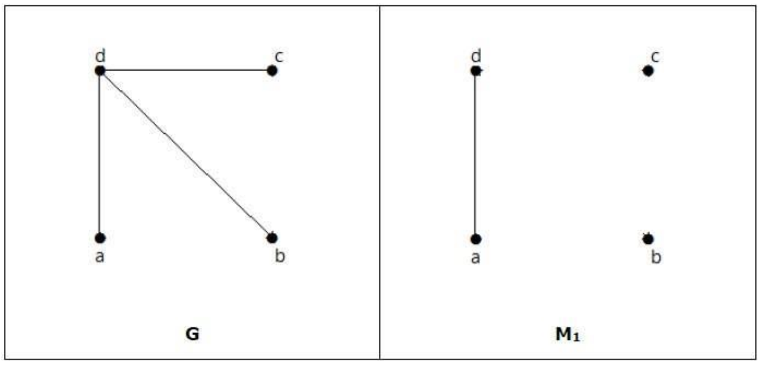
A maximum matching of graph need not be perfect. If a graph 'G' has a perfect match, then the number of vertices |V(G)| is even. If it is odd, then the last vertex pairs with the other vertex, and finally there remains a single vertex which cannot be paired with any other vertex for which the degree is zero. It clearly violates the perfect matching principle.

Example



Note − The converse of the above statement need not be true. If G has even number of vertices, then M_1 need not be perfect.

Example



It is matching, but it is not a perfect match, even though it has even number of vertices.

ALGORITHM TO COMPUTE MAXIMUM MATCHING:

Hopcroft–Karp Algorithm for Maximum Matching

A matching in a Bipartite Graph is a set of the edges chosen in such a way that no two edges share an endpoint. A maximum matching is a matching of maximum size (maximum number of edges). In a maximum matching, if any edge is added to it, it is no longer a matching. There can be more than one maximum matching for a given Bipartite Graph.

Hopcroft Karp algorithm is an improvement that runs in $O(\sqrt{V} \times E)$ time. Let us define few terms before we discuss the algorithm.

Free Node or Vertex*:* Given a matching M, a node that is not part of matching is called free node. Initially all vertices as free (from first graph of below diagram). In second graph, u2 and v2 are free. In third graph, no vertex is free.

Matching and Not-Matching edges*:* Given a matching M, edges that are part of matching are called Matching edges and edges that are not part of M (or connect free nodes) are called Not-Matching edges. In first graph, all edges are non-matching. In second graph, (u0, v1), (u1, v0) and (u3, v3) are matching and others not-matching.

Alternating Paths: Given a matching M, an alternating path is a path in which the edges belong alternatively to the matching and not matching. All single edges paths are alternating paths. Examples of alternating paths in middle graph are u0-v1-u2 and u2-v1-u0-v2.

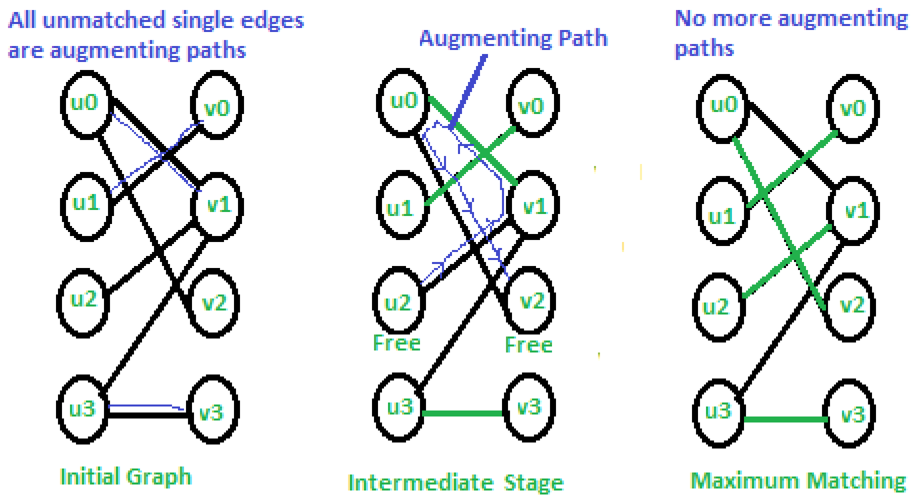
Augmenting path: Given a matching M, an augmenting path is an alternating path that starts from and ends on free vertices. All single edge paths that start and end with free vertices are augmenting paths. In below diagram, augmenting paths are highlighted with blue color. Note that the augmenting path always has one extra matching edge.
The Hopcroft Karp algorithm is based on below concept.

A matching M is not maximum if there exists an augmenting path. It is also true other way, i.e, a matching is maximum if no augmenting path exists. So the idea is to one by one look for augmenting paths. And add the found paths to current matching.

Hopcroft Karp Algorithm

1. Initialize Maximal Matching M as empty.
2. While there exists an Augmenting Path p, remove matching edges of p from M and add not-matching edges of p to M(This increases size of M by 1 as p status and ends with a free vertex)
3. Return M.

Below diagram shows working of the algorithm.



In the initial graph all single edges are augmenting paths and we can pick in any order. In the middle stage, there is only one augmenting path. We remove matching edges of this path from M and add not-matching edges. In final matching, there are no augmenting paths so the matching is maximum.

Ford-Fulkerson algorithm

The Ford–Fulkerson method or Ford–Fulkerson algorithm (FFA) is a greedy algorithm that computes the maximum flow in a flow network. It is sometimes called a "method" instead of an "algorithm" as the approach to finding augmenting paths in a residual graph is not fully specified or it is specified in several implementations with different running times.

Max Flow Problem Introduction

The max flow problem is an optimization problem for determining the maximum amount of *stuff* that can flow at a given point in time through a single source/sink flow network. A flow network is essentially just a directed graph where the edge weights represent the flow capacity of each edge. The *stuff* that flows through these networks could be literally anything. Maybe it's traffic driving through a city, water flowing through pipes or bits traveling across the internet.

Algorithm of Ford-Fulkerson algorithm:

1. Start with initial flow as 0.
2. While there is a augmenting path from source to sink. Add this path-flow to flow.
3. Return flow.

Time Complexity: Time complexity of the above algorithm is O(max_flow * E). We run a loop while there is an augmenting path. In worst case, we may add 1 unit flow in every iteration. Therefore the time complexity becomes O(max_flow * E).

How to implement the above simple algorithm?

Let us first define the concept of Residual Graph which is needed for understanding the implementation.

Residual Graph of a flow network is a graph which indicates additional possible flow. If there is a path from source to sink in residual graph, then it is possible to add flow. Every edge of a residual graph has a value called ***residual capacity*** which is equal to original capacity of the edge minus current flow.

- Residual capacity is basically the current capacity of the edge. Residual capacity is 0 if there is no edge between two vertices of residual graph.
- We can initialize the residual graph as original graph as there is no initial flow and initially residual capacity is equal to original capacity. To find an augmenting path, we can either do a BFS or DFS of the residual graph. We have used BFS in below implementation.
- Using BFS, we can find out if there is a path from source to sink. BFS also builds parent[] array. Using the parent[] array, we traverse through the found path and find possible flow through this path by finding minimum residual capacity along the path. We later add the found path flow to overall flow.
- The important thing is, we need to update residual capacities in the residual graph. We subtract path flow from all edges along the path and we add path flow along the reverse edges. We need to add path flow along reverse edges because may later need to send flow in reverse direction.(See following link for example).

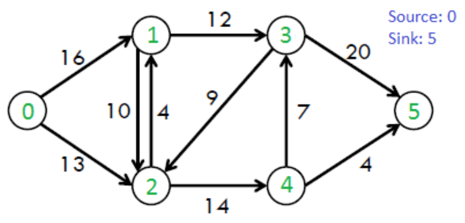
Ford-Fulkerson Algorithm for Maximum Flow Problem

Given a graph which represents a flow network where every edge has a capacity. Also given two vertices *source* ‘s’ and *sink* ‘t’ in the graph, find the maximum possible flow from s to t with following constraints:

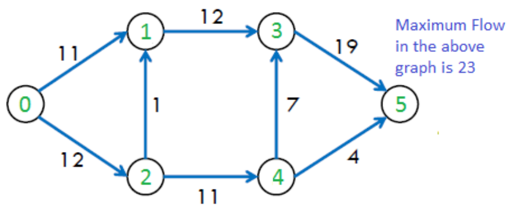
a) Flow on an edge doesn’t exceed the given capacity of the edge.

b) Incoming flow is equal to outgoing flow for every vertex except s and t.

For example, consider the following graph.



The maximum possible flow in the above graph is 23.

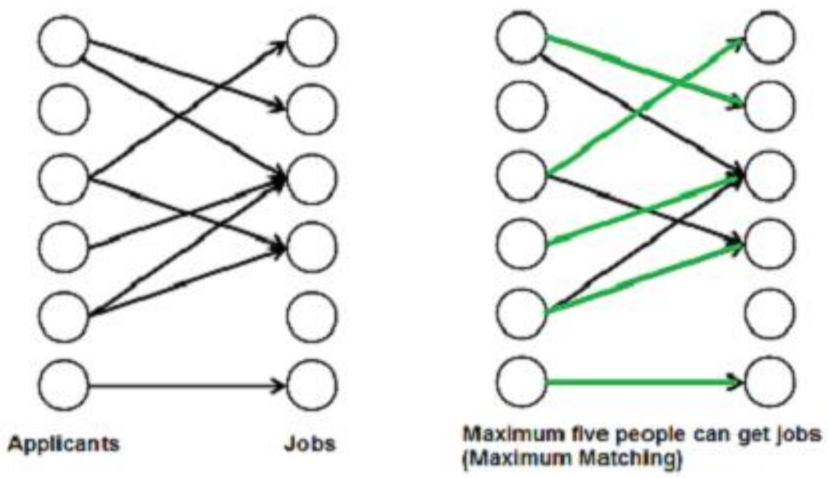


Maximum Bipartite Matching

A matching in a Bipartite Graph is a set of the edges chosen in such a way that no two edges share an endpoint. A maximum matching is a matching of maximum size (maximum number of edges). In a maximum matching, if any edge is added to it, it is no longer a matching. There can be more than one maximum matchings for a given Bipartite Graph.

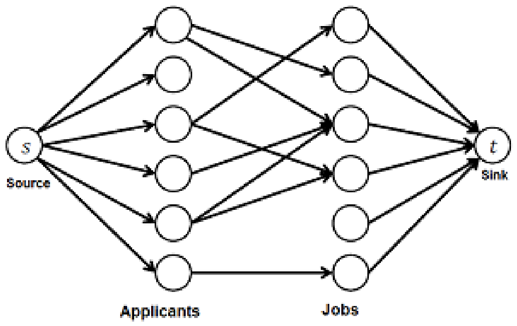
Why do we care?

There are many real world problems that can be formed as Bipartite Matching.
For example, consider the following problem:
There are M job applicants and N jobs. Each applicant has a subset of jobs that he/she is interested in.
Each job opening can only accept one applicant and a job applicant can be appointed for only one job.
Find an assignment of jobs to applicants in such that as many applicants as possible get jobs.

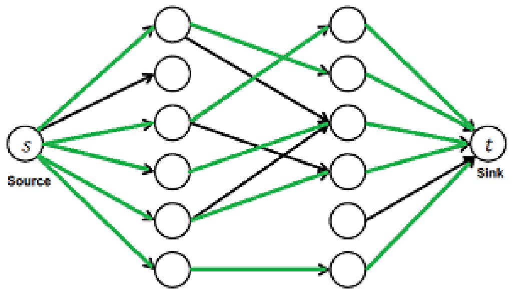


Example: Ford-Fulkerson Algorithm for Maximum Flow Problem

Maximum Bipartite Matching and Max Flow Problem
Maximum **B**ipartite **M**atching (**MBP**) problem can be solved by converting it into a flow network Following are the steps:
1) Build a Flow Network
There must be a source and sink in a flow network. So we add a source and add edges from source to all applicants. Similarly, add edges from all jobs to sink. The capacity of every edge is marked as 1 unit.



2) Find the maximum flow.
We use Ford-Fulkerson algorithm to find the maximum flow in the flow network built in step 1. The maximum flow is actually the MBP we are looking for.



The maximum flow from source to sink is five units. Therefore, maximum five people can get jobs.

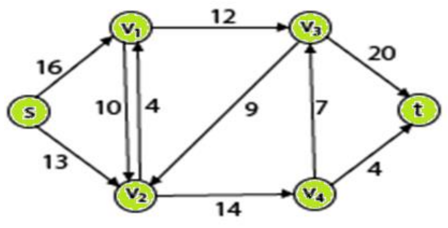
How to implement the above approach?

Let us first define input and output forms. Input is in the form of Edmonds matrix which is a 2D array 'bpGraph[M][N]' with M rows (for M job applicants) and N columns (for N jobs). The value bpGraph[i][j] is 1 if i'th applicant is interested in j'th job, otherwise 0. Output is number maximum number of people that can get jobs.

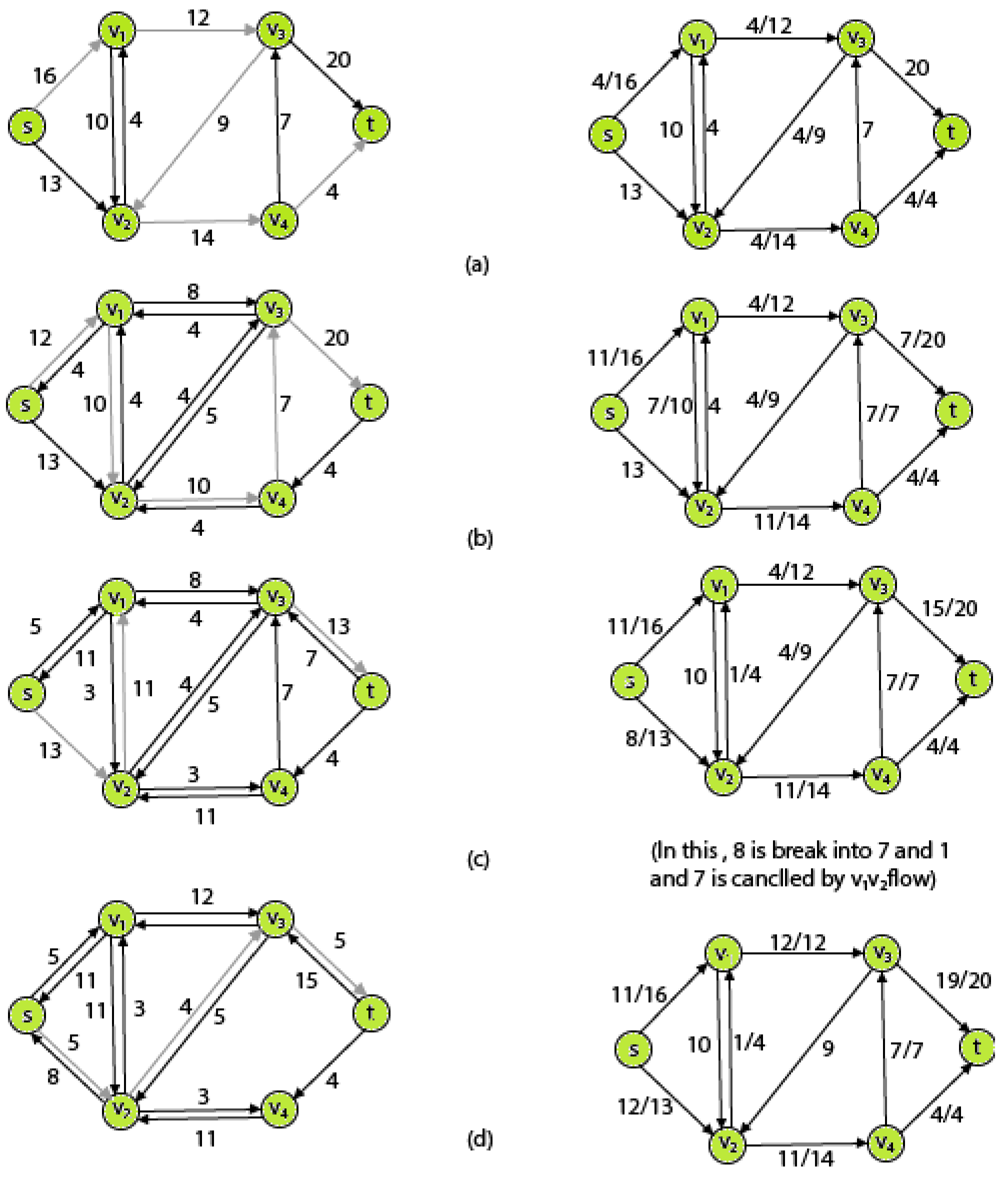
- A simple way to implement this is to create a matrix that represents adjacency matrix representation of a directed graph with M+N+2 vertices. Call the fordFulkerson() for the matrix. This implementation requires O((M+N)*(M+N)) extra space.
- Extra space can be be reduced and code can be simplified using the fact that the graph is bipartite and capacity of every edge is either 0 or 1. The idea is to use DFS traversal to find a job for an applicant (similar to augmenting path in Ford-Fulkerson). We call bpm() for every applicant, bpm() is the DFS based function that tries all possibilities to assign a job to the applicant.
- In bpm(), we one by one try all jobs that an applicant 'u' is interested in until we find a job, or all jobs are tried without luck. For every job we try, we do following. If a job is not assigned to anybody,
- We simply assign it to the applicant and return true. If a job is assigned to somebody else say x, then we recursively check whether x can be assigned some other job. To make sure that x doesn't get the same job again, we mark the job 'v' as seen before we make recursive call for x. If x can get other job, we change the applicant for job 'v' and return true. We use an array maxR[0..N-1] that stores the applicants assigned to different jobs.
- If bmp() returns true, then it means that there is an augmenting path in flow network and 1 unit of flow is added to the result in maxBPM().

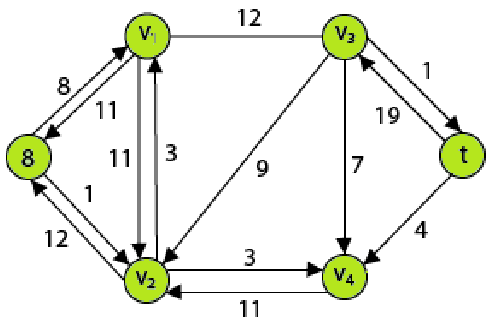
Example- Ford-Fulkerson algorithm:

Each Directed Edge is labeled with capacity. Use the Ford-Fulkerson algorithm to find the maximum flow.



The left side of each part shows the residual network G_f with a shaded augmenting path p,and the right side of each part shows the net flow f.





(e)

Now , it has no augmenting paths .So, the maximum flow shown in (d) is 23 is a maximum flow .

CHARACTERIZATION OF MAXIMUM MATCHING BY AUGMENTING PATHS:

Augmenting Paths for Matchings:

Definitions: ñ Given a matching M in a graph G, a vertex that is not incident to any edge of M is called a free vertex w. r. t. M. ñ for a matching M a path P in G is called an alternating path if edges in M alternate with edges not in M. ñ An alternating path is called an augmenting path for matching M if it ends at distinct free vertices.

Theorem: A matching M is a maximum matching if and only if there is no augmenting path w. r. t. M.

Augmenting Paths in Action
Proof.

- If M is maximum there is no augmenting path P, because we could switch matching and non-matching edges along P. This gives matching M^0M P with larger cardinality.
- Suppose there is a matching M^0 with larger cardinality. Consider the graph H with edge-set M^0M (i.e., only edges that are in either M or M^0 but not in both).
- Each vertex can be incident to at most two edges (one from M and one from M^0). Hence, the connected components are alternating cycles or alternating path.
- As jM^0j>jMj there is one connected component that is a path P for which both endpoints are incident to edges from M^0. P is an alternating path.

Algorithmic idea:
As long as we find an augmenting path augment our matching using this path. When we arrive at a matching for which no augmenting path exists we have a maximum matching.

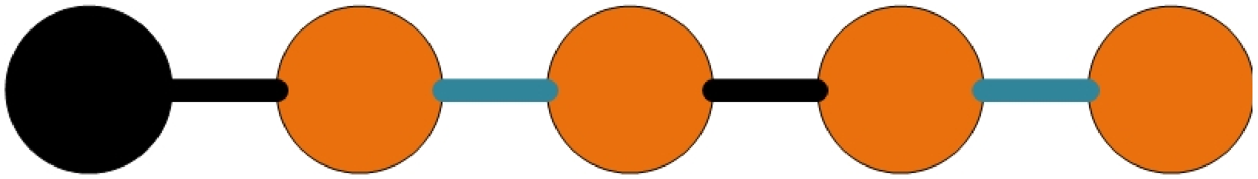
EDMOND'S BLOSSOM ALGORITHM TO COMPUTE AUGMENTING PATH:

Introduction

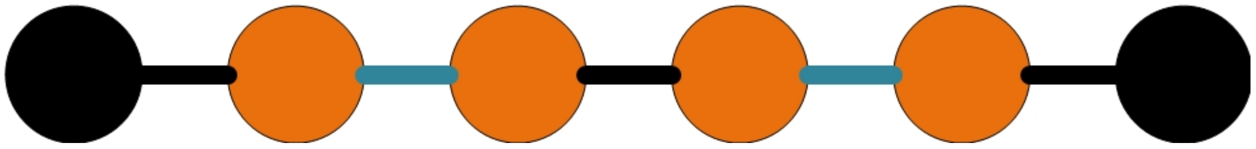
The blossom algorithm, created by Jack Edmunds in 1961, was the first polynomial time algorithm that could produce a maximum matching on any graph. Previous to the discovery of this algorithm, there were no fast algorithms that were able to find a maximum matching on a graph with odd length cycles. The Hungarian Algorithm came the closest still created a maximum matching on a graph under the condition that there were no odd length cycles. A matching is a graph that occurs when all vertices are connected to at most one other vertex through a single edge and therefore the set of all edges in the original graph do not contain any common vertices. A maximum matching occurs when the number of edges in the matching graph is maximized.

Background Understanding - Augmenting and Alternating Path

This algorithm uses the ideas of finding augmenting and alternating paths in a graph. An augmenting path is a path along within a graph where the edges alternate between unmatched and matched edges and it ultimately ends with an unmatched edge. This is in contrast to an alternating path where the path in the original graph alternates between unmatched and matched edges, but it does not start and end with an unmatched edge. Furthermore, this algorithm exploits the theorem that for any matching graph, it is only maximum if and only if there are no augmenting paths (Berge's Lemma). The brief intuition behind this is that if an augmenting path exists, all of the edges' labels can be inverted to create a path that increases the magnitude of the matching edge set by one. Furthermore, to prove that if a graph is maximum, then there must be no augmenting paths by assuming that for a matching M there is a larger matching M'. Then take the graph G that consists of the edges M x or M'. This graph only has vertices with degree at most 2, and therefore it must contain only simple paths or cycles. All cycles must have an equal number of edges from M and M', but for the paths, there will be more edges from M' than from M. Therefore there is at least one path with more edges from M' than M, so this path must start and end with an edge from M' and thus this path is an augmenting path. An example of an alternating path where matched vertices are orange and unmatched vertices are black. The edges that make up the matching are blue.

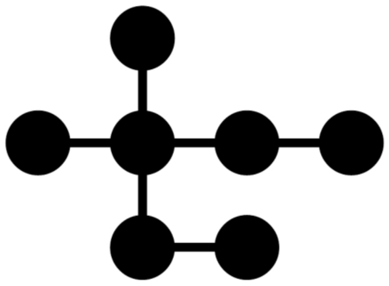


An example of an augmenting path where matched vertices are orange and unmatched vertices are black. The edges that make up the matching are blue.



Finding an Augmenting Path

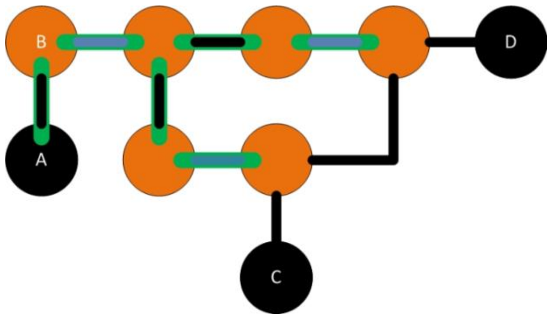
Finding an augmenting path happens as follows. Starting with a free (not part of the matching) vertex V, explore all of its neighbors. Either V will discover another free vertex (and thus find an augmenting path) or it will find a vertex W that is part of the matching. If the latter occurs, mark the edge from V to W and then extend the path to W's mate X. This will always produce an alternating path starting from the original vertex to some other matched vertex in the graph. Finally, recurse with vertex X. This process will continue as long as there is a vertex X that can recursed on.



1. To start all vertices are unmatched and colored black.
2. An unmatched vertex V is chosen.
3. A neighbor (yellow highlight) of V is searched
4. If the neighbor is unmatched, then the augmenting path back to V is inverted (and colored blue).All vertices on the path are marked as matched (colored orange). The graph is then recursed on from (2).
5. Else, the neighbor is matched and the path from V to X is stored (highlighed in green). X is then added to a Queue Q.
    a. If V has unexplored neighbors, recurse from (2) with V
    b. Else, all the neighbors of V are matched. Then X' is dequeued from Q and labled V, and then recurse from (2) with the new V.

Blossoms

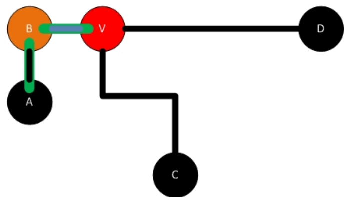
A blossom occurs when an odd length cycle is found while searching for augmenting paths. When a blossom is found, all of the nodes that are part of the cycle are contracted into one super node. This super node retains all the information of the contracted nodes and it gets all of the edges that connect the cycle to the rest of the graph.



The search for an augmenting path already occured as described as steps 1-5 for finding an augmenting path. The starting point for this potential augmenting path is vertex A

1. The yellow edge connects two nodes that could be part of an alternating path back towards A. So a blossom is present.
2. All nodes that are part of the cycle are then contracted into the supernode (red).
3. The supernode is then enqueued into Q.

After the nodes are contracted, the search for an augmenting path continues as if the supernode were a single node. Once the search for an augmenting path is completed (regardless of if it was found or not), the supernode is then lifted. This lifting means that the supernode once again becomes all of the original nodes. If an augmented path was found, then there is an augmenting path from the original free node to another free node with some vertices of the odd length cycle in the path. This augmenting path is then inverted (just like before) in order to increase the size of the matching by one.

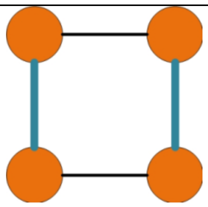
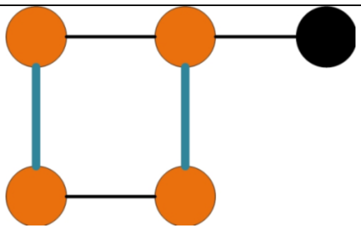


The search for an augmenting path is continued from supernode V as previously described, and it is found that the augmenting path passes through the supernode.

1. The supernode is lifted from its current state so that it is expanded.
2. The path through the supernode that forms an augmenting path is found (green highlight).
3. The augmenting path is then inverted.

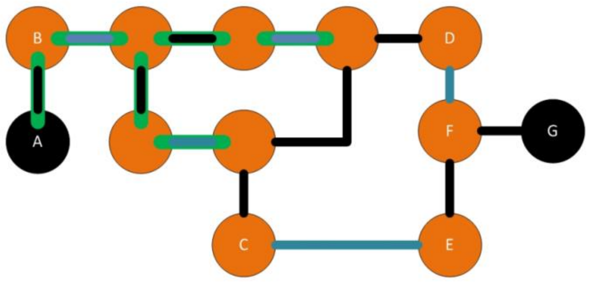
The Algorithm

Intuitively the algorithm works by repeatedly trying to find augmenting paths (specifically paths starting and ending with free vertices) in the graph until no more augmenting paths can be found. This occurs when either all vertices are matched with a mate or when a free vertex is only connected to matched vertices which recursively only match to other matched vertices.

All Vertices Matched and a maximum matching	**Not All Vertices Matched and a maximum matching**
	

Furthermore, the process of finding augmented paths can be slightly modified so that it simultaneously can check to see if there is a blossom in the graph. This can be accomplished by labeling each vertex in an alternating manner O (for odd) and E (for even) depending on how far away a vertex is from the root. To start, label the original free vertex E, and then alternate labeling from there. The algorithm that finds augmeted paths will always search for neighboring vertices from a vertex labeled E (the matched vertex’s mate always is two edges further away from the root) and thus if an adjacent vertex also has the label E, there is a blossom that contains both of those vertices. This blossom is then contracted into one supernode with a label E and then the search for augmenting paths continues.

All together, this algorithm starts by finding augmenting paths within any graph while labeling vertices E or O and then inverting these paths to increase the size of the matching by one. If a blossom is found, it is contracted into a single node and the augmenting path algorithm continues as normal. This repeats until all vertices of the graph have been either searched or are already part of the matching set.



Blossoms can be nested within each other, and therefore, supernodes may end up containing other supernodes.

Time Complexity

Overall, this algorithm takes O(E * V^2) time. Every attempt to find an augmenting path takes O(E) time in order to iterate over all of the edges. Assuming that an attempt to find an augmenting path succeeds, it will take O(V) time to flip all of the incident edges. Finally, there are O(V) blossoms in any graph and for each blossom the smaller graph with the supernode may need to be recursed on. This means that the total time is O(E * V^2) or simply O(V^4) for a dense graph.

UNIT-3

FLOW-NETWORKS:

A **network** is a directed graph G with vertices V and edges E combined with a function cc, which assigns each edge ee a non-negative integer value, the **capacity** of ee. Such a network is called a **flow network**, if we additionally label two vertices, one as **source** and one as **sink**.

A **flow** in a flow network is function ff, that again assigns each edge ee a non-negative integer value, namely the flow. The function has to fulfill the following two conditions:
The flow of an edge cannot exceed the capacity.

$$f(e) \leq c(e)$$

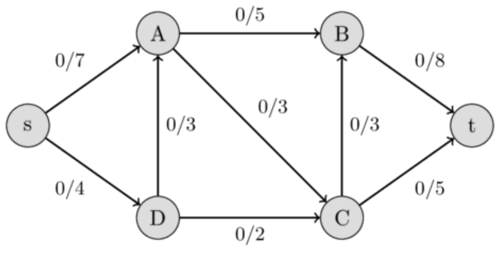
And the sum of the incoming flow of a vertex uu has to be equal to the sum of the outgoing flow of u except in the source and sink vertices.

$$\sum_{(v,u)\in E} f((v,u)) = \sum_{(u,v)\in E} f((u,v))$$

The source vertex ss only has an outgoing flow and the sink vertex tt has only incoming flow.
It is easy to see that the following equation holds:

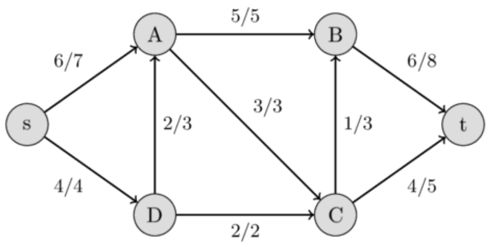
$$\sum_{(s,u)\in E} f((s,u)) = \sum_{(u,t)\in E} f((u,t))$$

A good analogy for a flow network is the following visualization: We represent edges as water pipes, the capacity of an edge is the maximal amount of water that can flow through the pipe per second, and the flow of an edge is the amount of water that currently flows though the pipe per second. This motivates the first flow condition. There cannot flow more water through a pipe than its capacity. The vertices act as junctions, where water comes out of some pipes and distributes it in some way to other pipes. This also motivates the second flow condition. In each junction all the incoming water has to be distributed to the other pipes. It cannot magically disappear or appear. The source s is origin of all the water and the water can only drain in the sink t. The following image shows a flow network. The first value of each edge represents the flow, which is initially 0, and the second value represents the capacity.



The value of a flow of a network is the sum of all flows that gets produced in source ss, or equivalently of the flows that are consumed in the sink tt. A **maximal flow** is a flow with the maximal possible value. Finding this maximal flow of a flow network is the problem that we want to solve.
In the visualization with water pipes, the problem can be formulated in the following way: how much water can we push through the pipes from the source to the sink. The following image shows the

maximal flow in the flow network.



In graph theory, a flow network is defined as a directed graph involving a source(S) and a sink(T) and several other nodes connected with edges. Each edge has an individual capacity which is the maximum limit of flow that edge could allow.
Flow in the network should follow the following conditions:

- For any non-source and non-sink node, the input flow is equal to output flow.
- For any edge (Ei) in the network, 0≤flow (Ei) ≤Capacity (Ei).
- Total flow out of the source node = total flow in to the sink node.
- Net flow in the edges follows skew symmetry i.e. F(u,v)=−F(v,u) where F(u,v) is flow from node u to node v. This leads to a conclusion where you have to sum up all the flows between two nodes (either directions) to find net flow between the nodes initially.

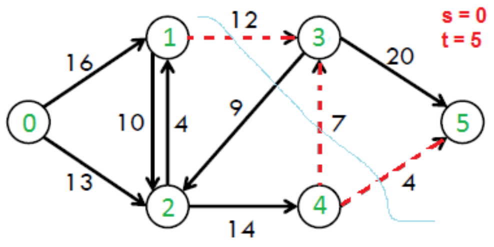
Maximum Flow:
It is defined as the maximum amount of flow that the network would allow to flow from source to sink. Multiple algorithms exist in solving the maximum flow problem. Two major algorithms to solve these kind of problems are Ford-Fulkerson algorithm and Dinic's Algorithm.

MAXFLOW-MINCUT THEOREM:

In computer science and optimization theory, the max-flow min-cut theorem states that in a flow network, the maximum amount of flow passing from the source to the sink is equal to the total weight of the edges in the minimum cut, i.e the smallest total weight of the edges which if removed would disconnect the source from the sink.

Find minimum s-t cut in a flow network

In a flow network, an s-t cut is a cut that requires the source ‘s’ and the sink ‘t’ to be in different subsets and it consists of edges going from the source’s side to the sink’s side. The capacity of an s-t cut is defined by the sum of the capacity of each edge in the cut-set. The problem discussed here is to find minimum capacity s-t cut of the given network. Expected output is all edges of the minimum cut.
For example, in the following flow network, example s-t cuts are {{0 ,1}, {0, 2}}, {{0, 2}, {1, 2}, {1, 3}}, etc. The minimum s-t cut is {{1, 3}, {4, 3}, {4 5}} which has capacity as 12+7+4 = 23.



Minimum Cut and Maximum Flow

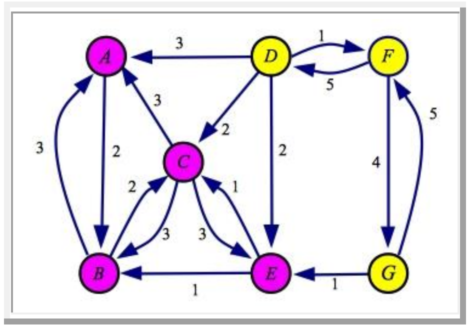
Maximum Bipartite Matching, this is another problem which can be solved using Ford-Fulkerson Algorithm. This is based on max-flow min-cut theorem. The max-flow min-cut theorem states that in a flow network, the amount of maximum flow is equal to capacity of the minimum cut. From Ford-Fulkerson, we get capacity of minimum cut. How to print all edges that form the minimum cut? The idea is to use residual graph.

Following are steps to print all edges of the minimum cut.

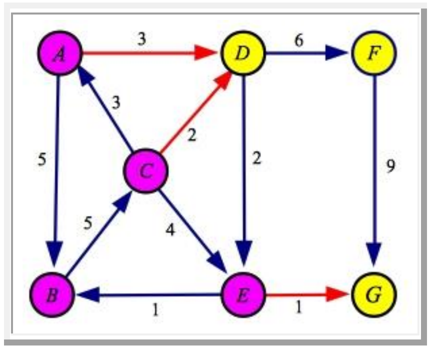
1. Run Ford-Fulkerson algorithm and consider the final residual graph.
2. Find the set of vertices that are reachable from the source in the residual graph.
3. All edges which are from a reachable vertex to non-reachable vertex are minimum cut edges. Print all such edges.

Example: Finding the Minimum Cut

The final residual graph is used to find the minimum cut. First, you find all nodes reachable from the source in the residual graph. This is one set of nodes, which we color purple below:



Now, in the original graph, we divide our nodes into two sets: the set determined above, and all of the remaining nodes. They are drawn purple and yellow in the original graph below:



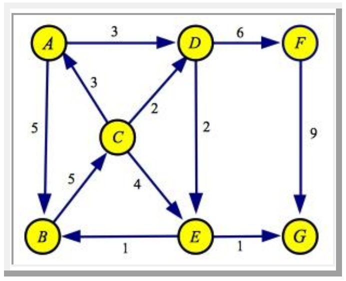
The minimum cut is composed of all the edges that go from the source set to the sink set. These are edges AD, CD and EG, which I've drawn in red above. The sum of their capacities equals the maximum flow of six.

Example: Finding the Maximum Flow Path through the Graph

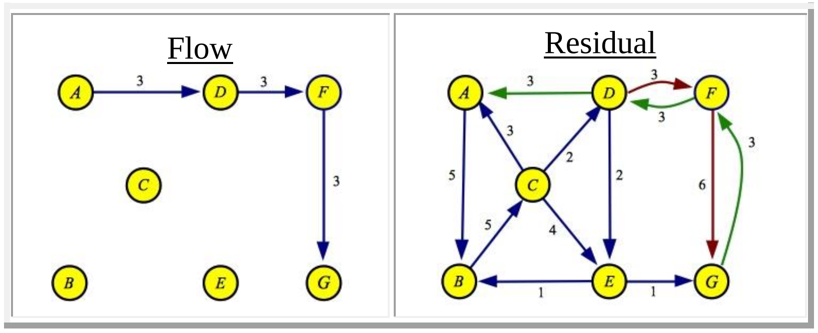
Instead of using a depth-first search, you can use a modification of Dijkstra's algorithm to find the path through the residual that has the maximum flow. When processing a node, Dijkstra's algorithm traverses the node's edges, and if shorter paths to any other nodes is discovered, the nodes are updated. At each step, the node with the shortest known path is processed next.

The modification works as follows. When processing a node, again the algorithm traverses the node's edges, and if paths with more flow to any other nodes are discovered, then the nodes are updated. At each step, the node with the maximum flow is processed next.

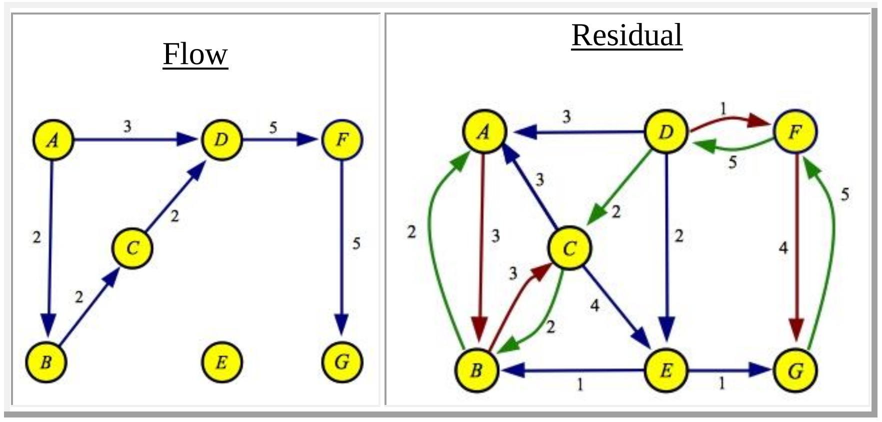
We'll work an example with the same graph as above:



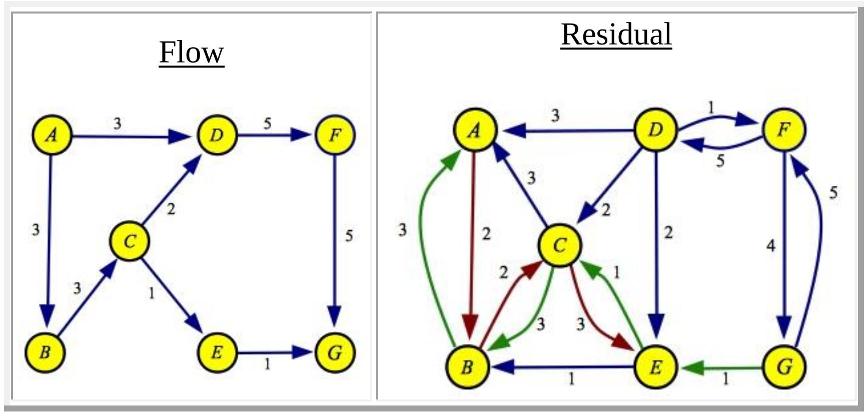
The maximum flow path here is *A->D->F->G* with a flow of three:



The next maximum flow path is *A->B->C->D->F->G* with a flow of 2:



The final path is *A->B->C->E->G* with a flow of one:



FORD-FULKERSON METHOD TO COMPUTE MAXIMUM FLOW

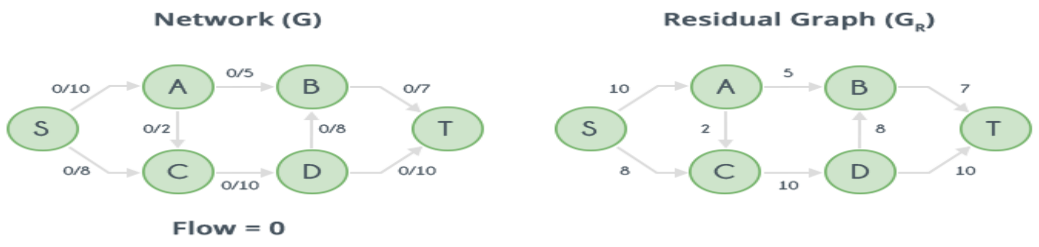
The Concept of Residual Graph is needed for understanding the implementation. ***Residual Graph*** of a flow network is a graph which indicates additional possible flow. If there is a path from source to sink in residual graph, then it is possible to add flow. Every edge of a residual graph has a value called ***residual capacity*** which is equal to original capacity of the edge minus current flow. Residual capacity is basically the current capacity of the edge.
Let us now talk about implementation details. Residual capacity is 0 if there is no edge between two vertices of residual graph.

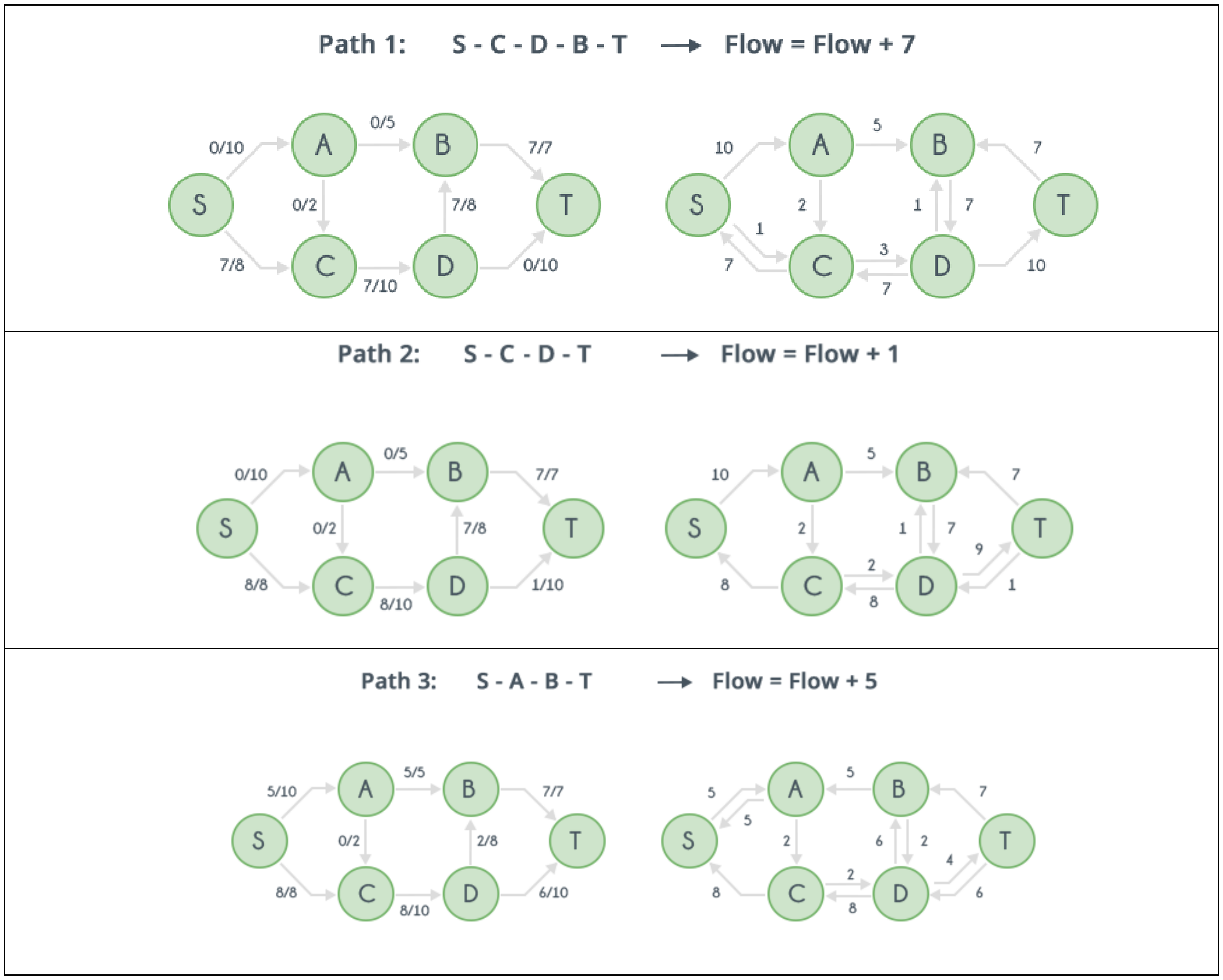
- We can initialize the residual graph as original graph as there is no initial flow and initially residual capacity is equal to original capacity. To find an augmenting path, we can either do a BFS or DFS of the residual graph. We have used BFS in below implementation.
- Using BFS, we can find out if there is a path from source to sink. BFS also builds parent [ ] array. Using the parent [ ] array, we traverse through the found path and find possible flow through this

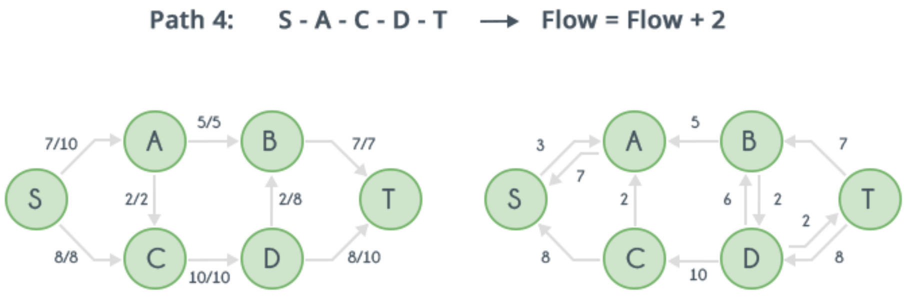
path by finding minimum residual capacity along the path. We later add the found path flow to overall flow.

- The important thing is, we need to update residual capacities in the residual graph. We subtract path flow from all edges along the path and we add path flow along the reverse edges. We need to add path flow along reverse edges because may later need to send flow in reverse direction

A demonstration of working of Ford-Fulkerson algorithm is shown below with the help of diagrams.




Path 1: S - C - D - B - T → Flow = Flow + 7
Path 2: S - C - D - T → Flow = Flow + 1
Path 3: S - A - B - T → Flow = Flow + 5

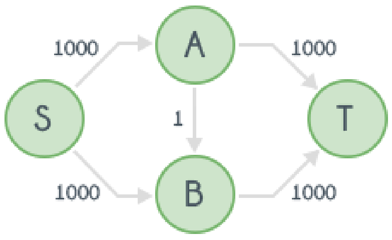


No more Path Left Max Flow : 15

Implementation:

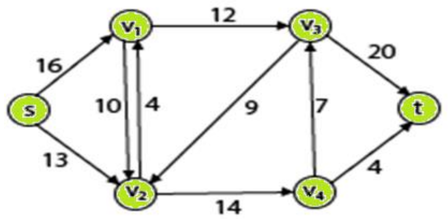
- An augmenting path in residual graph can be found using DFS or BFS.
- Updating residual graph includes following steps: (refer the diagrams for better understanding)
    - For every edge in the augmenting path, a value of minimum capacity in the path is subtracted from all the edges of that path.
    - An edge of equal amount is added to edges in reverse direction for every successive nodes in the augmenting path.

The complexity of Ford-Fulkerson algorithm cannot be accurately computed as it all depends on the path from source to sink. For example, considering the network shown below, if each time, the path chosen are S−A−B−T and S−B−A−T alternatively, then it can take a very long time. Instead, if path chosen are only S−A−T and S−B−T, would also generate the maximum flow.

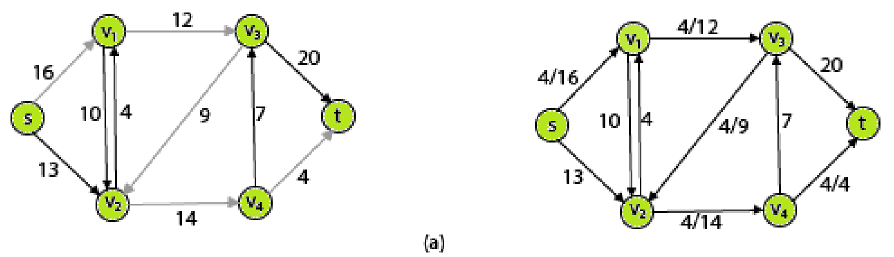


Example- Ford-Fulkerson algorithm:

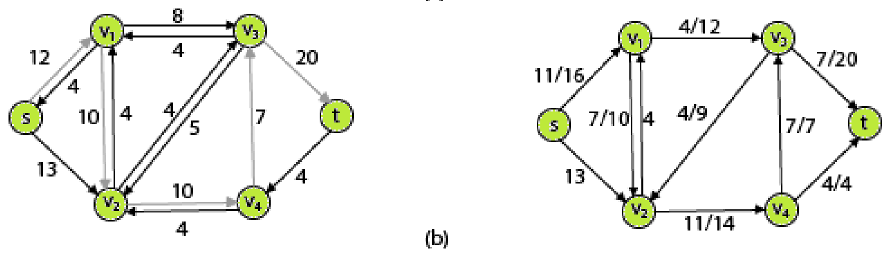
Each Directed Edge is labeled with capacity. Use the Ford-Fulkerson algorithm to find the maximum flow.



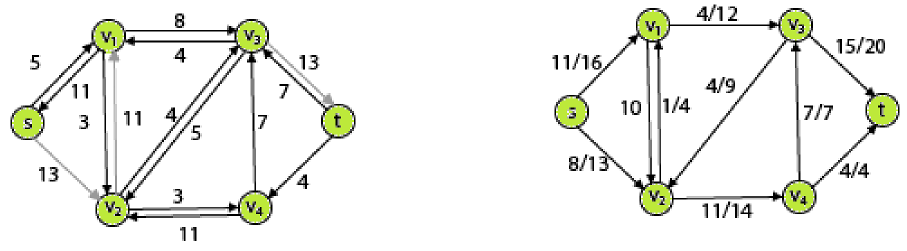
The left side of each part shows the residual network G_f with a shaded augmenting path p,and the right side of each part shows the net flow f.



(a)

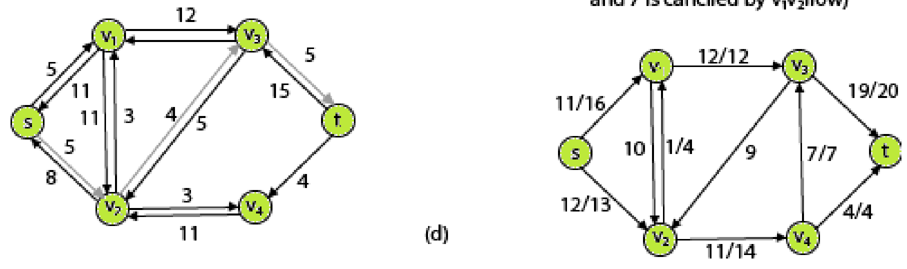


(b)

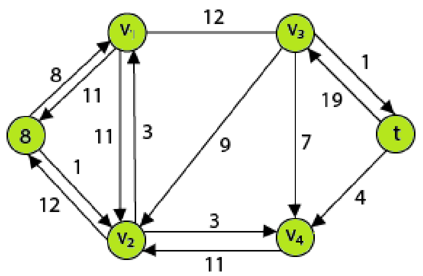


(c)

(In this , 8 is break into 7 and 1 and 7 is canclled by $v_1v_2$flow)



(d)



Now , it has no augmenting paths .So, the maximum flow shown in (d) is 23 is a maximum flow .

(e)

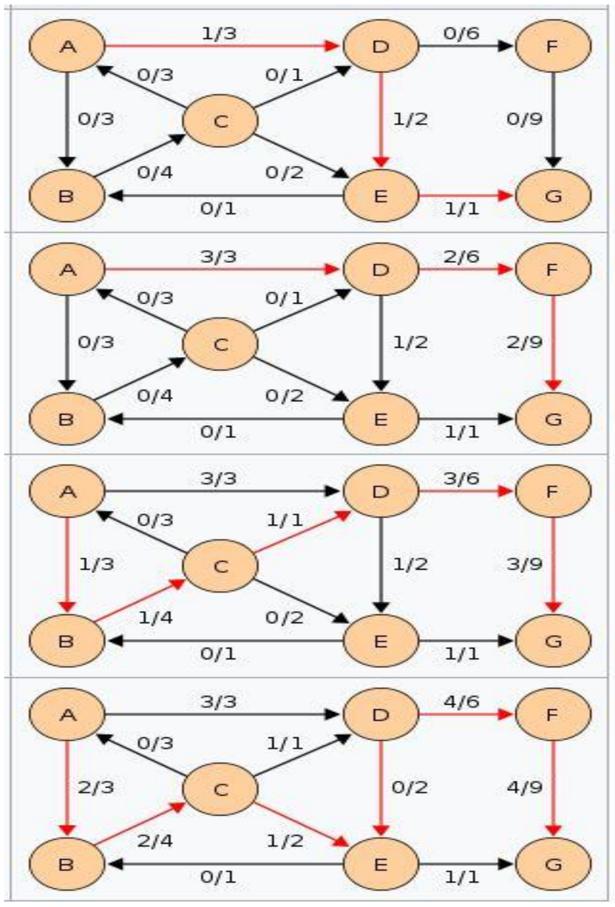
EDMOND-KARP MAXIMUM-FLOW ALGORITHM:

Edmonds–Karp algorithm is an implementation of the Ford–Fulkerson method for computing the maximum flow in a flow network in much more optimized approach. Edmonds-Karp is identical to Ford-Fulkerson except for one very important trait. The search order of augmenting paths is well defined. Here augmenting paths, along with residual graphs, are the two important concepts to understand when finding the max flow of a network.

Edmonds–Karp algorithm

```
Algorithm EdmondsKarp(G)
1: f ← 0; Gf ← G
2: while Gf contains an s − t path P do
3: Let P be an s − t path in Gf with the minimum number of edges.
4: Augment f using P.
5: Update Gf
6: end while
7: return f
```

Augmenting paths are simply any path from the source to the sink that can currently take more flow. Over the course of the algorithm, flow is monotonically increased. So, there are times when a path from the source to the sink can take on more flow and that are an augmenting path. This can be found by a breadth-first search, as we let edges have unit length. The running time of O(V E2) is found by showing that each augmenting path can be found in O(E) time, that every time at least one of the E edges becomes saturated (an edge which has the maximum possible flow), that the distance from the saturated edge to the source along the augmenting path must be longer than last time it was saturated and that the length is at most V. Another property of this algorithm is that the length of the shortest augmenting path increases monotonically.



Notice how the length of the augmenting path found by the algorithm (in red) never decreases. The paths found are the shortest possible. The flow found is equal to the capacity across the minimum cut in the graph separating the source and the sink. There is only one minimal cut in this graph, partitioning the nodes into the sets { A , B , C , E } and { D , F , G }, with the capacity is c(A,D) + c (C,D) +c (E,G) =3+1+1 =5

Complexity

- Worst case time complexity: **Θ(V * E * E)**
- Average case time complexity: **Θ(V * E * E)**
- Best case time complexity: **Θ(V * E * E)**
- Space complexity: **Θ(E + V)**

Proof of complexity: **2 Key ideas**:

- In the residual network, the distance of any vertex from the source never decreases during the algorithm.
- Any edge can become critical at most O(V) times.

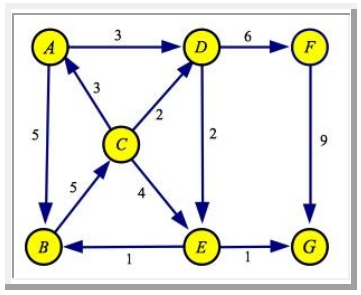
Applications of Edmonds Karp Algorithm are:

- Finding the maximum flow in a flow network
- Maximizing the transportation with given traffic limits
- Maximizing packet flow in computer networks.

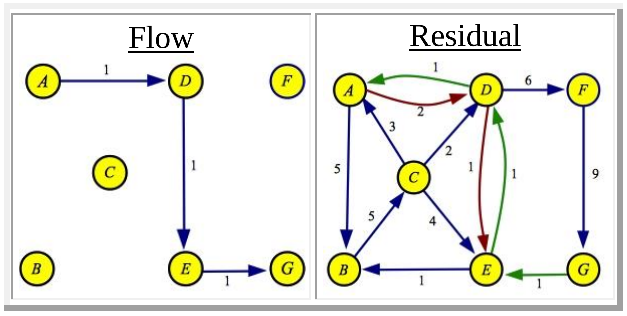
Example: Edmonds-Karp: Finding the minimum hop path

Finally, the Edmonds-Karp algorithm uses a straightforward breadth-first search to find the minimum hop path through the residual graph. This is equivalent to treating the residual as an unweighted and performing a shortest path search on it.

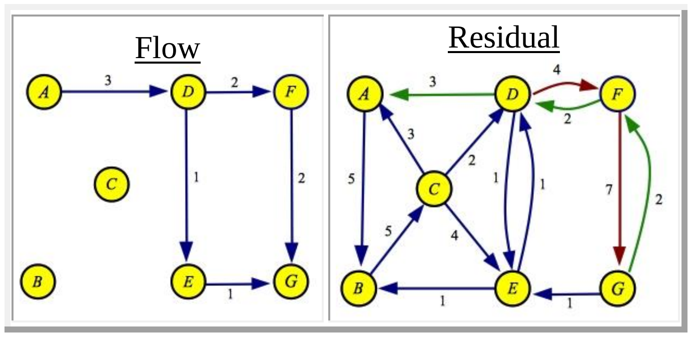
Again, we use the same graph as an example:



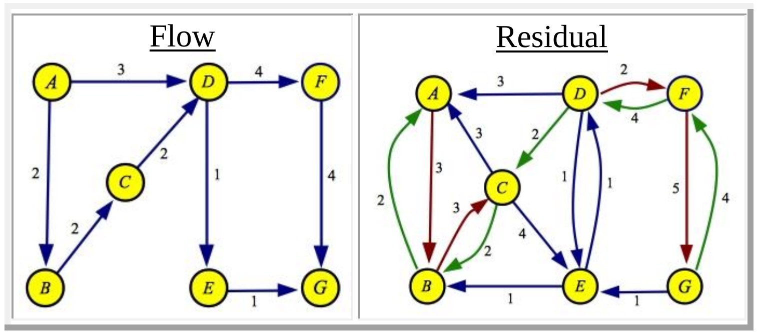
There are two minimum hop paths: *A->D->E->G* and *A->D->F->G*. Suppose we process the former of these, with a flow of one:



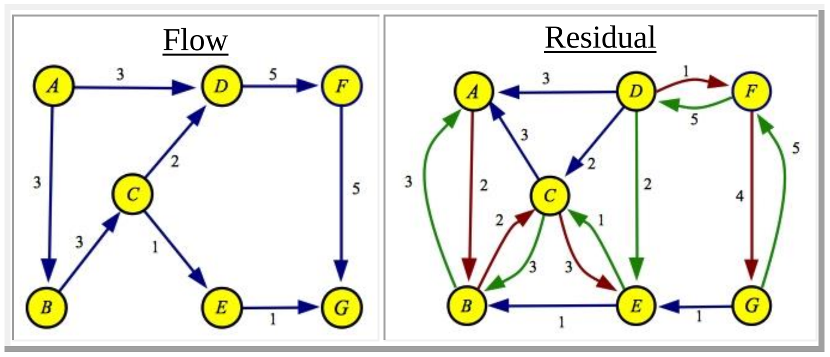
Now, there is only one minimum hop path through the residual: *A->D->F->G*, with a flow of two:



At this point, there are only two paths through the residual: *A->B->C->D->F->G* and *A->B->C->E->D->F->G*. The first of these has fewer hops, so we process it. It has a flow of two:



The final path through the residual is *A->B->C->E->D->F->G* with a flow of one. When we process it, we get the same flow and residual graphs as the other two algorithms:



MATRIX COMPUTATIONS:

A **matrix** is a rectangular array of numbers arranged in rows and columns. The array of numbers below is an example of a **matrix**. The number of rows and columns that a **matrix** has is called its dimension or its order. By convention, rows are listed first; and columns, second

Use of Matrix Computation:

They are **used** for plotting graphs, statistics and also to do scientific studies and research in almost different fields. **Matrices** are also **used** in representing the **real world** data's like the population of people, infant mortality rate, etc. They are best representation methods for plotting surveys.

Matrix Multiplication:

Input: $A = [a_{ij}], B = [b_{ij}].$
Output: $C = [c_{ij}] = A \cdot B.$ $\quad \} \; i, j = 1, 2, \ldots, n.$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Standard algorithm:

```
for $i \leftarrow 1$ to $n$
    do for $j \leftarrow 1$ to $n$
            do $c_{ij} \leftarrow 0$
                for $k \leftarrow 1$ to $n$
                    do $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$
```

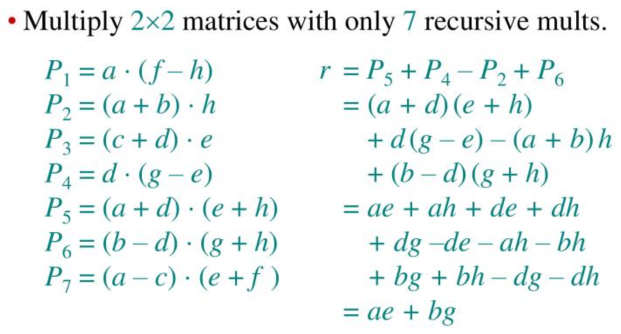
Running time $= \Theta(n^3)$

Idea of Strassen's Algorithm:

- Multiply 2×2 matrices with only 7 recursive mults.

$P_1 = a \cdot (f - h)$ $\qquad r = P_5 + P_4 - P_2 + P_6$
$P_2 = (a + b) \cdot h$ $\qquad s = P_1 + P_2$
$P_3 = (c + d) \cdot e$ $\qquad t = P_3 + P_4$
$P_4 = d \cdot (g - e)$ $\qquad u = P_5 + P_1 - P_3 - P_7$
$P_5 = (a + d) \cdot (e + h)$
$P_6 = (b - d) \cdot (g + h)$
$P_7 = (a - c) \cdot (e + f)$

7 mults, 18 adds/subs.
Note: No reliance on commutativity of mult!



• Multiply 2×2 matrices with only 7 recursive mults.

$$
\begin{aligned}
P_1 &= a \cdot (f-h) & r &= P_5 + P_4 - P_2 + P_6 \\
P_2 &= (a+b) \cdot h & &= (a+d)(e+h) \\
P_3 &= (c+d) \cdot e & &\quad + d(g-e) - (a+b)h \\
P_4 &= d \cdot (g-e) & &\quad + (b-d)(g+h) \\
P_5 &= (a+d) \cdot (e+h) & &= ae + ah + de + dh \\
P_6 &= (b-d) \cdot (g+h) & &\quad + dg - de - ah - bh \\
P_7 &= (a-c) \cdot (e+f) & &\quad + bg + bh - dg - dh \\
& & &= ae + bg
\end{aligned}
$$

STRASSEN'S ALGORITHM:

1.*Divide:* Partition *A* and *B* into (*n*/2)x(*n*/2) submatrices. Form terms to be multiplied using + and –.

2.*Conquer:* Perform 7 multiplications of (*n*/2)x(*n*/2) submatrices recursively.

3.*Combine:* Perform +and – on (*n*/2)x(*n*/2) submatrices. Combine the result of two matrices to find the final product or final matrix.

In this context, using Strassen's Matrix multiplication algorithm, the time consumption can be improved a little bit. Strassen's Matrix multiplication can be performed only on **square matrices** where **n** is a **power of 2**. Order of both of the matrices are **n × n**.

Divide **X**, **Y** and **Z** into four (n/2)×(n/2) matrices as represented below −

$$Z = \begin{bmatrix} I & J \\ K & L \end{bmatrix} \quad X= \begin{bmatrix} A & B \\ C & D \end{bmatrix} \quad Y=\begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Using Strassen's Algorithm compute the following:

$$M_1:=(A+C)\times(E+F)$$
$$M_2:=(B+D)\times(G+H)$$
$$M_3:=(A-D)\times(E+H)$$
$$M_4:=A\times(F-H)$$
$$M_5:=(C+D)\times(E)$$
$$M_6:=(A+B)\times(H)$$
$$M_7:=D\times(G-E)$$

Then,

$$I:=M_2+M_3-M_6-M_7$$
$$J:=M_4+M_6$$
$$K:=M_5+M_7$$
$$L:=M_1-M_3-M_4-M_5$$

Analysis

$T(n) = \begin{cases} c \\ 7 \text{ x } T\left(\frac{n}{2}\right) + dxn^2 ifn = 1\, otherwise \end{cases}$ where *c* and *d* are constants

- ➢ Using this recurrence relation, we get T(n)=O($n^{\log 7}$)T(n)=O($n^{\log 7}$)
- ➢ Addition and Subtraction of two matrices takes O(N^2) time. So time complexity can be written asT(N) =7T (N/2) + O (N^2)
- ➢ Hence, the complexity of Strassen's matrix multiplication algorithm is O($n^{\log 7}$)

$$T(n) = 7\,T(n/2) + \Theta(n^2)$$

$$n^{\log_b a} = n^{\log_2 7} \approx n^{2.81} \;\Rightarrow\; \text{CASE 1} \;\Rightarrow\; T(n) = \Theta(n^{\lg 7}).$$

- The number 2.81 may not seem much smaller than 3, but because the difference is in the exponent, the impact on running time is significant. In fact, Strassen's algorithm beats the ordinary algorithm today's machines for n ≥ 30 so on.

Generally Strassen's Method is not preferred for practical applications for following reasons.

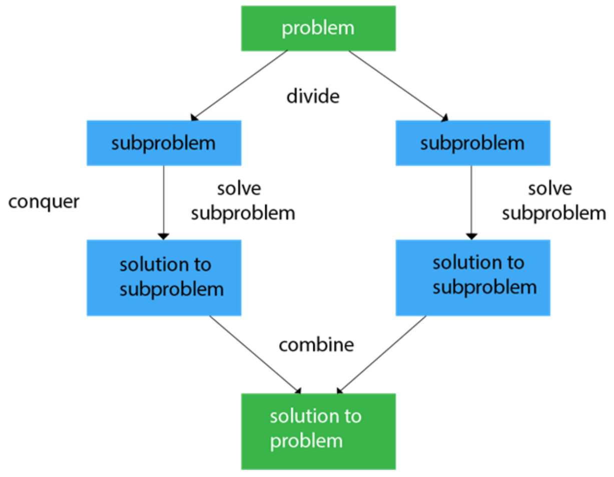
1. The constants used in Strassen's method are high and for a typical application Naive method works better.
2. For Sparse matrices, there are better methods especially designed for them.
3. The submatrices in recursion take extra space.
4. Because of the limited precision of computer arithmetic on noninteger values, larger errors accumulate in Strassen's algorithm than in Naive Method

INTRODUCTION TO DIVIDE AND CONQUER PARADIGM:

Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.

Divide and Conquer algorithm consists of a dispute using the following three steps.

1. **Divide:** The original problem into a set of sub-problems.
2. **Conquer:** Solve every sub-problem individually, recursively.
3. **Combine:** Put together the solutions of the sub-problems to get the solution to the whole problem.



Fundamental of Divide & Conquer Strategy:

There are two fundamentals of Divide & Conquer Strategy:

1. Relational Formula
2. Stopping Condition

1. Relational Formula: It is the formula that we generate from the given technique. After generation of Formula we apply D&C Strategy, i.e. we break the problem recursively & solve the broken sub-problems.

2. Stopping Condition: When we break the problem using Divide & Conquer Strategy, then we need to know that for how much time, we need to apply divide & Conquer. So the condition where we need to stop our recursion steps of D&C is called as Stopping Condition.

Standard Algorithms based on divide and conquer

The following algorithms are based on divide and conquer design paradigm

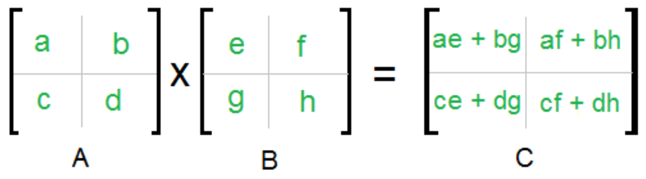
- Binary Search
- Strassen's matrix multiplication
- Closestpair(points)
- Cooley-Tukey Fast Fourier Transform (FFT) Algorithm
- Karatsuba Algorithm for Fast Multiplication
- Maximum and Minimum Problem
- Sorting (merge sort, quick sort)
- Tower of Hanoi.

Divide and Conquer

Following is simple Divide and Conquer method to multiply two square matrices.

1) Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.

2) Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.



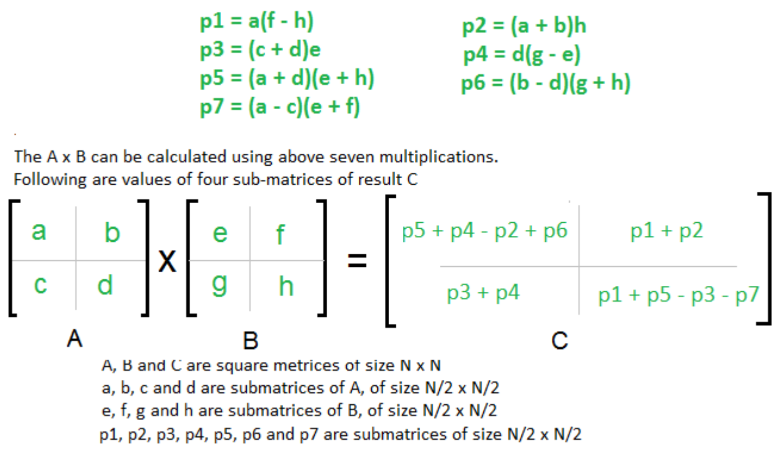
A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. Addition of two matrices takes O(N^2) time. So the time complexity can be written as$T(N) = 8T(N/2) + O(N^2)$

From Master's Theorem, time complexity of above method is O (N^3)

Simple Divide and Conquer also leads to O(N^3), can there be a better way?
In the above divide and conquer method, the main component for high time complexity is 8 recursive calls. The idea of **Strassen's method** is to reduce the number of recursive calls to 7. Strassen's method is similar to above simple divide and conquer method in the sense that this method also divide matrices

to sub-matrices of size N/2 x N/2 as shown in the below diagram, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.



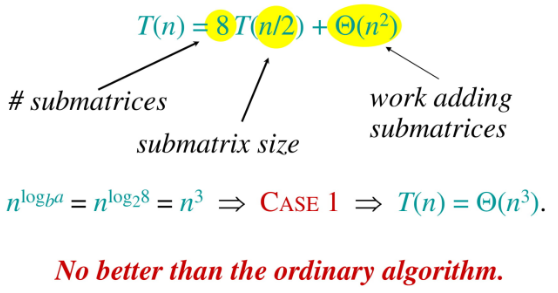
p1 = a(f - h)
p2 = (a + b)h
p3 = (c + d)e
p4 = d(g - e)
p5 = (a + d)(e + h)
p6 = (b - d)(g + h)
p7 = (a - c)(e + f)

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\underset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \underset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

Analysis of Divide and Conquer Algorithm:



$$T(n) = 8\,T(n/2) + \Theta(n^2)$$

$$n^{\log_b a} = n^{\log_2 8} = n^3 \Rightarrow \textsc{Case } 1 \Rightarrow T(n) = \Theta(n^3).$$

No better than the ordinary algorithm.

Pros of Divide and Conquer Strategy

1. Solves difficult problems with ease.
2. Algorithms designed with Divide and Conquer strategies are efficient when compared to its counterpart Brute-Force approach for e.g. if we compare time complexity of simple matrix multiplication i.e. O(n3) and that of Strassen's matrix multiplication i.e. O(n2.81).
3. Algorithms designed under Divide and Conquer strategy does not require any modification as they inhibit parallelism and can easily be processed by parallel processing systems.
4. It makes efficient use of memory cache. This happens so because that problems when divided gets so small that they can be easily solved in the cache itself.
5. If we use floating numbers then the results may improve since round off control is very efficient in such algorithms.
6. The most recognizable benefit of the divide and conquer paradigm is that it allows us to solve difficult problem, such as the Tower of Hanoi, which is a mathematical game or puzzle. Being given a difficult problem can often be discouraging if there is no idea how to go about solving it. However, with the divide and conquer method, it reduces the degree of difficulty since it divides the problem into sub-problems that are easily solvable and usually runs faster than other algorithms would.

7. It also uses memory caches effectively. When the sub-problems become simple enough, they can be solved within a cache, without having to access the slower main memory.

Cons of Divide and Conquer Strategy

1. **Divide and Conquer** strategy uses recursion that makes it a little slower and if a little error occurs in the code the program may enter into an infinite loop.
2. Usage of explicit stacks may make use of extra space.
3. If performing a recursion for number of times greater than the stack in the CPU than the system, may crash.
4. Choosing base cases is also a limitation as picking small cases may work, if the cases are taken much higher than the capacity of the system than problems may occur.
5. Sometimes a case where the problem when broken down results in same sub-problems which may needlessly increase the solving time and extra space may be consumed.
6. One of the most common issues with this sort of algorithm is the fact that the recursion is slow, which in some cases outweighs any advantages of this divide and conquer process.
7. Sometimes it can become more complicated than a basic iterative approach, especially in cases with a large n. In other words, if someone wanted to add a large amount of numbers together, if they just create a simple loop to add them together., it would turn out to be a much simpler approach than it would be to divide the numbers up into two groups, add these groups recursively and then add the sums of the two groups together.

INVERSE OF A TRIANGULAR MATRIX:

- Computation of inverse using co-factor matrix
- Properties of the inverse of a matrix
- Inverse of special matrices
  - ✓ Unit Matrix
  - ✓ Diagonal Matrix
  - ✓ Orthogonal Matrix
  - ✓ Lower/Upper Triangular Matrices

Matrix Inverse

- Inverse of a matrix can only be defined for square matrices.
- Inverse of a square matrix exists only if the determinant of that matrix is non-zero.
- Inverse matrix of A is noted as A^{-1}.
- $AA^{-1} = A^{-1}A = \mathrm{I}$

- Example:
  - $A = \begin{bmatrix} 2 & -1 \\ 1 & 0 \end{bmatrix}, A^{-1} = \begin{bmatrix} 0 & 1 \\ -1 & 2 \end{bmatrix},$
  - $AA^{-1} = \begin{bmatrix} 2 & -1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 0 & 1 \\ -1 & 2 \end{bmatrix} = A^{-1}A = \begin{bmatrix} 0 & 1 \\ -1 & 2 \end{bmatrix}\begin{bmatrix} 2 & -1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

Inverse of a 3 x 3 matrix (using cofactor matrix):

Calculating the inverse of a 3 × 3 matrix is:

- Compute the matrix of minors for A.
- Compute the cofactor matrix by alternating + and – signs.
- Compute the adjugate matrix by taking a transpose of cofactor matrix.
- Divide all elements in the adjugate matrix by determinant of matrix.

$$A^{-1} = \frac{1}{\det(A)}\, adj(A)$$

Properties of Inverse of a Matrix

- $(A^{-1})^{-1} = A$
- $(AB)^{-1} = B^{-1}A^{-1}$
- $(kA)^{-1} = k^{-1}A^{-1}$ where k is a non-zero scalar.
- $(A^{T})^{-1} = (A^{-1})^{T}$

Inverse of Identity matrices

• Inverse of identity matrix is itself.

• Because: • II = I

Example:

- $AI = A,\ \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$
- $I^{-1}A = A, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

Inverse of a 3 x 3 matrix (using cofactor matrix)

$$A = \begin{bmatrix} 3 & 0 & 2 \\ 2 & 0 & -2 \\ 0 & 1 & 1 \end{bmatrix}$$

$$\text{Matrix of Minors} = \begin{bmatrix} \begin{vmatrix} 0 & -2 \\ 1 & 1 \end{vmatrix} & \begin{vmatrix} 2 & -2 \\ 0 & 1 \end{vmatrix} & \begin{vmatrix} 2 & 0 \\ 0 & 1 \end{vmatrix} \\ \begin{vmatrix} 0 & 2 \\ 1 & 1 \end{vmatrix} & \begin{vmatrix} 3 & 2 \\ 0 & 1 \end{vmatrix} & \begin{vmatrix} 3 & 0 \\ 0 & 1 \end{vmatrix} \\ \begin{vmatrix} 0 & 2 \\ 0 & -2 \end{vmatrix} & \begin{vmatrix} 3 & 2 \\ 2 & -2 \end{vmatrix} & \begin{vmatrix} 3 & 0 \\ 2 & 0 \end{vmatrix} \end{bmatrix} = \begin{bmatrix} 2 & 2 & 2 \\ -2 & 3 & 3 \\ 0 & -10 & 0 \end{bmatrix}$$

$$\text{Cofactor of A } (\mathbf{C}) = \begin{bmatrix} 2 & 2 & 2 \\ -2 & 3 & 3 \\ 0 & -10 & 0 \end{bmatrix} .* \begin{bmatrix} 1 & -1 & 1 \\ -1 & 1 & -1 \\ 1 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 2 & -2 & 2 \\ 2 & 3 & -3 \\ 0 & 10 & 0 \end{bmatrix}$$

$$\text{adj(A)} \;= \text{CT} = \begin{bmatrix} 2 & 2 & 0 \\ -2 & 3 & 10 \\ 2 & -3 & 0 \end{bmatrix}$$

$$\text{A}^{-1} = \frac{1}{|A|} * \text{adj(A)} = \frac{1}{10}\begin{bmatrix} 2 & 2 & 0 \\ -2 & 3 & 10 \\ 2 & -3 & 0 \end{bmatrix} = \begin{bmatrix} 0.2 & 0.2 & 0 \\ -0.2 & 0.3 & 1 \\ 0.2 & -0.3 & 0 \end{bmatrix}$$

Inverse of Diagonal matrices

• The determinant of a diagonal matrix is the product of its diagonal elements.

• If they all are non-zero, then determinant is non-zero and the matrix is invertible.

• The inverse of a diagonal matrix A is another diagonal matrix B whose diagonal elements are the reciprocals of the diagonal elements of A.

• Example:

- $A = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \quad |A| = 2 \times 1 \times (-1) = -2 \;\; (\neq 0)$
- $A^{-1} = \begin{bmatrix} \frac{1}{2} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$

Inverse of Orthonormal matrices

- ➢ Earlier, we saw that multiplication of an orthogonal (orthonormal) matrix in its transpose results in identity matrix
- ➢ If A is an orthonormal matrix, its inverse is equal to its transpose
- ➢ A is an orthonormal $n \times n$ matrix
- ➢ Recall $AA^T = I_n$, where I_n is a $n \times n$ identity matrix
- ➢ So, $A^T = A^{-1}$

Inverse of Upper/Lower Triangular Matrices

- ➢ Inverse of an upper/lower triangular matrix is another upper/lower triangular matrix.
- ➢ Inverse exists only if none of the diagonal element is zero.
- ➢ Can be computed from first principles: Using the definition of an Inverse. $AA^{-1} = I$. No need to compute determinant. Diagonal elements of A^{-1} is the reciprocal of the elements of A.
- ➢ Other elements are iteratively computed such that the product of the matrices is I.

Inverse of Upper/Lower Triangular Matrices

Upper Triangular Matrix:

$$A = \begin{bmatrix} 2 & 1 \\ 0 & -1 \end{bmatrix}; \quad A^{-1} \begin{bmatrix} 0.5 & X \\ 0 & -1 \end{bmatrix}; \quad \begin{bmatrix} 2 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 0.5 & X \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$Solving\ for\ X\ we\ get\ X = 0.5$$

Lower Triangular Matrix:

$$B = \begin{bmatrix} 2 & 0 \\ 2 & 1 \end{bmatrix}; \quad B^{-1} = \begin{bmatrix} 0.5 & 0 \\ X & 1 \end{bmatrix}; \quad \begin{bmatrix} 2 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 0.5 & 0 \\ X & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$Solving\ for\ X\ we\ get\ X = -1$$

RELATION BETWEEN THE TIME COMPLEXITIES OF BASIC MATRIX OPERATIONS:

A matrix is a rectangular or square grid of numbers arranged into rows and columns. Each number in the matrix is called an element, and they are arranged in what is called an array. The plural of “matrix” is “matrices”. Matrices are often used in algebra to solve for unknown values in linear equations, and in geometry when solving for vectors and vector operations.

The four "**basic operations**" on numbers are addition, subtraction, multiplication, and division. For **matrices**, there are three **basic** row **operations**; that is, there are three procedures that you can do with the rows of a **matrix**.

Basic Arithmetic Operations				
Operation	**Input**	**Output**	**Algorithm**	**Complexity**
Addition	Two n-digit numbers N, N	One n+1-digit number	Basic addition with carry	Θ(n), Θ(log(N))
Subtraction	Two n-digit numbers N, N	One n+1-digit number	Basic subtraction with borrow	Θ(n), Θ(log(N))
Multiplication	Two n-digit numbers	One 2n-digit number	Basic long multiplication	O(n2)
			Karatsuba algorithm	O(n1.585)
			3-way Toom–Cook multiplication	O(n1.465)
			k-way Toom–Cook multiplication	O(nlog (2k − 1)/log k)
			Mixed-level Toom–Cook (Knuth 4.3.3-T)[2]	O(n 2√2 log n log n)

			Schönhage–Strassen algorithm	O(n log n log log n)
			Fürer's algorithm[3]	O(n log n 22 log* n)
			Harvey-Hoeven algorithm[4] [5]	O(n log n)
Division	Two n-digit numbers	One n-digit number	Basic long division	O(n2)
			Burnikel-Ziegler Divide-and-Conquer Division [6]	O(M(n) log n)
			Newton–Raphson division	O(M(n))
Square root	One n-digit number	One n-digit number	Newton's method	O(M(n))
Modular exponentiation	Two n-digit integers and a k-bit exponent	One n-digit integer	Repeated multiplication and reduction	O(M(n) 2k)
			Exponentiation by squaring	O(M(n) k)
			Exponentiation with Montgomery reduction	O(M(n) k)
Matrix Algebric Function				
Operation	**Input**	**Output**	**Algorithm**	**Complexity**
Matrix multiplication	Two $n\times n$ matrices	One $n\times n$ matrix	Basic matrix multiplication	$O(n^3)$
			Strassen algorithm	$O(n^{2.807})$
			Coppersmith–Winograd algorithm	$O(n^{2.376})$
			Optimized CW-like algorithms	$O(n^{2.373})$
Matrix multiplication	One $n\times m$ matrix & one $m\times p$ matrix	One $n\times p$ matrix	Schoolbook matrix multiplication	$O(nmp)$
Matrix multiplication	One $n\times\lceil n\rceil$ matrix & one $\lceil n\rceil\times n$ matrix	One $n\times n$ matrix	Algorithms	$O(n^{\omega(k)+\epsilon})$where upper bounds on ω(k)
Matrix inversion*	One $n\times n$ matrix	One $n\times n$ matrix	Gauss–Jordan elimination	$O(n^3)$

			Strassen algorithm	$O(n^{2.807})$
			Coppersmith–Winograd algorithm	$O(n^{2.376})$
			Optimized CW-like algorithms	$O(n^{2.373})$
Singular value decomposition	One $m\times n$ matrix	One $m\times m$ matrix, one $m\times n$ matrix, & one $n\times n$ matrix	Bidiagonalizationand QR algorithm	$O(mn^2 + m^2 n)$ $(m \geq n)$
		One $m\times n$ matrix, one $n\times n$ matrix, & one $n\times n$ matrix	Bidiagonalization and QR algorithm	$O(mn^2)$ $(m \geq n)$
Determinant	One $n\times n$ matrix	One number	Laplace expansion	$O(n!)$
			Division-free algorithm	$O(n^4)$
			LU decomposition	$O(n^3)$
			Bareiss algorithm	$O(n^3)$
			Fast matrix multiplication	$O(n^{2.373})$
Back substitution	Triangular matrix	n solutions	Back substitution	$O(n^2)$

LUP-DECOMPOSITION:

Definition:

A m×n matrix is said to have a **LU-decomposition** if there exists matrices L and U with the following Properties:

(i) L is a m×n lower triangular matrix with all diagonal entries being 1.

(ii) U is a m×n matrix in some echelon form.

(iii) $A = LU$.

Advantages of LU-Decomposition:

Suppose we want to solve a m×n system $AX = \boldsymbol{b}$.

If we can find a LU-decomposition for A , then to solve $AX = \boldsymbol{b}$, it is enough to solve the systems

$$\left.\begin{array}{l} LY = b \\ UX = Y \end{array}\right\}.$$

Thus the system LY = b can be solved by the method of forward substitution and the system ***UX*** = ***Y*** can be solved by the method of backward substitution. The following examples will illustrate this.

Example:

Consider the given system AX = b, where

$$A = \begin{bmatrix} 1 & 2 & 0 \\ 3 & 6 & -1 \\ 1 & 2 & 1 \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} 2 \\ 8 \\ 0 \end{bmatrix}.$$

It is easy to check that A = LU, where

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & -0 \\ 1 & -1 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & 2 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix}.$$

Thus, to solve AX = b, we first solve LY = b by forward substitution

$$\begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & -0 \\ 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 8 \\ 0 \end{bmatrix},$$

to get

$$\boldsymbol{Y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \\ 0 \end{bmatrix},$$

Now, we solve UX =Y by backward substitution:

$$\begin{bmatrix} 1 & 2 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \\ 0 \end{bmatrix}.$$

and get

$$\boldsymbol{X} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2(1-x_2) \\ x_2 \\ 2 \end{bmatrix}.$$

The above discussion raises the following question. Which matrices have LU-decomposition, and how to find them? The answer is given by the following:

Theorem:

Let A be a m×n matrix and $E_1, E_2, \ldots E_k$ be elementary matrices such that

U = $E_k \ldots E_1$ A

is in non-echelon form. If none of the E_i's corresponds to the operation of row interchange, then

C = $E_k \dots E_1$ is a lower triangular invertible matrix. Further L = C^{-1} is also a lower triangular matrix with A = LU .

Example:

Let

$$A = \begin{bmatrix} 2 & -1 & 3 \\ 4 & 2 & 1 \\ -6 & -1 & 2 \end{bmatrix}.$$

Then the row operation -2 $R_1 + R_2$, 3 $R_1 + R_3$, $R_2 + R_3$ give

$$A \sim \begin{bmatrix} 2 & -1 & 3 \\ 0 & 4 & -5 \\ 0 & -4 & 11 \end{bmatrix}$$

$$\sim \begin{bmatrix} 2 & -1 & 3 \\ 0 & 4 & -5 \\ 0 & 0 & 6 \end{bmatrix}$$

The corresponding elementary matrices and their inverses are

$$E_1 = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad E_1^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$E_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad E_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$E_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}, \quad E_3^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix}.$$

Thus

$$L = E_1^{-1}E_2^{-1}E_3^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -3 & -1 & 1 \end{bmatrix}.$$

Let us observe that for

$$A=\begin{bmatrix}2&-1&3\\4&2&1\\-6&-1&2\end{bmatrix}.$$

$$L=\begin{bmatrix}1&0&0\\\boxed{2}&1&0\\\boxed{-3}&\boxed{-1}&1\end{bmatrix}\text{ and }U=\begin{bmatrix}2&-1&3\\0&4&-5\\0&0&6\end{bmatrix}$$

Note that the element below the main diagonal in L is the pivotal columns are the negatives of the scalars used in the respective column to make all the entries in that column below the pivot zero. For example, in first column, we operated -2R_1+R_2 and 3R_1+R_3. Thus, in the first column, the second entry is 2 and the third entry is -3.

Example:
Let

$$A=\begin{bmatrix}0&\boxed{2}&-6&-2&4\\0&\boxed{-1}&3&3&2\\0&\boxed{-1}&3&7&10\end{bmatrix}$$

To reduce A to a row -echelon form, we operate ½R_1+R_2 and ½R_1+R_3 for the first pivotal columns, namely the second column to get

$$A\sim\begin{bmatrix}0&2&-6&-2&4\\0&0&0&\boxed{2}&4\\0&0&0&\boxed{6}&12\end{bmatrix}.$$

Next we operate -3R_2+R_3 for the next pivotal column, the 4^{th} column to get

$$A\sim\begin{bmatrix}0&2&-6&-2&4\\0&0&0&2&4\\0&0&0&0&0\end{bmatrix}:=U$$

This has only two non-zero rows. Thus L is given by replacing entries below diagonal pivotal columns by -ve of the multiples: in column 1 these are -½ , -½ ,and in second column this is 3. Thus

$$L_{3\times3}=\begin{bmatrix}1&0&0\\-\frac{1}{2}&1&0\\-\frac{1}{2}&3&1\end{bmatrix}$$

Let us check

$$LU = \begin{bmatrix} 1 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 \\ -\frac{1}{2} & 3 & 1 \end{bmatrix} \begin{bmatrix} 0 & 2 & -6 & -2 & 4 \\ 0 & 0 & 0 & 2 & 4 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 2 & -6 & -2 & 4 \\ 0 & -1 & 3 & 3 & 2 \\ 0 & -1 & 3 & 7 & 10 \end{bmatrix}.$$

The method followed in examples can in fact be proved mathematically as follows:

Theorem:

Let E_{ij} be the m $\times$ m elementary matrix which corresponds to the elementary row operation of multiplying the ith row by the scalar $-\alpha_{ij}$ and adding it to the jth row. Then

(i)
$$\begin{array}{cc} \begin{array}{c} \text{j}^{\text{th}} column \\ \downarrow \end{array} & \\ E_{ij} = \begin{bmatrix} 1 & 0 \cdots\cdots 0 \cdots & \cdots & 0 \\ 0 & 1 \cdots\cdots 0 \cdots & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 \cdots 0 & -\alpha_{ij} & 0 \cdots & 0 \\ \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ 0 & 0 \cdots & \cdots \cdots\cdots & 1 \end{bmatrix} & \begin{array}{c} \\ \\ \\ \leftarrow - - i^{th} row \\ \\ \\ \\ \end{array} \end{array}$$

(ii)
$$E_{ij}^{-1} = \begin{bmatrix} 1 & 0 \cdots\cdots 0 \cdots & \cdots & 0 \\ 0 & 1 \cdots\cdots 0 \cdots & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 \cdots 0 & \alpha_{ij} & 0 \cdots & 0 \\ \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ 0 & 0 \cdots & \cdots \cdots\cdots & 1 \end{bmatrix} \begin{array}{c} \\ \\ \\ \leftarrow - - i^{th} row \\ \\ \\ \\ \end{array}$$

(iii) If $A_{m\times n}$ is such that

$U := E_{m,m-1}, \dots, E_{2,1}, A$ is in non-echelon form, then

$$L = E_{2,1}^{-1} \dots E_{m,m-1}^{-1} = \begin{bmatrix} 1 & 0 & \cdots\cdots & 0 \\ \alpha_{21} & 1 & \cdots\cdots & 0 \\ \vdots & & \ddots & 0 \\ \alpha_{m1} & \alpha_{m2} & \cdots & \alpha_{mn} \end{bmatrix}$$

Till now we had assumed that no row interchanges were required to transform A to upper triangular form. If such row operations are needed we can proceed by the following:

Theorem

Let A be an m × n matrix which requires row interchanges $P_1, P_2, \dots, P_r$, in addition to the other elementary row operations to transform it to row echelon form U : Then , the matrix PA requires no row interchanges to reduce it to row echelon form and hence can be written as PA = LU .

The system AX = b is equivalent to a system PAX = Pb , Where PA has LU-decomposition .

Definition

A m×m matrix which arises from I_m by finite number of row interchanges is called a permutation matrix.

Example:

$$A = \begin{bmatrix} 1 & 4 & 2 & 3 \\ 1 & 2 & 1 & 0 \\ 2 & 6 & 3 & 1 \\ 0 & 0 & 1 & 4 \end{bmatrix}$$

$$\underset{\sim}{-R_1 + R_2, -2R_2 + R_3} \begin{bmatrix} 1 & 4 & 2 & 3 \\ 0 & -2 & -1 & -3 \\ 0 & -2 & -1 & -5 \\ 0 & 0 & 1 & 4 \end{bmatrix}$$

$$\underset{\sim}{-R_2 + R_3} \begin{bmatrix} 1 & 4 & 2 & 3 \\ 0 & -2 & -1 & -3 \\ 0 & 0 & 0 & -2 \\ 0 & 0 & 1 & 4 \end{bmatrix}$$

To continue we need to interchange $R_3 \leftrightarrow R_4$ to get

$$A \sim \begin{bmatrix} 1 & 4 & 2 & 3 \\ 0 & -2 & -1 & -3 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & -2 \end{bmatrix} := U$$

and now U is in r.e.f..

Thus, we consider PA and transform it to r.e.f

$$\left[I_4 \mid PA \right] = \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 1 & 4 & 2 & 3 \\ 0 & 1 & 0 & 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 & 2 & 6 & 3 & 1 \end{array}\right]$$

$$\sim \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 1 & 4 & 2 & 3 \\ -1 & 1 & 0 & 0 & 0 & -2 & -1 & -3 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 4 \\ -2 & 0 & 0 & 1 & 0 & -2 & -1 & -5 \end{array}\right]$$

$$\sim \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 1 & 4 & 2 & 3 \\ -1 & 1 & 0 & 0 & 0 & -2 & -1 & -3 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 4 \\ -1 & -1 & 0 & 1 & 0 & 0 & 0 & -2 \end{array}\right]$$

$$= \left[L^{-1} \mid U \right].$$

Hence,

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

UNIT-4

SHORTEST PATH IN GRAPHS:

The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the edges weights is minimum.

Shortest paths:

An edge-weighted digraph is a digraph where we associate weights or costs with each edge. A *shortest path* from vertex s to vertex t is a directed path from s to t with the property that no other such path has a lower weight.

Properties:

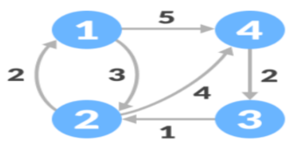
- **Paths are directed**. A shortest path must respect the direction of its edges.
- **The weights are not necessarily distances.** Geometric intuition can be helpful, but the edge weights weights might represent time or cost.
- **Not all vertices need be reachable.** If t is not reachable from s, there is no path at all, and therefore there is no shortest path from s to t.
- **Negative weights introduce complications.** For the moment, we assume that edge weights are positive (or zero).
- **Shortest paths are normally simple.** Our algorithms ignore zero-weight edges that form cycles, so that the shortest paths they find have no cycles.
- **Shortest paths are not necessarily unique.** There may be multiple paths of the lowest weight from one vertex to another; we are content to find any one of them.
- **Parallel edges and self-loops may be present.** In the text, we assume that parallel edges are not present and use the notation v->w to refer to the edge from v to w, but our code handles them without difficulty.

FLOYD WARSHALL ALGORITHM:

- Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).
- A weighted graph is a graph in which each edge has a numerical value associated with it.
- Floyd-Warhshall algorithm is also called as Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm or WFI algorithm.
- This algorithm follows the dynamic programming approach to find the shortest paths.

Working of Floyd warshall algorithm:

Let the given graph be:



Follow the steps below to find the shortest path between all the pairs of vertices.

1. Create a matrix A^1 of dimension n*n where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph. Each cell A[i][j] is filled with the distance from the ith vertex to the jth vertex. If there is no path from ith vertex to jth vertex, the cell is left as infinity.

$$A^0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & \infty & 4 \\ 3 & \infty & 1 & 0 & \infty \\ 4 & \infty & \infty & 2 & 0 \end{array}$$

2. Now, create a matrix A^1 using matrix A^0. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way. Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex.A[i][j] is filled with (A[i][k] + A[k][j]) if (A[i][j] > A[i][k]+A[k][j]). That is, if the direct distance from the source to the destination is greater than the path through the vertex k, then the cell is filled with A[i][k] + A[k][j]. In this step, k is vertex 1. We calculate the distance from source vertex to destination vertex throughthisvertexk.

$$A^1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & & \\ 3 & \infty & & 0 & \\ 4 & \infty & & & 0 \end{array} \longrightarrow \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & \infty & 1 & 0 & 8 \\ 4 & \infty & \infty & 2 & 0 \end{array}$$

For example: For A^1[2, 4], the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (ie. from vertex 2 to 1 and from vertex 1 to 4) is 7. Since 4 < 7, A^0[2, 4] is filled with 4.

3. In a similar way, A^2 is created using A^3. The elements in the second column and the second row are left as they are.In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in **step 2**.

$$A^2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & & \\ 2 & 2 & 0 & 9 & 4 \\ 3 & & 1 & 0 & \\ 4 & & \infty & & 0 \end{array} \longrightarrow \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 9 & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & \infty & \infty & 2 & 0 \end{array}$$

4. Similarly, A^3 and A^4 is also created.

$$A^3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & & \infty & \\ 2 & & 0 & 9 & \\ 3 & \infty & 1 & 0 & 8 \\ 4 & & & 2 & 0 \end{array} \rightarrow \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 9 & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array} \qquad A^4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & & & 5 \\ 2 & & 0 & & 4 \\ 3 & & & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array} \rightarrow \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 7 & 5 \\ 2 & 2 & 0 & 6 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array}$$

A^4 gives the shortest path between each pair of vertices.

Floyd-Warshall Algorithm

```
n = no of vertices
A = matrix of dimension n*n
for k = 1 to n
for i = 1 to n
for j = 1 to n
$A^k[i, j] = \min (A^{k-1}[i, j], A^{k-1}[i, k] + A^{k-1}[k, j])$
return A
```

Floyd Warshall Algorithm Complexity

Time Complexity

There are three loops. Each loop has constant complexities. So, the time complexity of the Floyd-Warshall algorithm is $O(n^3)$.

Space Complexity

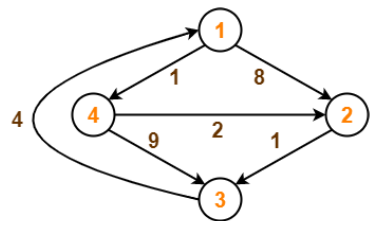
The space complexity of the Floyd-Warshall algorithm is $O(n^2)$.

Floyd Warshall Algorithm Applications

- To find the shortest path is a directed graph
- To find the transitive closure of directed graphs
- To find the Inversion of real matrices
- For testing whether an undirected graph is bipartite

Example Problem:

Consider the following directed weighted graph



Using Floyd Warshall Algorithm, find the shortest path distance between every pair of vertices.

<u>Solution:</u>

Step-01:

- Remove all the self loops and parallel edges (keeping the lowest weight edge) from the graph.
- In the given graph, there are neither self edges nor parallel edges.

Step-02:

- Write the initial distance matrix.
- It represents the distance between every pair of vertices in the form of given weights.
- For diagonal elements (representing self-loops), distance value = 0.
- For vertices having a direct edge between them, distance value = weight of that edge.
- For vertices having no direct edge between them, distance value = ∞.

Initial distance matrix for the given graph is-

$$D_0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & \infty & 0 & \infty \\ 4 & \infty & 2 & 9 & 0 \end{array}$$

Step-03:

Using Floyd Warshall Algorithm, write the following 4 matrices-

$$D_1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 9 & 0 \end{array} \qquad D_3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & 9 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 12 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{array}$$

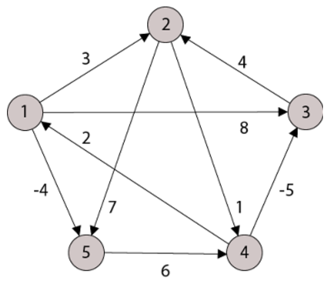
$$D_2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & 9 & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 3 & 0 \end{array} \qquad D_4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 4 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 7 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{array}$$

- The last matrix D_4 represents the shortest path distance between every pair of vertices.
- In the above problem, there are 4 vertices in the given graph.

- So, there will be total 4 matrices of order 4 x 4 in the solution excluding the initial distance matrix.
- Diagonal elements of each matrix will always be 0.

Example: Floyd-Warshall algorithm

This algorithm is used for constructing the shortest path. Show that matrices $D^{(k)}$ and $\pi^{(k)}$ computed by the Floyd-Warshall algorithm for the graph.



Solution:

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

$$\pi_{ij}^{(k)} = \begin{Bmatrix} \pi_{ij}^{(k-1)} \text{ if } d_{ij}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} \text{ if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{Bmatrix}$$

Step (i) When k = 0

$$D^{(0)} = \begin{matrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & -5 & \infty \\ 2 & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{matrix} \qquad \pi^{(0)} = \begin{matrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 3 & \text{NIL} \\ 4 & \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{matrix}$$

Step (ii) When k =1

$$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

$$d_{14}^{(1)} = \min(d_{14}^{(0)}, d_{11}^{(0)} + d_{14}^{(0)})$$

$$d_{14}^{(1)} = \min(\infty, 0 + \infty) = \infty$$

$$d_{15}^{(1)} = \min(d_{15}^{(0)}, d_{11}^{(0)} + d_{15}^{(0)})$$

$$d_{15}^{(1)} = \min(-4, 0 + -4) = -4$$

$$d_{21}^{(1)} = \min(d_{21}^{(0)}, d_{21}^{(0)} + d_{11}^{(0)})$$

$$d_{21}^{(1)} = \min(\infty, \infty + 0) = \infty$$

$$d_{23}^{(1)} = \min(d_{23}^{(0)}, d_{21}^{(0)} + d_{13}^{(0)})$$

$$d_{23}^{(1)} = \min((\infty, \infty + 8) = \infty$$

$$d_{31}^{(1)} = \min(d_{31}^{(0)}, d_{31}^{(0)} + d_{11}^{(0)})$$

$$d_{31}^{(1)} = \min(\infty, \infty + 0) = \infty$$

$$d_{35}^{(1)} = \min(d_{35}^{(0)}, d_{31}^{(0)} + d_{15}^{(0)})$$

$$d_{35}^{(1)} = \min(\infty, \infty + (-4)) = \infty$$

$$d_{42}^{(1)} = \min(d_{42}^{(0)}, d_{41}^{(0)} + d_{12}^{(0)})$$

$$d_{42}^{(1)} = \min(\infty, 2 + 3) = 5$$

$$d_{43}^{(1)} = \min(d_{43}^{(0)}, d_{41}^{(0)} + d_{13}^{(0)})$$

$$d_{43}^{(1)} = \min(\infty, 2 + 8) = 10$$

$$d_{45}^{(1)} = \min(d_{45}^{(0)}, d_{41}^{(0)} + d_{15}^{(0)})$$

$$d_{45}^{(1)} = \min(\infty, 2 + (-4)) = -2$$

$$d_{51}^{(1)} = \min(d_{51}^{(0)}, d_{51}^{(0)} + d_{11}^{(0)})$$

$$d_{51}^{(1)} = \min(\infty, \infty + 0) = \infty$$

$$D_{ij}^{(1)} = \begin{matrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & -5 & \infty \\ 2 & 5 & 10 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix} \qquad \pi^{(1)} = \begin{matrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 3 & NIL \\ 4 & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{matrix}$$

Step (iii) When k = 2

$$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

$$d_{14}^{(2)} = \min(d_{14}^{(1)}, d_{12}^{(1)} + d_{24}^{(1)})$$

$$d_{14}^{(2)} = \min(\infty, 3 + 1) = 4$$

$$d_{21}^{(2)} = \min(d_{21}^{(1)}, d_{22}^{(1)} + d_{21}^{(1)})$$

$$d_{21}^{(2)} = \min(\infty, 0 + \infty) = \infty$$

$$d_{34}^{(2)} = \min(d_{34}^{(1)}, d_{32}^{(1)} + d_{24}^{(1)})$$

$$d_{34}^{(2)} = \min(-5, 4 + 1) = -5$$

$$d_{35}^{(2)} = \min(d_{35}^{(1)}, d_{32}^{(1)} + d_{25}^{(1)})$$

$$d_{35}^{(2)} = \min(\infty, 4 + 7) = 11$$

$$d_{43}^{(2)} = \min(d_{43}^{(1)}, d_{42}^{(1)} + d_{23}^{(1)})$$

$$d_{43}^{(2)} = \min(10, 5 + \infty) = 10$$

$$D_{ij}^{(2)} = \begin{matrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & -5 & 11 \\ 2 & 5 & 10 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix} \qquad \pi^{(2)} = \begin{matrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 3 & 2 \\ 4 & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{matrix}$$

Step (iv) When k = 3

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

$$d_{14}^{(3)} = \min\left(d_{14}^{(2)}, d_{13}^{(2)} + d_{34}^{(2)}\right)$$

$$d_{14}^{(3)} = \min(4,\ 8 + (-5)) = 3$$

$$D_{ij}^{(3)} = \begin{array}{ccccc} 0 & 3 & 8 & 3 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & -5 & 11 \\ 2 & 5 & 10 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{array} \qquad \pi^{(3)} = \begin{array}{ccccc} \text{NIL} & 1 & 1 & 3 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 3 & 2 \\ 4 & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{array}$$

Step (v) When k = 4

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

$$d_{21}^{(4)} = \min\left(d_{21}^{(3)}, d_{24}^{(3)} + d_{41}^{(3)}\right)$$

$$d_{21}^{(4)} = \min(\infty,\ 1 + 2) = 3$$

$$d_{23}^{(4)} = \min\left(d_{23}^{(3)}, d_{24}^{(3)} + d_{43}^{(3)}\right)$$

$$d_{23}^{(4)} = \min(\infty,\ 1 + 10) = 11$$

$$d_{25}^{(4)} = \min\left(d_{25}^{(3)}, d_{24}^{(3)} + d_{45}^{(3)}\right)$$

$$d_{25}^{(4)} = \min(7,\ 1 + (-2)) = -1$$

$$d_{31}^{(4)} = \min\left(d_{31}^{(3)}, d_{34}^{(3)} + d_{41}^{(3)}\right)$$

$$d_{31}^{(4)} = \min(\infty,\ -5 + 2) = -3$$

$$d_{32}^{(4)} = \min\left(d_{32}^{(3)}, d_{34}^{(3)} + d_{42}^{(3)}\right)$$

$$d_{32}^{(4)} = \min(4,\ -5 + 5) = 0$$

$$d_{51}^{(4)} = \min\left(d_{51}^{(3)}, d_{54}^{(3)} + d_{41}^{(3)}\right)$$

$$d_{51}^{(4)} = \min(\infty,\ 6 + 2) = 8$$

$$d_{52}^{(4)} = \min\left(d_{52}^{(3)}, d_{54}^{(3)} + d_{42}^{(3)}\right)$$

$$d_{52}^{(4)} = \min(\infty,\ 6 + 5) = 11$$

$$d_{53}^{(4)} = \min\left(d_{53}^{(3)}, d_{54}^{(3)} + d_{43}^{(3)}\right)$$

$$d_{53}^{(4)} = \min(\infty,\ 6 + 10) = 16$$

$$D_{ij}^{(4)} = \begin{array}{ccccc} 0 & 3 & 8 & 3 & -4 \\ 3 & 0 & 11 & 1 & -1 \\ -3 & 0 & 0 & -5 & -7 \\ 2 & 5 & 10 & 0 & -2 \\ 8 & 11 & 16 & 6 & 0 \end{array} \qquad \pi^{(4)} = \begin{array}{ccccc} \text{NIL} & 1 & 1 & 3 & 1 \\ 4 & \text{NIL} & 4 & 2 & 2 \\ 4 & 4 & \text{NIL} & 3 & 4 \\ 4 & 1 & 1 & \text{NIL} & 1 \\ 4 & 4 & 4 & 5 & \text{NIL} \end{array}$$

Step (vi) When k = 5

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

$$d_{25}^{(5)} = \min\left(d_{25}^{(4)}, d_{25}^{(4)} + d_{55}^{(3)}\right)$$

$$d_{25}^{(5)} = \min(-1, -1 + 0) = -1$$

$$d_{23}^{(5)} = \min\left(d_{23}^{(4)}, d_{25}^{(4)} + d_{53}^{(3)}\right)$$

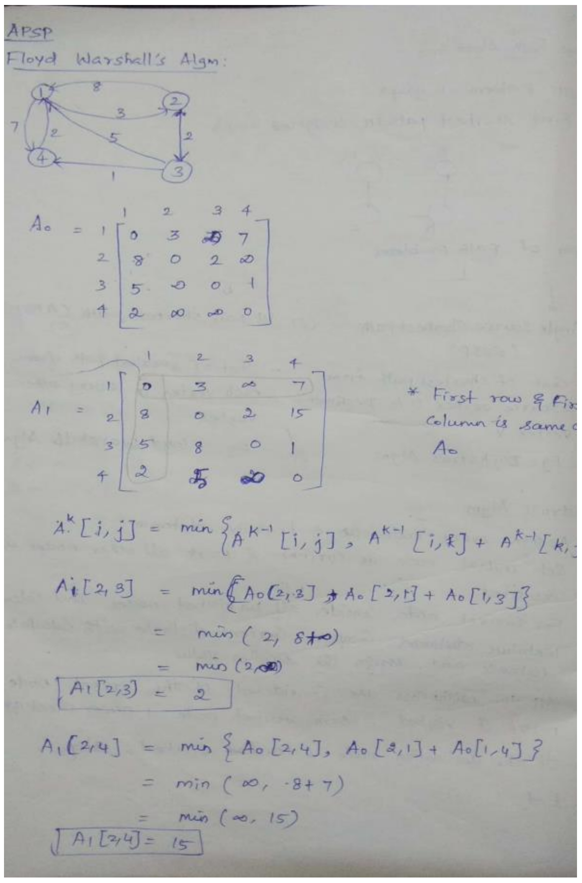
$$d_{23}^{(5)} = \min(11, -1 + 16) = 11$$

$$d_{35}^{(5)} = \min\left(d_{35}^{(4)}, d_{35}^{(4)} + d_{55}^{(3)}\right)$$

$$d_{35}^{(5)} = \min(-7, -7 + 0) = -7$$

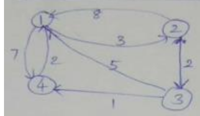
$$D_{ij}^{(5)} = \begin{matrix} 0 & 3 & 8 & 3 & -4 \\ 3 & 0 & 11 & 1 & -1 \\ -3 & 0 & 0 & -5 & -7 \\ 2 & 5 & 10 & 0 & -2 \\ 8 & 11 & 16 & 6 & 0 \end{matrix} \qquad \pi^{(5)} = \begin{matrix} NIL & 1 & 1 & 5 & 1 \\ 4 & NIL & 4 & 2 & 4 \\ 4 & 4 & NIL & 3 & 4 \\ 4 & 1 & 1 & NIL & 1 \\ 4 & 4 & 4 & 5 & NIL \end{matrix}$$

Example 2:



APSP

Floyd Warshall's Algm:



$$A_0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 7 \\ 2 & 8 & 0 & 2 & \infty \\ 3 & 5 & \infty & 0 & 1 \\ 4 & 2 & \infty & \infty & 0 \end{array}$$

$$A_1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 7 \\ 2 & 8 & 0 & 2 & 15 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & \infty & 0 \end{array}$$

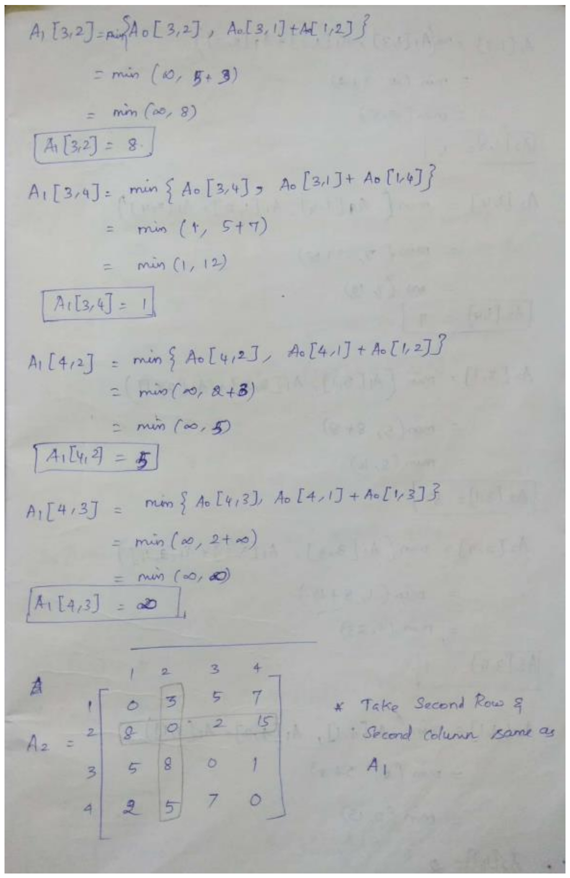
* First row & First Column is same as A_0

$$A^k[i,j] = \min\left\{A^{k-1}[i,j], A^{k-1}[i,k] + A^{k-1}[k,j]\right\}$$

$$A_1[2,3] = \min\{A_0[2,3], A_0[2,1] + A_0[1,3]\}$$
$$= \min(2, 8+\infty)$$
$$= \min(2, \infty)$$
$$\boxed{A_1[2,3] = 2}$$

$$A_1[2,4] = \min\{A_0[2,4], A_0[2,1] + A_0[1,4]\}$$
$$= \min(\infty, 8+7)$$
$$= \min(\infty, 15)$$
$$\boxed{A_1[2,4] = 15}$$

Page 1



$$A_1[3,2] = \min\{A_0[3,2], A_0[3,1] + A_0[1,2]\}$$
$$= \min(\infty, 5+3)$$
$$= \min(\infty, 8)$$
$$\boxed{A_1[3,2] = 8}$$

$$A_1[3,4] = \min\{A_0[3,4], A_0[3,1] + A_0[1,4]\}$$
$$= \min(1, 5+7)$$
$$= \min(1, 12)$$
$$\boxed{A_1[3,4] = 1}$$

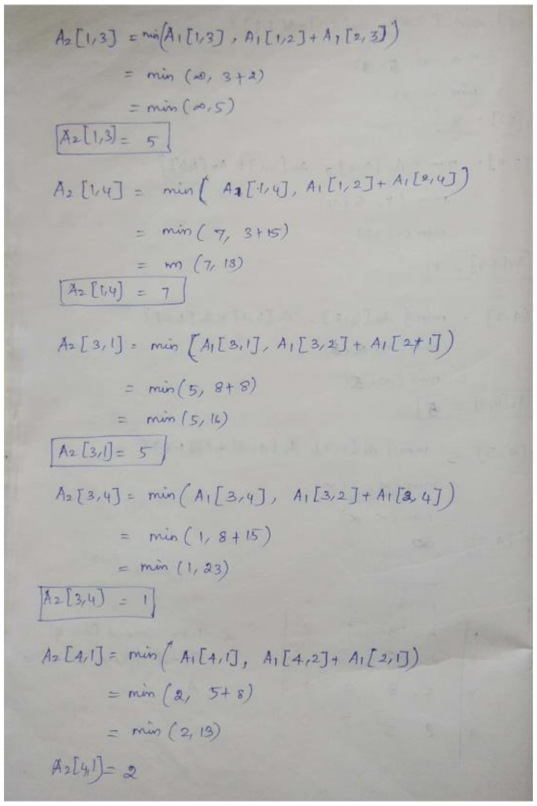
$$A_1[4,2] = \min\{A_0[4,2], A_0[4,1] + A_0[1,2]\}$$
$$= \min(\infty, 2+3)$$
$$= \min(\infty, 5)$$
$$\boxed{A_1[4,2] = 5}$$

$$A_1[4,3] = \min\{A_0[4,3], A_0[4,1] + A_0[1,3]\}$$
$$= \min(\infty, 2+\infty)$$
$$= \min(\infty, \infty)$$
$$\boxed{A_1[4,3] = \infty}$$

$$A_2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 5 & 7 \\ 2 & 8 & 0 & 2 & 15 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{array}$$

* Take Second Row & Second Column same as A_1

Page 2



$A_2[1,3] = \min(A_1[1,3], A_1[1,2] + A_1[2,3])$
$= \min(\infty, 3+2)$
$= \min(\infty, 5)$
$\boxed{A_2[1,3] = 5}$

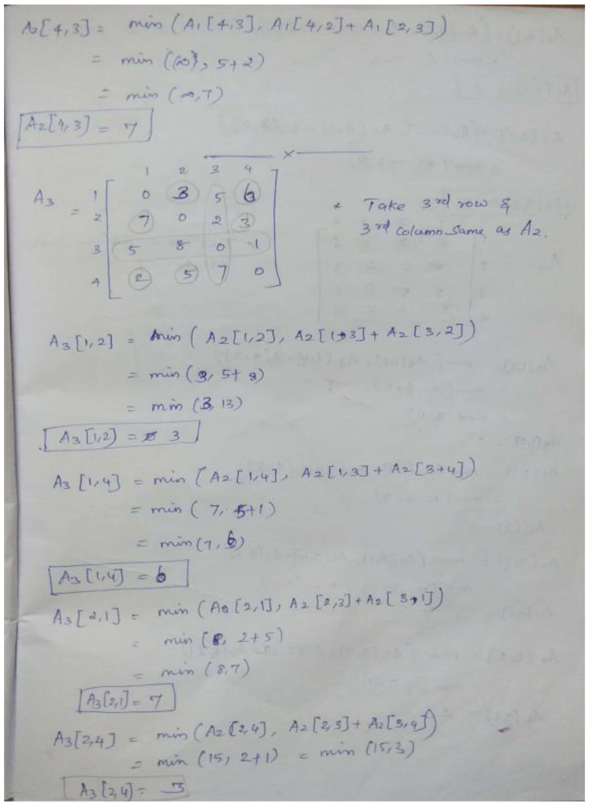
$A_2[1,4] = \min(A_1[1,4], A_1[1,2] + A_1[2,4])$
$= \min(7, 3+15)$
$= \min(7, 18)$
$\boxed{A_2[1,4] = 7}$

$A_2[3,1] = \min(A_1[3,1], A_1[3,2] + A_1[2,1])$
$= \min(5, 8+8)$
$= \min(5, 16)$
$\boxed{A_2[3,1] = 5}$

$A_2[3,4] = \min(A_1[3,4], A_1[3,2] + A_1[2,4])$
$= \min(1, 8+15)$
$= \min(1, 23)$
$\boxed{A_2[3,4] = 1}$

$A_2[4,1] = \min(A_1[4,1], A_1[4,2] + A_1[2,1])$
$= \min(2, 5+8)$
$= \min(2, 13)$
$A_2[4,1] = 2$

Page 3



$A_2[4,3] = \min(A_1[4,3], A_1[4,2] + A_1[2,3])$
$= \min(\infty, 5+2)$
$= \min(\infty, 7)$
$\boxed{A_2[4,3] = 7}$

$$A_3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 5 & 6 \\ 2 & 7 & 0 & 2 & 3 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{array}$$

* Take 3rd row & 3rd Column Same as A_2.

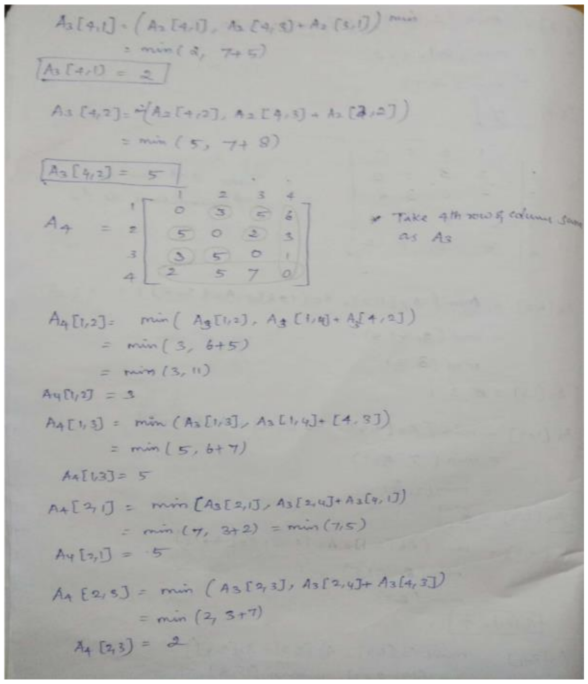
$A_3[1,2] = \min(A_2[1,2], A_2[1,3] + A_2[3,2])$
$= \min(3, 5+8)$
$= \min(3, 13)$
$\boxed{A_3[1,2] = 3}$

$A_3[1,4] = \min(A_2[1,4], A_2[1,3] + A_2[3,4])$
$= \min(7, 5+1)$
$= \min(7, 6)$
$\boxed{A_3[1,4] = 6}$

$A_3[2,1] = \min(A_2[2,1], A_2[2,3] + A_2[3,1])$
$= \min(8, 2+5)$
$= \min(8, 7)$
$\boxed{A_3[2,1] = 7}$

$A_3[2,4] = \min(A_2[2,4], A_2[2,3] + A_2[3,4])$
$= \min(15, 2+1) = \min(15, 3)$
$\boxed{A_3[2,4] = 3}$

Page 4



$A_3[4,1] = \min(A_2[4,1], A_2[4,3] + A_2[3,1])$
$= \min(2, 7+5)$
$\boxed{A_3[4,1] = 2}$

$A_3[4,2] = \min(A_2[4,2], A_2[4,3] + A_2[3,2])$
$= \min(5, 7+8)$
$\boxed{A_3[4,2] = 5}$

$$A_4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 5 & 6 \\ 2 & 5 & 0 & 2 & 3 \\ 3 & 3 & 5 & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{array}$$

* Take 4th row & column Same as A_3

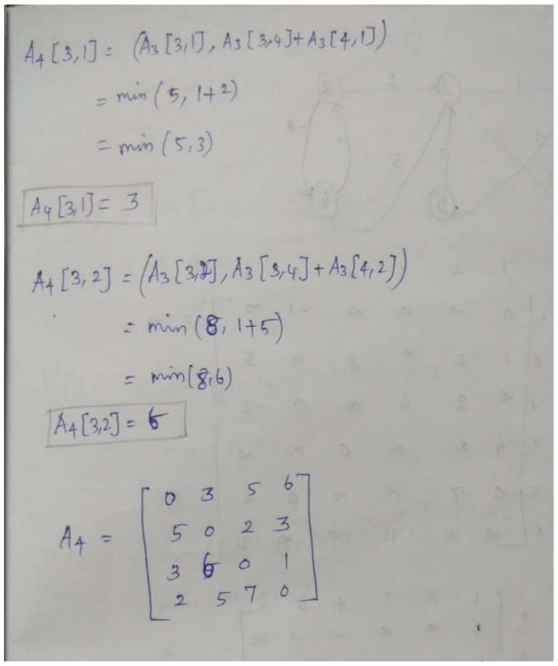
$A_4[1,2] = \min(A_3[1,2], A_3[1,4] + A_3[4,2])$
$= \min(3, 6+5)$
$= \min(3, 11)$
$A_4[1,2] = 3$

$A_4[1,3] = \min(A_3[1,3], A_3[1,4] + [4,3])$
$= \min(5, 6+7)$
$A_4[1,3] = 5$

$A_4[2,1] = \min(A_3[2,1], A_3[2,4] + A_3[4,1])$
$= \min(7, 3+2) = \min(7,5)$
$A_4[2,1] = 5$

$A_4[2,3] = \min(A_3[2,3], A_3[2,4] + A_3[4,3])$
$= \min(2, 3+7)$
$A_4[2,3] = 2$

Page 5



$A_4[3,1] = (A_3[3,1], A_3[3,4] + A_3[4,1])$
$= \min(5, 1+2)$
$= \min(5, 3)$
$\boxed{A_4[3,1] = 3}$

$A_4[3,2] = (A_3[3,2], A_3[3,4] + A_3[4,2])$
$= \min(8, 1+5)$
$= \min(8, 6)$
$\boxed{A_4[3,2] = 6}$

$$A_4 = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 3 & 6 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

Page 6

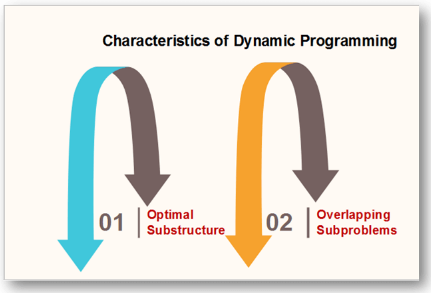
INTRODUCTION TO DYNAMIC PROGRAMMING PARADIGM:

Dynamic Programming is the most powerful design technique for solving optimization problems. Divide& Conquer algorithm partition the problem into disjoint sub-problems, solves the sub-problems recursively and then combine their solution to solve the original problems.

- It is used when the sub-problems are not independent, e.g. when they share the same sub-problems. In this case, divide and conquer may do more work than necessary, because it solves the same sub problem multiple times.
- It solves each sub-problem just once and stores the result in a table so that it can be repeatedly retrieved if needed again.
- It is a **Bottom-up approach.** We solve all possible small problems and then combine to obtain solutions for bigger problems.
- It is a paradigm of algorithm design in which an optimization problem is solved by a combination of achieving sub-problem solutions and appearing to the "**principle of optimality**".

Characteristics of Dynamic Programming:

Dynamic Programming works when a problem has the following features:-

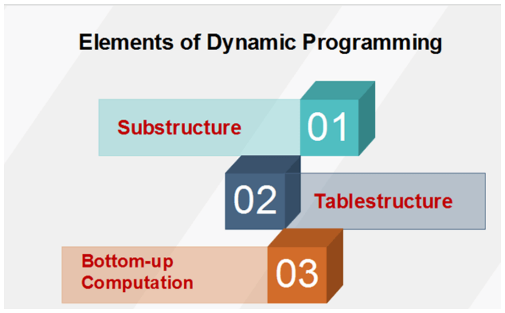


- ○ **Optimal Substructure:** If an optimal solution contains optimal sub solutions then a problem exhibits optimal substructure.
- ○ **Overlapping sub-problems:** When a recursive algorithm would visit the same sub-problems repeatedly, then a problem has overlapping sub-problems.

- ➢ If a problem has optimal substructure, then we can recursively define an optimal solution. If a problem has overlapping sub-problems, then we can improve on a recursive implementation by computing each sub-problem only once.
- ➢ If a problem doesn't have optimal substructure, there is no basis for defining a recursive algorithm to find the optimal solutions. If a problem doesn't have overlapping sub problems, we don't have anything to gain by using dynamic programming.
- ➢ If the space of sub-problems is enough (i.e. polynomial in the size of the input), dynamic programming can be much more efficient than recursion.

Elements of Dynamic Programming:

There are basically three elements that characterize a dynamic programming algorithm:-

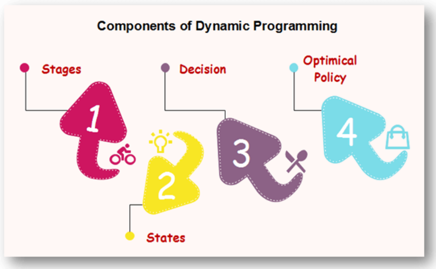


1. **Substructure:** Decompose the given problem into smaller sub-problems. Express the solution of the original problem in terms of the solution for smaller problems.
2. **Table Structure:** After solving the sub-problems, store the results to the sub problems in a table. This is done because sub-problem solutions are reused many times and we do not want to repeatedly solve the same problem over and over again.
3. **Bottom-up Computation:** Using table, combine the solution of smaller sub-problems to solve larger sub-problems and eventually arrives at a solution to complete problem.

Bottom-up means

i. Start with smallest sub-problems.
ii. Combining their solutions obtain the solution to sub-problems of increasing size.
iii. Until solving at the solution of the original problem.

Components of Dynamic programming:



1. **Stages:** The problem can be divided into several sub-problems, which are called stages. A stage is a small portion of a given problem. For example, in the shortest path problem, they were defined by the structure of the graph.
2. **States:** Each stage has several states associated with it. The states for the shortest path problem were the node reached.
3. **Decision:** At each stage, there can be multiple choices out of which one of the best decisions should be taken. The decision taken at every stage should be optimal; this is called a stage decision.

4. **Optimal policy:** It is a rule which determines the decision at each stage; a policy is called an optimal policy if it is globally optimal. This is known as Bellman principle of optimality.
5. Given the current state, the optimal choice for each of the remaining states does not depend on the previous states or decisions. In the shortest path problem, it was not necessary to know how we got a node only that we did.
6. There exist recursive relationships that identify the optimal decisions for stage j, given that stage j+1, has already been solved.
7. The final stage must be solved by itself.

Development of Dynamic Programming Algorithm:

It can be broken into four steps:

1. Characterize the structure of an optimal solution.
2. Recursively defined the value of the optimal solution. Like Divide and Conquer, divide the problem into two or more optimal parts recursively. This helps to determine what the solution will look like.
3. Compute the value of the optimal solution from the bottom up (starting with the smallest sub-problems)
4. Construct the optimal solution for the entire problem from the computed values of smaller sub-problems.

Applications of dynamic programming

1. 0/1 knapsack problem
2. Mathematical optimization problem
3. All pair Shortest path problem
4. Reliability design problem
5. Longest common subsequence (LCS)
6. Flight control and robotics control
7. Time-sharing: It schedules the job to maximize CPU usage
8. Matrix Chain Multiplication
9. Longest Common Subsequence
10. Travelling Salesman Problem

MORE EXAMPLES OF DYNAMIC PROGRAMMING:

Knapsack

Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack.

Hence, in case of 0-1 Knapsack, the value of $\boldsymbol{x_i}$ can be either ***0*** or ***1***, where other constraints remain the same.

0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.

Example 1

Let us consider that the capacity of the knapsack is W = 25 and the items are as shown in the following table.

Item	**A**	**B**	**C**	**D**
Profit	24	18	18	10
Weight	24	10	10	7

Without considering the profit per unit weight ($\boldsymbol{p_i/w_i}$), if we apply Greedy approach to solve this problem, first item ***A*** will be selected as it will contribute max$_i$mum profit among all the elements.

After selecting item ***A***, no more item will be selected. Hence, for this given set of items total profit is **24**. Whereas, the optimal solution can be achieved by selecting items **B** and C, where the total profit is 18 + 18 = 36.

Example 2

Instead of selecting the items based on the overall benefit, in this example the items are selected based on ratio p_i/w_i. Let us consider that the capacity of the knapsack is W = 60 and the items are as shown in the following table.

Item	**A**	**B**	**C**
Price	100	280	120
Weight	10	40	20
Ratio	10	7	6

Using the Greedy approach, first item ***A*** is selected. Then, the next item ***B*** is chosen. Hence, the total profit is **100 + 280 = 380**. However, the optimal solution of this instance can be achieved by selecting items, ***B*** and ***C***, where the total profit is **280 + 120 = 400**.

Hence, it can be concluded that Greedy approach may not give an optimal solution.

To solve 0-1 Knapsack, Dynamic Programming approach is required.

Problem Statement

A thief is robbing a store and can carry a maximal weight of ***W*** into his knapsack. There are ***n*** items and weight of **i**$^{\textbf{th}}$ item is w_i and the profit of selecting this item is p_i. What items should the thief take?

Dynamic-Programming Approach

Let ***i*** be the highest-numbered item in an optimal solution **S** for **W** dollars. Then $S' = S - \{i\}$ is an optimal solution for $W - w_i$ dollars and the value to the solution ***S*** is V_i plus the value of the sub-problem.

We can express this fact in the following formula: define **c[i, w]** to be the solution for items **1,2, … , i** and the max$_i$mum weight **w**.

The algorithm takes the following inputs

- The maximum weight **W**
- The number of items **n**
- The two sequences $\mathbf{v} = \langle v_1, v_2, \ldots, v_n \rangle$ and $\mathbf{w} = \langle w_1, w_2, \ldots, w_n \rangle$

Dynamic-0-1-knapsack (v, w, n, W)

```
for w = 0 to W do
c[0, w] = 0
for i = 1 to n do
c[i, 0] = 0
for w = 1 to W do
ifw_i ≤ w then
if v_i + c[i-1, w-w_i] then
c[i, w] = v_i + c[i-1, w-w_i]
else c[i, w] = c[i-1, w]
else
c[i, w] = c[i-1, w]
```

The set of items to take can be deduced from the table, starting at **c[n, w]** and tracing backwards where the optimal values came from.

If *c[i, w] = c[i-1, w]*, then item ***i*** is not part of the solution and we continue tracing with **c[i-1, w]**. Otherwise, item ***i*** is part of the solution and we continue tracing with **c[i-1, w-W]**.

Analysis

This algorithm takes $\theta(n, w)$ times as table c has $(n + 1).(w + 1)$ entries, where each entry requires $\theta(1)$ time to compute.

Longest Common Subsequence (LCS)

The longest common subsequence problem is finding the longest sequence which exists in both the given strings.

Subsequence

Let us consider a sequence S = <s_1, s_2, s_3, s_4, …,s_n>.

A sequence Z = <z_1, z_2, z_3, z_4, …,z_m> over S is called a subsequence of S, if and only if it can be derived from S deletion of some elements.

Common Subsequence

Suppose, ***X*** and ***Y*** are two sequences over a finite set of elements. We can say that ***Z*** is a common subsequence of ***X*** and ***Y***, if ***Z*** is a subsequence of both ***X*** and ***Y***.

Longest Common Subsequence

If a set of sequences are given, the longest common subsequence problem is to find a common subsequence of all the sequences that is of maximal length.

Naïve Method

Let ***X*** be a sequence of length ***m*** and ***Y*** a sequence of length ***n***. Check for every subsequence of ***X*** whether it is a subsequence of ***Y***and return the longest common subsequence found.

There are 2^m subsequences of ***X***. Testing sequences whether or not it is a subsequence of ***Y*** takes ***O(n)*** time. Thus, the naïve algorithm would take $O(n2^m)$ time.

Dynamic Programming

Let $X = < x_1, x_2, x_3, \ldots, x_m >$ and $Y = < y_1, y_2, y_3, \ldots, y_n >$ be the sequences. To compute the length of an element the following algorithm is used.In this procedure, table ***C[m, n]*** is computed in row major order and another table ***B[m,n]*** is computed to construct optimal solution.

Algorithm: LCS-Length-Table-Formulation (X, Y)

```
m := length(X) n := length(Y)
for i = 1 to m do
C[i, 0] := 0
for j = 1 to n do
C[0, j] := 0
for i = 1 to m do
for j = 1 to n do
if $x_i = y_j$
C[i, j] := C[i - 1, j - 1] + 1
B[i, j] := ‘D’
else
if C[i -1, j] ≥ C[i, j -1]
```

```
C[i, j] := C[i - 1, j] + 1
B[i, j] := 'U'
else
C[i, j] := C[i, j - 1]
B[i, j] := 'L'
return C and B
```

Algorithm: Print-LCS (B, X, i, j)

```
if i = 0 and j = 0
return
if B[i, j] = 'D'
  Print-LCS(B, X, i-1, j-1)
Print(x_i)
else if B[i, j] = 'U'
  Print-LCS(B, X, i-1, j)
else
  Print-LCS(B, X, i, j-1)
```

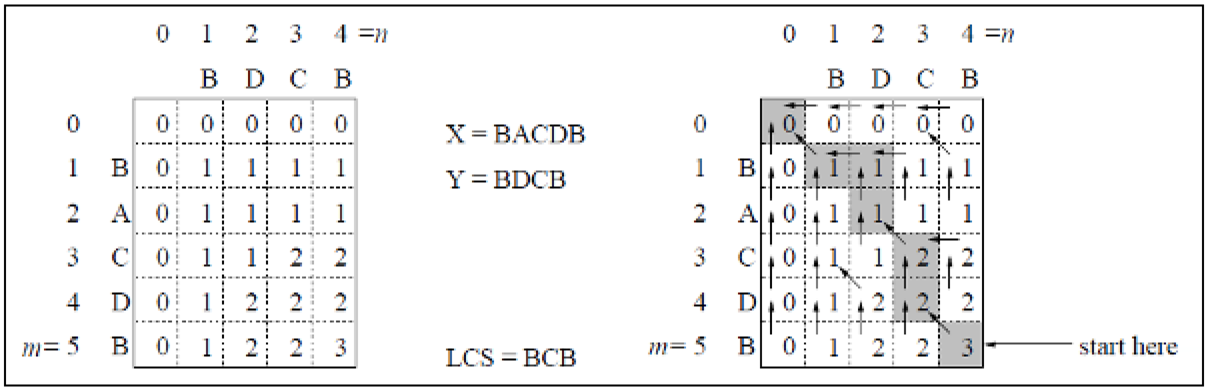
This algorithm will print the longest common subsequence of **X** and **Y**.

Analysis

To populate the table, the outer **for** loop iterates ***m*** times and the inner **for** loop iterates ***n*** times. Hence, the complexity of the algorithm is *O(m, n)*, where ***m*** and ***n*** are the length of two strings.

Example

In this example, we have two strings ***X = BACDB*** and ***Y = BDCB*** to find the longest common subsequence. Following the algorithm LCS-Length-Table-Formulation (as stated above), we have calculated table C (shown on the left hand side) and table B (shown on the right hand side).In table B, instead of 'D', 'L' and 'U', we are using the diagonal arrow, left arrow and up arrow, respectively. After generating table B, the LCS is determined by function LCS-Print. The result is BCB.



		0	1	2	3	4 =*n*
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
m= 5	B	0	1	2	2	3

X = BACDB

Y = BDCB

LCS = BCB

MODULO REPRESENTATION OF POLYNOMIALS/INTEGERS:

Modulo Representation of Integers:

- The Modulo representation is one that is very different from the place value system.
- Consider any three numbers which are relatively prime to each other. Relatively prime numbers are those numbers which have no common factors. The numbers need not necessarily be prime. Examples of relatively prime numbers are –

  1)2,3,5
  2)5,7,9
  3) 9, 16, 25
- Note here that 9,16 and 25 are not prime, but they have no common factors and hence, are relatively prime to each other.
- Coming back to modulo representations, let us try to understand it with an example.
- Consider the 3 numbers 2, 3 and 5.

  The LCM of these numbers is $2 \times 3 \times 5 = 30$.

  So, any number from 0 to 29 can be represented in terms of its remainder with respect to each of 2, 3 and 5.
- Take the number 23. The **remainder** when

  i) 23 is divided by 2 is **1**.

  ii) 23 is divided by 3 is **2**.

  iii) 23 is divided by 5 is **3**.

  Hence, **23** can be represented as **(1, 2, 3)**.
- This is called a **triplet** as there are 3 numbers to represent another number. Each number from 0 to 29 will have a unique triplet to represent it. Numbers greater than 29 will have repeating representations, identical to those between 0 and 29.
  For example, the representation for 31 is (1, 1, 1) which is the same as that for 1.
- Let us look at another number - say 3.
  i) 3 leaves a remainder of **1** when divided by 2.
  ii) 3 leaves a remainder of **0** when divided by 3.
  iii) 3 leaves a remainder of **3** when divided by 5.
  The representation for **3** in this system is **(1, 0, 3)**.

Addition:

Suppose to find the sum of 23 & 3. Just adding the triplets for 23 & 3 will give the triplet for the sum. This is a little tricky, though. it has to be done !**23 + 3= (1, 2, 3) + (1, 0, 3) = (2, 2, 6)**

- just add the corresponding remainders when divided by 2,3 and 5.

In this triplet, the 1st number represents the remainder when divided by 2, the second the remainder when divided by 3 and the third, the remainder when divided by5. However, the remainder when divided by 2 can never be greater than **1**, that when divided by 3 can never be greater than **2** & that when divided by 5 can never be greater than **4**. So, the solution we have obtained above has to be reduced further. For that, take each number in the representation of the sum. If it is greater than the allowed limit, take the modulo of that number again. So, for (2, 2, 6),

i. 2 > 1, which is the maximum possible remainder when divided by 2. Hence $[2]_2$=0.
ii. 2 = 2, which is the maximum possible remainder when divided by 3.So,let's leave it as it is!
iii. 6 > 4, which is the maximum possible remainder when divided by 5. Hence $[6]_5$= 1.Hence the representation for 23 + 3 =26 is **(0, 2, 1)**.

Subtraction:

If we wish to subtract 3 from 23,23 -3 = (1, 2, 3) - (1, 0, 3) = (0, 2, 0)
- just subtract the respective remainders. Thus, **20 = (0,2,0)**

Now, let us see what to do if we get a negative number in the remainder.
Take the example 17 - 9.
17 = (1, 2, 2)
9 = (1, 0, 4)
17 - 9 = (1, 2, 2) - (1, 0, 4) = (0, 2, -2)
Now, we have -2 as the remainder when divided by 5. As in addition, the modulo has to be found for -2. But, how do you find the modulo of a negative number ? Well, it's very simple !
Add the base to the remainder. In this case, the base is 5 and the remainder is -2. Adding, we get 3. Now, take the modulo of 3 with respect to the base 5 i.e.$[3]_5$. The result is 3.
Hence 17 - 9 = 8 = **(0, 2, 3)**.

Multiplication:

Multiplication is as simple as addition and subtraction. Just multiply the respective remainders!
Take the numbers 4 and 7.
4 = (0, 1, 4)
7 = (1, 1, 2)
4 × 7 = (0, 1, 4) × (1, 1, 2) = (0, 1, 8).
As before, the remainders must be less than the allowable limit. Here 8 > 4, which is the maximum possible remainder when divided by 5. Taking $[8]_5$, we get 3.Hence 4 × 7 = **28 = (0, 1, 3)**
This representation will hold for any number of bases relatively prime to each other. For example, if we wish to represent a number with respect to 4 bases, we will have 4 numbers in the representation.

Polynomials modulo m:

Let us look at m = 17 as an example. Then mod-m-polynomials are expressions like

$10x^4 + 14x + 2$ or $x^3 + 2x$ or $x^{10} + 7$.

This means there are powers
$x^0 = 1, x^1 = x, x^2 , x^3 , x^4 , \ldots ,$ of a symbol x (the variable).
Among these powers there is the expression x^0, to be read as "1" and to be omitted when it appears as a factor: $2 \cdot x^0 = 2$. Instead of x^1 we write x. Such powers of x we can multiply with numbers c between 0 and m − 1 to obtain terms cx^j .The factor c here is then called a coefficient. We get polynomials by adding arbitrary terms with different powers of x. In order not to get confused we normally order the terms according to falling powers of x, but we may also write $5x + 3x^2 + 1$ or $5x + 2x^2 + 1 + x^2$: these are just a different way of writing $3x^2 + 5x + 1$. 1
If we are given an expression like

$2x^4 - 3x^3 + 30x^2 + 3$

that is not really a polynomial modulo 17, because coefficients that are negative or larger than 16 are illegal, we may just take all numbers modulo 17 to obtain a proper mod-17-polynomial:

$$2x^4 + 14x^3 + 13x^2 + 3.$$

The general format for a mod-m-polynomial is the following:

$$(2)\quad c_n \cdot x^n + c_{n-1} \cdot x^{n-1} + \cdot \cdot \cdot + c_2 \cdot x^2 + c_1 \cdot x + c_0.$$

Here $c_n, \ldots, c_0$ are numbers between 0 and $m - 1$ (endpoints included).

It is interesting to note that if one wants to calculate with polynomials there is no need to deal with the "x" and its powers at all. One just stores the numbers $c_0, c_1, \ldots, c_n$, e. g. in an array C[0..n], and has the complete information.

CHINESE REMAINDER THEOREM:

The Chinese Remainder Theorem: If $m_1, m_2, \ldots, m_k$ are pairwise relatively prime positive integers, and if $a_1, a_2, \ldots, a_k$ are any integers, then the simultaneous congruences

$x \equiv a_1 \pmod{m_1}$, $x \equiv a_2 \pmod{m_2}$, ..., $x \equiv a_k \pmod{m_k}$ have a solution and the solution is unique modulo m, where $m = m_1 m_2 \cdots m_k$.

Proof that a solution exists:

To keep the notation simpler, we will assume k = 4. Note the proof is constructive, i.e., it shows us how to actually construct a solution. Our simultaneous congruences are

$x \equiv a_1 \pmod{m_1}$, $x \equiv a_2 \pmod{m_2}$, $x \equiv a_3 \pmod{m_3}$, $x \equiv a_4 \pmod{m_4}$.

Our goal is to find integers

	value mod m_1	value mod m_2	value mod m_3	value mod m_4
w_1	1	0	0	0
w_2	0	1	0	0
w_3	0	0	1	0
w_4	0	0	0	1

Once we have found w_1, w_2, w_3, w_4, it is easy to construct x:

$x = a_1w_1 + a_2w_2 + a_3w_3 + a_4w_4$.

Moreover, as long as the moduli (m_1, m_2, m_3, m_4) remains the same, we can use the same w_1, w_2, w_3, w_4 with any a_1, a_2, a_3, a_4.

First define:

$z_1 = m / m_1 = m_2m_3m_4$

$z_2 = m / m_2 = m_1m_3m_4$

$z_3 = m / m_3 = m_1m_2m_4$

$z_4 = m / m_4 = m_1m_2m_3$

Note that

i) $z_1 \equiv 0 \pmod{m_j}$ for j = 2, 3, 4.

ii) $\gcd(z_1, m_1) = 1$. (If a prime p dividing m_1 also divides $z_1= m_2m_3m_4$, then p divides m_2, m_3, or m_4.)and likewise for z_2, z_3, z_4.

Next define:

$y_1 \equiv z_1^{-1} \pmod{m_1}$

$y_2 \equiv z_2^{-1} \pmod{m_2}$

$y_3 \equiv z_3^{-1} \pmod{m_3}$

$y_4 \equiv z_4^{-1} \pmod{m_4}$

The inverses exist by (ii) above and we can find them by Euclid's extended algorithm.

Note that

i) $y_1z_1 \equiv 0 \pmod{m_j}$ for $j = 2, 3, 4$. (Recall $z_1 \equiv 0 \pmod{m_j}$)

ii) $y_1z_1 \equiv 1 \pmod{m_1}$ and likewise for y_2z_2, y_3z_3, y_4z_4.

Lastly define:

$w_1 \equiv y_1z_1 \pmod{m}$

$w_2 \equiv y_2z_2 \pmod{m}$

$w_3 \equiv y_3z_3 \pmod{m}$

$w_4 \equiv y_4z_4 \pmod{m}$

Then w1, w2, w3, and w4 have the properties in the table

Example:

Solve the simultaneous congruences

$x \equiv 6 \pmod{11}$,

$x \equiv 13 \pmod{16}$,

$x \equiv 9 \pmod{21}$,

$x \equiv 19 \pmod{25}$.

Solution:

Since 11, 16, 21, and 25 are pairwise relatively prime, the Chinese Remainder Theorem tells us that there is a unique solution modulo m, where $m = 11\cdot16\cdot21\cdot25 = 92400$.

We apply the technique of the Chinese Remainder Theorem with

$k = 4$, $m_1 = 11$, $m_2 = 16$, $m_3 = 21$, $m_4 = 25$,

$a_1 = 6$, $a_2 = 13$, $a_3 = 9$, $a_4 = 19$, to obtain the solution.

We compute

$z_1 = m / m_1 = m_2m_3m_4 = 16\cdot21\cdot25 = 8400$

$z_2 = m / m_2 = m_1m_3m_4 = 11\cdot21\cdot25 = 5775$

$z_3 = m / m_3 = m_1m_2m_4 = 11\cdot16\cdot25 = 4400$

$z_4 = m / m_4 = m_1m_3m_3 = 11\cdot16\cdot21 = 3696$

$y_1 \equiv z_1^{-1} \pmod{m_1} \equiv 8400^{-1} \pmod{11} \equiv 7^{-1} \pmod{11} \equiv 8 \pmod{11}$

$y_2 \equiv z_2^{-1} \pmod{m_2} \equiv 5775^{-1} \pmod{16} \equiv 15^{-1} \pmod{16} \equiv 15 \pmod{16}$

$y_3 \equiv z_3^{-1} \pmod{m_3} \equiv 4400^{-1} \pmod{21} \equiv 11^{-1} \pmod{21} \equiv 2 \pmod{21}$

$y_4 \equiv z_4^{-1} \pmod{m_4} \equiv 3696^{-1} \pmod{25} \equiv 21^{-1} \pmod{25} \equiv 6 \pmod{25}$

$w_1 \equiv y_1z_1 \pmod{m} \equiv 8\cdot8400 \pmod{92400} \equiv 67200 \pmod{92400}$

$w_2 \equiv y_2z_2 \pmod{m} \equiv 15\cdot5775 \pmod{92400} \equiv 86625 \pmod{92400}$

$w_3 \equiv y_3z_3 \pmod{m} \equiv 2\cdot4400 \pmod{92400} \equiv 8800 \pmod{92400}$

$w_4 \equiv y_4z_4 \pmod{m} \equiv 6\cdot3696 \pmod{92400} \equiv 22176 \pmod{92400}$

The solution, which is unique modulo 92400, is

$x \equiv a1w1 + a2w2 + a3w3 + a4w4 \pmod{92400}$

$\equiv 6\cdot 67200 + 13\cdot 86625 + 9\cdot 8800 + 19\cdot 22176 \pmod{92400}$

$\equiv 2029869 \pmod{92400} \equiv 51669 \pmod{92400}$.

We are given two arrays num[0..k-1] and rem[0..k-1]. In num[0..k-1], every pair is co-prime (gcd for every pair is 1). We need to find minimum positive number x such that:

```
x % num[0]    =  rem[0],

x % num[1]    =  rem[1],

     .......................

x % num[k-1]  =  rem[k-1]
```

Basically, we are given k numbers which are pairwise co-prime and given remainders of these numbers when an unknown number x is divided by them. We need to find the minimum possible value of x that produces given remainders.

Examples:

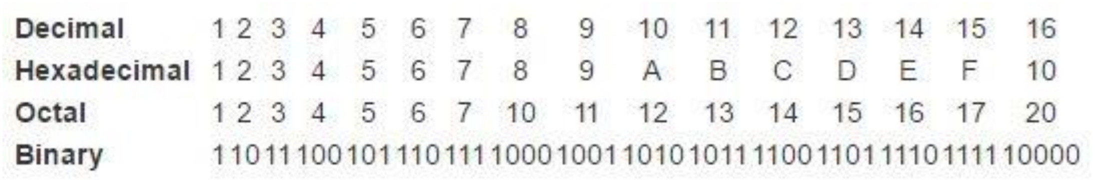
```
Input:  num[] = {5, 7}, rem[] = {1, 3}

Output: 31

Explanation: 31 is the smallest number such that:

(1) When we divide it by 5, we get remainder 1.

(2) When we divide it by 7, we get remainder 3.

Input:  num[] = {3, 4, 5}, rem[] = {2, 3, 1}

Output: 11

Explanation:

11 is the smallest number such that:

  (1) When we divide it by 3, we get remainder 2.

  (2) When we divide it by 4, we get remainder 3.

  (3) When we divide it by 5, we get remainder 1.
```

Chinese Remainder Theorem states that there always exists an x that satisfies given congruences.

CONVERSION BETWEEN BASE REPRESENTION AND MODULO REPRESENTION:

Representing a number in a base different from the customary 10 may have a great advantage.

- For example, binary representation is instrumental in solving Nim, Scoring, Turning Turtles and other puzzles. Along with the binary, the science of computers employs bases 8 and 16 for it's very easy to convert between the three while using bases 8 and 16 shortens considerably number representations.
- To represent 8 first digits in the binary system we need 3 bits. Thus we have, 0 = 000, 1 = 001, 2 = 010, 3 = 011,4 = 100, 5 = 101, 6 = 110, 7 = 111. Assume M = (2065)8.
- In order to obtain its binary representation, replace each of the four digits with the corresponding triple of bits: 010 000 110 101. After removing the leading zeros, binary representation is immediate: M = (10000110101)2. (For the hexadecimal system, conversion is quite similar, except that now one should use 4-bit representation of numbers below 16.)
- This fact follows from the general conversion algorithm and the observation that 8 = 23 (and, of course, 16 = 24.) Thus it appears that the shortest way to convert numbers into the binary system is to first convert them into either octal or hexadecimal representation. Now let us see how to implement the general algorithm programmatically. Let us combine the first few conversion in a table:



Decimal	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Hexadecimal	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
Octal	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17	20
Binary	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111	10000

Representation of a number in a system with base (***radix***) N may only consist of digits that are less than N. More accurately, if

(1) M = akNk + ak-1Nk-1 + ... + a1N1 + a0

with 0 ≤ ai < N. we have a representation of M in base N system and write

M = (akak-1...a0)N

If we rewrite (1) as

(2) M = a0 + N·(a1 + N·(a2 + N·...))

The algorithm for obtaining coefficients ai becomes more obvious. For example, a0 = M modulo n and a1 = M/N (modulo n) and so on.

Recursive implementation

Let's represent the algorithm mnemonically: (***result*** is a string or character variable where I shall accumulate the digits of the result one at a time)

```
1. result = ""
2. if M < N, result = 'M' + result. Stop.
3. S = M mod N, result = 'S' + result
   M = M/N
4. goto 2
```

A few words of explanation.

1. It is an empty string. You may remember it's a ***zero*** element for string concatenation.
2. Here we check whether the conversion procedure is over. It's over if M is less than N in which case M is a digit (with some qualification for N>10) and no additional action is necessary. Just prepend it in front of all other digits obtained previously. The '+' plus sign stands for the string concatenation.
3. If we got this far, M is not less than N. First we extract its remainder of division by N, prepend this digit to the result as described previously and reassign M to be M/N.
4. This says that the whole process should be repeated starting with step 2.

I would like to have a function say called Conversion that takes two arguments M and N and returns representation of the number M in base N. The function might look like this

```
String Conversion(int M, int N)      // return string, accept two integers
{
   if (M < N)                        // see if it's time to return
      return new String(""+M);       // ""+M makes a string out of a digit
   Else                              // the time is not yet ripe
      return Conversion(M/N, N) +    // continue
        new String(""+(M mod N));
 }
```

At some point the function calls itself with a different first argument. One may say that the function is defined in terms of itself. Such functions are called ***recursive***. (The best known recursive function is ***factorial***: n! = n·(n-1)!.) The function calls (applies) itself to its arguments and then (naturally) applies itself to its new arguments and then ... and so on. We can be sure that the process will eventually stop because the sequence of arguments (the first ones) is decreasing. Thus sooner or later the first argument will be less than the second and the process will start emerging from the recursion, still a step at a time.

Iterative implementation

The iterative implementation of the Conversion function might look as the following.

```
String Conversion(int M, int N)           // return string, accept two integers
{
   Stack stack = new Stack();             // create a stack
   while (M >= N)                         // now the repetitive loop is clearly seen
   {
   7     stack.push(M mod N);                    // store a digit
       M = M/N;                                   // find new M
   }
   // now it's time to collect the digits together
   String str = new String(""+M);                // create a string with a single digit M
   while (stack.NotEmpty())
       str = str+stack.pop()                      // get from the stack next digit
   return str;
}
```

The function is by far longer than its recursive counterpart; but, as I said, sometimes it's the one we want to use and sometimes it's the only one you may actually use.

Introduction to Polynomials:

Definition: A polynomial is an expression of the form:

an · x n + an −1 · x n −1 + … + a2 · x 2 + a1 · x + a0,where x is a variable, n is a positive integer and a0, a1, … , an −1, an are constants. The highest power of x that occurs is called the degree of the polynomial. The terms are usually written in order from highest power of x to lowest power.

Examples:

- Any quadratic is a polynomial of degree 2.
- $x^4 - 8x^2$ is a polynomial of degree 4.
- $x^5 - 8x^3 + 10x + 6$ is a polynomial of degree 5.

A polynomial is an expression consisting of variables and coefficients,that involves only the operations of addition, subtraction, multiplication and non-negative integer exponents.

To create a polynomial imagines carrying out the following steps:

- Start with a variable *x*.
- Raise *x* to various integer powers, starting with the power 0 and ending with the power *n* (where *n* is a positive integer):

$1,\ x,\ x^2,\ x^3,\ \dots x^n.$

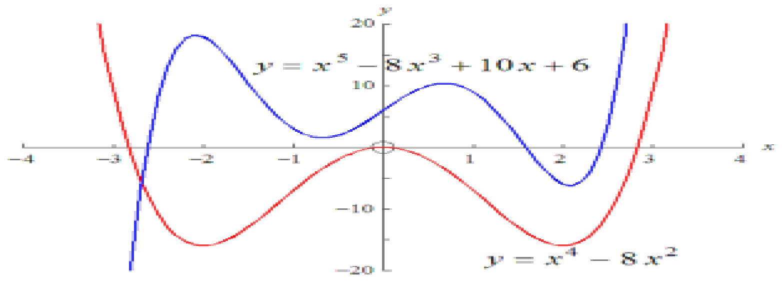
- Multiply each power of *x* by a coefficient. Let a_3 denote the coefficient of x^3, and so on.
- Add all the terms together.

The result is a **polynomial**. Note that some of the coefficients could be zero so that some of the powers of *x* could be absent.

Graph of a polynomial

We can create a **polynomial function**, called say *f*, whose input is *x* and whose output, *f* (*x*), is the polynomial evaluated at *x*:

$$x \rightarrow \boxed{f} \rightarrow a_n x^n + a_{n-1} x^{n-1} + \ \dots\ + a_2 x^2 + a_1 x + a_0$$



We can then call the output of the function "*y*" and make a **graph** of *y* versus *x*. We will get a curve like the two curves shown to the right. The graph of a polynomial function oscillates smoothly up and down several times before finally "taking off for good" in either the up or down direction. The degree of the polynomial gives the maximum number of "*ups and downs*" that the graph of the polynomial can have. It also gives the maximum number of crossings of the *x* axis that the polynomial can have.

Polynomial equation

If we set the polynomial equal to zero or if we set $y = 0$ or $f(x) = 0$ then we get a so-called **polynomial equation**:

$a_n x^n + a_{n-1} x^{n-1} + \ \dots\ + a_2 x^2 + a_1 x + a_0 = 0.$

(Note that setting $y = 0$ in the polynomial's graph means that we are looking at points where the graph crosses the *x* axis and setting $f(x) = 0$ in the polynomial function means that we are looking for values of *x* for which the output of the polynomial function is zero.

There is a close connection between:

- The values of *x* that cause a polynomial to equal zero.
- The places where a polynomial function's graph crosses the *x* axis.
- The solutions of a polynomial equation.
- The factors of a polynomial.

This connection is made formal by the *Factor theorem* and the *Fundamental theorem of algebra*.

The factor theorem

Let f (x) be a polynomial.

If (x − r) is a factor of the polynomial, then r is a root of the polynomial equation f (x) = 0.

Conversely, if the polynomial equation f (x) = 0 has a root r, then (x − r) is a factor of the polynomial f (x).

The fundamental theorem of algebra

Over the complex numbers, a polynomial equation of degree n has exactly n roots. Over the real numbers it may have less than n.

- Some of the roots may be real.
- Some of the roots may be complex. If so then their complex conjugates will also be roots.
- Some of the roots may be equal (these are called multiple or repeated roots).

Notes on the Factor theorem and the Fundamental theorem of algebra:

- The factor theorem essentially says that finding the roots of a polynomial equation amounts to finding the factors of the polynomial and *vice versa*.
- If a is a real root, then $x - a$ is a factor of the polynomial and the graph of the polynomial crosses the x axis at $x = a$.
- If a is a double root, then $(x - a)^2$ is a factor of the polynomial and the graph of the polynomial just *touches* the x axis at $x = a$ rather than crosses it.
- Over the real numbers, complex roots are not allowed and must be excluded from the count of roots. That is why there may be less than n solutions over the real numbers.
- If $a + b\,i$ is a complex root then so is $a - b\,i$ and the expression $(x - a + b\,i)\,(x - a - b\,i)$ is a factor of the polynomial. Over the real numbers this complex expression is not allowed and is replaced by the real quadratic expression $x^2 - 2\,a\,x + (a^2 + b^2)$ (which is the previous expression, but in unfactored form). A complex root $a + b\,i$ does not correspond to a graph crossing of the x axis.

Extension of Polynomials:

A **polynomial extension** operator is an **extension** operator with the additional property that whenever the function on ∂K to be extended is the trace of a **polynomial** on K, the extended function is also a **polynomial**. u is a **polynomial** of degree at most p on K.

INTERPOLATION PROBLEM:

The process of fitting a function through given data is called interpolation. Usually when we have data, we don't know the function f(x) that generated the data. So we fit a certain class of functions. The most usual class of functions fitted through data are polynomials. Polynomials are fitted through data when we don't know f(x).The process of fitting a polynomial through given data is called polynomial interpolation. Polynomials are often used because they have the property of approximating any continuous function.

Given:

f(x) continuous on [a,b], ε>0 (called tolerance), then there is a polynomial P(x) of appropriate degree which approximates the function within the given tolerance. **Interpolation**, in mathematics, the determination or estimation of the value of $f(x)$, or a function of x, from certain known values of the function.

If $x0 < \ldots < xn$ and $y0 = f(x0),\ldots, yn = f(xn)$ are known and if $x0 < x < xn$, then the estimated value of $f(x)$ is said to be an interpolation. If $x < x0$ or $x > xn$, the estimated value of $f(x)$ is said to be an extrapolation.If $\boldsymbol{x}0, \ldots, \boldsymbol{xn}$ are given, along with corresponding values $\boldsymbol{y}0, \ldots, \boldsymbol{yn}$ (see the figure), interpolation may be regarded as the determination of a function $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$ whose **graph** passes through the $\boldsymbol{n} + 1$ points, $(\boldsymbol{xi}, \boldsymbol{yi})$ for $\boldsymbol{i} = 0, 1, \ldots, \boldsymbol{n}$. There are infinitely many such functions, but the simplest is a polynomial interpolation function

$$\boldsymbol{y} = \boldsymbol{p}(\boldsymbol{x}) = \boldsymbol{a}0 + \boldsymbol{a}1\boldsymbol{x} + \ldots + \boldsymbol{anxn}$$

with constant $\boldsymbol{ai}$'s such that $\boldsymbol{p}(\boldsymbol{xi}) = \boldsymbol{yi}$ for $\boldsymbol{i} = 0, \ldots, \boldsymbol{n}$.

DISCRETE FOURIER TRANSFORM:

Definition:

Let $x_0, \ldots, x_{N-1}$ be **complex numbers**. The **DFT** is defined by the formula

$$X_K = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N} \qquad k = 0, \ldots \ldots N-1$$

Any function that periodically repeats itself can be expressed as the sum of sines and/or cosines of different frequencies, each multiplied by a different coefficient (Fourier series).

Even functions that are not periodic (but whose area under the curve is finite) can be expressed as the integral of sines and/or cosines multiplied by a weighting function (Fourier transform).

The **frequency domain** refers to the plane of the two dimensional discrete Fourier transform of an image. The purpose of the Fourier transform is to represent a signal as a linear combination of sinusoidal signals of various frequencies. The one-dimensional Fourier transform and its inverse

- Fourier transform (continuous case)

$$F(u) = \int_{-\infty}^{\infty} f(x) e^{-j2\pi ux} dx \quad \text{where } j = \sqrt{-1}$$

- Inverse Fourier transform:

$$f(x) = \int_{-\infty}^{\infty} F(u) e^{j2\pi ux} du$$

The two-dimensional Fourier transform and its inverse

- Fourier transform (continuous case)

$$F(u,v) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y) e^{-j2\pi(ux+vy)} dxdy$$

- Inverse Fourier transform:

$$f(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} F(u,v) e^{j2\pi(ux+vy)} dudv$$

The one-dimensional Fourier transform and its inverse

– Fourier transform (discrete case) DTC

$$F(u) = \frac{1}{M}\sum_{x=0}^{M-1} f(x) e^{-j2\pi ux/M} \quad \text{for } u = 0,1,2,\ldots,M-1$$

– Inverse Fourier transform:

$$f(x) = \sum_{u=0}^{M-1} F(u) e^{j2\pi ux/M} \quad \text{for } x = 0,1,2,\ldots,M-1$$

- Since $e^{j\theta} = \cos\theta + j\sin\theta$ and the fact $\cos(-\theta) = \cos\theta$ then discrete Fourier transform can be redefined

$$F(u) = \frac{1}{M}\sum_{x=0}^{M-1} f(x)[\cos 2\pi ux/M - j\sin 2\pi ux/M]$$

for $u = 0,1,2,\ldots,M-1$

  - Frequency (time) domain: the domain (values of *u*) over which the values of *F*(*u*) range; because *u* determines the frequency of the components of the transform.
  - Frequency (time) component: each of the *M* terms of *F*(*u*).
- *F*(*u*) can be expressed in polar coordinates:

$$F(u) = |F(u)| e^{j\phi(u)}$$

where $|F(u)| = \left[R^2(u) + I^2(u)\right]^{\frac{1}{2}}$ (magnitude or spectrum)

$$\phi(u) = \tan^{-1}\left[\frac{I(u)}{R(u)}\right] \text{ (phase angle or phase spectrum)}$$

  - *R*(*u*): the real part of *F*(*u*)
  - *I*(*u*): the imaginary part of *F*(*u*)
- Power spectrum:

$$P(u) = |F(u)|^2 = R^2(u) + I^2(u)$$

• The two-dimensional Fourier transform and its inverse
  - Fourier transform (discrete case) DTC

$$F(u,v) = \frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y) e^{-j2\pi(ux/M + vy/N)}$$

for $u = 0,1,2,\ldots,M-1, v = 0,1,2,\ldots,N-1$

  - Inverse Fourier transform:

$$f(x,y) = \sum_{u=0}^{M-1}\sum_{v=0}^{N-1} F(u,v) e^{j2\pi(ux/M + vy/N)}$$

for $x = 0,1,2,\ldots,M-1, y = 0,1,2,\ldots,N-1$

    • *u*, *v* : the transform or frequency variables
    • *x*,*y* : the spatial or image variables

• We define the Fourier spectrum, phase angle and power spectrum as follows:

$$|F(u,v)| = \left[R^2(u,v) + I^2(u,v)\right]^{\frac{1}{2}} \text{ (spectrum)}$$

$$\phi(u,v) = \tan^{-1}\left[\frac{I(u,v)}{R(u,v)}\right] \text{ (phase angle)}$$

$$P(u,v) = |F(u,v)|^2 = R^2(u,v) + I^2(u,v) \text{ (power spectrum)}$$

  - *R*(*u*,*v*): the real part of *F*(*u*,*v*)
  - *I*(*u*,*v*): the imaginary part of *F*(*u*,*v*)

• Some properties of Fourier transform:

$$\Im\left[f(x,y)(-1)^{x+y}\right] = F(u-\frac{M}{2}, v-\frac{N}{2}) \text{ (shift)}$$

$$F(0,0) = \frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y) \qquad \text{(average)}$$

$$F(u,v) = F^*(-u,-v) \qquad \text{(conujgate symmetric)}$$

$$|F(u,v)| = |F(-u,-v)| \qquad \text{(symmetric)}$$

- In mathematics, the discrete Fourier transform (DFT) converts a finite sequence of equally-spaced samples of a function into a same-length sequence of equally-spaced samples of the discrete-time Fourier transform (DTFT), which is a complex-valued function of frequency.
- The interval at which the DTFT is sampled is the reciprocal of the duration of the input sequence. Discrete Fourier Transform (DFT) is the discrete version of the Fourier Transform (FT) that transforms a signal (or discrete sequence) from the time domain representation to its representation in the frequency domain. Whereas, Fast Fourier Transform (FFT) is any efficient algorithm for calculating the DFT.
- An inverse DFT is a Fourier series, using the DTFT samples as coefficients of complex sinusoids at the corresponding DTFT frequencies. It has the same sample-values as the original input sequence. The DFT is a frequency domain representation of the original input sequence.
- If the original sequence spans all the non-zero values of a function, its DTFT is continuous (and periodic) and the DFT provides discrete samples of one cycle. If the original sequence is one cycle of a periodic function, the DFT provides all the non-zero values of one DTFT cycle. The DFT is the most important discrete transform, used to perform Fourier analysis in many practical applications.
- In digital signal processing, the function is any quantity or signal that varies over time, such as the pressure of a sound wave, a radio signal, or daily temperature readings, sampled over a finite time interval (often defined by a window function).
- The samples can be the values of pixels along a row or column of a raster image. The DFT is also used to efficiently solve partial differential equations and to perform other operations such as convolutions or multiplying large integers. Since it deals with a finite amount of data, it can be implemented in computers by numerical algorithms or even dedicated hardware.
- These implementations usually employ efficient fast Fourier transform (FFT) algorithms so much so that the terms "FFT" and "DFT" are often used interchangeably. Prior to its current usage, the "FFT" initialisation may have also been used for the ambiguous term "finite Fourier transform".

Definition:

$$X_K = \sum_{n=0}^{N-1} x_n . \mathrm{e}^{-\frac{i2\pi}{N}kn} \qquad k = 0, \dots \dots N-1$$

$$= \sum_{n=0}^{N-1} \mathrm{x}_n . \left[\cos\left(\frac{2\pi}{\mathrm{N}}\mathrm{kn}\right) - \mathrm{i}.\sin\left(\frac{2\pi}{\mathrm{N}}\mathrm{kn}\right)\right]$$

Where the last expression follows from the first one by Euler's formula.

The transform is sometimes denoted by the symbol, as in X=f(x) or f(x) or F_x.

- The discrete analog of the formula for the coefficients of a Fourier series is

$$x_{\mathrm{n}} = \frac{1}{N}\sum_{k=0}^{\mathrm{N}-1} x_{\mathrm{k}} . \mathrm{e}^{-\mathrm{i}2\pi\mathrm{kn}/\mathrm{N}} \qquad \mathrm{n} \in \mathrm{Z}$$

IN COMPLEX FIELD DFT:

The FFT is based on the complex DFT, a more sophisticated version of the DFT. These transform are named for the way each represent data, this is using complex numbers or using real numbers. The term complex does not mean that this representation is difficult or complicated but that a specific type of mathematics is used.

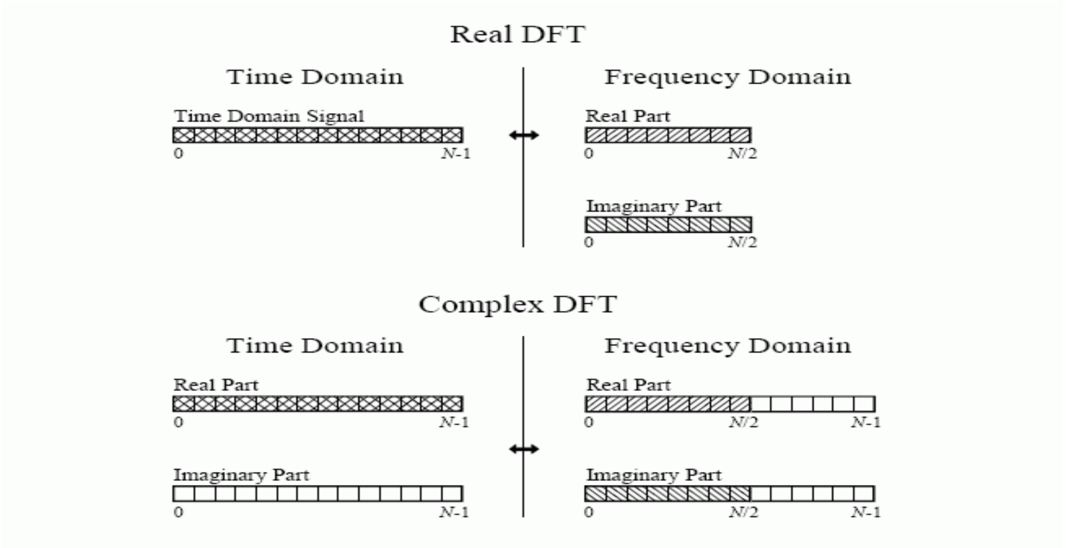


FIGURE 12-1
Comparing the real and complex DFTs. The real DFT takes an N point time domain signal and creates two $N/2+1$ point frequency domain signals. The complex DFT takes two N point time domain signals and creates two N point frequency domain signals. The crosshatched regions shows the values common to the two transforms.

Consider the case of N-point complex DFT, it takes in N samples of complex-valued time domain waveform $x[n]$ and produces an array $X[k]$ of length N.

$$X[k] = \frac{1}{N}\sum_{n=0}^{N-1} x[n]e^{-j2\pi kn/N}$$

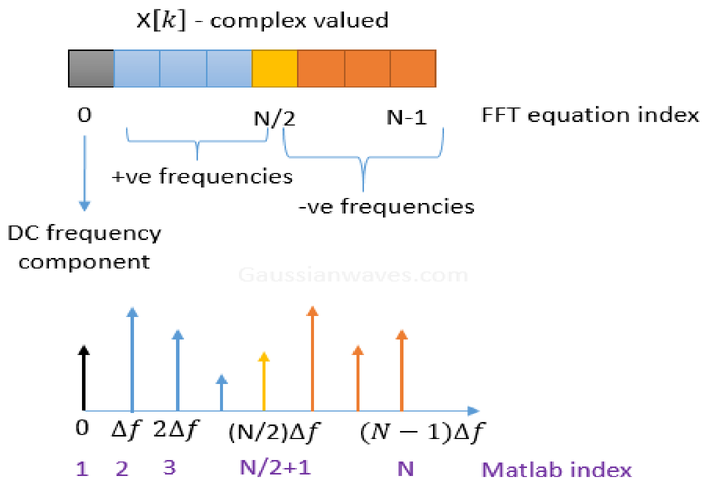
The arrays values are interpreted as follows

- $X[0]$ represents DC frequency component

- Next $N/2$ terms are positive frequency components with $X[N/2]$ being the Nyquist frequency (which is equal to half of sampling frequency).
- Next $N/2-1$ terms are negative frequency components (note: negative frequency components are the phasors rotating in opposite direction, they can be optionally omitted depending on the application).The corresponding synthesis equation (reconstruct $x[n]$ from frequency domain samples $X[k]$) is

$$x[n] = \sum_{k=0}^{N-1} X[k] e^{j2\pi kn/N}$$

From these equations we can see that the real DFT is computed by projecting the signal on cosine and sine basis functions. However, the complex DFT projects the input signal on exponential basis functions (Euler's formula connects these two concepts).When the input signal in the time domain is real valued, the complex DFT zero-fills the imaginary part during computation (That's its flexibility and avoids the caveat needed for real DFT). The following figure shows how to interpret the raw FFT results in Matlab that computes complex DFT.



FAST FOURIER TRANSFORM:

The Fourier Transform is one of deepest insights ever made. Unfortunately, the meaning is buried within dense equations:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N}$$

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{i2\pi kn/N}$$

- Evaluating this definition directly requires $O(N^2)$ operations: there are *N* outputs X_k, and each output requires a sum of *N* terms.
- An FFT is any method to compute the same results in o(N log N) operations. All known FFT algorithms require Θ(NlogN) operations, although there is no known proof that a lower complexity score is impossible.

To illustrate the savings of an FFT, consider the count of complex multiplications and additions for N=4096 data points.

- Evaluating the DFT's sums directly involves N^2 complex multiplications and $N(N-1)$ complex additions, of which O(N) operations can be saved by eliminating trivial operations such as multiplications by 1, leaving about 30 million operations.
- On the other hand, the radix-2 Cooley–Tukey algorithm, for *N* a power of 2, can compute the same result with only $(N/2)\log_2(N)$ complex multiplications (again, ignoring simplifications of multiplications by 1 and similar) and $N\log_2(N)$ complex additions, in total about 30,000 operations - a thousand times less than with direct evaluation. In practice, actual performance on modern computers is usually dominated by factors other than the speed of arithmetic operations and the analysis is a complicated subject but the overall improvement from $O(N^2)$ to $O(N \log N)$ remains.

Cooley–Tukey algorithm

- By far the most commonly used FFT is the Cooley–Tukey algorithm. This is a divide and conquer algorithm that recursively breaks down a DFT of any composite size $N = N_1N_2$into many smaller DFTs of sizes N_1 and N_2, along with O(N) multiplications by complex roots of unity traditionally called twiddle factors.
- This method (and the general idea of an FFT) was popularized by a publication of Cooley and Tukey in 1965, but it was later discovered that those two authors had independently re-invented an algorithm known to Carl Friedrich Gauss around 1805 (and subsequently rediscovered several times in limited forms).
- The best known use of the Cooley–Tukey algorithm is to divide the transform into two pieces of size N/2 at each step and is therefore limited to power-of-two sizes, but any factorization can be used in general (as was known to both Gauss and Cooley/Tukey).
- These are called the radix-2 and mixed-radix cases, respectively (and other variants such as the split-radix FFT have their own names as well).
- Although the basic idea is recursive, most traditional implementations rearrange the algorithm to avoid explicit recursion.
- Also, because the Cooley–Tukey algorithm breaks the DFT into smaller DFTs, it can be combined arbitrarily with any other algorithm for the DFT.

Other FFT algorithms:

- Main articles: Prime-factor FFT algorithm, Bruun's FFT algorithm, Rader's FFT algorithm, Bluestein's FFT algorithm and Hexagonal fast Fourier transform.
- There are FFT algorithms other than Cooley–Tukey. Cornelius Lanczos did pioneering work on the FFT and FFS (fast Fourier sampling method) with G. C. Danielson (1940).
- For $N = N_1N_2$ with co-prime N_1 and N_2, one can use the prime-factor (Good–Thomas) algorithm (PFA), based on the Chinese remainder theorem, to factorize the DFT similarly to Cooley–Tukey but without the twiddle factors.
- The Rader–Brenner algorithm (1976) is a Cooley–Tukey-like factorization but with purely imaginary twiddle factors, reducing multiplications at the cost of increased additions and reduced numerical stability.
- It was later superseded by the split-radix variant of Cooley–Tukey (which achieves the same multiplication count but with fewer additions and without sacrificing accuracy).
- Algorithms that recursively factorize the DFT into smaller operations other than DFTs include the Bruun and QFT algorithms.
- The Rader–Brenner and QFT algorithms were proposed for power-of-two sizes, but it is possible that they could be adapted to general composite N. Bruun's algorithm applies to arbitrary even composite sizes.
- Bruun's algorithm, in particular, is based on interpreting the FFT as a recursive factorization of the polynomials$^N - 1$, here into real-coefficient polynomials of the form $z^M - 1$ and $z^{2M} + az^M + 1$.

SCONHAGE-STRASSEN INTEGER MULTIPLICATION ALGORITHM:

Basics of Integer Multiplication

1. By polynomial multiplication
2. Evaluation/Interpolation
3. Karatsuba's method
4. Toom-Cook method
5. FFT and weighted transforms
6. Schonhage-Strassen's Algorithm

Schonhage-Strassen's Algorithm:

The **Schönhage–Strassen algorithm** is an asymptotically fast multiplication algorithm for large integers. It was developed by Arnold Schönhage and VolkerStrassen in 1971.

The run-time bit complexity is, in Big O notation, O(n.log n .log.log n) for two *n*-digit numbers. The algorithm uses recursive Fast Fourier transforms in rings with 2^n+1 elements, a specific type of number theoretic transform.

Basic Idea:

- First algorithms to use FFT (Schonhage and Strassen 1971)
- Uses ring Rn = Z/(2n + 1)Z for transform, with l = 2k | n
- Then 2n/l ≡ −1 (mod 2n + 1): so 2n/l ∈Rn is 2l-th root of unity, multiplication by powers of 2 is fast! (O(n))
- Allows length l weighted transform for negacyclic convolution
- Write input a = A(2M), b = B(2M), compute C(x) = A(x)B(x) (mod xl + 1). Then c = C(2M) = ab (mod 2Ml + 1)
- Point-wise products modulo 2n + 1 use SSA recursively: choose next level's l' , M' so that M' l' = n

Motivation for Improving SSA:

- Integer multiplication is fundamental to arithmetic, used in PRP testing, ECPP, polynomial multiplication.
- Schonhage-Strassen's algorithm [SSA]: good asymptotic complexity O(n log n log log n), fast in practice for large operands, exact (only integer arithmetic)
- Used in GMP, widely deployed
- Validated by implementation based on GMP 4.2.1 [GMP]

Schonhage-Strassen's Algorithm:

- SSA reduces multiplication of two S-bit integers to l- multiplications of approx. 4S/l-bit integers
- Example: multiply two numbers a, b of 220 bits each ⇒ product has at most 2^21 bits
  1. Choose N = 2^21 and a good l, for this example l = 512. We compute ab mod (2N + 1)
  2. Write a as polynomial of degree l, coefficients ai< 2M with M = N/l, a = a(2M). Same for b
  3. ab = a(2M)b(2M) mod (2N + 1), compute c(x) = a(x)b(x) mod (xl + 1)
  4. Convolution theorem: Fourier transform and pointwise multiplication Alexander Kruppa 14 INRIA RocquencourtSchonhage-Strassen's Algorithm
  5. FFT needs l-th root of unity: map to Z/(2n + 1)Z[x] with l | n. Then 22n/l has order l
  6. We need 2n + 1 > ci: choose n ≥ 2M + log2(l)+1
  7. Compute c(x) = a(x)b(x) mod (xl + 1), evaluate ab = c(2M) - and we are done!
  8. Benefits: - Root of unity is power of 2 - Reduction mod(2n + 1) is fast - Point-wise products can use SSA recursively without padding

Mersenne Transform:

- Convolution theorem implies reduction mod(xl − 1)
- Convolution mod(xl + 1) needs weights θi with ord(θ)=2l, needs l | n to get 2l-th root of unity in Rn
- Computing ab mod (2N + 1) to allow recursive use of SSA, but is not required at top level
- Map a and b to Z/(2N − 1)Z instead: compute c(x) = a(x)b(x) mod (xl − 1)

- Condition relaxes to l | 2n. Twice the transform length, smaller n
- No need to apply/unapply weights

CRT Reconstruction:
At least one of (2rN −1, 2sN +1) and (2rN +1, 2sN −1) is co-prime
- Our implementation uses (2rN +1, 2N −1) : always co-prime, good speed
- Smaller convolution, finer-grained parameter selection

The √2 Trick
- If 4 | n, 2 is a quadratic residue in Z/(2n + 1)Z
- In that case, √2 ≡ 23n/4 − 2n/4: simple form, multiplication by √2 takes only 2 shift, 1 subtraction modulo 2n + 1
- Offers root of unity of order 4n, allows l | 2n for Fermat transform, l | 4n for Mersenne transform
- Sadly, higher roots of 2 usually not available in Z/(2n + 1)Z, or have no simple form

Arithmetic modulo:
- 2n + 1 Residues stored semi-normalized (< 2n+1), each with m = n/w full words plus one extra word ≤ 1
- Adding two semi-normalized values: c = a[m] + b[m] + mpn_add_n (r, a, b, m); r[m] = (r[0] < c); MPN_DECR_U (r, m + 1, c - r[m]); Assures r[m] = 0 or 1, c − r[m] > r[0] ⇒ r[m] = 1 so carry propagation must terminate.
- Conditional branch only in mpn_add_n loop and (almost always non-taken) in carry propagation of MPN_DECR_U • Similar for subtraction, multiplication by 2k

UNIT-5

LINEAR PROGRAMMING

Linear programming (LP) is a method to achieve the optimum outcome under some requirements represented by linear relationships. LP is an optimization technique for a system of linear constraints and a linear objective function. An objective function defines the quantity to be optimized and the goal of linear programming is to find the values of the variables that maximize or minimize the objective function subject to some linear constraints. In general, the standard form of LP consists of

- Variables: $\boldsymbol{x} = (x_1, x_2, \ldots, x_d)^\top$
- Objective function: $\boldsymbol{c} \cdot \boldsymbol{x}$
- Inequalities (constraints): $A\boldsymbol{x} \leq \boldsymbol{b}$, where A is a $n \times d$ matrix

and we maximize the objective function subject to the constraints and $x \geq 0$.

LP has many different applications, such as flow, shortest paths, and even politics. In this lecture, we will be covering different examples of LP, and present an algorithm for solving them. Linear programming is part of an important area of mathematics called "optimization techniques" as it is literally used to find the most optimized solution to a given problem. A very basic example of linear optimization usage is in logistics or the "method of moving things around efficiently."

For example, suppose there are 1000 boxes of the same size of 1 cubic meter each; 3 trucks that are able to carry 100 boxes, 70 boxes and 40 boxes respectively; several possible routes and 48 hours to deliver all the boxes. Linear programming provides the mathematical equations to determine the optimal truck loading and route to be taken in order to meet the requirement of getting all boxes from point A to B with the least amount of going back and forth and of course the lowest cost at the fastest time possible.

The basic components of linear programming are as follows:

- Decision variables - These are the quantities to be determined.
- Objective function - This represents how each decision variable would affect the cost, or, simply, the value that needs to be optimized.
- Constraints - These represent how each decision variable would use limited amounts of resources.
- Data - These quantify the relationships between the objective function and the constraints.

ASSUMPTIONS OF LINEAR PROGRAMMING MODELS

Linear programs are constrained optimization models that satisfy three requirements.

1. The decision variables must be continuous; they can take on any value within some restricted range.
2. The objective function must be a linear function.
3. The left-hand sides of the constraints must be linear functions.

Thus, linear programs are written in the following form:

Maximize or minimize $z = c_1x_1 + c_2x_2 + \ldots\ldots\ldots + c_nx_n$

$\leq$

Subject to $a_{11}x_1 + a_{12}x_2 \ldots\ldots\ldots + a_{1n}x_n = b_1$

$\geq$

$\leq$

$a_{21}x_1 + a_{22}x_2 \ldots\ldots\ldots + a_{2n}x_n = b_2$

$\geq$

$\leq$

$a_{m1}x_1 + a_{m2}x_2 \ldots\ldots\ldots + a_{mn}x_n = b_m$

$\geq$

where the xj values are decision variables and cj, aijand bi values are constants, called parameters or coefficients, that are given or specified by the problem assumptions. Most linear programs require that all decision variables be nonnegative.

Linear programs make the following implicit assumptions.

1. **Proportionality:** With linear programs, we assume that the contribution of individual variables in the objective function and constraints is proportional to their value. That is, if we double the value of a variable, we double the contribution of that variable to the objective function and each constraint in which the variable appears. The contribution per unit of the variable is constant. For example, suppose the variable xj is the number of units of product j produced and cj is the cost per unit to produce product j. If doubling the amount of product j produced doubles its cost, per unit cost is constant and the proportionality assumption is satisfied.

2. **Additivity:** Additivity means that the total value of the objective function and each constraint function is obtained by adding up the individual contributions from each variable.

3. **Divisibility.** The decision variables are allowed to take on any real numerical values within some range specified by the constraints. That is, the variables are not restricted to integer values. When fractional values do not make a sensible solution, such as the number of flights an airline should have each day between two cities, the problem should be formulated and solved as an integer program.

4. **Certainty:** We assume that the parameter values in the model are known with certainty or are at least treated that way. The optimal solution obtained is optimal for the specific problem formulated. If the parameter values are wrong, then the resulting solution is of little value. In practice, the assumptions of proportionality and additivity need the greatest care and are most likely to be violated by the modeler. With experience, we recognize when integer solutions are needed and the variables must be modeled explicitly.In practice, the assumptions of proportionality and additivity need the greatest care and are most likely to be violated by the modeler. With experience, we recognize when integer solutions are needed and the variables must be modeled explicitly.

Applications of Linear Programming

Linear programming and Optimization are used in various industries. The manufacturing and service industry uses linear programming on a regular basis. In this section, we are going to look at the various applications of Linear programming.

1. Manufacturing industries use linear programming for **analyzing their supply chain operations**. Their motive is to maximize efficiency with minimum operation cost. As per the recommendations from the linear programming model, the manufacturer can reconfigure their storage layout, adjust their workforce and reduce the bottlenecks.
2. Linear programming is also used in organized retail for **shelf space optimization**. Since the number of products in the market has increased in leaps and bounds, it is important to understand what does the customer want. Optimization is aggressively used in stores like Walmart, Hypercity, Reliance, Big Bazaar, etc. The products in the store are placed strategically keeping in mind the customer shopping pattern. The objective is to make it easy for a customer to locate & select the right products. This is subject to constraints like limited shelf space, a variety of products, etc.
3. Optimization is also used for **optimizing Delivery Routes**. This is an extension of the popular traveling salesman problem. The service industry uses optimization for finding the best route for multiple salesmen traveling to multiple cities. With the help of clustering and greedy algorithm, the delivery routes are decided by companies like FedEx, Amazon, etc. The objective is to minimize the operation cost and time.
4. Optimizations are also used in **Machine Learning**. Supervised Learning works on the fundamental of linear programming. A system is trained to fit on a mathematical model of a function from the labeled input data that can predict values from an unknown test data.

FORMULATING LINEAR PROGRAMS

Steps in Problem Formulation

1. Identify and define the decision variables for the problem. Define the variables completely and precisely. All units of measure need to be stated explicitly, including time units if appropriate. For example, if the variables represent quantities of a product produced, these should be defined in terms of tons per hour, units per day, barrels per month, or some other appropriate units.
2. Define the objective function. Determine the criterion for evaluating alternative solutions. The objective function will normally be the sum of terms made up of a variable multiplied by some appropriate coefficient (parameter). For example, the coefficients might be profit per unit of production, distance travel per unit transported, or cost per person hired.
3. Identify and express mathematically all of the relevant constraints. It is often easier to express each constraint in words before putting it into mathematical form. The written constraint is decomposed into its fundamental components. Then substitute the appropriate numerical coefficients and variable names for the written terms. A common mistake is using variables that have not been defined in the problem, which is not valid. This mistake is frequently caused by not defining the original variables precisely. The formulation process is iterative, and sometimes additional variables must be defined or existing variables redefined. For example, if one of the variables is the total production of the company and five other variables represent the production at the company's five plants, then there must be a constant that forces total production to equal the sum of the production at the plants.

Feed Mix or Diet Problem one of the first problems solved using linear programming is the feed mix problem,

International Wool Company operates a large farm on which sheep are raised. The farm manager determined that for the sheep to grow in the desired fashion, they need at least minimum amounts of four nutrients (the nutrients are nontoxic so the sheep can consume more than the minimum without harm). The manager is considering three different grains to feed the sheep. Table B-2 lists the number of units of each nutrient in each pound of grain, the minimum daily requirements of each nutrient for each sheep and the cost of each grain. The manager believes that as long as a sheep receives the minimum daily amount of each nutrient, it will be healthy and produce a standard amount of wool. The manager wants to raise the sheep at minimum cost.

Table: International Wool Data

		Grain			Minimum Daily Grain Requirement(units)
		1	2	3	
Nutrient	A	20	30	70	110
Nutrient	B	10	10	0	18
Nutrient	C	50	30	0	90
Nutrient	D	6	2.5	10	14
Cost (¢/lb)		41	36	96	

Solution

The quantities that the manager controls are the amounts of each grain to feed each sheep daily. We define xj number of pounds of grain j (= 1, 2, 3) to feed each sheep daily. Note that the units of measure are completely specified. In addition, the variables are expressed on a per sheep basis. If we minimize the cost per sheep, we minimize the cost for any group of sheep. The daily feed cost per sheep will be

(cost per lb of grain j)/(lb. of grain j fed to each sheep daily)

That is, the objective function is to

Minimize $z = 41x_1 + 36x_2 + 96x_3$nutrient

Why can’t the manager simply make all the variables equal to zero? This keeps costs at zero, but the manager would have a flock of dead sheep, because there are minimum constraints that must be satisfied. The values of the variables must be chosen so that the number of units of nutrient A consumed daily by each sheep is equal to or greater than 110. Expressing this in terms of the variables yields

$20x_1 30x_2 70x_3 \quad 110$

The constraints for the other nutrients are

$10x_1 10x_2 \quad 18$

$50x_1 30x_2 \quad 90$

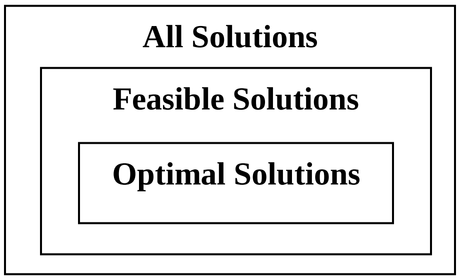
$6x_1 2.5x_2 10x_3 \quad 110$ and

finally all xj s 0

The optimal solution to this problem (obtained using a computer software package) is x_1= 0.595, x_2 =2.008, x_3 =0.541 and z 148.6 cents.

THE GEOMETRY OF LINEAR PROGRAMS

The characteristic that makes linear programs easy to solve is their simple geometric structure. Let's define some terminology. A solution for a linear program is any set of numerical values for the variables. These values need not be the best values and do not even have to satisfy the constraints or make sense. For example, in the Healthy Pet Food problem, M = 25 and Y = -800 is a solution, but it does not satisfy the constraints, nor does it make physical sense. A feasible solution is a solution that satisfies all of the constraints. The feasible set or feasible region is the set of all feasible solutions. Finally, an optimal solution is the feasible solution that produces the best objective function value possible.



The above figure shows the relationships among these types of solutions. Let's use the Healthy Pet Food example to show the geometry of linear programs and to show how two-variable problems can be solved graphically. The linear programming formulation for the Healthy Pet Food problem is:

Maximize z 0.65M 0.45Y

Subject to $2M + 3Y \leq 400{,}000$

$3M + 1.5Y \leq 300{,}000$

$M \leq 90{,}000$

$M, Y \geq 0$

THE SIMPLEX ALGORITHM

In 1949, George Dantzig developed an efficient procedure for solving linear programs called the **simplex method** or **simplex algorithm**. This is the most widely used method in instructional and commercial computer packages. A method developed in 1984 is gaining popularity, but since it requires more sophisticated mathematics.

- The fundamental theorem of linear programming reduces to a finite value the number of feasible solutions that need to be evaluated. One solution strategy might be to identify the coordinates of every extreme point and then evaluate the objective function at each. The one that produces the best objective function value is the optimum. In practice, this approach is not efficient because the number of extreme points can be very large for real problems with hundreds or thousands of variables and constraints.

- The simplex algorithm begins by identifying an initial extreme point of the feasible set. The algorithm then looks along each edge intersecting at the extreme point and computes the net effect on the objective function if we were to move along the edge.

- If the objective function value does not improve by moving along at least one of these edges, it can be proved that the extreme point is optimal. If movement along one or more of the edges improves the objective function value, we move along one of these edges until we reach a new extreme point. We repeat the previous steps: checking along each edge intersecting at the extreme point and then either stopping or sliding along another edge that improves the objective function value.

This algorithm has many desirable features in practice.
1. It only moves from one extreme point to a better or equally good extreme point, thereby skipping large numbers of suboptimal extreme points without explicitly identifying them. Thus it usually only has to check a small subset of the extreme points to find an optimum.
2. When it finds an optimum, it identifies this fact and stops.
3. The algorithm detects whether the problem is infeasible, is unbounded, or has multiple optima.
4. The algorithm uses very simple mathematics that are easy to implement on a computer.

NP hard and NP Complete problems:

Basic Concepts
The computing times of algorithms fall into two groups.
Group1– consists of problems whose solutions are bounded by the polynomial of small degree.
Example – Binary search o (log n) , sorting o(n log n), matrix multiplication 0(n 2.81).

NP –HARD AND NP – COMPLETE PROBLEMS
Group2 – contains problems whose best known algorithms are non-polynomial.
Example – knapsack problem 0(2n/2) etc. There are two classes of non-polynomial time problems

1. NP- hard
2. NP-complete

A problem which is NP complete will have the property that it can be solved in polynomial time if all other NP – complete problems can also be solved in polynomial time. The class NP (meaning non-deterministic polynomial time) is the set of problems that might appear in a puzzle magazine: ``Nice puzzle."
What makes these problems special is that they might be hard to solve, but a short answer can always be printed in the back and it is easy to see that the answer is correct once you see it.

Example... Does matrix A have LU decomposition
No guarantee if answer is ``no".

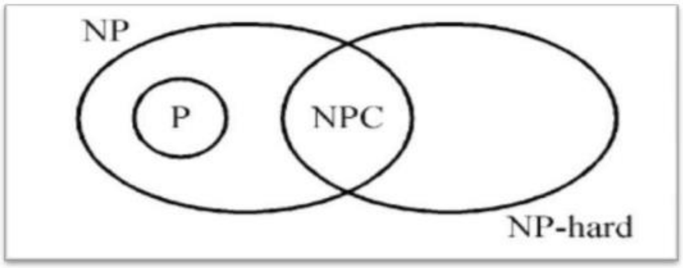
Exponential Upper bound
- Another useful property of the class NP is that all NP problems can be solved in exponential time (EXP).
- This is because we can always list out all short certificates in exponential time and check all O (2nk) of them.
- Thus, P is in NPand NP is in EXP. Although we know that P is not equal to EXP, it is possible that NP = P, or EXP, or neither.

NP-hardness

Some problems are at least as hard to solve as any problem in NP. We call such problems NP-hard.
How might we argue that problem X is at least as hard (to within a polynomial factor) as problem Y?
If X is at least as hard as Y, how would we expect an algorithm that is able to solve X to behave?

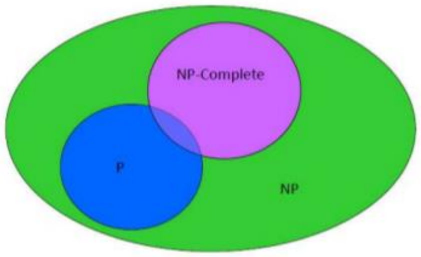
NP –HARD and NP – Complete Problems Basic Concepts:

- If an NP-hard problem can be solved in polynomial time, then all NP-complete problems can be solved in polynomial time.
- All NP-complete problems are NP-hard, but all NP- hard problems are not NP-complete.
- The class of NP-hard problems is very rich in the sense that it contains many problems from a wide variety of disciplines



P: The class of problems which can be solved by a deterministic polynomial algorithm.
NP: The class of decision problem which can be solved by a non-deterministic polynomial algorithm.
NP-hard: The class of problems to which every NP problem reduces
NP-complete (NPC): the class of problems which are NP-hard and belong to NP. NP-Competence

- How we would we define NP-Complete
- They are the "hardest" problems in NP. Below diagram shows the relation between P,NP and NP-Complete problems.



Deterministic and Nondeterministic Algorithms:

- Algorithmswiththepropertythattheresultofeveryoperationisuniquelydefined are termed deterministic.
- Such algorithms agree with the way programs are executed on a computer.
- In a theoretical framework, we can allow algorithms to contain operations whose outcome are not uniquely defined but are limited to a specified set of possibilities.

- The machines executing such operations are allowedtochooseanyoneoftheseoutcomessubjecttoaterminationcondition.
- This leads to the concept of non-deterministic algorithms.
- To specify such algorithms in SPARKS, we introduce three statements Choice (s) ……… arbitrarily chooses one of the elements of the set S. Failure …. Signals an unsuccessful completion. Success: Signals a successful completion.
- Whenever there is a set of choices that leads to a successful completion then one such set of choices is always made and the algorithm terminates.
- A non-deterministic algorithm terminates unsuccessfully if and only if there exists no set of choices leading to a successful signal.

Nondeterministic algorithms

A non deterministic algorithm consists of

Phase 1: guessing

Phase 2: checking

- If the checking stage of a non-deterministic algorithm is of polynomial timecomplexity, then this algorithm is called an NP (nondeterministic polynomial) algorithm.
- NP problems : (must be decision problems)

  e.g. Searching, MST Sorting, Satisfy ability problem (SAT), Travelling salesperson problem (TSP)

Example of a non deterministic algorithm

```
// The problem is to search for an element x //
// Output j such that A(j) =x; or j=0 if x is not in A //
j choice (1 :n ) if A(j) =x then print(j) ;
success endif
print (‘0’) ;
failure complexity 0(1);
```

Non-deterministic decision algorithms

Generate a zero or one as their output.

Deterministic search algorithm complexity. (n)

- Many optimization problems can be recast into decision problems with the property that the decision problem can be solved in polynomial time if and only if the corresponding optimization problem can.
- The decision is to determine if there is a 0/1 assignment of values to xi $1 \leq i \leq n$ such that $\sum$pi xi $\geq$ R and $\sum$ wi xi $\leq$ M, R, M are given numbers pi, wi ≥ 0, $1 \leq i \leq n$.
- It is easy to obtain polynomial time nondeterministic algorithms for many problems that can be deterministically solved by a systematic search of a solutions, pace of exponential size.

Example of NP-Complete and NP-Hard Problem

Chromatic Number Decision Problem (CNP)

i. A coloring of a graph G = (V,E) is a function f : V { 1,2, …, k} i V
ii. If (U, V) E then f(u) f(v).
iii. The CNP is to determine if G has a coloring for a given K.
iv. Satisfiability with at most three literals per clause chromatic number problem. CNP is NP-hard.

Directed Hamiltonian Cycle(DHC)

i. Let G=(V,E) be a directed graph and length n=1V1
ii. The DHC is acycle that goes through every vertex exactly once and then returns to the starting vertex.
iii. The DHC problem is to determine if G has a directed Hamiltonian Cycle.
iv. Theorem: CNF (Conjunctive Normal Form) satisfiability DHC is NP-hard.

Travelling Salesperson Decision Problem (TSP) :

i. The problem is to determine if a complete directed graph G = (V,E) with edge costs C(u,v) has a tour of cost at most M Theorem: Directed Hamiltonian Cycle (DHC) TSP
ii. But from problem (2) satisfiability DHC Satisfiability TSP, TSP is NP-hard.

➢ CNF-Satisfiability with at most three literals per clause is NP-hard. If each clause is restricted to have at most two literals then CNF-satisfiability is polynomial solvable.

➢ Generating optimal code for a parallel assignment statement is NP-hard, However if the expressions are restricted to be simple variables, then optimal code can be generated in polynomial time.

➢ Generating optimal code for level one directed a-cyclic graphs is NP-hard but optimal code for trees can be generated in polynomial time. Determining if a planner graph is three colorable is NP-Hard. To determine if it is two colorable is a polynomial complexity problem. (It only have to see if it is bipartite)

General Definitions

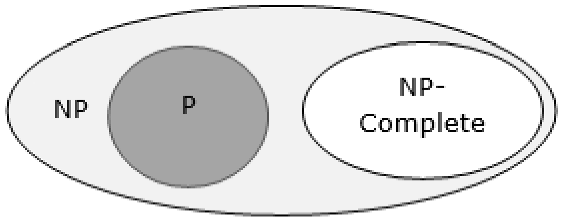
P, NP, NP-hard, NP-easy and NP-complete... - Polynomial-time reduction Examples of NP-complete problems

- P - Decision problems (decision problems) that can be solved in polynomial time - can be solved “efficiently”.
- NP - Decision problems whose “YES” answer can be verified in polynomial time, if we already have the proof (or witness).
- Co-NP - Decision problems whose “NO” answer can be verified in polynomial time, if we already have the proof (or witness) E.g. the satisfy ability problem (SAT) - Given a Boolean formula is it possible to assign the input x1...x9, so that the formula evaluates to TRUE?

- If the answer is YES with a proof (i.e. an assignment of input value), then we can check the proof in polynomial time (SAT is in NP). We may not be able to check the NO answer in polynomial time. (Nobody really knows.)
- NP-hard - A problem is NP-hard if and only if an polynomial-time algorithm for it implies a polynomial-time algorithm for every problem in NPNP-hard problems are at least as hard as NP problems
- NP-complete - A problem is NP-complete if it is NP-hardand is an element of NP.

PROOF OF NP-COMPLETE AND NP-HARD PROBLEM

A problem is in the class NPC if it is in NP and is as **hard** as any problem in NP. A problem is **NP-hard** if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself.



If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called **NP-complete**. The phenomenon of NP-completeness is important for both theoretical and practical reasons.

Definition of NP-Completeness

A language **B** is ***NP-complete*** if it satisfies two conditions

- **B** is in NP
- Every **A** in NP is polynomial time reducible to **B**.

If a language satisfies the second property, but not necessarily the first one, the language **B** is known as **NP-Hard**. Informally, a search problem **B** is **NP-Hard** if there exists some **NP-Complete** problem **A** that Turing reduces to **B**.

The problem in NP-Hard cannot be solved in polynomial time, until **P = NP**. If a problem is proved to be NPC, there is no need to waste time on trying to find an efficient algorithm for it. Instead, we can focus on design approximation algorithm.

NP-Complete Problems

Following are some NP-Complete problems, for which no polynomial time algorithm is known.

- Determining whether a graph has a Hamiltonian cycle
- Determining whether a Boolean formula is satisfiable, etc.

NP-Hard Problems

The following problems are NP-Hard

- The circuit-satisfiability problem

- Set Cover
- Vertex Cover
- Travelling Salesman Problem

In this context, now we will discuss TSP is NP-Complete

TSP is NP-Complete

The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain one and returning to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip

Proof

To prove ***TSP is NP-Complete***, first we have to prove that ***TSP belongs to NP***. In TSP, we find a tour and check that the tour contains each vertex once. Then the total cost of the edges of the tour is calculated. Finally, we check if the cost is minimum. This can be completed in polynomial time. Thus ***TSP belongs to NP***.

Secondly, we have to prove that ***TSP is NP-hard***. To prove this, one way is to show that ***Hamiltonian cycle*** $\leq_p$ ***TSP*** (as we know that the Hamiltonian cycle problem is NP-complete).

Assume ***G = (V, E)*** to be an instance of Hamiltonian cycle.

Hence, an instance of TSP is constructed. We create the complete graph ***G' = (V, E')***, where

E′={(i,j):i,j∈Vandi≠jE′={(i,j):i,j∈Vandi≠j

Thus, the cost function is defined as follows −

t(i,j)={01if(i,j)∈Eotherwiset(i,j)={0if(i,j)∈E1otherwise

Now, suppose that a Hamiltonian cycle ***h*** exists in ***G***. It is clear that the cost of each edge in ***h*** is **0** in ***G'*** as each edge belongs to ***E***. Therefore, ***h*** has a cost of **0** in ***G'***. Thus, if graph ***G*** has a Hamiltonian cycle, then graph ***G'*** has a tour of **0** cost.

Conversely, we assume that ***G'*** has a tour ***h'*** of cost at most **0**. The cost of edges in ***E'*** are **0** and **1** by definition. Hence, each edge must have a cost of **0** as the cost of ***h'*** is **0**. We therefore conclude that ***h'*** contains only edges in ***E***.We have thus proven that ***G*** has a Hamiltonian cycle, if and only if ***G'*** has a tour of cost at most **0**. TSP is NP-complete.

APPROXIMATION ALGORITHM:

Introduction:

An Approximate Algorithm is a way of approach **NP-COMPLETENESS** for the optimization problem.The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at the most polynomial time. Such algorithms are called approximation algorithm or heuristic algorithm.

- For the traveling salesperson problem, the optimization problem is to find the shortest cycle, and the approximation problem is to find a short cycle.
- For the vertex cover problem, the optimization problem is to find the vertex cover with fewest vertices and the approximation problem is to find the vertex cover with few vertices.

Performance Ratios:

Suppose we work on an optimization problem where every solution carries a cost. An Approximate Algorithm returns a legal solution, but the cost of that legal solution may not be optimal.

- For Example, suppose we are considering for a **minimum size vertex-cover (VC)**. An approximate algorithm returns a VC for us, but the size (cost) may not be minimized.
- Another Example is we are considering for a **maximum size Independent set (IS)**. An approximate algorithm returns an IS for us, but the size (cost) may not be maximum. Let C be the cost of the solution returned by an approximate algorithm and C* is the cost of the optimal solution. We say the approximate algorithm has an approximate ratio P (n) for an input size n, where

$$\boxed{\max\left(\frac{C}{C^*},\frac{C^*}{C}\right) \leq P(n)}$$

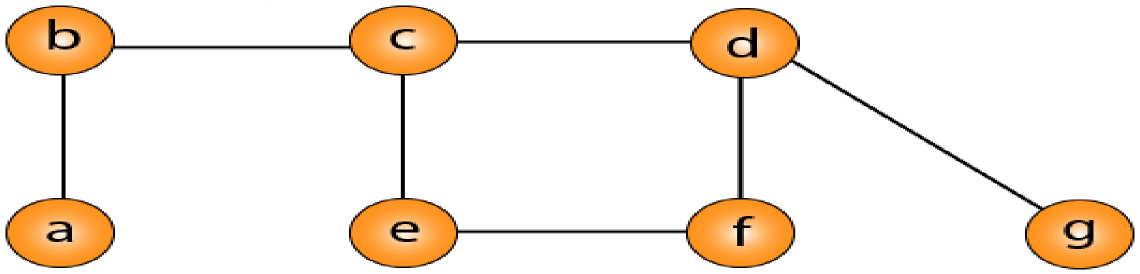
Intuitively, the approximation ratio measures how bad the approximate solution is distinguished with the optimal solution. A large (small) approximation ratio measures the solution is much worse than (more or less the same as) an optimal solution.

Observe that P (n) is always ≥ 1, if the ratio does not depend on n, we may write P. Therefore, a 1-approximation algorithm gives an optimal solution. Some problems have polynomial-time approximation algorithm with small constant approximate ratios, while others have best-known polynomial time approximation algorithms whose approximate ratios grow with n.

Vertex Cover

A Vertex Cover of a graph G is a set of vertices such that each edge in G is incident to at least one of these vertices. The decision vertex-cover problem was proven NPC. Now, we want to solve the optimal version of the vertex cover problem, i.e., we want to find a minimum size vertex cover of a given graph. We call such vertex cover an optimal vertex cover C*.

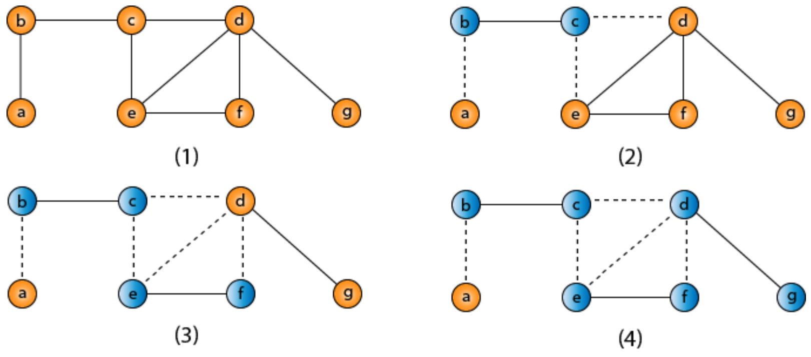
An approximate algorithm for vertex cover:



```
Approx-Vertex-Cover (G = (V, E))
{
   C = empty-set;
   E'= E;
   While E' is not empty do
     {
  Let (u, v) be any edge in E': (*)
```

```
    Add u and v to C;
    Remove from E' all edges incident to
      u or v;
     }
    Return C;
}
```

The idea is to take an edge (u, v) one by one, put both vertices to C, and remove all the edges incident to u or v. We carry on until all edges have been removed. C is a VC. But how good is C?



VC = {b, c, d, e, f, g}

Traveling-salesman Problem

- In the traveling salesman Problem, a salesman must visits n cities. We can say that salesman wishes to make a tour or Hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. There is a non-negative cost c (i, j) to travel from the city i to city j.
- The goal is to find a tour of minimum cost. We assume that every two cities are connected. Such problems are called Traveling-salesman problem (TSP).
- We can model the cities as a complete graph of n vertices, where each vertex represents a city.It can be shown that TSP is NPC.
- If we assume the cost function c satisfies the triangle inequality, then we can use the following approximate algorithm.

Triangle inequality

Let u, v, w be any three vertices, we have

$$\boxed{c\ (u, w) \le c\ (u, v) + c\ (v, w)}$$

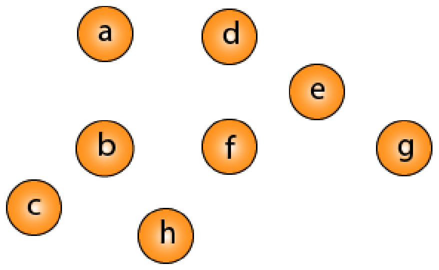
One important observation to develop an approximate solution is if we remove an edge from H*, then the tour becomes a spanning tree.

Algorithm:

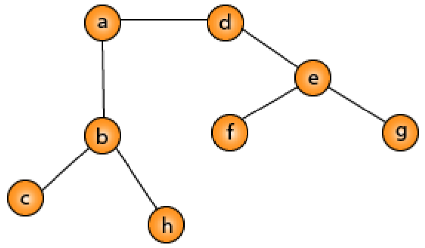
```
Approx-TSP (G= (V, E))
{ 1. Compute a MST T of G;
  2. Select any vertex r is the root of the tree;
  3. Let L be the list of vertices visited in a preorder tree walk of T;
```

4. Return the Hamiltonian cycle H that visits the vertices in the order L;
}

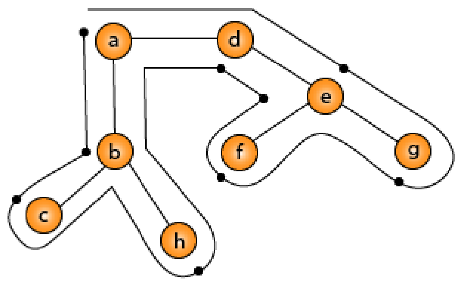
Traveling-Salesman Problem



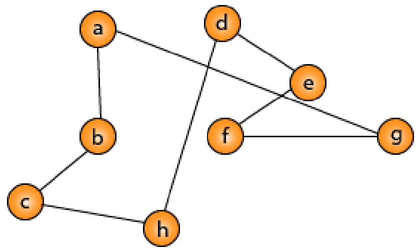
(1) A given set of points



(2) MST T



(3) Full tree walk on T.



(4) A preorder sequence gives a tour H.

Intuitively, Approx-TSP first makes a full walk of MST T, which visits each edge exactly two times. To create a Hamiltonian cycle from the full walk, it bypasses some vertices (which corresponds to making a shortcut)

RANDOMIZED ALGORITHM

What is a Randomized Algorithm?

- An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm.
- For example, in Randomized Quick Sort, we use random number to pick the next pivot (or we randomly shuffle the array) and in Karger's algorithm, we randomly pick an edge.

How to analyse Randomized Algorithms?

- Some randomized algorithms have deterministic time complexity. For example, this implementation of Karger's algorithm has time complexity as O(E).
- Such algorithms are called Monte Carlo Algorithms and are easier to analyse for worst case.
- On the other hand, time complexity of other randomized algorithms is dependent on value of random variable. Such Randomized algorithms are called Las Vegas Algorithms.
- These algorithms are typically analysed for expected worst case. To compute expected time taken in worst case, all possible values of the used random variable needs to be considered in worst case and time taken by every possible value needs to be evaluated.

- Average of all evaluated times is the expected worst case time complexity. Below facts are generally helpful in analysis of such algorithms. Linearity of expectation expected number of trials until Success.

For example consider below a randomized version of Quick Sort.

A **Central Pivot** is a pivot that divides the array in such a way that one side has at-least 1/4 elements.

```
// Sorts an array arr[low..high]randQuickSort(arr[], low, high)
1. If low >= high, then EXIT.
2. While pivot 'x' is not a Central Pivot.
  (i)   Choose uniformly at random a number from [low..high].
        Let the randomly picked numberbex.
  (ii)  Count elements in arr[low..high] that are smaller
thanarr[x]. Let this count be sc.
  (iii) Count elements in arr[low..high] that are greater
thanarr[x]. Let this count be gc.
(iv)  Letn = (high-low+1). If sc>= n/4 and
gc>= n/4, then x is a central pivot.
3. Partition arr[low..high] around the pivot x.
4. // Recur for smaller elements
randQuickSort(arr, low, sc-1)
5. // Recur for greater elements
randQuickSort(arr, high-gc+1, high)
```

Based on analysis is, the time taken by step 2 is O(n).

How many times while loop runs before finding a central pivot?

The probability that the randomly chosen element is central pivot is 1/2.

- Therefore, expected number of times the while loop runs is 2
- Thus, the expected time complexity of step 2 is O(n)

What is overall Time Complexity in Worst Case?

- In worst case, each partition divides array such that one side has n/4 elements and other side has 3n/4 elements. The worst case height of recursion tree is Log 3/4 n which is O(Log n).
  $T(n) < T(n/4) + T(3n/4) + O(n)$
  $T(n) < 2T(3n/4) + O(n)$
- Solution of above recurrence is O(n Log n)
- Note that the above randomized algorithm is not the best way to implement randomized Quick Sort. The idea here is to simplify the analysis as it is simple to analyse.
- Typically, randomized Quick Sort is implemented by randomly picking a pivot (no loop) or by shuffling array elements. Expected worst case time complexity of this algorithm is also O(n Log n), but analysis is complex.

INTERIOR POINT METHOD

Interior-point methods (also referred to as **barrier methods** or **IPMs**) are a certain class of algorithms that solve linear and nonlinear convex optimization problems.

Example: John von Neumann suggested an interior-point method of linear programming, which was neither a polynomial-time method nor an efficient method in practice. In fact, it turned out to be slower than the commonly used simplex method.

- An interior point method discovered by Soviet mathematician I. I. Dikin in 1967 and reinvented in the U.S. in the mid-1980s. In 1984, NarendraKarmarkar developed a method for linear programming called Karmarkar's algorithm, which runs in provably polynomial time and is also very efficient in practice.
- It enabled solutions of linear programming problems that were beyond the capabilities of the simplex method. Contrary to the simplex method, it reaches a best solution by traversing the interior of the feasible region. The method can be generalized to convex programming based on a self-concordant barrier function used to encode the convex set.
- Any convex optimization problem can be transformed into minimizing (or maximizing) a linear function over a convex set by converting to the epigraph form.
- A special class of such barriers that can be used to encode any convex set. They guarantee that the number of iterations of the algorithm is bounded by a polynomial in the dimension and accuracy of the solution.
- Karmarkar's breakthrough revitalized the study of interior-point methods and barrier problems, showing that it was possible to create an algorithm for linear programming characterized by polynomial complexity and, moreover, that was competitive with the simplex method. Already Khachiyan's ellipsoid method was a polynomial-time algorithm; however, it was too slow to be of practical interest.
- The class of primal-dual path-following interior-point methods is considered the most successful. Mehrotra's predictor–corrector algorithm provides the basis for most implementations of this class of methods.

Primal dual interior Point Method for non-linear Optimization

The primal-dual method's idea is easy to demonstrate for constrained non-linear optimization. For simplicity, consider the all-inequality version of a non-linear optimization problem:

minimize $f(x)$ subject

to $c_i(x) \geq 0 \text{ for } i = 1, \ldots, m, \; x \in \mathbb{R}^n,$ where

$f : \mathbb{R}^n \to \mathbb{R}, c_i : \mathbb{R}^n \to \mathbb{R} \quad (1).$

The logarithmic barrier function associated with (1) is

$$B(x,\mu) = f(x) - \mu \sum_{i=1}^{m} \log(c_i(x)). \quad (2)$$

- Here μ is a small positive scalar, sometimes called the "barrier parameter". As μ converges to zero the minimum of B(x,μ) should converge to a solution of (1).

The barrier function gradient is

$$g_b = g - \mu \sum_{i=1}^{m} \frac{1}{c_i(x)} \nabla c_i(x), \quad (3)$$

where g is the gradient of the original function f(x) .

ADVANCED NUMBER THEORY ALGORITHMS

1. All 4 digit palindromic numbers are divisible by 11.
2. If we repeat a three-digit number twice, to form a six-digit number. The result will be divisible by 7, 11 and 13 and dividing by all three will give our original three-digit number.
3. A number of form 2N has exactly N+1 divisors. For example 4 has 3 divisors, 1, 2 and 4.
4. To calculate sum of factors of a number, we can find the number of prime factors and their exponents. Let p1, p2, …pk be prime factors of n. Let a1, a2, ..ak be highest powers of p1, p2, .. pk respectively that divide n, i.e., we can write n as **n = (p1a1)*(p2a2)* … (pkak)**.Sum of divisors = (1 + p1 + p12 ... p1a1) * (1 + p2 + p22 ... p2a2) * .... (1 + pk + pk2 ... pkak)

We can notice that individual terms of above formula are Geometric Progressions (GP). Wecan rewrite the formula as.

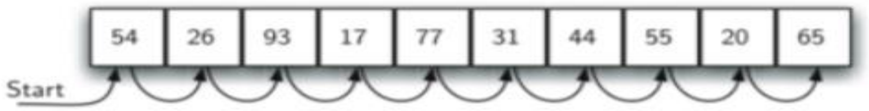
Sum of divisors = (p1a1+1 - 1)/(p1 -1) *(p2a2+1 - 1)/(p2 -1) *............................ (pkak+1 - 1)/(pk -1)

1. For a product of N numbers, if we have to subtract a constant K such that the product gets its maximum value, then subtract it from a largest value such that largest value-k is greater than 0. If we have to subtract a constant K such that the product gets its minimum value, then subtract it from the smallest value where smallest value-k should be greater than 0.
2. **Goldbech`s conjecture:** Every even integer greater than 2 can be expressed as the sum of 2 primes.
3. **Perfect numbers or Amicable numbers:** Perfect numbers are those numbers which are equal to the sum of their proper divisors. Example: 6 = 1 + 2 + 3
4. **Lychrel numbers:** Are those numbers that cannot form a palindrome when repeatedly reversed and added to itself. For example 47 is not a Lychrel Number as 47 + 74 = 121
5. **Lemoine's Conjecture :** Any odd integer greater than 5 can be expressed as a sum of an odd prime (all primes other than 2 are odd) and an even semi-prime. A semi-prime number is a product of two prime numbers. This is called Lemoine's conjecture.
6. **Fermat's Last Theorem :** According to the theorem, no three positive integers a, b, c satisfy the equation, $a^n + b^n = c^n$ for any integer value of n greater than 2. For n = 1 and n = 2, the equation have infinitely many solutions.

RECENT TRENDS IN PROBLEM SOLVING USING SEARCHING AND SORTING TECHNIQUES IN DATASTRUCTURES:

Searching

Searching is the algorithmic process of finding a particular item in a collection of items. A search typically answers either True or False as to whether the item is present. It turns out that there are many different ways to search for the item. What we are interested in here is how these algorithms work and how they compare to one another.



The Sequential Search:

When data items are stored in a collection such as a list, we say that they have a linear or sequential relationship. Each data item is stored in a position relative to the others. In Python lists, these relative positions are the index values of the individual items. Since these index values are ordered, it is possible for us to visit them in sequence. This process gives rise to our first searching technique, the sequential search. Figure shows how this search works. Starting at the first item in the list, we simply move from item to item, following the underlying sequential ordering until we either find what we are looking for or run out of items. If we run out of items, we have discovered that the item we were searching for was not present.

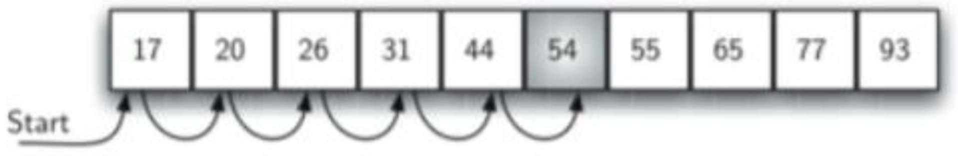
The implementation for this algorithm is shown below. The function needs the list and the item we are looking for and returns a boolean value as to whether it is present. The boolean variable found is initialized to False and is assigned the value True if we discover the item in the list.

```
defsequential_search(a_list, item):
        pos = 0
        found = False
                whilepos<len(a_list) and not found:
                        ifa_list[pos] == item:
                                found = True
                        else: pos = pos+1
                return found
test_list = [1, 2, 32, 8, 17, 19, 42, 13, 0]
print(sequential_search(test_list, 3))
print(sequential_search(test_list, 13))
```

Analysis of Sequential Search

To analyze searching algorithms, we need to decide on a basic unit of computation. Recall that this is typically the common step that must be repeated in order to solve the problem. For searching, it makes sense to count the number of comparisons performed. Each comparison may or may not discover the item we are looking for. In addition, we make another assumption here.

Case	Best Case	Worst Case	Average Case
Item is present	1	n	n/2
Item is not present	n	n	N



The list of items is not ordered in any way. The items have been placed randomly into the list in other words, the probability that the item we are looking for is in any particular position is exactly the same for each position of the list. If the item is not in the list, the only way to know it is to compare it against every item present. If there are n items, then the sequential search requires $n$comparisons to discover that the item is not there. In the case where the item is in the list, the analysis is not so straightforward. There are actually three different scenarios that can occur.

In the best case we will find the item in the first place we look, at the beginning of the list. We will need only one comparison. In the worst case, we will not discover the item until the very last comparison, the nth comparison.

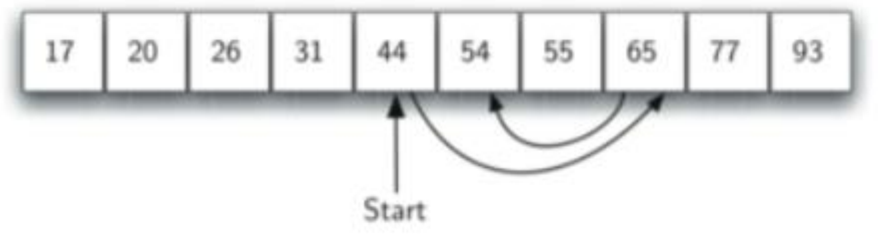
What about the average case? On average, we will find the item about halfway into the list; that is, we will compare against $n/2$ items. Recall, however, that as n gets large, the coefficients, no matter what they are, become insignificant in our approximation, so the complexity of the sequential search, is (n).

Table summarizes these results. We assumed earlier that the items in our collection had been randomly placed so that there is no relative order between the items. What would happen to the sequential search if the items were ordered in some way? Would we be able to gain any efficiency in our search technique? Assume that the list of items was constructed so that the items were in ascending order, from low to high.

If the item we are looking for is present in the list, the chance of it being in any one of the n positions is still the same as before. We will still have the same number of comparisons to find the item. However, if the item is not present there is a slight advantage. Figure shows this process as the algorithm looks for the item 50.Notice that items are still compared in sequence until 54. At this point, however, we know something extra. Not only is 54 not the item we are looking for, but no other elements beyond 54 can work either since the list is sorted. In this case, the algorithm does not have to continue looking through all of the items to report that the item was not found. It can stop immediately.

The Binary Search

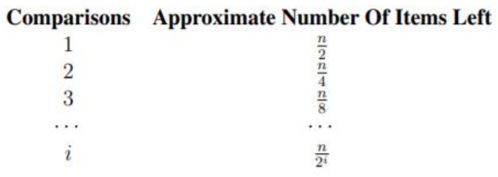
It is possible to take greater advantage of the ordered list if we are clever with our comparisons. In the sequential search, when we compare against the first item, there are at most $n - 1$ more items to look through if the first item is not what we are looking for. Instead of searching the list in sequence, a binary search will start by examining the middle item. If that item is the one we are searching for, we are done.



If it is not the correct item, we can use the ordered nature of the list to eliminate half of the remaining items. If the item we are searching for is greater than the middle item, we know that the entire lower half of the list as well as the middle item can be eliminated from further consideration. The item, if it is in the list, must be in the upper half. We can then repeat the process with the upper half. Start at the middle item and compare it against what we are looking for. Again, we either find it or split the list in half, therefore eliminating another large part of our possible search space. Figure shows how this algorithm can quickly find the value 54.

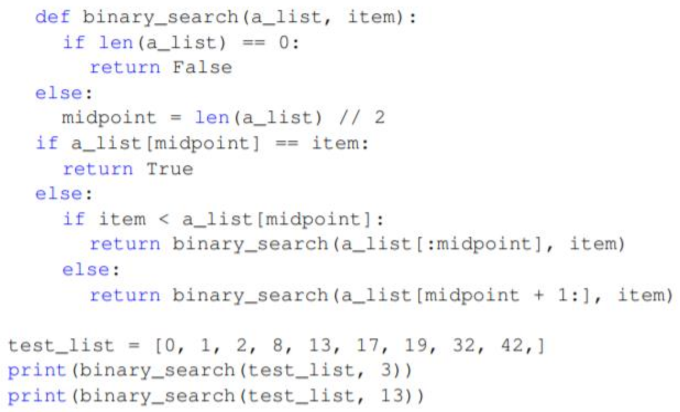
```
def binary_search(a_list, item):
   first = 0
   last = len(a_list) - 1
   found = False

   while first <= last and not found:
      midpoint = (first + last) // 2
      if a_list[midpoint] == item:
         found = True
      else:
         if item < a_list[midpoint]:
            last = midpoint - 1
      else:
         first = midpoint + 1
   return found

test_list = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binary_search(test_list, 3))
print(binary_search(test_list, 13))
```

Before we move on to the analysis, we should note that this algorithm is a great example of a divide and conquer strategy. Divide and conquer means that we divide the problem into smaller pieces, solve the smaller pieces in some way and then reassemble the whole problem to get the result. When we perform a binary search of a list, we first check the middle item. If the item we are searching for is less than the middle item, we can simply perform a binary search of the left half of the original list. Likewise, if the item is greater, we can perform a binary search of the right half. Either way, this is a recursive call to the binary search function passing a smaller list.



Comparisons	Approximate Number Of Items Left
1	$\frac{n}{2}$
2	$\frac{n}{4}$
3	$\frac{n}{8}$
...	...
i	$\frac{n}{2^i}$

Table 5.3: Tabular Analysis for a Binary Search



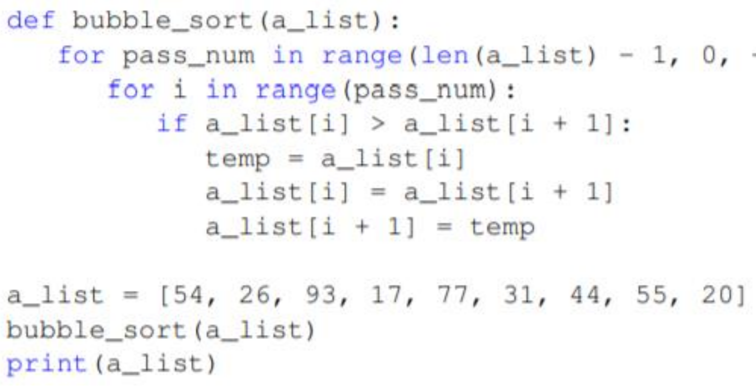
```
  def binary_search(a_list, item):
    if len(a_list) == 0:
      return False
  else:
    midpoint = len(a_list) // 2
  if a_list[midpoint] == item:
    return True
  else:
    if item < a_list[midpoint]:
      return binary_search(a_list[:midpoint], item)
    else:
      return binary_search(a_list[midpoint + 1:], item)

test_list = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binary_search(test_list, 3))
print(binary_search(test_list, 13))
```

Analysis of Binary Search:

To analyze the binary search algorithm, we need to recall that each comparison eliminates about half of the remaining items from consideration. What is the maximum number of comparisons this algorithm will require to check the entire list? If we start with n items, about $n/2$ items will be left after the first comparison. After the second comparison, there will be about $n/4$ .Then $n/8$,n/16 and so on. How many times can we split the list? Table 5.3 helps us to see the answer. When we split the list enough times, we end up with a list that has just one item. Either that is the item we are looking for or it is not. Either way, we are done. The number of comparisons necessary to get to this point is i where n 2 i = 1. Solving for i gives us i = log n. The maximum number of comparisons is logarithmic with respect to the number of items in the list. Therefore, the time complexity for binary search is (log n).Even though a binary search is generally better than a sequential search, it is important to note that for small values of n, the additional cost of sorting is probably not worth it. In fact, we should always consider whether it is cost effective to take on the extra work of sorting to gain searching benefits. If we can sort once and then search many times, the cost of the sort is not so significant. However, for large lists, sorting even once can be so expensive that simply performing a sequential search from the start may be the best choice.

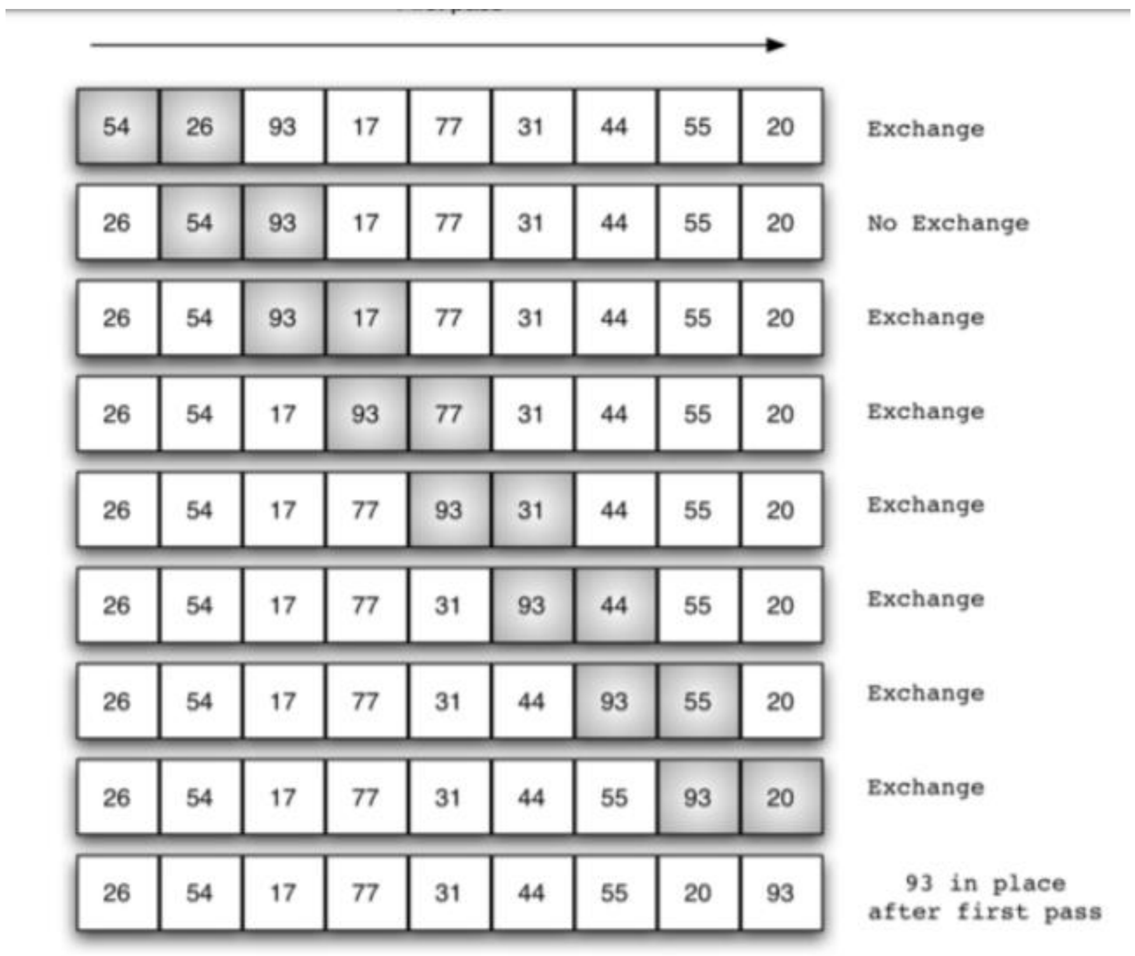
Sorting
Sorting is the process of placing elements from a collection in some kind of order. For example, a list of words could be sorted alphabetically or by length. A list of cities could be sorted by population, by area, or by zip code. There are many, many sorting algorithms that have been developed and analyzed. This suggests that sorting is an important area of study in computer science. Sorting a large number of items can take a substantial amount of computing resources. Like searching, the efficiency of a sorting algorithm is related to the number of items being processed. For small collections, a complex sorting method may be more trouble than it is worth. The overhead may be too high. On the other hand, for larger collections, we want to take advantage of as many improvements as possible. In this section we will discuss several sorting techniques and compare them with respect to their running time. Before getting into specific algorithms, we should think about the operations that can be used to analyze a sorting process. First, it will be necessary to compare two values to see which is smaller (or larger). In order to sort a collection, it will be necessary to have some systematic way to compare values to see if they are out of order. The total number of comparisons will be the most common way to measure a sort procedure. Second, when values are not in the correct position with respect to one another, it may be necessary to exchange them. This exchange is a costly operation and the total number of exchanges will also be important for evaluating the overall efficiency of the algorithm.

1. Bubble Sort:
The bubble sort makes multiple passes through a list. It compares adjacent items and exchanges those that are out of order. Each pass through the list places the next largest value in its proper place. In essence, each item "bubbles" up to the location where it belongs. Figure shows the first pass of a bubble sort. The shaded items are being compared to see if they are out of order. If there are n items in the list, then there are □ − 1 pairs of items that need to be compared on the first pass. It is important to note that once the largest value in the list is part of a pair, it will continually be moved along until the pass is complete. At the start of the second pass, the largest value is now in place. There are □ − 1 items left to sort, meaning that there will be □ − 2 pairs. Since each pass places the next largest value in place, the total number of passes necessary will be □ − 1. After completing the □ − 1 passes, the smallest item must be in the correct position with no further processing required. The code below shows the complete bubble_sort function. It takes the list as a parameter and modifies it by exchanging items as necessary.



```
def bubble_sort(a_list):
   for pass_num in range(len(a_list) - 1, 0, -1):
      for i in range(pass_num):
         if a_list[i] > a_list[i + 1]:
            temp = a_list[i]
            a_list[i] = a_list[i + 1]
            a_list[i + 1] = temp

a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
bubble_sort(a_list)
print(a_list)
```

The exchange operation, sometimes called a "swap," is slightly different in Python than in most other programming languages. Typically, swapping two elements in a list requires a temporary storage location (an additional memory location),will exchange the ith and □th items in the list. Without the temporary storage, one of the values would be overwritten.

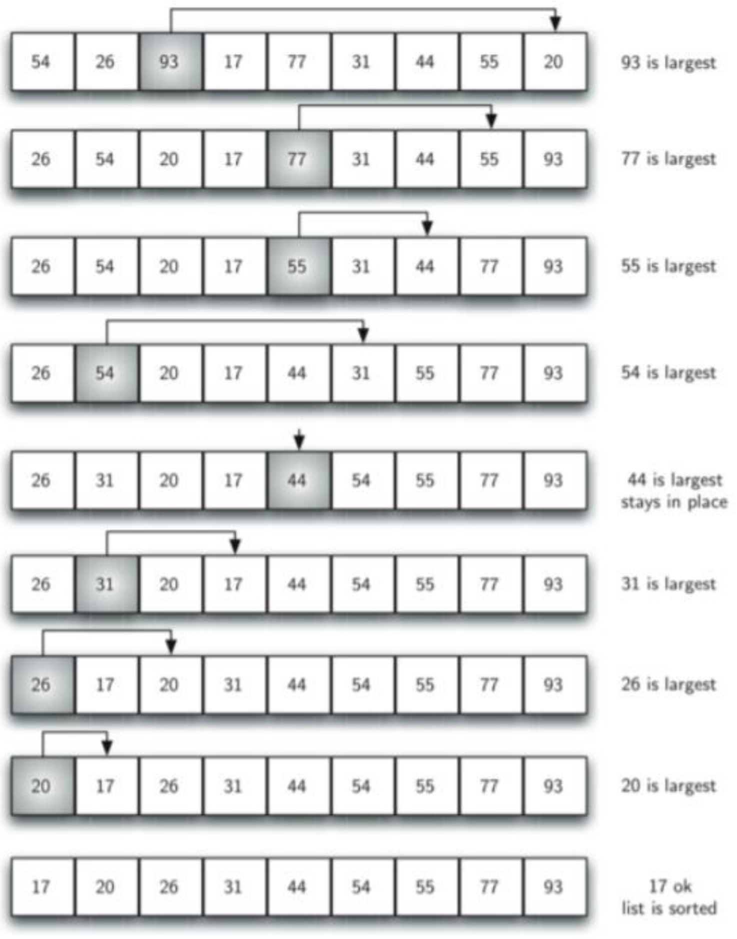


2. Selection Sort:

The selection sort improves on the bubble sort by making only one exchange for every pass through the list. In order to do this, a selection sort looks for the largest value as it makes a pass and after completing the pass, places it in the proper location. As with a bubble sort, after the first pass, the largest item is in the correct place. After the second pass, the next largest is in place. This process continues and requires □ − 1 passes to sort □ items, since the final item must be in place after the (□ − 1)st pass. Figure shows the entire sorting process. On each pass, the largest remaining item is selected and then placed in its proper location. The first pass places 93, the second pass places 77, the third places 55, and so on. The function is shown below.

```
def selection_sort(a_list):
   for fill_slot in range(len(a_list) - 1, 0, -1):
      pos_of_max = 0
      for location in range(1, fill_slot + 1):
         if a_list[location] > a_list[pos_of_max]:
            pos_of_max = location

      temp = a_list[fill_slot]
      a_list[fill_slot] = a_list[pos_of_max]
      a_list[pos_of_max] = temp

a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
selection_sort(a_list)
print(a_list)
```

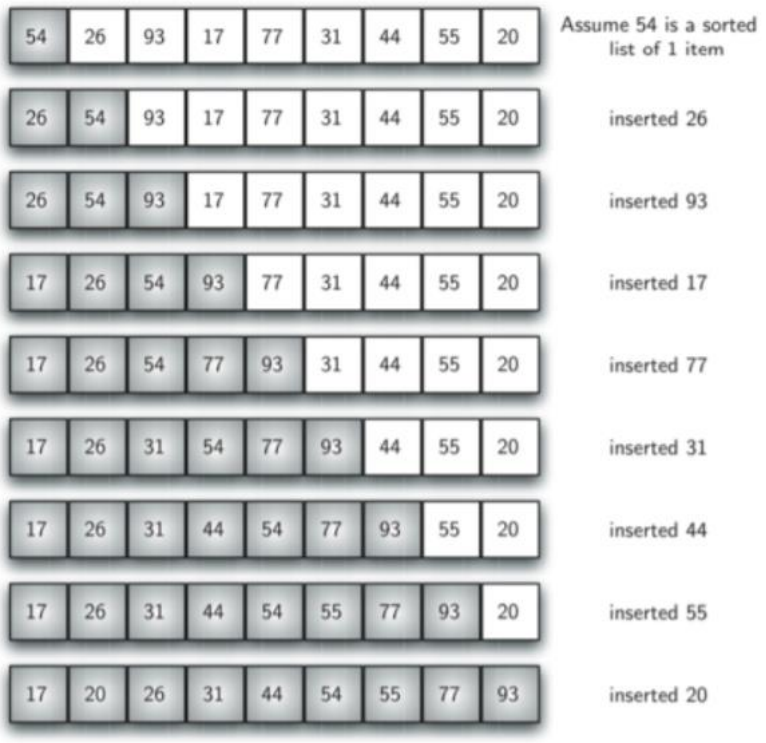
The selection sort makes the same number of comparisons as the bubble sort and is therefore also (□2). However, due to the reduction in the number of exchanges, the selection sort typically executes faster in benchmark studies. In fact, for our list, the bubble sort makes 20 exchanges, while the selection sort makes only 8



3. Insertion Sort:

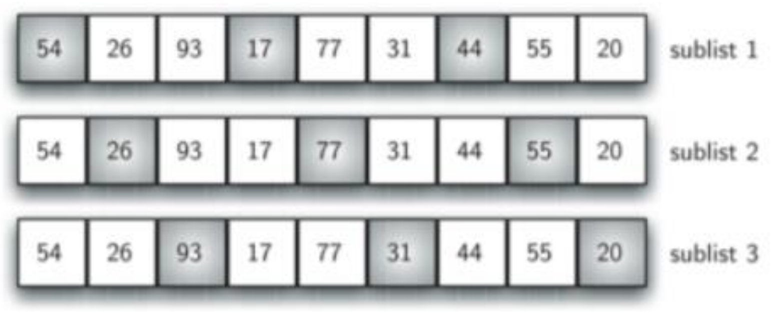
The insertion sort, although still, works in a slightly different way. It always maintains a sorted sub-list in the lower positions of the list. Each new item is then “inserted” back into the previous sub-list such that the sorted sub-list is one item larger. Figure shows the insertion sorting process. The shaded items represent the ordered sub-lists as the algorithm makes each pass. We begin by assuming that a list with one item (position 0) is already sorted. On each pass, one for each item 1 through, the current item is checked against those in the already sorted sub-list. As we look back into the already sorted sub-list, we shift those items that are greater to the right. When we reach a smaller item or the end of the sub-list, the current item can be inserted. Figure below shows the fifth pass in detail. At this point in the algorithm, a sorted sub-list of five items consisting of 17, 26, 54, 77, and 93 exists. We want to insert 31 back into the already sorted items. The first comparison against 93 causes 93 to be shifted to the right. 77 and 54 are also shifted. When the item 26 is encountered, the shifting process stops and 31 is placed in the open position. Now we have a sorted sub-list of six items. The implementation of insertion sort shows that there are again n − 1 passes to sort n items. The iteration starts at position 1 and moves through position n − 1, as these are the items that need to be inserted back into the sorted sub-lists. Line 8 performs the shift operation that moves a value up one position in the list, making room behind it for the insertion. Remember that this is not a complete exchange as was performed in the previous algorithms. The maximum number of comparisons for an insertion sort is the sum of the first n−1 integers. Again, this is (n2). However, in the best case, only one comparison needs to be done on each pass. This would be the case for an already sorted list. One note about shifting versus exchanging is also important. In general, a shift operation requires approximately a third of the processing work of an exchange since only one assignment is performed. In benchmark studies, insertion sort will show very good performance.

```
def insertion_sort(a_list):
  for index in range(1, len(a_list)):

    current_value = a_list[index]
    position = index

    while position > 0 and a_list[position - 1] > current_value:
       a_list[position] = a_list[position - 1]
       position = position - 1

    a_list[position] = current_value

a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
insertion_sort(a_list)
print(a_list)
```

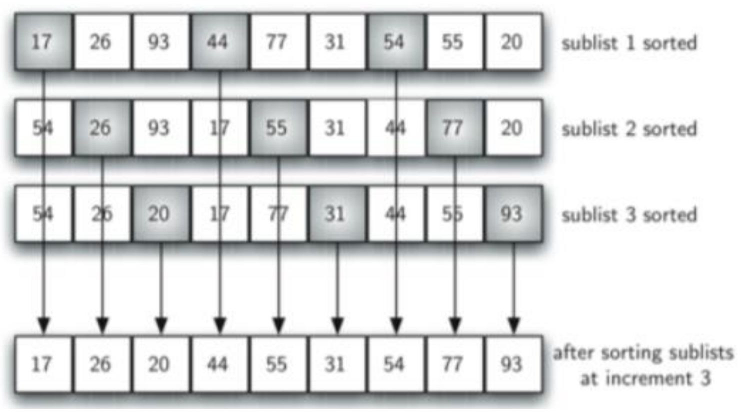


4. Shell Sort:

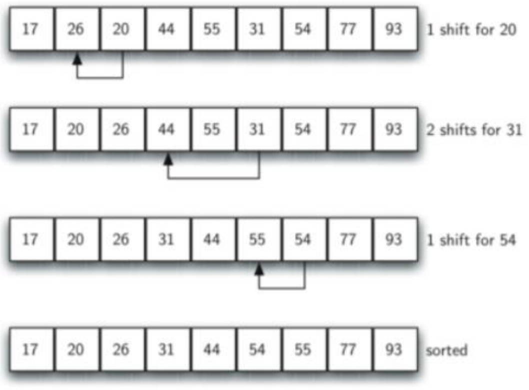
The shell sort, sometimes called the “diminishing increment sort,” improves on the insertion sort by breaking the original list into a number of smaller sub-lists, each of which is sorted using an insertion sort. The unique way that these sub-lists are chosen is the key to the shell sort. Instead of breaking the list into sub-lists of contiguous items, the shell sort uses an increment i, sometimes called the gap, to create a sub-list by choosing all items that are i items apart. This can be seen in figure. This list has nine items. If we use an increment of three, there are three sub-lists, each of which can be sorted by an insertion sort. After completing these sorts, we get the list shown in figure. Although this list is not completely sorted, something very interesting has happened. By sorting the sub-lists, we have moved the items closer to where they actually belong. Figure shows a final insertion sort using an increment of one; in other words, a standard insertion sort. Note that by performing the earlier sub-list sorts, we have now reduced the total number of shifting operations necessary to put the list in its final order. For this case, we need only four more shifts to complete the process. In the way in which the increments are chosen is the unique feature of the shell sort. The function shell sort shown below uses a different set of increments. In this case, we begin with 2 sub-lists. On the next pass, 4 sub-lists are sorted. Eventually, a single list is sorted with the basic insertion sort. Figure shows the first sub-lists for our example using this increment.



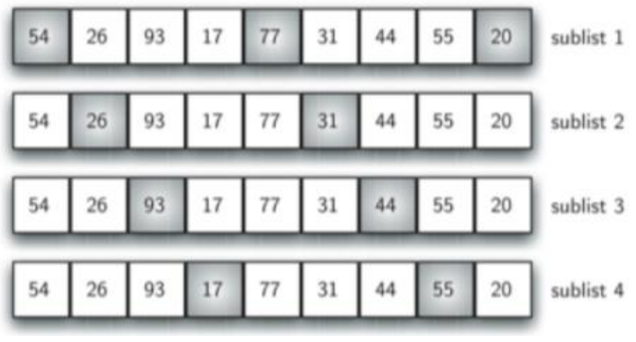
54 26 93 17 77 31 44 55 20
sublist 1
sublist 2
sublist 3



17 26 93 44 77 31 54 55 20
sublist 1 sorted
54 26 93 17 55 31 44 77 20
sublist 2 sorted
54 26 20 17 77 31 44 55 93
sublist 3 sorted
17 26 20 44 55 31 54 77 93
after sorting sublists at increment 3



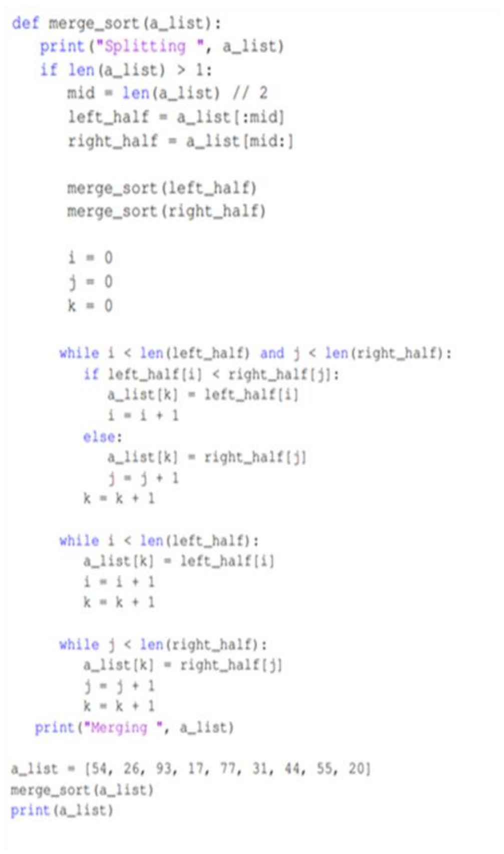
17 26 20 44 55 31 54 77 93
1 shift for 20
17 20 26 44 55 31 54 77 93
2 shifts for 31
17 20 26 31 44 55 54 77 93
1 shift for 54
17 20 26 31 44 54 55 77 93
sorted



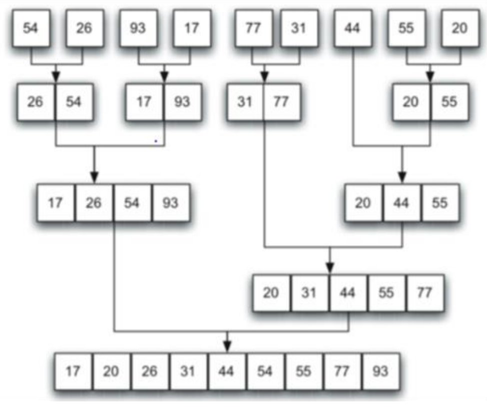
54 26 93 17 77 31 44 55 20
sublist 1
sublist 2
sublist 3
sublist 4

5. **The Merge Sort:**

Merge sort using a divide and conquer strategy as a way to improve the performance of sorting algorithms. Merge sort is a recursive algorithm that continually splits a list in half. If the list is empty or has one item, it is sorted by definition (the base case). If the list has more than one item, we split the list and recursively invoke a merge sort on both halves. Once the two halves are sorted, the fundamental operation, called a merge, is performed. Merging is the process of taking two smaller sorted lists and combining them together into a single, sorted, new list. Figure below shows our familiar example list as it is being split by merge sort. Figure below also shows the simple lists, now sorted, as they are merged back together. The merge sort function shown below begins by asking the base case question. If the length of the list is less than or equal to one, then we already have a sorted list and no more processing is necessary. If, on the other hand, the length is greater than one, then we use the Python slice operation to extract the left and right halves. It is important to note that the list may not have an even number of items. That does not matter, as the lengths will differ by at most one.

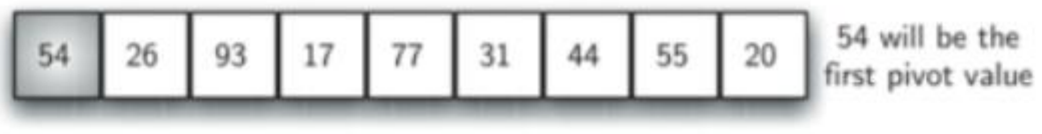


```
def merge_sort(a_list):
   print("Splitting ", a_list)
   if len(a_list) > 1:
      mid = len(a_list) // 2
      left_half = a_list[:mid]
      right_half = a_list[mid:]

      merge_sort(left_half)
      merge_sort(right_half)

      i = 0
      j = 0
      k = 0

     while i < len(left_half) and j < len(right_half):
        if left_half[i] < right_half[j]:
           a_list[k] = left_half[i]
           i = i + 1
        else:
           a_list[k] = right_half[j]
           j = j + 1
        k = k + 1

     while i < len(left_half):
        a_list[k] = left_half[i]
        i = i + 1
        k = k + 1

     while j < len(right_half):
        a_list[k] = right_half[j]
        j = j + 1
        k = k + 1
  print("Merging ", a_list)

a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
merge_sort(a_list)
print(a_list)
```

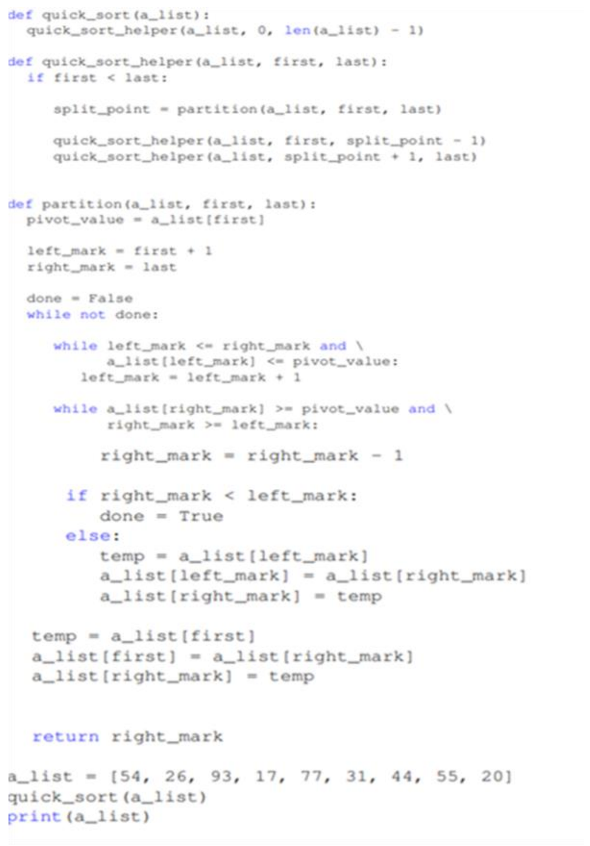


6. **Quick Sort:**

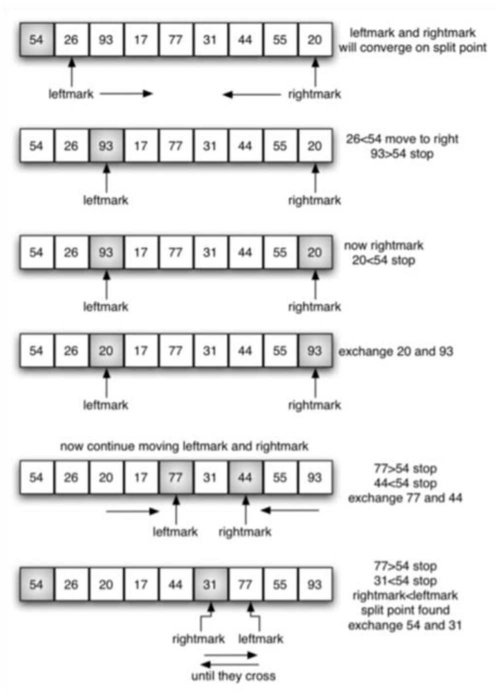
The quick sort uses divide and conquer to gain the same advantages as the merge sort, while not using additional storage. As a trade-off, however, it is possible that the list may not be divided in half. When this happens, we will see that performance is diminished. A quick sort first selects a value, which is called the pivot value. Although there are many different ways to choose the pivot value, we will simply use the first item in the list. The role of the pivot value is to assist with splitting the list. The actual position where the pivot value belongs in the final sorted list, commonly called the split point, will be used to divide the list for subsequent calls to the quick sort. Figure below shows that 54 will serve as our first pivot value. Since we have looked at this example a few times already, we know that 54 will eventually end up in the position currently holding 31. The partition process will happen next. It will find the split point and at the same time move other items to the appropriate side of the list, either less than or greater than the pivot value.

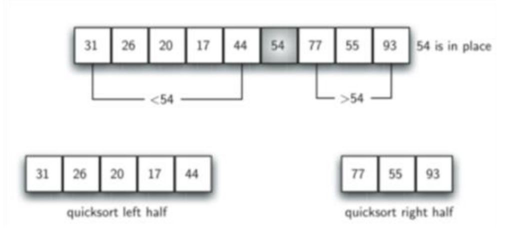


Partitioning begins by locating two position markers – let’s call them left_mark and right_mark – at the beginning and end of the remaining items in the list (positions 1 and 8 in Figure). The goal of the partition process is to move items that are on the wrong side with respect to the pivot value while also converging on the split point. Figure shows this process as we locate the position of 54.We begin by incrementing left_mark until we locate a value that is greater than the pivot value. We then decrement right_mark until we find a value that is less than the pivot value. At this point we have discovered two items that are out of place with respect to the eventual split point. For our example, this occurs at 93 and 20. Now we can exchange these two items and then repeat the process again. At the point where right_mark becomes less than left_mark, we stop. The position of right_mark is now the split point. The pivot value can be exchanged with the contents of the split point and the pivot value is now in place. In addition, all the items to the left of the split point are less than the pivot value and all the items to the right of the split point are greater than the pivot value. The list can now be divided at the split point and the quick sort can be invoked recursively on the two halves. The quick_sort function shown below invokes a recursive function, quick_sort_helper. quick_sort_helper begins with the same base case as the merge sort. If the length of the list is less than or equal to one, it is already sorted. If it is greater, then it can be partitioned and recursively sorted. The partition function implements the process described earlier.



```
def quick_sort(a_list):
  quick_sort_helper(a_list, 0, len(a_list) - 1)

def quick_sort_helper(a_list, first, last):
  if first < last:

     split_point = partition(a_list, first, last)

     quick_sort_helper(a_list, first, split_point - 1)
     quick_sort_helper(a_list, split_point + 1, last)


def partition(a_list, first, last):
  pivot_value = a_list[first]

  left_mark = first + 1
  right_mark = last

  done = False
  while not done:

     while left_mark <= right_mark and \
             a_list[left_mark] <= pivot_value:
         left_mark = left_mark + 1

     while a_list[right_mark] >= pivot_value and \
             right_mark >= left_mark:

           right_mark = right_mark - 1

       if right_mark < left_mark:
           done = True
       else:
           temp = a_list[left_mark]
           a_list[left_mark] = a_list[right_mark]
           a_list[right_mark] = temp

   temp = a_list[first]
   a_list[first] = a_list[right_mark]
   a_list[right_mark] = temp


   return right_mark

a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
quick_sort(a_list)
print(a_list)
```





Web Resources:

Unit I

https://www.slideshare.net/mohamedloey/algorithms-lecture-4-sorting-algorithms-i
http://vssut.ac.in/lecture_notes/lecture1428551222.pdf
https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_selection_sort.htm
https://yourbasic.org/algorithms/graph/
https://www.tutorialspoint.com/index.htm
https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
https://www.geeksforgeeks.org/topological-sorting/
https://www.geeksforgeeks.org/strongly-connected-components/
http://www.csc.kth.se/utbildning/kth/kurser/DD1339/inda14/lecture/PDF/lecture01a-correctness.pdf
https://www.cs.cornell.edu/courses/cs3110/2013sp/supplemental/recitations/rec21.html

Unit II

https://codeforces.com/blog/entry/69287
https://inf.ethz.ch/personal/ladickyl/Matroids.pdf
https://www.guru99.com/greedy-algorithm.html
https://www.geeksforgeeks.org/divide-and-conquer/
https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_greedy_method.htm
https://www.javatpoint.com/minimum-spanning-tree-introduction
https://www.javatpoint.com/applications-of-minimum-spanning-tree
https://www.javatpoint.com/kruskals-minimum-spanning-tree-algorithm
https://www.javatpoint.com/prims-minimum-spanning-tree-algorithm
https://en.wikipedia.org/wiki/Graph_matching
https://www.tutorialspoint.com/graph_theory/graph_theory_matchings.htm
https://www.geeksforgeeks.org/hopcroft-karp-algorithm-for-maximum-matching-set-1-introduction/
https://www.geeksforgeeks.org/maximum-bipartite-matching/
https://www.ti.inf.ethz.ch/ew/courses/GT03/lectures/PDF/lecture5f.pdf
http://www14.in.tum.de/lehre/2014WS/ea/split/sec-Augmenting-Paths-for-Matchings-single.pdf
https://www.eecs.tufts.edu/~gdicks02/Blossom/Blossom_Algorithm.html

Unit III

https://tutorialspoint.dev/data-structure/graph-data-structure/minimum-cut-in-a-directed-graph
https://www.hackerearth.com/practice/algorithms/graphs/maximum-flow/tutorial/
https://cp-algorithms.com/graph/edmonds_karp.html#toc-tgt-5
https://sahebg.github.io/computersceince/maximum-flow-edmond-karp-algorithm-c-program-example/
https://www.geeksforgeeks.org/strassens-matrix-multiplication/

https://shivathudi.github.io/jekyll/update/2017/06/15/matr-mult.html
http://www.facweb.iitkgp.ac.in/~sourav/Lecture-03.pdf
https://developerinsider.co/introduction-to-divide-and-conquer-algorithm-design-paradigm/
https://www.javatpoint.com/divide-and-conquer-introduction
https://www.includehelp.com/algorithms/divide-and-conquer-paradigm.aspx
http://graphics.ics.uci.edu/ICS6N/NewLectures/Lecture5.pdf
https://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations
https://en.wikipedia.org/wiki/LU_decomposition

Unit IV
https://www.javatpoint.com/floyd-warshall-algorithm
https://www.javatpoint.com/dynamic-programming-introduction
http://eiche.theoinf.tu-ilmenau.de/fingerprint/ueber_polynome_modulo-englisch.pdf
http://sabarirb.org/insight/Reading%20Materials/maths/algebra/indet/modrep.htm
http://homepages.math.uic.edu/~leon/mcs425-s08/handouts/chinese_remainder.pdf
http://algo.inria.fr/seminars/sem08-09/kruppa-slides.pdf

Unit V
http://www.uky.edu/~dsianita/300/online/LP.pdf
https://www.techopedia.com/definition/20403/linear-programming-lp
https://optimization.mccormick.northwestern.edu/index.php/Interior-point_method_for_LP
https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_np_hard_complete_classes.htm
https://www.slideshare.net/JyotsnaSuryadevara/9-chapter-8-np-hard-and-np-complete-problems
https://www.javatpoint.com/daa-approximate-algorithms
https://www.cs.auckland.ac.nz/compsci105s1c/resources/ProblemSolvingwithAlgorithmsandDataStructures.pdf