MCS-211: Design and Analysis of Algorithms
Guess Paper-I

Q. Discuss about Analysis and Complexity of Algorithm.

Ans. The term "analysis of algorithms" was introduced by Donald Knuth. It has become now an important computer science discipline whose overall objective is to understand the complexity of an algorithm in terms of time complexity and storage requirement.

System performance is directly dependent on the efficiency of algorithm in terms of both the time complexity as well the memory. An algorithm designed for time sensitive application takes too long to run can render its results of no use.

Suppose M is an algorithm with n the input data size. The time and space used by the algorithm M are the two main measures for the efficiency of M. The time is measured by counting the number of key operations, for example, in case of sorting and searching algorithms, the number of comparisons is the number of key operations. That is because key operations are so defined that the time for the other operations is much less than or at most proportional to the time for the key operations. The space is measured by counting the maximum of memory needed by the algorithm.

The complexity of an algorithm M is the function f(n), which give the running time and/or storage space requirement of the algorithm in terms of the size n of the input data. Frequently, the storage space required by an algorithm is simply a multiple of the data size n. In general the term "complexity" given anywhere simply refers to the running time of the algorithm. There are 3 cases, in general, to find the complexity function f(n):

- **Worst-case:** The maximum number of steps taken on any instance of size a.
- **Best-case:** The minimum number of steps taken on any instance of size a.
- **Average case:** An average number of steps taken on any instance of size a.

Average case: The value of $f(n)$ which is in between maximum and minimum for any possible input. Generally the Average case implies the ***expected value*** of $f(n)$.

The analysis of the average case assumes a certain probabilistic distribution for the input data; one such assumption might be that all possible permutations of an input data set are equally likely. The Average case also uses the concept of probability theory. Suppose the numbers $N_1, N_2, \ldots\ldots$ occur with respective probabilities $p_1, p_2, \ldots\ldots p_k$. Then the expectation or average value of E is given by $E = N_1 p_1 + N_2 p_2 + \cdots\ldots\ldots + N_k p_k$

To understand the Best, Worst and Average cases of an algorithm, consider a linear array $A[1 \ldots . n]$, where the array A contains n-elements. Students may you are having some problem in understanding. Suppose you want *either* to find the location LOC of a given element (say x) in the given array A or to x send some message, such as LOC=0, to indicate that x does not appear in A. Here the linear search algorithm solves this problem by comparing given , one-by-one, with each element in A. That is, we compare with A[1], then A[2], and so on, until we find LOC such that $x = A[LOC]$.

Analysis of linear search algorithm

The complexity of the search algorithm is given by the number C of comparisons between x and array elements A[K].

Best case: Clearly the best case occurs when x is the first element in the array A. That is $x = A[LOC]$. In this case $C(n) = 1$

Worst case: Clearly the worst case occurs when x is the last element in the array A or x is not present in given array A (to ensure this we have to search entire array A till last element). In this case, we have
$C(n) = n$.

Average case: Here we assume that searched element x appear array A, and it is equally likely to occur at any position in the array. Here the number of comparisons can be any of numbers *1,2,3,...,n,* and each

number occurs with the probability $p = \frac{1}{n}$. then

$$C(n) = 1.\frac{1}{n} + 2.\frac{1}{n} + \cdots \ldots \ldots + n.\frac{1}{n}$$
$$= (1 + 2 + \cdots \ldots \ldots \ldots + n).\frac{1}{n}$$
$$= \frac{n(n+1)}{2}.\frac{1}{n} = \frac{n+1}{2}$$

It means the average number of comparisons needed to find the location of x is approximately equal to half the number of elements in array A. From above discussion, it may be noted that the complexity of an algorithm in the average case is much more complicated to analyze than that of worst case. Unless otherwise stated or implied, we always find and write the complexity of an algorithm in the worst case.

There are three basic asymptotic $(i.e.\ n \to \infty)$ notations which are used to express the running time of an algorithm in terms of function, whose domain is the set of natural numbers N={1,2,3,.....}. These are:

- *0(Big – 'Oh')* [This notation is used to express Upper bound (maximum steps) required to solve a problem]
- *Ω0(Big – 'Oh')* [This notation is used to express Lower bound i.e. minimum (at least) steps required to solve a problem]
- Θ ('Theta') Notations.[Used to express both Upper & Lower bound, also called tight bound]

Asymptotic notation gives the *rate of growth*, i.e. performance, of the run time for "*sufficiently large input sizes*" $(i.e.\ n \to \infty)$ and is **not** a measure of the *particular* run time for a specific input size (which should be done empirically). O-notation is used to express the Upper bound (worst case); Ω-notation is used to express the Lower bound (Best case) and Θ- Notations is used to express both upper and lower bound (i.e. Average case) on a function.

Q. Explain the concept of complexity/efficiency of an algorithm.

Ans. In order to understand the complexity/efficiency of an algorithm, it is very important to understand the notion of the size of an instance of the problem under consideration and the role of size in determining complexity of the solution. For example finding the product of two 2 × 2 matrices will take much less time than the time taken by the same algorithm for multiplying say two 100 × 100 matrices. This explains intuitively the notion of the size of an instance of a problem and also the role of size in determining the (time) complexity of an algorithm. If the size (to be later considered formally) of general instance is n then time complexity of the algorithm solving the problem (not just the instance) under consideration is some function of n.

In view of the above explanation, the notion of size of an instance of a problem plays an important role in determining the complexity of an algorithm for solving the problem under consideration. However, it is difficult to define precisely the concept of size in general, for all problems that may be attempted for algorithmic solutions. In some problems number of bits is required in representing the size of an instance. However, for all types of problems, this does not serve properly the purpose for which the notion of size is taken into consideration. Hence different measures of size of an instance of a problem are used for different types of problems. Let us take two examples:

(i) In sorting and searching problems, the number of elements, which are to be sorted or are considered for searching, is taken as the size of the instance of the problem of sorting/searching

(ii) In the case of solving polynomial equations or while dealing with the algebra of polynomials, the degrees of polynomial instances, may be taken as the sizes of the corresponding instances.

To measure the efficiency of an algorithm, we will consider the theoretical approach and follow the following steps:

- Calculation of time complexity of an algorithm- Mathematically determine the time needed by the algorithm, for a general instance of size, say, n of the problem under consideration. In this approach, generally, each of the basic instructions like assignment, read and write, arithmetic operations, comparison operations are assigned some constant number of (basic) units of time for execution. Time for looping statements will depend upon the number of times the loop executes. Adding basic units of time of all the instructions of an algorithm will give the total amounts of time of the algorithm.
- The approach does not depend on the programming language in which the algorithm is coded and on how it is coded in the language as well as the computer system used for executing (a programmed version of) the algorithm. But different computers have different execution speeds. However, the speed of one computer is generally some constant multiple of the speed of the other.
- Instead of applying the algorithm to many different-sized instances, the approach can be applied for a general size say n of an arbitrary instance of the problem but the size n may be arbitrarily large under consideration.

An important consequence of the above discussion is that if the time taken by one machine in executing a solution of a problem is a polynomial (or exponential) function in the size of the problem, then time taken by every machine is a polynomial (or exponential) function respectively, in the size of the problem.

Q. Give a review of Asymptotic Notations.

Ans. The complexity analysis of algorithm is required to measure the time and space required to run an algorithm. In this unit we focus on only the time required to execute an algorithm. Let us quickly review some asymptotic notations (Please refer to the previous unit for detailed discussion). The central idea of these notations is to compare the relative rate of growth of functions.

Assume T(n) and f(n) are two functions

(i) $T(n) = O(f(n))$ if there are two positive constants C and n_0 such that $T(n) \leq Cf(n)$ where $n \geq n_0$

(ii) $T(n) = \Omega(f(n))$ if there are two positive constants C and n_0 such that $T(n) \geq C\Omega$ f(n) where $n \geq n_0$

(iii) $T(n) = \theta(f(n))$ if and only if $T(n) = O(f(n)) and T(n) = \Omega(f(n))$

If Assume $T(n) = 1000\, n \; and \; f(n) = n^3$. A function 1000 n is larger than n^3 for small value of n, but n^3 will grow at faster rate if the value n become large. Therefore n^3 is a larger function. The definition of $T(n) = O(f(n))$ says that T(n) will grow slower or equal to C.f(n) after the point where $n \geq n_0$.

The second definition, $T(n) = \Omega(f(n))$ says that the growth rate of T(n) is faster than or equal to (≥) f(n).

Q. Discuss the procedure of linear research.

Ans. Linear search or sequential search is a very simple search algorithm which is used to search for a particular element in an array. Unlike binary search algorithm, in linear search algorithm, elements are not arranged in a particular order. In this type of search, an element is searched in an array sequentially one by one .If a match is found then that particular item is returned and the search process stops. Otherwise, the search continues till the end of an array. The following steps are used in the linear search algorithm:

Linear_ Search(A[ ], X)

Step 1: Initialize i to 1

Step 2: if i exceeds the end of an array then print "element not found" and Exit

Step 3: if A[i] = X then Print "Element X Found at index i in the array" and Exit

Step 4: Increment i and go to Step 2

We are given with a list of items. The following table shows a data set for linear search:

7	17	3	9	25	18

In the above table of data set, start at the first item/element in the list and compared with the key. If the key is not at the first position, then we move from the current item to next item in the list sequentially until we either find what we are looking for or run out of items i.e the whole list of items is exhausted. If we run out of items or the list is exhausted, we can conclude that the item we were searching from the list is not present.

The key to be searched=25 from the given data set.

In the given data set key 25 is compared with first element i.e 7 , they are not equal then move to next element in the list and key is again compared with 17 , key 25 is not equal to 17. Like this key is

compared with element in the list till either element is found in the list or not found till end of the list. In this case key element is found in the list and search is successful.

Let us write the algorithm for the linear search process first and then analyze its complexity.

```
// a is the list of n elements, key is an element to be searched in the list function
linear_search(a,n,key)
{
        found=false // found is a boolean variable which will store either true or
false
        for(i=0;i<n;i++)
        {
                if (a[i]==key)
                        found = true
                        break;
        }
        if (i==n)
                found =
false
        return found
}
```

For the complexity analysis of this algorithm, we will discuss the following cases:

a. best case time analysis
b. worst-case time analysis
c. average case time analysis

To analyze searching algorithms, we need to decide on a basic unit of computation. This is the common step that must be repeated in order to solve the problem. For searching, comparison operation is the key operation in the algorithm so it makes sense to count the number of comparisons performed. Each comparison may or may not discover the item we are looking for. If the item is not in the list, the only way to know it is to compare it against every item present.

Best Case: The best case - we will find the key in the first place we look, at the beginning of the list i.e the first comparison returns a match or return found as true. In this case we only require a single comparison and complexity will be O (1).

Worst Case: In worst case either we will find the key at the end of the list or we may not find the key until the very last comparison i.e nth comparison. Since the search requires n comparisons in the worst case, complexity will be O (n).

Average Case: On average, we will find the key about halfway into the list; that is, we will compare against n/2 data items. However, that as n gets larger, the coefficients, no matter what they are, become insignificant in our approximation, so the complexity of the linear search, is O(n). The average time depends on the probability that the key will be found in the collection - this is something that we would not expect to know in the majority of cases. Thus in this case, as in most others, estimation of the average time is of little utility.

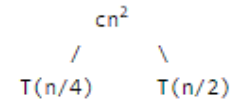
If the performance of the system is crucial, i.e. it's part of a life-critical system, and then we must use the worst case in our design calculations and complexity analysis as it tends to the best guaranteed performance.

The following table summarizes the above discussed results.

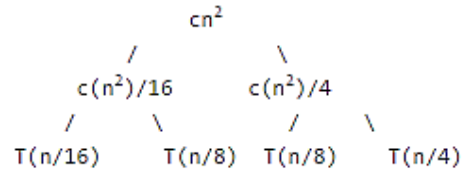
Case	Best Case	Worst Case	Average Case
item is present	O(1)	O(n)	O(n/2) = O(n)
item is not present	O(n)	O(n)	O(n)

However, we will generally be most interested in the worst-case time calculations as worst-case times can lead to guaranteed performance predictions.
Most of the times an algorithm run for the longest period of time as defined in worst case. Information provide by best case is not very useful. In average case, it is difficult to determine probability of occurrence of input data set. Worst case provides an upper bound on performance i.e the algorithm will never take more time than computed in worse case. So, the worst-case time analysis is easier to compute and is useful than average time case.

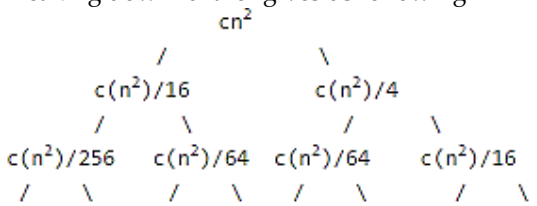
Q. Explain various methods of solving recurrence relations.
Ans. There are mainly three ways for solving recurrences.
(1) Substitution Method: We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.
For example consider the recurrence T(n) = 2T(n/2) + n
We guess the solution as T(n) = O(nLogn). Now we use induction to prove our guess.
We need to prove that T(n) <= cnLogn. We can assume that it is true for values smaller than n.
T(n) = 2T(n/2) + n
<= cn/2Log(n/2) + n
= cnLogn - cnLog2 + n
= cnLogn - cn + n
<= cnLogn
(2) Recurrence Tree Method: In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically a arithmetic or geometric series.
For example consider the recurrence relation



If we further break down the expression T(n/4) and T(n/2), we get following recursion tree.
T(n) = T(n/4) + T(n/2) + cn^2



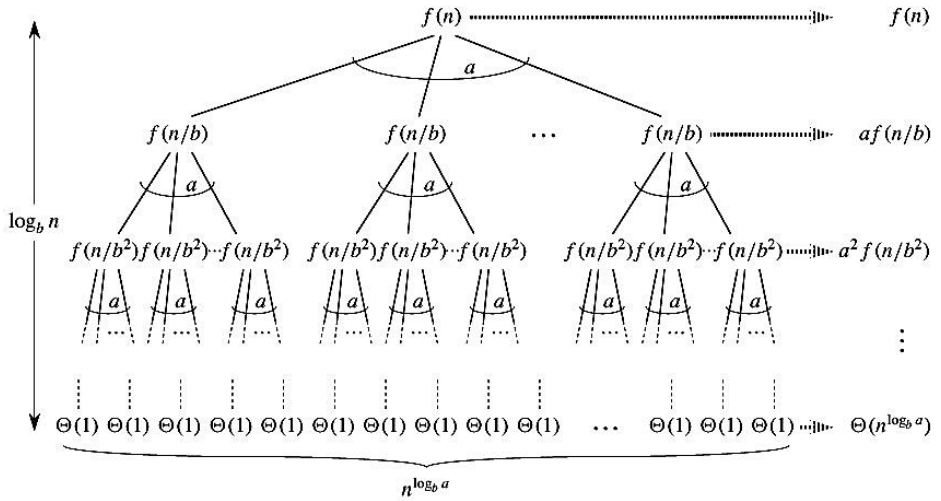
Breaking down further gives us following



To know the value of T(n), we need to calculate sum of tree nodes level by level. If we sum the above tree level by level, we get the following series
T(n) = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ...
The above series is geometrical progression with ratio 5/16.
To get an upper bound, we can sum the infinite series.
We get the sum as (n^2)/(1 - 5/16) which is O(n^2)
(3) Master Method: Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.
T(n) = aT(n/b) + f(n) where a >= 1 and b > 1
There are following three cases:

- If f(n) = Θ(nc) where c < Logba then T(n) = Θ(nLogba)
- If f(n) = Θ(nc) where c = Logba then T(n) = Θ(ncLog n)
- If f(n) = Θ(nc) where c > Logba then T(n) = Θ(f(n))

Master method is mainly derived from recurrence tree method. If we draw recurrence tree of T(n) = aT(n/b) + f(n), we can see that the work done at root is f(n) and work done at all leaves is Θ(nc) where c is Logba. And the height of recurrence tree is Logbn



In recurrence tree method, we calculate total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1). If work done at leaves and root is asymptotically same, then our result becomes height multiplied by work done at any level (Case 2). If work done at root is asymptotically more, then our result becomes work done at root (Case 3).
Example:
Merge Sort: T(n) = 2T(n/2) + Θ(n). It falls in case 2 as c is 1 and Logba] is also 1. So the solution is Θ(n Logn)
Binary Search: T(n) = T(n/2) + Θ(1). It also falls in case 2 as c is 0 and Logba is also 0. So the solution is Θ(Logn)

Q. What is a 'Greedy Algorithm'?
Ans. In an algorithm design there is no one 'silver bullet' that is a cure for all computation problems. Different problems require the use of different kinds of techniques. A good programmer uses all these techniques based on the type of problem. Some commonly-used techniques are:

- Divide and conquer
- Randomized algorithms
- Greedy algorithms (This is not an algorithm, it is a technique.)
- Dynamic programming

A greedy algorithm, as the name suggests, always makes the choice that seems to be the best at that moment. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.

How do you decide which choice is optimal?

Assume that you have an objective function that needs to be optimized (either maximized or minimized) at a given point. A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized. The Greedy algorithm has only one shot to compute the optimal solution so that it never goes back and reverses the decision.

Greedy algorithms have some advantages and disadvantages:

- It is quite easy to come up with a greedy algorithm (or even multiple greedy algorithms) for a problem.
- Analyzing the run time for greedy algorithms will generally be much easier than for other techniques (like Divide and conquer). For the Divide and conquer technique, it is not clear whether the technique is fast or slow. This is because at each level of recursion the size of gets smaller and the number of sub-problems increases.
- The difficult part is that for greedy algorithms you have to work much harder to understand correctness issues. Even with the correct algorithm, it is hard to prove why it is correct. Proving that a greedy algorithm is correct is more of an art than a science. It involves a lot of creativity.

Note: Most greedy algorithms are **not** correct. An example is described later in this article.

C. How to create a Greedy Algorithm?

Being a very busy person, you have exactly T time to do some interesting things and you want to do maximum such things.

You are given an array **A** of integers, where each element indicates the time a thing takes for completion. You want to calculate the maximum number of things that you can do in the limited time that you have.

This is a simple Greedy-algorithm problem. In each iteration, you have to greedily select the things which will take the minimum amount of time to complete while maintaining two variables current Time and number Of Things. To complete the calculation, you must:

- Sort the array **A** in a non-decreasing order.
- Select each to-do item one-by-one.
- Add the time that it will take to complete that to-do item into current Time.
- Add one to number Of Things.

Repeat this as long as the current Time is less than or equal to T.

Let A = {5, 3, 4, 2, 1} and T = 6

After sorting, A = {1, 2, 3, 4, 5}

After the 1st iteration:

- Current Time = 1
- number Of Things = 1

After the 2nd iteration:

- current Time is 1 + 2 = 3
- number Of Things = 2

After the 3rd iteration:

- current Time is 3 + 3 = 6
- number Of Things = 3

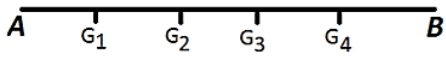
After the 4th iteration, current Time is 6 + 4 = 10, which is greater than T. Therefore, the answer is 3.

Some examples to understand Greedy Techniques

Example 1: Suppose there is a list of tasks along with the time taken by each task. However, you are given with limited time only. The problem is which set of tasks will you be doing so that you can complete maximum number of tasks in the given amount of time.

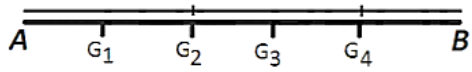
Solution: The intuitive approach would be greedy approach and the task selection criteria would be to select the task with the slowest amount of time. So, the first task will be that task which takes minimum time. The next task would be the one that takes minimum time among the remaining set of tasks and so on.

Example 2: Consider a bus which can travel up to 40 kilometers (Km) with full tank. We need to travel from location 'A' to location 'B' which has distance of 95 Km.



In between 'A' and 'B', there are four gas stations, G1, G2, G3, and G4, which are at distance of 20 KM, 37.5 KM, 55 KM, and 75 KM, from location 'A' respectively. The problem is to determine the minimum number of refills needed to reach the location 'B' from location 'A'.

Solution: Suppose, the tank refill decision criteria is considered to refill at the gas station which is nearest when the tank is about to get empty. According to the criteria, if we start from location 'A', the tank will be refilled at the second gas station (G2)as it is 37.5 KM from 'A'. After this, we need to refill at the fourth gas station (G4) which is at a distance of 37.5 KM from G2. From G4, we can easily reach the location 'B' as it is 20 KM. Therefore, the number of refills required is two.



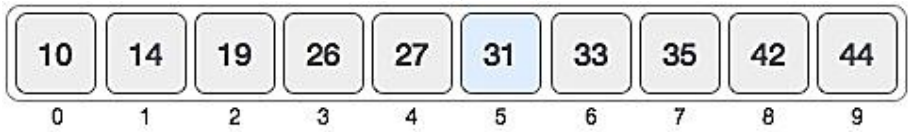
Q. What is Binary search?

Ans. Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

How Binary Search Works?

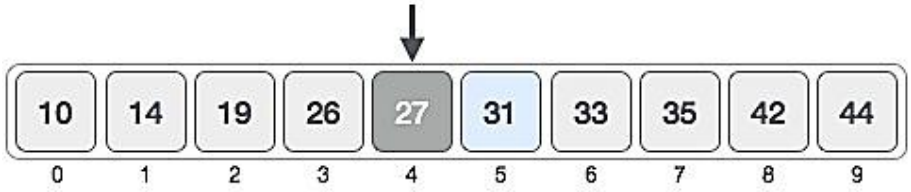
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



First, we shall determine half of the array by using this formula −

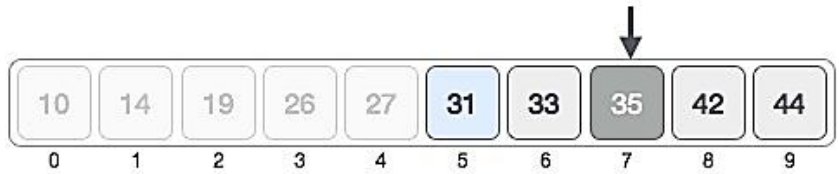
mid = low + (high - low) / 2

Here it is, 0 + (9 - 0)/2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



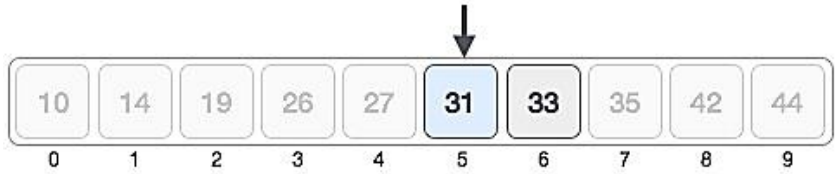
Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

We change our low to mid + 1 and find the new mid value again.
low = mid + 1
mid = low + (high - low)/2
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.
Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Q. Discuss about various graph representation schemes.
Ans. The purpose of graph representation is typically to search a graph most systematically such that the edges of the graphs can be used to effectively visit all the vertices of it. In order to have an effective search algorithm, the logical representation of the graph plays a very critical role. When it comes to representations of graphs, two most standard and common computational representations are in practice such as Adjacency Matrix and Adjacency List. Apart from these other additional representations such as Linked lists, contiguous lists and combinations are also used in fewer cases. The selection of a particular representation depends on applications and functions one wants to perform on these graphs.
In case the graph is sparse for which the number of edges also written as $|E|$ is quite less than $|v|^2$, adjacency list is preferred but if the graph is dense where $|E|$ is close to $|v|^2$ an adjacency matrix is selected.
Adjacency Matrix:
Adjacency matrix representation is typically used to represent both directed and undirected graphs, where the concentration of nodes is a graph is dense in nature. Usually in such graphs $|E|$ is much

close to $|V|^2$ these adjacency matrix are typically drawn for Directed Acyclic Graph (DAG), where we have no loops or double directions.

- *Adjacency matrix A is a square matrix which is used to represent a finite graph. The matrix elements shows whether pairs of vertices are adjacent (connected) or not in the graph. It is 2 dimensional array , which have the size V x V , where V are the numbers of vertices present in graph* otherwise it

  $A[i,j] = 1$ if there is edges between A_i & A_j

  $A[i,j] = 0$, otherwise

Adjacency Matrix of the following graph( ) is given below.

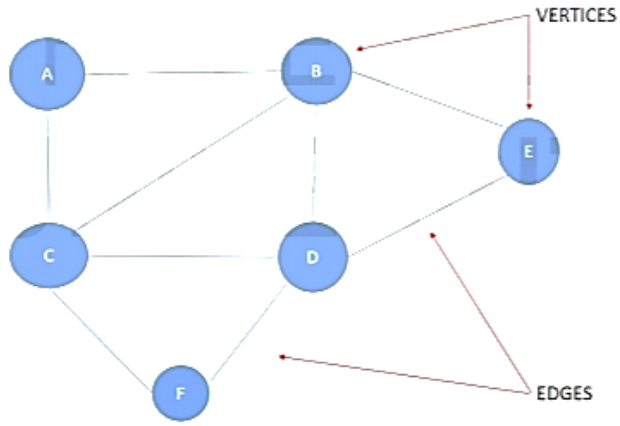


Figure 8: Example graph for matrix representation

In the given graph, if there is a link between the vertices we mark as‘1’ in the adjacency matrix, else we mark as ‘0’.

In the given graph, if there is a link between the vertices we mark as‘1’ in the adjacency matrix, else we mark as ‘0’.

	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	1	1	1	0
C	1	1	0	1	0	1
D	0	1	1	0	1	1
E	0	1	0	1	0	0
F	0	0	1	1	0	0

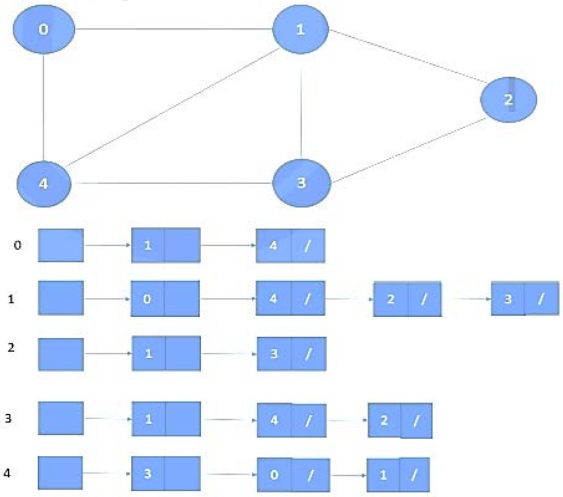
Figure 9: Adjacency matrix representation of a graph given at figure 8

It is easy to determine if there is an edge between two vertices in a graph if it is represented through adjacency matrix. To find out how many edges are in a graph, it will require at least (v 2) time as there are v 2 entries of matrix have to be examined except the diagonal element which are zeroes. But when the most of the enteries are $0s$ (in case of a sparse graph) then it might take (e + v) where $e << v$ 2 which is significantly less time.

Adjacency matrix for a directed graph may or may not be symmetric but in case of undirected graph. It is always symmetric Space complexity = (v 2) time complexity. Where V is the number of vertex is independent of a number of edges.

Adjacency List

Adjacency list representation is typically used to represent graphs, where the number edges |E| is much less than |V|2. Adjacency list is represented as an array of |V| linked lists. There is one linked list for every vertex node in a linked list í Each Node in this linked list is a reference to the other vertices which share an edge with the current vertex. The following figure, (b) shows adjacency list of a given graph.



If there is undirected with V vertices and E edges adjacency list representation requires |V| head nodes and 2|E| adjacency list nodes. In an undirected edges (í, *j*), í appear in *j*'s adjacency list and *j* appears in í's adjacency list. In terms of memory requires it is O(V + E) for both types of graphs representations: adjacency matrix and adjacency list.

Q. Discuss about Direct Acyclic graph.

Ans. A directed graph without a cycle is called a directed acyclic graph (or a (DAG (for short) which is a frequently used graph structure to represent precedence relation or dependence is a network. The following is a example of a directed acyclic task graph.

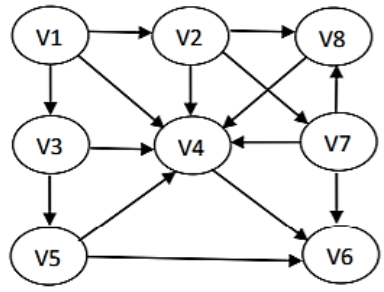


Figure 14: Directed Acyclic Graph

In this graph except vertex V1, all other vertices are dependent upon other vertices.

Any major task can be broken down into several subtasks. The successful completion of the task is possible only when all the subtasks are completed successfully. Dependencies among subtasks are represented through a directed graph. In such representation, subtasks are represented through vertices and an edge between two vertices define a precedence relation. After showing dependency,

the next task is to ordering is to ordering of these subtasks for execution. This is also called topological sorting or topological ordering.

Topological sort or ordering of a $DAG\ G = (V, E)$ is a liner ordering of its vertices $V_1, V_2 \ldots . V_n$ so that if a graph G contains an edge $from\ v_i\ to\ v_j$ then v_i comes before v_j in the ordering. In other words, all edges between vertices representing tasks show forward direction in ordering or sorting. It the graph G contains a cycle then no topological order is possible.

Therefore, if a graph G has a topological sort, then G is a DAG.

Proof- Applying contradiction that G has a typological sorting of all its vertices (tasks): $V_1, V_2 \ldots . V_n$ and G is also having a cycle. let all the vertices be indexed : $V_1, V_2 \ldots . V_n$. let V_i be the lowest index in topological ordering. let there be an edges $(V_j, V_i) in$ a cyclic graph G. If $V_j > V_i$ i.e. V_j, comes before V_i which contradicts that all the vertices $V_1, V_2 \ldots . V_n$ are topological sorted is G.

The following is a pseudo-code topological for computing a DAG.
Pseudo-code to compute topological sorting

```
Function topological_sort(G)
Input  G = (V, E) //G is a DAG.
{
search for a node V with  zero in-degree (no incoming edges) and order it first in
topological sorting
remove V from G
topological_sort(G-{V})  // recursively compute topological sorting
append the ordering
}
```

Pseudo-code for topological sorting

MCS-211: Design and Analysis of Algorithms
Guess Paper-II

Q. Explain Topological sorting time complexity.

Ans. Finding out a vertex with no incoming edge, deleting it from a graph and appending it in the linear ordering would take O(n) time. Since there are n number of vertices is G, there will be n times loop, the total time will be (n 2). But if the graph is sparse where the $|E| << n$ 2 *and* the graph is represented through an adjacency list it is possible to have $(m + n)$ time complexity, where m is |E|.

Illustration of an example

The following figure show the application of the algorithm to the example given in the figure 15

Step 1: V_1 - does not have incoming edges so it is deleted first.

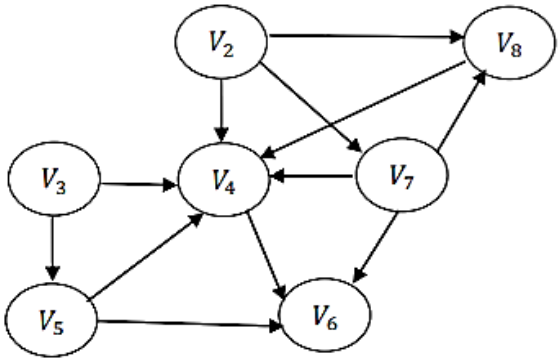


Figure 16 (a) : Deletion of V_1

Step 2 : In the graph there are the two nodes V_2 & V_3 with no incoming edges. One can pick up either of the two vertices. Let us remove V_2 and append it after V_1, i.e., V_1 , V_2

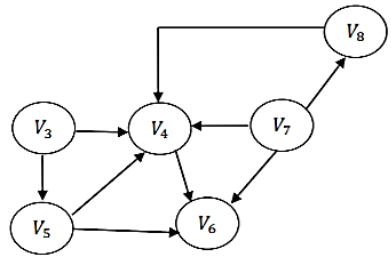


Figure 16(b): Deletion of V_2

Step 3: - V_3 is the only node with no incoming edge. Therefore V_3 will be removed and appended after V_2 , *i. e.*, $V_1V_2V_3$

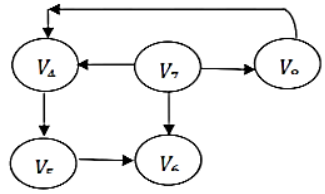


Figure 16(c): Deletion of V_3

Step 4 : V_5 is the only node with no incoming edge. V_5 will be removed and appended to the list ,i.e.,

$V_1\ V_2\ V_3\ V_5$

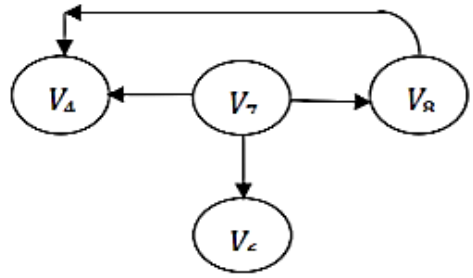


Figure 16(d): Deletion of V_5

Step 5 : V_7 is the only node with no incoming edge. Therefore V_7 will be removed and appended, i.e.,

$V_1\ V_2\ V_3\ V_5\ V_7$

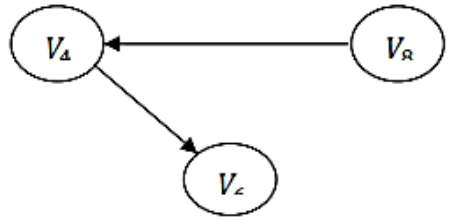


Figure 16(e): Deletion of V_7

Step 6: - V_8 will be removed and appended, i.e., $V_1\ V_2\ V_3\ V_5\ V_7\ V_8$

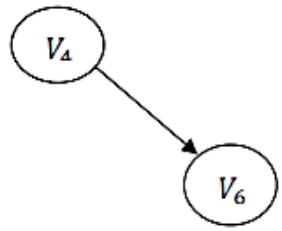
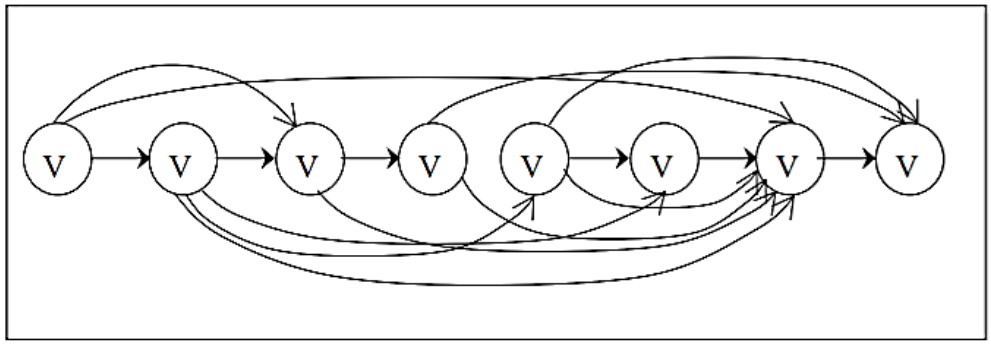


Figure 16(f) : Deletion of V_8

Step 7: V_4 will be removed and appended as: $V_1, V_2, V_3, V_5, V_7, V_8, V_4$

Step 8: Finally V_6 will be appended: $V_1, V_2, V_3, V_5, V_7, V_8, V_4, V_6$

The linear ordering of vertices is shown in the following figure:



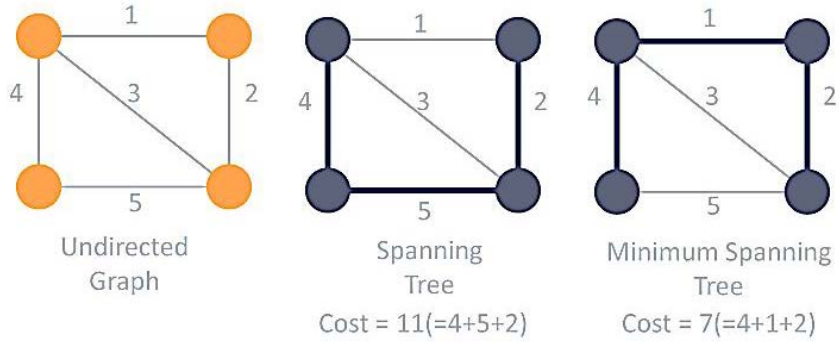
Topological Sorted order of vertices

Q. What is a Minimum Spanning Tree?

Ans. The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:

- Cluster Analysis
- Handwriting recognition
- Image segmentation



There are two famous algorithms for finding the Minimum Spanning Tree:

Kruskal's Algorithm

Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

Algorithm Steps:

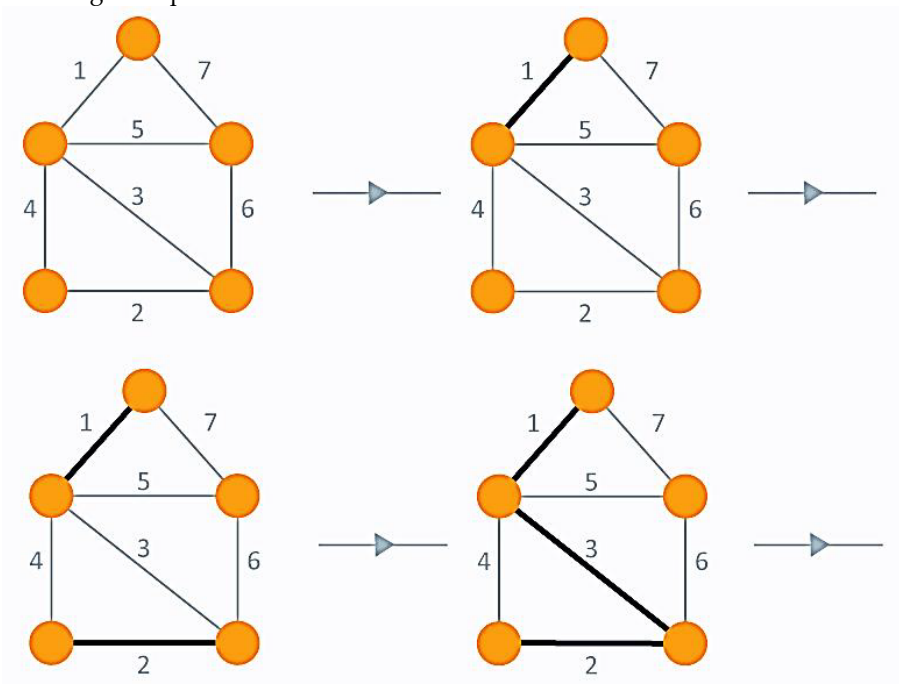
- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle , edges which connect only disconnected components.

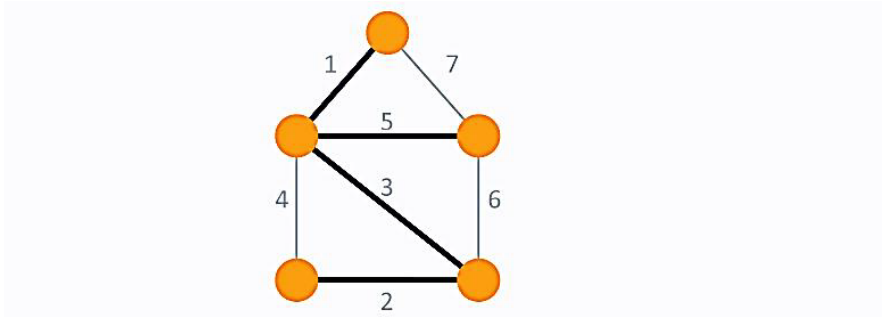
So now the question is how to check if 2 vertices are connected or not?

This could be done using DFS which starts from the first vertex, then check if the second vertex is visited or not. But DFS will make time complexity large as it has an order of O(V+E) where V is the number of vertices, E is the number of edges. So the best solution is "Disjoint Sets":

Disjoint sets are sets whose intersection is the empty set so it means that they don't have any element in common.

Consider following example:





In Kruskal's algorithm, at each iteration we will select the edge with the lowest weight. So, we will start with the lowest weighted edge first i.e., the edges with weight 1. After that we will select the second lowest weighted edge i.e., edge with weight 2. Notice these two edges are totally disjoint. Now, the next edge will be the third lowest weighted edge i.e., edge with weight 3, which connects the two disjoint pieces of the graph. Now, we are not allowed to pick the edge with weight 4, that will create a cycle and we can't have any cycles. So we will select the fifth lowest weighted edge i.e., edge with weight 5. Now the other two edges will create cycles so we will ignore them. In the end, we end up with a minimum spanning tree with total cost 11 (= 1 + 2 + 3 + 5).

Time Complexity:

In Kruskal's algorithm, most time consuming operation is sorting because the total complexity of the Disjoint-Set operations will be O(ElogV), which is the overall Time Complexity of the algorithm.

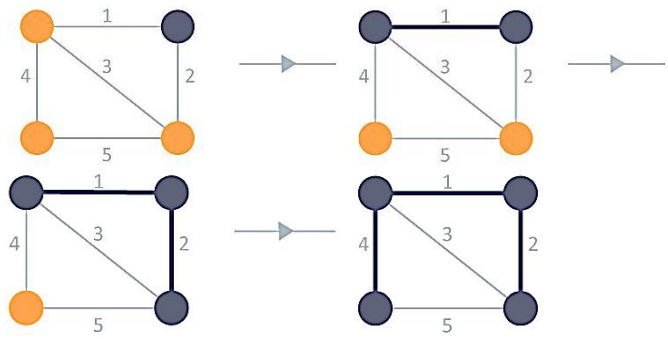
Prim's Algorithm

Prim's Algorithm also use Greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an **edge** in Kruskal's, we add **vertex** to the growing spanning tree in Prim's.

Algorithm Steps:

- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices that are connected to growing spanning tree, into the Priority Queue.
- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

Consider the example below:

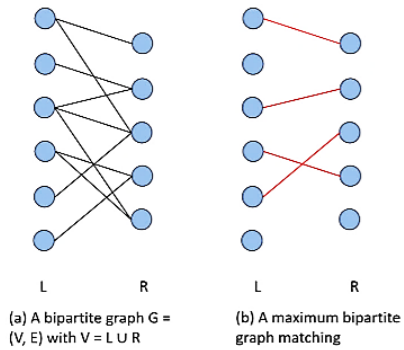


In Prim's Algorithm, we will start with an arbitrary node (it doesn't matter which one) and mark it. In each iteration we will mark a new vertex that is adjacent to the one that we have already marked. As a greedy algorithm, Prim's algorithm will select the cheapest edge and mark the vertex. So we will simply choose the edge with weight 1. In the next iteration we have three options, edges with weight 2, 3 and 4. So, we will select the edge with weight 2 and mark the vertex. Now again we have three options, edges with weight 3, 4 and 5. But we can't choose edge with weight 3 as it is creating a cycle. So we will select the edge with weight 4 and we end up with the minimum spanning tree of total cost 7 (= 1 + 2 +4).

Q. What is Maximum Bipartite Matching?

Ans. Maximum bipartite matching problem is a subset of matching problem specific for bipartite graphs (graphs where the vertex set V can be partitioned into two disjoint sets L and R i.e. V = L∪R and the edges E is present only between L and R). A matching in a bipartite graph consists of a set of edges such that no two edges e ∈ E share a vertex v ∈ V. A matching of maximum size (i.e. maximum number of edges) is known to be maximum matching if the addition of any edge makes it no longer a matching

Many real-world problems can be represented as bipartite matching. For example, there are L applicants and R jobs with each applicant has a subset of job interest but can be hired for only one job. Figure 1 illustrates the matching problem and a solution.



(a) A bipartite graph G = (V, E) with V = L ∪ R

(b) A maximum bipartite graph matching

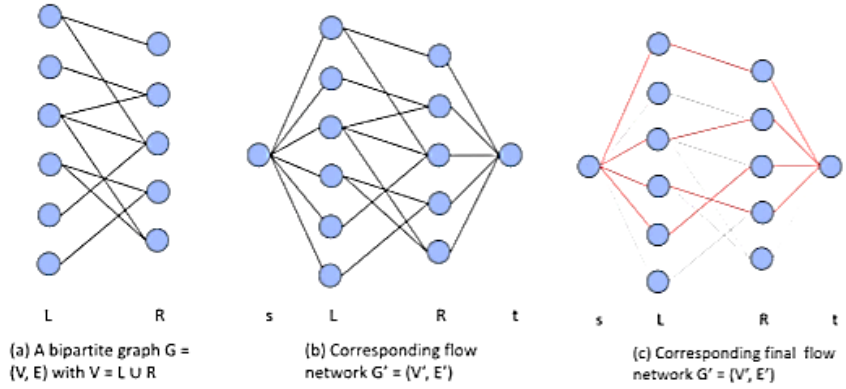
The maximum bipartite matching problem can be solved by transforming the problem into a flow network. Next, Ford-Fulkerson algorithm can be used to find a maximum matching. Following are the steps:

A. Construct a flow network:

A flow network **G' = (V', E')** is defined by adding a source node **s** and sink node to the bipartite graph **G = (V, E)** such that **V' = V ∪ {s, t}** and **E' = {(s, u): u ∈ L} ∪ {(u, v): (u, v) ∈ E} ∪ {(v, t): v ∈ R}**. Here **L** and **R** are the partitions of vertex **V** as shown in Figure 1 (a). The capacity of every new edge is marked as **1**. The flow network constructed for bipartite graph given in Figure 1 is shown in Figure 2.

B. Find the maximum flow:

Ford-Fulkerson algorithm is the most efficient method to find a solution for the flow network. It relies on the Breadth First Search (BFS) to pick a path with minimum number of edges. Ford-Fulkerson, computes a maximum flow in a network by applying the greedy method to the augmenting path approach used to prove max-flow. The intuition behind it that keep augmenting flow among an augmenting path until there is no augmenting path.



(a) A bipartite graph G = (V, E) with V = L ∪ R

(b) Corresponding flow network G' = (V', E')

(c) Corresponding final flow network G' = (V', E')

*Figure 14: Flow network corresponding to the bipartite graph. Right most graph shows the solution for a maximum flow problem (the final flow network of (b)). Each edge have unit capacity, and shaded edge from **L** to **R** corresponds to those in the maximum matching from (b).*

Pseudo code for Ford-Fulkerson algorithm:

- The main idea of algorithm is to incrementally increase the value of flow in stages, where at each stage some amount of flow is pushed along an augmenting path from source to the sink.
- Initially the flow of each edge is equal to **0**.
- In line 3, at each stage path **p** is calculated and amount of flow equal to the residual capacity c_f of **p** is pushed along **p**.
- The algorithm terminates when current flow does not accept an augmenting path.
- An augmenting path is a path p from the start vertex **(s)** to the end vertex **(t)** that can receive additional flow without going over capacity.
- Line 5 is adding flow and line 6 is reducing flow in the edge belongs to the path **p**.

Ford – Fulkerson(G, s, t)

Inputs Given a Network $G' = (V', E')$ with flow capacity c, a source node s, and a sink node t

Output Compute a flow f from s to t of maximum value

1. $f(u, v) \leftarrow 0$ for all edges (u, v)
2. *While* there is a path p from s to t in G_f, such that $c_f(u, v) > 0$ for all edges $(u, v) \in p$
3. Find $c_f(p) = \min\{c_f(u, v) : (u, v) \in p\}$
4. For each edge $(u, v) \in p$
5. $f(u, v) \leftarrow f(u, v) + c_f(p)$ *(Send flow along the path)*
6. $f(u, v) \leftarrow f(u, v) - c_f(p)$ *(The flow might be "returned" later)*

Since in the current implementation of Ford Fulkerson algorithm, it is using BFS, the algorithm is also known as Edmonds-Karp algorithm.

Q. Define the Principle of Optimality.

Ans. Dynamic programming follows the principal of optimality. If a problem have an optimal structure, then definitely it has principal of optimality. A problem has optimal sub structure if an optimal solution can be constructed efficiently from optimal solution of its sub-problems. Principal of optimality shows that a problem can be solved by taking a sequence of decision to solve the optimization problem. In dynamic programming in every stage we takes a decision. The Principle of Optimality states that components of a globally optimum solution must themselves be optimal.

A problem is said to satisfy the Principle of Optimality if the sub solutions of an optimal solution of the problem are themselves optimal solutions for their sub problems.

Examples:

- The shortest path problem satisfies the Principle of Optimality.
- This is because if $a, x_1, x_2, \ldots, x_n$ is a shortest path from node a to node b in a graph, then the portion of x_i to x_j on that path is a shortest path from x_i to x_j.
- The longest path problem, on the other hand, does not satisfy the Principle of Optimality. For example the undirected graph of nodes a, b, c, d, and e and edges $(a, b), (b, c), (c, d), (d, e)$ and (e, a). That is, G is a ring. The longest (noncyclic) path from a to d to a,b,c,d. The sub-path from b to c on that path is simply the edge b,c. But that is not the longest path from b to c. Rather b,a,e,d,c is the longest path. Thus, the sub path on a longest path is not necessarily a longest path.

Steps of Dynamic Programming:

Dynamic programming has involve four major steps:

1. Develop a mathematical notation that can express any solution and sub solution for the problem at hand.
2. Prove that the Principle of Optimality holds.

3. Develop a recurrence relation that relates a solution to its sub solutions, using the math notation of step 1. Indicate what the initial values are for that recurrence relation, and which term signifies the final solution.
4. Write an algorithm to compute the recurrence relation.

- Steps 1 and 2 need not be in that order. Do what makes sense in each problem.
- Step 3 is the heart of the design process. In high level algorithmic design situations, one can stop at step 3.
- Without the Principle of Optimality, it won't be possible to derive a sensible recurrence relation in step 3.
- When the Principle of Optimality holds, the 4 steps of DP are guaranteed to yield an optimal solution. No proof of optimality is needed.

Q. Discuss about Matrix chain multiplication.

Ans. Given a sequence of matrices that are conformable for multiplication in that sequence, the problem of matrix-chain-multiplication is the process of selecting the optimal pair of matrices in every step in such a way that the overall cost of multiplication would be minimal.

If there are total N matrices in the sequence then the total number of different ways of selecting matrix-pairs for multiplication will be ${}^{2n}C_n/(n+1)$. We need to find out the optimal one. In directly we can say that total number of different ways to perform the matrix chain multiplication will be equal to the total number of different binary trees that can be generated using N-1 nodes i.e. 2nCn/(n+1). The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.Since matrix multiplication follows associativity property, rearranging the parentheses also yields the same result of multiplication. That means any pair in the sequence can be multiplied and that will not affect the end result: But the total number of multiplication operations will change accordingly.A product of matrices is fully satisfied if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. For example, if the chain of matrices is $\langle A_1, A_2, A_3, A_4 \rangle$ then we can fully parenthesize the product $A_1A_2A_3A_4$ in five distinct ways:

$(A_1(A_2(A_3A_4)))$,
$(A_1((A_2A_3)A_4)$,
$((A_1A_2)(A_3A_4))$,
$(A_1(A_2A_3))A_4)$,
$(((A_1A_2)A_3)A_4)$.

How we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. We shall use the dynamic-programming method to determine how to optimally parenthesize a matrix chain.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.
4. Construct an optimal solution from computed information.

Step 1: The structure of an optimal parenthesization:

- An optimal parenthesization of $A_1 \ldots A_n$ must break the product into two expressions, each of which is parenthesized or is a single array
- Assume the break occurs at position k.
- In the optimal solution, the solution to the product $A_1 \ldots A_k$ must be optimal
  - Otherwise, we could improve $A_1 \ldots A_n$ by improving $A_1 \ldots A_k$.
  - But the solution to $A_1 \ldots A_n$ is known to be optimal
  - This is a contradiction
  - Thus the solution to $A_1 \ldots A_n$ is known to be optimal
- This problem exhibits the Principle of Optimality:
  - The optimal solution to product $A_1 \ldots A_n$ contains the optimal solution to two subproducts
- Thus we can use Dynamic Programming
  - Consider a recursive solution
  - Then improve it's performance with memorization or by rewriting bottom up

Step 2: A recursive solution

For the matrix-chain multiplication problem, we pick asour sub problems the problems of determining the minimum cost of parenthesizing $A_i A_{i+1} \ldots A_j$ for $1 \leq i \leq j \leq n$.

- Let $m[i,j]$ be the minimum number of scalar multiplication needed to compute the matrix $A_{i \ldots j}$; for the full problem, the lowest cost way to compute $A_{1 \ldots n}$ would thus be $m[1,n]$.
- $m[i,i] = 0$, when i=j, problem is trivial. Chain consist of just one matrix $A_{i \ldots i} = A_i$, so that no scalar multiplications are necessary to compute the product.

- The optimal solution of $A_i \times A_j$ must break at some point, k, with $\leq k < j$. Each matrix A_i is $p_{i-1} \times p_i$ and computing the matrix multiplication $A_{i \ldots k} A_{k+1 \ldots j}$ takes $p_{i-1} p_k p_j$ scalar multiplication.

  Thus, $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$
  Equation 1

3. Computing the optimal costs

It is a simple matter to write a recursive algorithm based on above recurrence to compute the minimum cost $m[1,n]$ for multiplying $A_1 A_2 \ldots A_n$.

MATRIX-CHAIN-ORDER (P)

1. $n \leftarrow length[p] - 1$
2. let $m[1..n, 1..n]$ and $s[1..n-1, 2..n]$ be new tables
3. for i←1 to n
4. do $m[i, i] \leftarrow 0$
5. For l← 2 to n
6. do for i←1 to n-1+1
7. do j←i+l-1
8. $m[i, j] \leftarrow \infty$
9. for k←l to j-1
10. do $q \leftarrow m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$
11. if q<$m[i, j]$
12. then $m[i, j] \leftarrow q$
13. $s[i, j] \leftarrow k$
14. return m and s

The following pseudocode assumes that matrix $\boldsymbol{A_i}$ has dimensions $\boldsymbol{p_{i-1} \times p_i}$ for **i=1,2,…,n**. The input is a sequence$\boldsymbol{p = p_0, p_1, \ldots, p_n}$, where **length[p]=n+1**. The procedure uses an auxiliary table $\boldsymbol{m[1 \ldots n, 1 \ldots n]}$ for storing the $\boldsymbol{m[i, j]}$ costs and the auxiliary table $\boldsymbol{s[1 \ldots n, 1 \ldots n]}$ that records which index of k achieved the optimal cost in computing $m[i, j]$. We will use the table **s** to construct an optimal solution.

- In line 3-4 algorithm first computes $m[i, i] = 0$ for $i = 1, 2, \ldots, n$ (the minimum costs for chains of length 1).
- During the first execution of for loop in line 5-13, it uses Equation (1) to computes$m[i, i+1]$ for $i = 1, 2, \ldots, n-1$ (the minimum costs for chains of length 2).
- The second time through the loop, it computes $m[i, i+2]$ for $i = 1, 2, \ldots, n-2$ (the minimum costs for chains of length 3, and so forth.
- At each step, the $m[i, j]$ cost computed in lines 10-13 depends only on table entries $m[i, k]$ and $m[k+1, j]$ already computed.

4. Constructing an Optimal Solution:

The MATRIX-CHAIN-ORDER determines the optimal number of scalar multiplication needed to compute a matrix-chain product. To obtain the optimal solution of the matrix multiplication, we call **PRINT_OPTIMAL_PARES(s,1,n)** to print an optimal parenthesization of $\boldsymbol{A_1 A_2, \ldots, A_n}$. Each entry $s[i, j]$ records a value of

k such that an optimal parenthesization of $\boldsymbol{A_1 A_2, \ldots, A_j}$splits the product between A_k and A_{k+1}.

Q. What is Binomial Theorem?

Ans. Binomial is also called as Binomial Expansion describe the powers in algebraic equations. Binomial Theorem helps us to find the expanded polynomial without multiplying the bunch of binomials at a time. The expanded polynomial will always contain one more than the power you are expanding. Following formula shows the General formula to expand the algebraic equations by using Binomial Theorem:

$$(x+a)^n = \sum_{k=0}^{n} \binom{n}{k} x^k a^{n-k}$$

Where, n= positive integer power of algebraic equation and $\binom{n}{k}$= read as "n choose k".
According to theorem, expansion goes as following for any of the algebraic equation containing any positive power,

$$(a+b)^n = \binom{n}{0}a^n b^0 + \binom{n}{1}a^{n-1}b^1 + \binom{n}{2}a^{n-2}b^2 + \ldots + \binom{n}{n-1}a^1 b^{n-1} + \binom{n}{n}a^n b^n$$

Where, $\binom{n}{k}$ are binomial coefficients and $\binom{n}{k} = n_{c_k}$ gives combinations to choose k elements from n-element set. The expansion of binomial coefficient can be calculated as: $\frac{n!}{k! \times (n-k)!}$.

We can compute n_{c_k} for any n and k using the recurrence relation as follows:

$$n_{C_k} = \begin{cases} 1, & if\ k=0\ or\ n=k \\ n-1_{C_{k-1}} + n-1_{C_k}, & for\ n>k>0 \end{cases}$$

Computing C(n, k): Pseudocode:

BINOMIAL(n, k)

Input: A pair of nonnegative integers $n \geq k \geq 0$

Output: The value of $c(n, k)$

```
1. for i←0 to n    do
2.     for j← 0 to min(i, k) do
3.         if  j=0   or   j=i
4.             c[i, j]←1
5.         else
6.             c[i, j]← c[i-1, j-1] + c[i-1, j]
7. return c[n, k]
```

Let's consider an example for calculating binomial coefficient:
Compute binomial coefficient for n = 5 and k = 5

n/k	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1

Table 3: gives the binomial coefficient

According to the formula when k=0 corresponding value in the table will be 1. All the values in column1 will be 1 when k=0. Similarly when n=k, value in the diagonal index will be 1.

c[2, 1] = c[1, 0]+c[1,1] = 1+1= 2 c[3, 1]= c[2,0] + c[2,1] = 1+2 = 3

c[4, 1] = c[3,0]+ c[3,1]= 1+3 = 4 c[5, 1] = c[4, 0] + c[4, 1] = 1 + 4 = 5

c[3, 2] = c[2,1] + c[2, 2] = 2+1=3 c[4, 2] = c[3, 1] + c[3, 2] = 3 + 3 = 6

c[5, 2] = c[4, 1]+ c[4, 2] = 4 + 6 =10 c[4, 3] = c[3,2] + c[3, 3] = 3 + 1= 4

c[5, 3] = c[4, 2] + c[4, 3] = 6 + 4 = 10 c[5, 4] = c[4, 3] + c[4, 4]= 4 + 1= 5

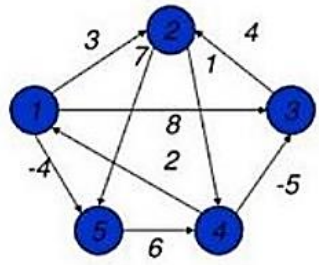
Table 3 represent the binomial coefficient of each term.

Time Complexity:

The cost of the algorithm is filing out the table. Addition is the basic operation. Because $k \leq n$, the sum needs to be split into two parts because only the half the table needs to be filled out for $i < k$ and remaining part of the table is filled out across the entire row. Time complexity of the binomial coefficient is the size of the table i.e, O(n*k)

Q. Discuss about All-Pairs Shortest Paths.

Ans. The all pair shortest path algorithm is also known as Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.



$$\begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

At first the output matrix is same as given cost matrix of the graph. After that the output matrix will be updated with all vertices k as the intermediate vertex.

The time complexity of this algorithm is O (V3), here V is the number of vertices in the graph.

Input − The cost matrix of the graph.

0 3 6 ∞ ∞ ∞ ∞

3 0 2 1 ∞ ∞ ∞

6 2 0 1 4 2 ∞

∞ 1 1 0 2 ∞ 4

∞ ∞ 4 2 0 2 1

∞ ∞ 2 ∞ 2 0 1

∞ ∞ ∞ 4 1 1 0

Output − Matrix of all pair shortest path.

0 3 4 5 6 7 7
3 0 2 1 3 4 4
4 2 0 1 3 2 3
5 1 1 0 2 3 3
6 3 3 2 0 2 1
7 4 2 3 2 0 1
7 4 3 3 1 1 0

Algorithm

Floyd Warshal(cost)

Input − The cost matrix of given Graph.

Output − Matrix to for shortest path between any vertex to any vertex.

```
Begin
   for k := 0 to n, do
      for i := 0 to n, do
         for j := 0 to n, do
            if cost[i,k] + cost[k,j] < cost[i,j], then
               cost[i,j] := cost[i,k] + cost[k,j]
            done
         done
      done
      display the current cost matrix
End
```

Q. Write a short note about Brute force algorithm.

Ans. Brute Force Algorithms are exactly what they sound like – straightforward methods of solving a problem that rely on sheer computing power and trying every possibility rather than advanced techniques to improve efficiency.

For example, imagine you have a small padlock with 4 digits, each from 0-9. You forgot your combination, but you don't want to buy another padlock. Since you can't remember any of the digits, you have to use a brute force method to open the lock.

So you set all the numbers back to 0 and try them one by one: 0001, 0002, and 0003, and so on until it opens. In the worst case scenario, it would take 10^4, or 10,000 tries to find your combination.

A classic example in computer science is the traveling salesman problem (TSP). Suppose a salesman needs to visit 10 cities across the country. How does one determine the order in which those cities should be visited such that the total distance traveled is minimized?

The brute force solution is simply to calculate the total distance for every possible route and then select the shortest one. This is not particularly efficient because it is possible to eliminate many possible routes through clever algorithms.

The time complexity of brute force is O(mn), which is sometimes written as O(n*m). So, if we were to search for a string of "n" characters in a string of "m" characters using brute force, it would take us n * m tries.

Q. Discuss briefly about Knuth Morris Pratt Algorithm.

Ans. This is linear time string matching algorithm. The complexity is O(m+n) where m and n are the length of a pattern string and a text string respectively. This happens because the KPS algorithm avoids frequent backtracking in the text string as it is done in the naïve algorithm. The key idea in KMP algorithm is to build a LPS (largest prefix as suffix) array to determine from which point in the pattern string to restart comparing for pattern matching in a text in case there is a mismatch of a character without moving the text pointer backward. In such a case first we go back one character

backward from the position where mismatch occurred, read its value in the LPS array which defines the length of a prefix also as a suffix if any, i.e., we check whether there is any occurrence of the largest prefix as a suffix in the pattern to decide how many characters in the pattern need to be skipped to start searching for the string matching hat the next stage. If there is a mismatch at the ith character of a pattern string, we move to (i-1) th character) in the pattern string and find out the LPS array value of this character. Suppose the LPS array value of (i-1)th character is 2 . This number defines the length of the largest prefix which is also a suffix in a pattern string. It also indicates the first two characters in the pattern string need to be skipped for the next comparison of pattern string with a text. If the length of a pattern string is m then only (m-2) characters will be compared with a text(not from the beginning of the text but from the position where there was a mismatch).

In the following example we first do the pattern matching exercise without building the LPS array to get the idea quickly. But efficient implementation of KPS would be done through LPS array only.

Example:

Text: abcfa b c x a b f a b a b c x a b c z

Pattern = abcxa b c z

Examining the text and the pattern string we notice that there is mismatch of characters at the fourth position. In the text string it is 'f' whereas in the pattern string it is 'x'. To decide from which position in the pattern string the search should restart, we go back and examine the substring in the pattern just before the position where there was a mismatch, i.e., we examine "a b c" substring of pattern (so far we have not built up the LPS array). Since all the characters are unique, there is no prefix also as a suffix in this substring, therefore we cannot skip any character in the pattern string, the comparison will start from the beginning character of the pattern, i.e., 'a' with 'f'(at this position , there was a mismatch). Again there is a mismatch, so we will move to the next character in the text, i.e., 'a' and start comparing with the first character in the pattern string ,i.e., 'a'. There is a match of a character, we will compare the next character of a pattern with the next character of a text and continue till there is a mismatch. Please notice that there is a mismatch at the 7th character in the pattern, i.e., 'c' with 'f' in the text. Now we examine the substring of a pattern just before the position of the mis-match, i.e., "a b c x a b " and decide how many characters to be skipped in the pattern. We try to find out the length of prefix also as a suffix in this substring. It is "ab" which is suffix as well as prefix and the length is 2. Therefore we will skip two characters in the pattern from the beginning, which is now, 'c' and start comparing with the same position of character in the text where there was a mismatch, i.e., 'f'. It makes a sense because "ab" is existing in the text just before 'f'. It need not be compared again. Now there is a mismatch, so we go back and examine the substring in pattern to find out if there is any prefix which is also as a suffix. The substring is "ab". There is no prefix and suffix information in the substring, because both are unique characters. Therefore the comparison starts with the first character of the pattern ('a') and the next character in the text('a'). The next character is 'b' in the pattern as well as in the text. Finally we find that the pattern is found in the text.

It is time now to build a LPS array for a pattern P.

P = d e f g d e f

I j

d	e	f	g	D	e	f	d
0	1	2	3	4	5	6	7
0	0	0	0	1	2	3	1

Figure : LPS Array (last row)

In the above table, the last row builds information about prefix and suffix of the pattern. The first and the second row indicate the pattern and its index values. We consider two pointers iand j. Initially iis pointing to the first character of the pattern and j is pointing to the second character of the pattern. The first entry in LPS array(last row of the table) is zero. For the second entry, we compare i with j

which are not equal. i is pointing to d and j is pointing to e. If i and jare not equal, the related entry in the LPS array will be zero. Then j will be incremented by one. Again i and j are not-equal. i is pointing to a and j is pointing to 'f'. The LPS entry for this index will be zero. Similarly the next entry in the LPS array is zero again. What will happen when j is pointing to d and i is also pointing to d. In this case the LPS entry of the character pointed to by j will be equal to the current index value of i and + 1. Therefore the entry will be 1 at this column. The next two entries will be 2 and 3. Each time there is a match, i will get incremented. Now i is pointing to g and j is pointing to d. There is mismatch. In this case i will be decremented by 1. i is pointing to 'f' now. Its LPS entry value is zero. Accordingly i will shift to the beginning of the array. Both i and j are pointing to the same character. The LPS entry for the last character will be the index value of i, which is zero plus 1,i.e., 1

The question is now how to use LPS array for pattern matching? Please refer to the last row of the array. There is entry 1 at the 4th column. We will ignore the remaining entries in the LPS array after this entry. It indicates that the length of a prefix which is also a suffix is one. Therefore skip one character from the beginning in the pattern for the next comparison. Let us take one example:

Text = d e f g d x . . . .

Pattern = d e f g d e

There is a mismatch between 'x' and 'e' characters. After mismatch we go back to the previous character, i.e., 'd'. The entry for d in the array is 1 , which says that the next comparison will start from 'e' in the pattern and 'x' in the text.

Time Complexity of KMP Algorithm

The worst case time complexity of KMP is O (m+n) where O (m) is time taken to build LPS array and O (n) is the time taken to search for entire text..