Introduction

26/04/21, Mon

• Algorithm:

Algorithm is a process or set of rules to be followed in calculations or other problem-solving operations. Therefore, algorithm refers to a set of rules/instructions that step-by-step defines how a work is to be executed upon in order to get the expected results.

An algorithm has some specific characteristics —

(i) Clear and Unambiguous: An algorithm should be clear and unambiguous which means that all of its steps should be mentioned clearly and should not contain any ambiguity.

(ii) Well-defined inputs: An algorithm can take zero or more inputs and in either of the cases it should be well defined

(iii) Outputs: An algorithm must yield one output.

(iv) Definiteness:

(iv) Finiteness: An algorithm must be finite. If the steps are traced then for all cases it must terminate after a finite number of steps.

(v) Effectiveness: Each instruction in an algorithm must be very basic so that it can be carried out, in principle, by a person using only pencil and paper

(vi) Language-independent: The algorithm must be independent of any programming language syntax. It should be readable by non-programmers as well.

• Program vs. Algorithm:

Program	Algorithm
(i) A program is written in the implementation phase of software engineering.	(i) An algorithm is written in the design phase of software engineering.
(ii) A program is totally dependent upon computer hardware and OS.	(ii) An algorithm is totally independent of computer hardware and OS.
(iii) Programmes are written in programming languages like C, C++, Java	(iii) Algorithms are usually written in plain english like readable statements.

(iv) Programs are written by programmers.	(iv) Any person having specific domain knowledge can write algorithms.

Example:

The following algorithm finds and returns the maximum of n given numbers.

```
Algorithm Max(A, n)
// A is an array of size n
{
    Result := A[1];
    for i := 2 to n do
    {
        if A[i] > Result then Result := A[i];
    }
    return Result;
}
```

The following algorithm shows how **selection sort** works

```
Algorithm SelectionSort(A, n)
// We'll sort array A of length n in ascending order
{
    for i ← 1 to n do
    {
        j ← i;
        for k ← i+1 to n do
        {
            if (A[k] < A[j]) then j ← k;
        }
        t ← A[i]; A[i] ← A[j]; A[j] ← t;
    }
}
```

Recursive Algorithms:

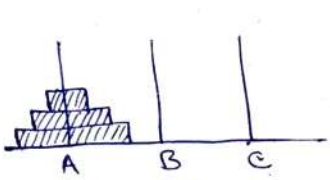
A function is called <u>recursive function</u> when it is defined in terms of itself. So, similarly an algorithm is said to be recursive if it invokes itself in its body.

Recursive algorithm
- → **Direct Recursive** (An algorithm that calls itself is called direct recursive)
- → **Indirect Recursive** (Algorithm A is called indirect recursive if it calls another algorithm which in turn calls A.)

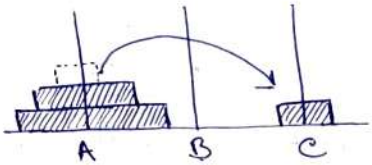
Example 1 [Towers of Hanoi]

Problem Statement : There are n disks arranged on a tower say A in decreasing order of their radius. The problem is to transfer all the disks from tower A to tower C and arrange them the ~~same~~ same way they were in tower A. ~~An empty~~ One disk can be moved at a time and larger disks cannot be put upon smaller ones. Another ~~two~~ tower B is given for help in moving the disks.

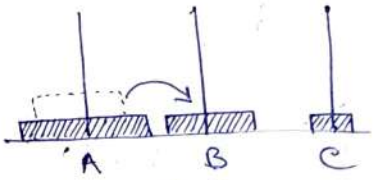
Solution : Initially, lets assume that there are 3 disks.



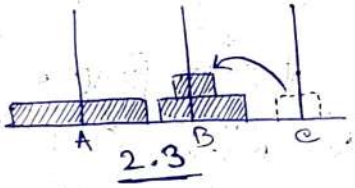
1. Initial



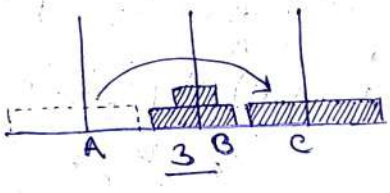
2.1



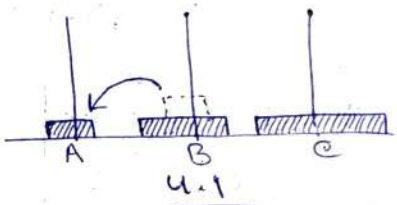
2.2



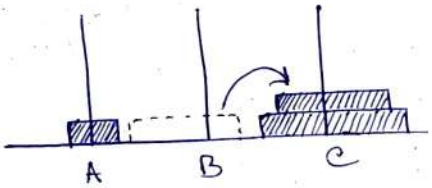
2.3



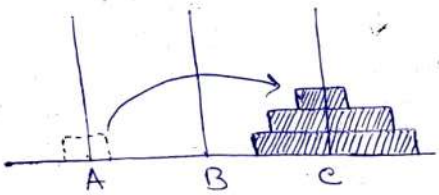
3



4.1



4.2



4.3

Step:1 Move two disks from A to B using C as described using diagram 2.1, 2.2 and 2.3.

Step:2 Move ~~the~~ a ~~longest~~ disk to the destination tower, here C.

Step:3 Move two disks from B to C using A as described using diagram 4.1, 4.2 and 4.3.

Using these 3 steps we can write the steps for n disks as well shown below :

Step:1 Move (n-1) disks from A to B using C.

Step:2 Move a disk from A to C.

Step:3 Move (n-1) disks from B to C using A.

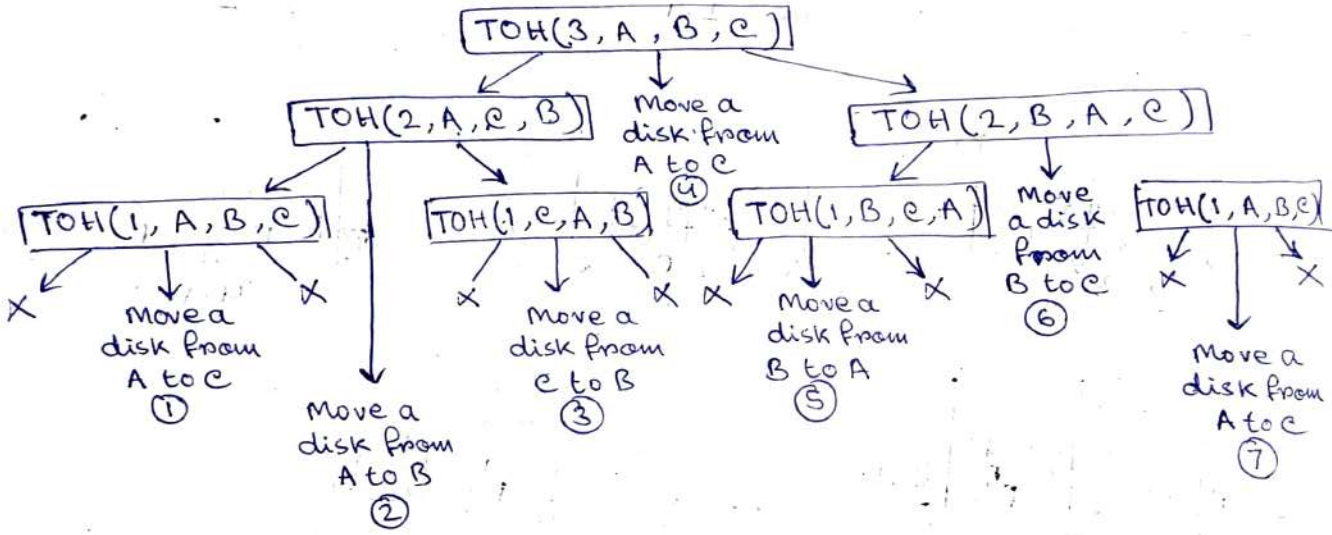
[Here also moving two disk means moving one disk at a time in the above ~~desc~~ diagramed approach]

So, the algorithm of this solution can be given recursively as below —

```
Algorithm TowerOfHanoi(n, A, B, C)
// Move top n disks from tower A to C using B
{
    if(n≥1) then
    {
        TowerOfHanoi(n-1, A, C, B);
        write("Move a disk from", A, "to", C);
        TowerOfHanoi(n-1, B, A, C);
    }
}
```

Trace:

[TowerOfHanoi is written is short as TOH]



So all the outputs are labelled 1 through 7 in order.

Asymptotic Complexity & Analysis of Algorithms

• Performance Analysis :

The criterias that have direct relationship to the performance of an algorithm are space and time complexity.

The space complexity of an algorithm is the amount of memory it needs to run to completion.

The time complexity of an algorithm is the amount of computer time it needs to run to completion.

Time complexity of an algorithm can be measured using two methods — Priori analysis and posteriori analysis.

Priori Analysis : A priori analysis is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured assuming that processor speed and other machine parameters to be constant, and have no effect on implementation.

Posteriori Analysis : A posteriori analysis is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language and executed on target computer machine.

The differences between priori and posteriori analysis are –

Posteriori Analysis	Priori Analysis
(i) Posteriori analysis is dependent on language of compiler and type of hardware.	(i) Priori analysis is indepent of all kinds of machine related parameters.
(ii) It doesn't use asymptotic notations to represent the time complexity of an algorithm.	(ii) It uses asymptotic notations to represent the time complexity of an algorithm.
(iii) The time complexity of an algorithm using posteriori analysis differ from system to system.	(iii) The time complexity of an algorithm using priori analysis is same for every system.
(iv) The calculated time complexity using posteriori analysis is an exact value.	(iv) The calculated time complexity using priori analysis is an approximate value.

• Space Complexity :

The space needed by any algorithm is seen to be the sum of the following components :

(i) A fixed or constant part that is independent of the number of inputs or size of input or output. This part typically includes the instruction space (space for code), space for simple variables, constants and so on.

(ii) A variable part that consists of the space needed by component variables whose size is dependent on the

particular problem instance being solved, the space needed by referenced variables and the recursion stack space.

Therefore, the space complexity S(P) of any algorithm P can be referred to as $S(P) = c + S_p$ where c = constant component and S_p = instance characteristics.

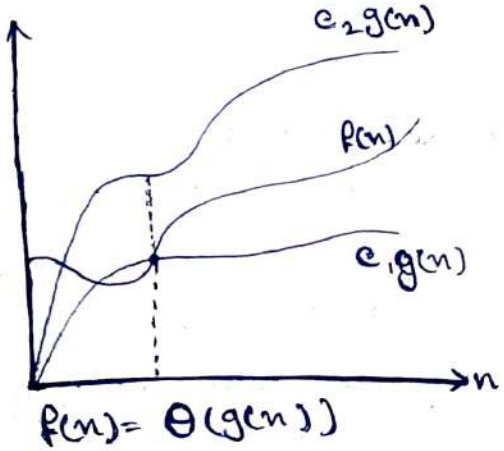
• Time Complexity :

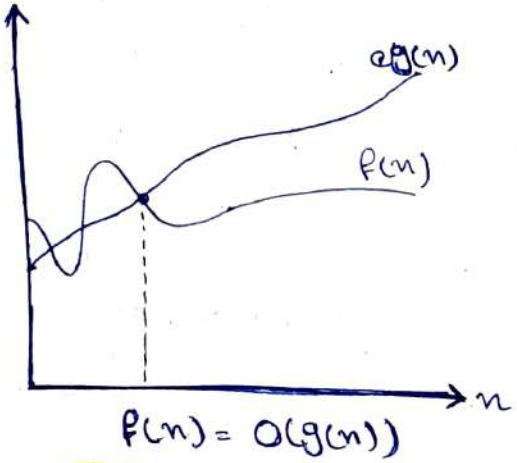
The time taken by a program is the sum of compile time and run time. The compile time does not depend upon instance characteristics. We may assume that a compiled program will run several times without recompilation. We concern ourselves with just the compile time.

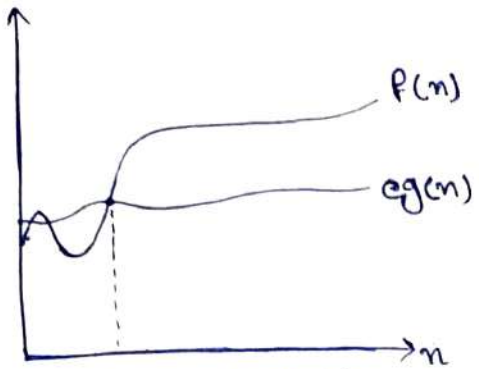
So, the time complexity of an algorithm is given by the number of steps taken by the algorithm to compute the function it was written for. The number of steps is dependent upon the instance characteristics. But although any specific instance may have several characteristics, the number of steps is computed as a function of some subset of these only. We choose those characteristics only which are important to us. For example, for any certain algorithm we may wish to know how the computing time increases as the number of inputs increases. In this case, the no of steps will be calculated as a function of the number of inputs alone only.

• Asymptotic Notation :

Asymptotic notations are the notations which we use to describe the asymptotic running time of an algorithm. Asymptotic notations are defined in terms of functions whose domains are set of natural numbers. We use asymptotic notations primarily to describe the running times of algorithms.







$f(n) = \Omega(g(n))$

classes of functions ⇒

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \cdots < 2^n < 3^n < \cdots < n^n$

Asymptotic Notations

- **O (big-oh)** [works as upper bound of a function]
- **Ω (big-omega)** [works as lower bound of a function]
- **θ (theta)** [works as average bound of a function] ★ Tight bound

(i) O (big-oh)

The function $f(n) = O(g(n))$ iff ∃ positive constants c and n_0 such that $f(n) \leq c * g(n)$ ∀ $n \geq n_0$

ex (i) $f(n) = 2n + 3$

$2n + 3 \leq 5n \; \forall n \geq 1 \;\therefore f(n) = O(n)$

(ii) Ω (big-omega)

The function $f(n) = \Omega(g(n))$ iff ∃ positive constants c and n_0 such that $f(n) \geq c * g(n)$ ∀ $n \geq n_0$

ex (i) $f(n) = 2n + 3$

$2n + 3 \geq n \; \forall n \geq 1 \;\therefore f(n) = \Omega(n)$

(iii) θ (theta)

The function $f(n) = \theta(g(n))$ iff ∃ positive constants c_1, c_2 and n_0 such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ ∀ $n \geq n_0$

ex (i) $f(n) = 2n + 3$

$n \leq 2n + 3 \leq 5n \quad \therefore f(n) = \theta(n)$

8th May, 2021, Sat

Ex ① Find O and Ω of $2n^2+3n+4$.

Ans. $f(n) = 2n^2+3n+4$

$2n^2+3n+4 \leq 9n^2 \quad \forall \; n>0$

$\therefore f(n) = O(n^2)$

$2n^2+3n+4 \geq n^2 \quad \forall \; n \geq 0$

$\therefore f(n) = \Omega(n^2)$

Ex ② Find O and Ω of $n^2\log n + n$

Ans. $f(n) = n^2\log n + n$

$n(n\log n + 1) \leq 2n^2 \log n$

$\therefore f(n) = O(n^2 \log n)$

$n^2\log n + n \geq n^2 \log n$

$\therefore f(n) = \Omega(n^2 \log n)$

Ex ③ Find O and Ω of $f(n) = n!$

Ans. $f(n) = n! = n(n-1)(n-2)(n-3)\cdots 1$

$= 1 \times 2 \times 3 \times \cdots \times \cdots \times n$

$1 \times 2 \times 3 \times \cdots \times n \leq n \times n \times n \times \cdots \times n$

$\therefore f(n) = O(n^n)$

$1 \cdot 2 \cdot 3 \cdots \cdot n \leq 1 \cdot 1 \cdot 1 \cdots \cdot 1$

$\therefore f(n) = \Omega(1)$ [Tight bound on Θ is not available for this function]

Ex ④ Find O and Ω of $f(n) = \log n!$

Ans $f(n) = \log n! = \log(1 \cdot 2 \cdot 3 \cdots n)$

$\log(1 \cdot 2 \cdot 3 \cdots n) \leq \log(n \cdot n \cdot n \cdots n)$

$\Rightarrow \log n! \leq \log n^n$

$\Rightarrow \log n! \leq n \log n$

$\therefore f(n) = O(n \log n)$

$\log n! \geq \log(1 \cdot 1 \cdot 1 \cdots 1)$

$\Rightarrow \log n! \geq 0 \quad \therefore f(n) = \Omega(1)$ [Tight bound is also not available for this function either]

Properties of asymptotic notation :-

General Properties ——

(i) If $f(n) = O(g(n))$ then $a * f(n) = O(g(n))$.

eg: $f(n) = 2n^2 + 5$; $f(n) = O(n^2)$

$\therefore 7 \cdot f(n) = 7(2n^2 + 5) = 14n^2 + 35$

$14n^2 + 35 \leq 49n^2$

$\therefore 7 \cdot f(n) = O(n^2)$

(ii) If $f(n) = \Omega(g(n))$ then $a * f(n) = \Omega(g(n))$.

(iii) If $f(n) = \Theta(g(n))$ then $a * f(n) = \Theta(g(n))$.

Reflexive Properties ——

(i) If $f(n)$ is given then $f(n) = O(f(n))$

eg: $f(n) = n^2$ $\therefore f(n) = O(n^2)$

(ii) If $f(n)$ is given then $f(n) = \Omega(f(n))$ (iii) If $f(n)$ is given, $f(n) = \Theta(f(n))$

Transitive Properties ——

(i) If $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then

$f(n) = O(h(n))$

eg: $f(n) = n$, $g(n) = n^2$ and $h(n) = n^3$

Here we can say,

$f(n) = O(n^2)$ and $g(n) = O(n^3)$

We can also say, $f(n) = O(n^3)$

(ii) If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ then

$f(n) = \Omega(h(n))$

(iii) If $f(n) = \Theta(g(n))$ an $g(n) = \Theta(h(n))$ then

$f(n) = \Theta(h(n))$

eg: Let $f(n) = 2n^2 + 4n + 5$

$\therefore n^2 \leq 2n^2 + 4n + 5 \leq 11n^2$

$\therefore f(n) = \Theta(n^2)$

Here $g(n) = n^2$ and clearly $g(n) = \Theta(h(n)) = \Theta(n^2)$

Therefore, $f(n) = \Theta(h(n)) = \Theta(n^2)$

Symmetric Properties

(i) If $f(n) = \Theta(g(n))$ then $g(n) = \Theta(f(n))$.

eg: Let $f(n) = n^2$

$\therefore f(n) = \Theta(g(n)) = \Theta(n^2)$

Also, $g(n) = \Theta(f(n)) = \Theta(n^2)$

Transpose Symmetric

(i) If $f(n) = O(g(n))$ then $g(n) = \Omega(f(n))$

eg: Let $f(n) = n$

$f(n) = O(n^2)$ this statement is true.

So, we can write $g(n) = n^2$ and $g(n) = \Omega(f(n))$
$= \Omega(n)$

(ii) If $f(n) = \Omega(g(n))$ then $g(n) = O(f(n))$

eg: Let $f(n) = n^2$

we can say that, $f(n) = \Omega(\log n)$

So, we can write $g(n) = \log n$ and $g(n) = O(f(n))$
$= O(n)$

> If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
> then, we can definitely say that
> $f(n) = \Theta(g(n))$

Ex (1) If $f(n) = O(g(n))$ and $d(n) = O(e(n))$ then find O of $f(n) + d(n)$.

Ans. Big-oh means upper bound. So if we add $f(n)$ and $d(n)$ then whichever term among $g(n)$ and $e(n)$ is maximum will represent the upper bound of the function $f(n) + d(n)$

$\therefore f(n) + d(n) = O(\max(g(n), e(n)))$.

Ex (2) If $f(n) = O(g(n))$ and $d(n) = O(e(n))$ then find O of $f(n) * d(n)$.

Ans. $f(n) * d(n) = O(g(n) * e(n))$

● Comparison of functions:

Some useful log formulas

- $\log ab = \log a + \log b$
- $\log a/b = \log a - \log b$
- $\log a^b = b \log a$
- $a^{\log_c b} = b^{\log_c a}$
- If $a^b = n$ then $b = \log_a n$

Ex① Among $f(n)$ and $g(n)$, find which one is greater. $f(n) = n^2 \log n$ and $g(n) = n(\log n)^{10}$

Ans. $\therefore \log f(n) = \log(n^2 \log n) = 2\log n + \log\log n$

$\therefore \log g(n) = \log[n(\log n)^{10}] = \log n + 10 \log\log n$

Now, among $(2\log n + \log\log n)$ is greater than $(\log n + 10 \log\log n)$.

So, $f(n)$ is greater than $g(n)$.

Ex② Among $f(n)$ and $g(n)$, find out which one is bigger.
$f(n) = 3n^{\sqrt{n}}$ and $g(n) = 2^{\sqrt{n}\log_2 n}$

Ans.

$g(n) = 2^{\sqrt{n}\log_2 n}$

$= n^{\sqrt{n}\log_2 2}$

$= n^{\sqrt{n}}$ [$\because \log_2 2 = 1$]

clearly $3n^{\sqrt{n}} > n^{\sqrt{n}}$. so, $f(n) > g(n)$

Ex③ Compare $f(n) = n^{\log n}$ and $g(n) = 2^{\sqrt{n}}$

Ans. $f(n) = n^{\log n}$

$\Rightarrow \log f(n) = \log(n^{\log n})$

$= \log n \log n$

$= \log^2 n$

$g(n) = 2^{\sqrt{n}}$

$\Rightarrow \log g(n) = \sqrt{n} \log 2$

$= \sqrt{n} = n^{1/2}$

We still can't say which one is bigger even after apply-ing log. So,

$\log(\log f(n)) = \log \log^2 n$

$= 2 \log\log n$

$\log(\log g(n)) = \log n^{1/2}$

$= 1/2 \log n$

Now, we can say that $2\log\log n < \frac{1}{2}\log n$ and so we can say that $f(n) < g(n)$.

Ex (4) Compare $f(n) = 2^{\log n}$ and $g(n) = n^{\sqrt{n}}$

Ans. $f(n) = 2^{\log n}$

$\Rightarrow \log f(n) = \log 2^{\log n}$

$= \log n \cdot \log 2$

$= \log n$

$g(n) = n^{\sqrt{n}}$

$\Rightarrow \log(g(n)) = \log n^{\sqrt{n}}$

$= \sqrt{n} \log n$

As, $\sqrt{n}\log n > \log n$, $g(n) > f(n)$

Ex (5) Compare the functions $g_1(n)$ and $g_2(n)$. $g_1(n)$ and $g_2(n)$ are given as follows.

$$g_1(n) = \begin{cases} n^3 & n < 100 \\ n^2 & n \geq 100 \end{cases}$$

$$g_2(n) = \begin{cases} n^2 & n < 10000 \\ n^3 & n \geq 10000 \end{cases}$$

Ans. For the value of n upto 99, $g_1(n) > g_2(n)$.

From $n = 100$ to 9999 $g_1(n) = g_2(n)$

But from $n = 10000$ onwards $g_2(n) > g_1(n)$ $\forall n \geq 10000$.

so, we can say $g_2(n) > g_1(n)$.

Ex (6) State true or false

(i) $(n+k)^m = \Theta(n^m)$

(ii) $2^{n+1} = O(2^n)$

(iii) $2^{2n} = O(2^n)$

(iv) $\sqrt{\log n} = O(\log\log n)$

(v) $n^{\log n} = O(2^n)$

Ans: (i) $(n+k)^m = n^m + \cdots + k^m$

Here the maximum term is n^m

so it is $\Theta(n^m)$

true

(ii) $2^{n+1} = 2^n \cdot 2^1 = O(2^n)$ true

(iii) $2^{2n} = 4^n > 2^n$

So, 2^n cannot be a upper bound of 4^n.

false

(iv) Let $f(n) = \sqrt{\log n}$ and $g(n) = \log \log n$

$\therefore \log f(n) = \log \sqrt{\log n}$ $\quad \log g(n) = \log(\log \log n)$

$= \frac{1}{2} \log \log n$

As $\frac{1}{2} \log \log n > \log \log \log n$, $f(n) > g(n)$ which implies that $g(n)$ that is $\log \log n$ cannot be a upper bound of $\sqrt{\log n}$

false

(v) Let $f(n) = n^{\log n}$ and $g(n) = 2^n$

$\therefore \log f(n) = \log n^{\log n}$

$= \log n \cdot \log n$

$= \log^2 n$

$\therefore \log g(n) = \log 2^n$

$= n \log 2$

$= n$

As $n > \log^2 n$, $g(n) > f(n)$ which implies that $g(n)$ that is 2^n can be an upper bound of $n^{\log n}$

$\therefore n^{\log n} = O(2^n)$ true

• Case Analysis of algorithms:

Linear Search

```
Algorithm LinearSearch(A, n, key){
// A is the array of elements, n is the no. of elements
// in A and key is the value to be found in A
    i:= 0;
    while (i < n){
        if (A[i] = key){
            return i;
            break;
        }else i:= i+1;
    }
    return (-1);
}
```

Best case : The key element is present at the beginning of the array that is at 0th index.

Best case time : $B(n) = O(1)$ [∵ Only one comparison]

Worst case : The key element is present at the last of the array that is at $(n-1)$th index.

Worst case time : $W(n) = O(n)$. [∵ n comparisons must be made]

Average case time : $\left[\frac{\text{All possible case time}}{\text{no. of elements}}\right]$

Avg. case time $= \frac{1+2+3+\cdots+n}{n}$

$= \frac{n(n+1)}{2n} = \frac{n+1}{2}$

$\therefore A(n) = \frac{n+1}{2} = O(n)$

20 August MONDAY

34th Week • 232-133

2018 AUGUST

WK	S	M	T	W	T	F	S
31				1	2	3	4
32	5	6	7	8	9	10	11
33	12	13	14	15	16	17	18
34	19	20	21	22	23	24	25
35	26	27	28	29	30	31	

Binary ~~sear~~ search

$$c(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 + c(\lfloor n/2 \rfloor) & \text{if } n \geq 2 \end{cases}$$

c ~~not~~ denotes no. of comparisons to be made based on number of elements n.

$$c(n) \leq 1 + c(\lfloor n/2 \rfloor)$$

$$c(n) \leq 1 + [1 + c(\lfloor n/4 \rfloor)]$$

$$c(n) \leq 1 + 1 + [1 + c(\lfloor n/8 \rfloor)]$$

$$c(n) \leq 1 + 1 + 1 + [1 + c(\lfloor n/16 \rfloor)]$$

$$\therefore c(n) \leq k + c(\lfloor n/2^k \rfloor)$$

The algo stops when $\frac{n}{2^k} = 1$

$$\therefore n = 2^k$$

$$\Rightarrow \log_2 n = k \log_2 2 = k$$

$$\Rightarrow k = \log_2 n$$

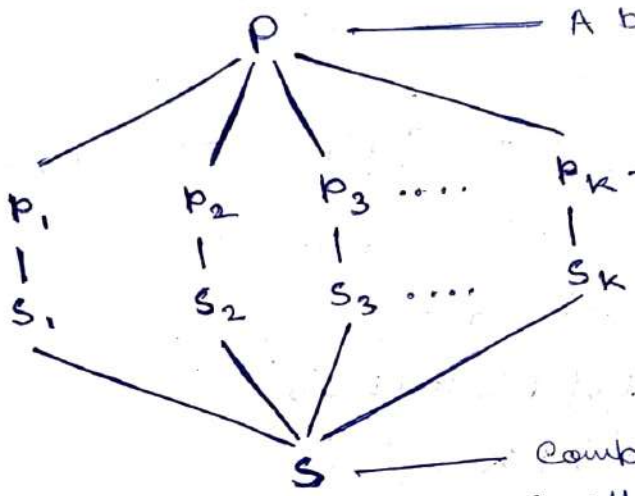
$$\therefore c(n) \leq \log_2 n + c(1)$$

$$c(n) \leq \log_2 n + 1$$

$$\therefore c(n) = O(\log n)$$

NOTES:

• Divide and Conquer :




```
DAC(P) {
    if (Small(P))
        Solve(P);
    Else {
        divide P into P1, P2, P3, ..., Pk
        DAC(P1), DAC(P2), ..., DAC(Pk)
        Combine(DAC(P1), DAC(P2), ..., DAC(Pk))
    }
}
```

<Merge Sort>

```
Algorithm MergeSort (A, l, u){ ——— T(n)
    if(l<u){
        mid := (l+u)/2; ——— 1
        MergeSort (A, l, mid); ——— T(n/2)
        MergeSort (A, mid+1, u); ——— T(n/2)
        Merge (A, l, mid, u); ——— n
    }
}

Algorithm Merge (A, l, mid, u){
    M := A[l : mid];
    N := A[mid+1 : u];
    i:=0, j:=0, K:=0;
    while(i < length(M) and j < length(N)) do{
        if (M[i] < N[j])
            O[K++] := M[i++];
        else
            O[K++] := N[j++];
    }
    for (; i < length(M);) do
        O[K++] := M[i++];
    for (; j < length(N);) do
        O[K++] := N[j++];
    return O;
}
```

The algorithm ends when $n/2^k = 1 \Rightarrow k = \log n$

$\therefore T(n) = n\,T(1) + \log n + n \log n$

$= n + \log n + n \log n$

$\therefore O(n \log n)$

Expt.No.

Page No.
Date.

——— T(n)

— 1
); ——— T(n/2)
u); ——— T(n/2)
; ——— n

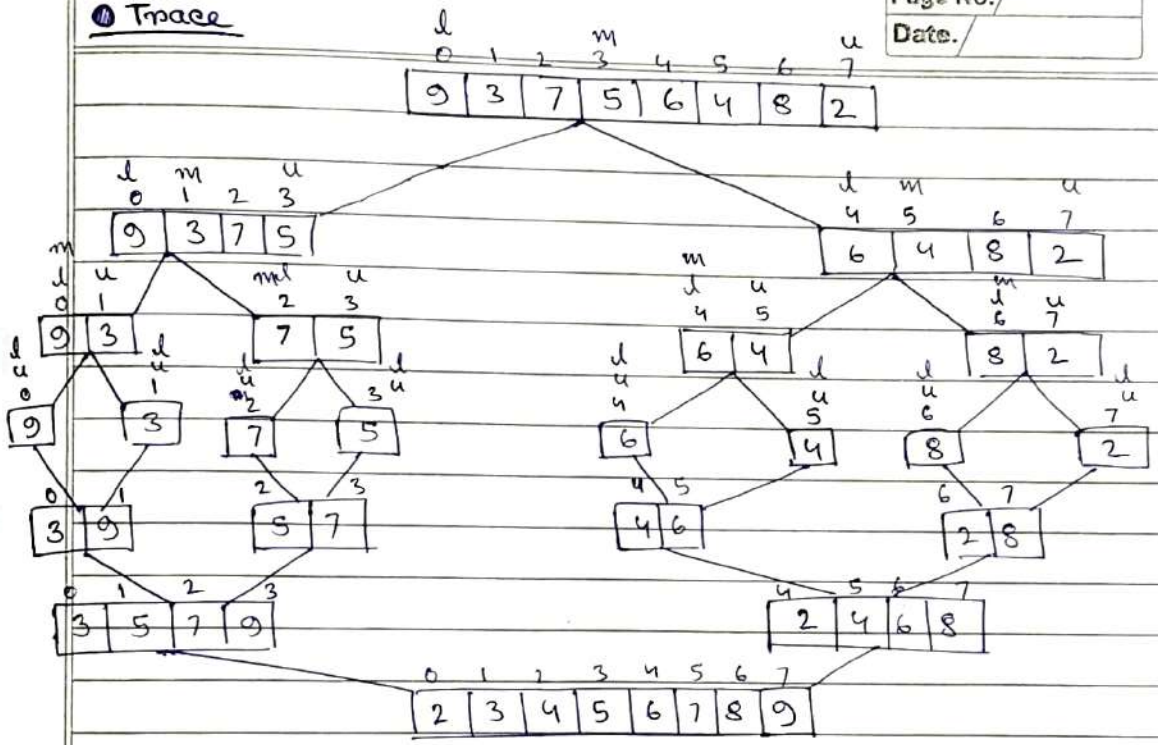
{

j < length(N)) do {

];

];

Trace



Analysis of MergeSort

Total time taken by the algorithm $T(n) = T(n/2) + T(n/2) + 1 + n = 2T(n/2) + 1 + n$

We can write the recurrence relation as ——

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n/2) + 1 + n & n > 1 \end{cases}$$

$\therefore T(n) = 2T(n/2) + 1 + n = 2 \times [2T(n/2^2)] + 1 + 1 + n + n$

$= 2^2 T(n/2^2) + 1 + 1 + n + n$

$= 2^3 T(n/2^3) + 1 + 1 + 1 + n + n + n$

$\vdots$

$= 2^k T(n/2^k) + k + k \cdot n$

The algorithm ends when $n/2^k = 1 \Rightarrow k = \log n$

$\therefore T(n) = n T(1) + \log n + n \log n$

$= n + \log n + n \log n$

$\therefore O(n \log n)$

Teacher's Signature .....

[MergeSort has no best or worst case, only average case — $n \log n$]

Pros of MergeSort

(i) It supports sorting very large list in $n \log n$ time.

(ii) Merge sort is suitable for linked lists. We can merge two sorted linked list into one sorted linked list.

(iii) Merge sort supports external sorting. Suppose we have two very large files in hard disk and we need to merge them in a single sorted file. The process of merging can only be done in main memory. We cannot just bring the whole files into main memory, merge them and then move it out of main memory as we have limited main memory. So, parts from the files can be merged in main memory and be moved back in hard disk and thus all the contents in the files can be merged in a sorted file using merge sort.

(iv) Merge sort is stable. Suppose we have some data like

$$8^{(1)}\ 6\ 4\ 3\ 8^{(2)}\ 5\ 9$$

where there are two 8s. $8^{(1)}$ and $8^{(2)}$ are nothing but to distinguishable notation. So after merge sort we get an expected list as $3\ 4\ 5\ 6\ 8^{(1)}\ 8^{(2)}\ 9$ where the order $8^{(1)}$ first and then $8^{(2)}$ is maintained.

Cons of MergeSort

(i) Merge sort is not an inplace sorting algorithm which means if we need to sort an array then the sorted elements should be kept in another array. It need extra space. [Not for linked lists only arrays].

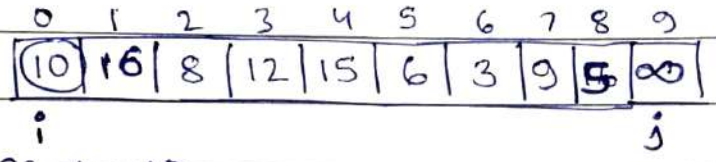
(ii) If the list to be sorted is small then merge sort is slower than algorithms having time complexity $O(n^2)$. For list of lenght ≤ 15, merge sort works slower than insertion sort [$O(n^2)$]. (Reason is recursion)

(iii) Merge sort uses recursion, meaning using stack and extra mem.

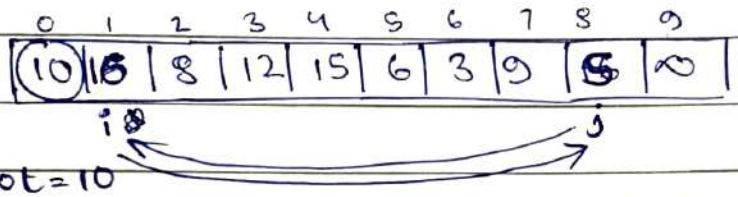
Expt.No.

Page No.
Date.

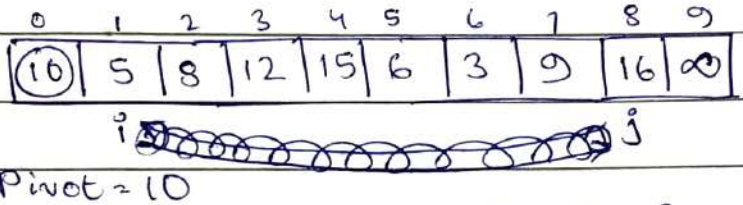
<Quick Sort>



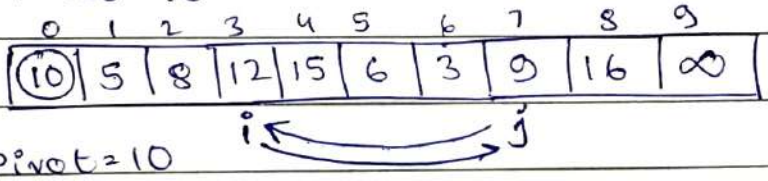
Pivot = 10



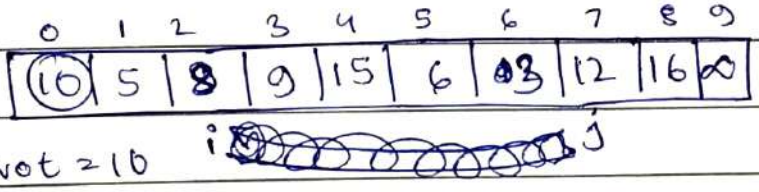
Pivot = 10



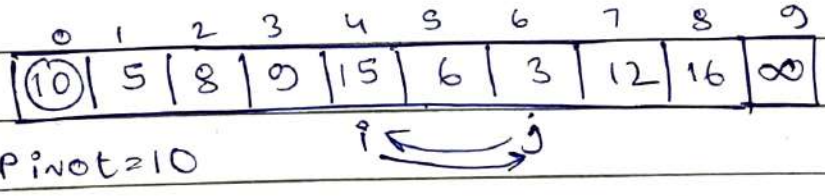
Pivot = 10



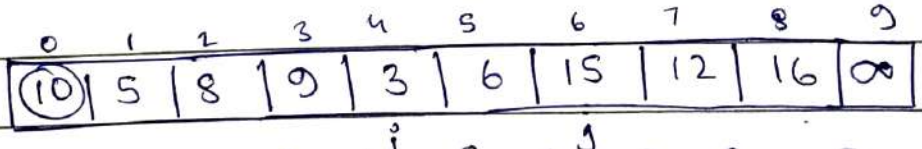
Pivot = 10

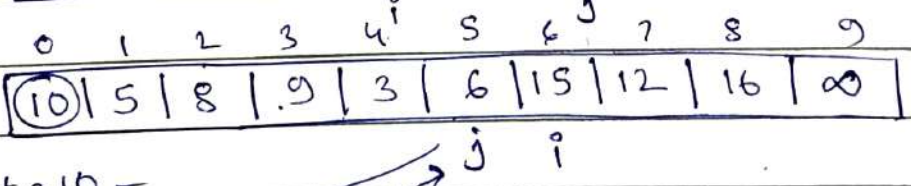


Pivot = 10

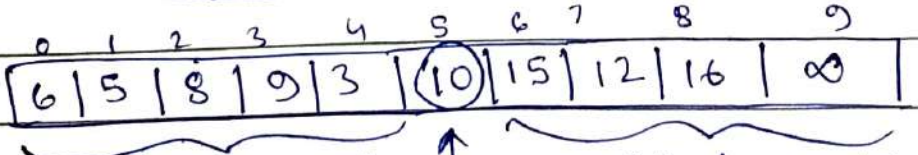


Pivot = 10





Pivot = 10



All elements < 10

At sorted pos

All elements > 10

Teacher's Signature .....

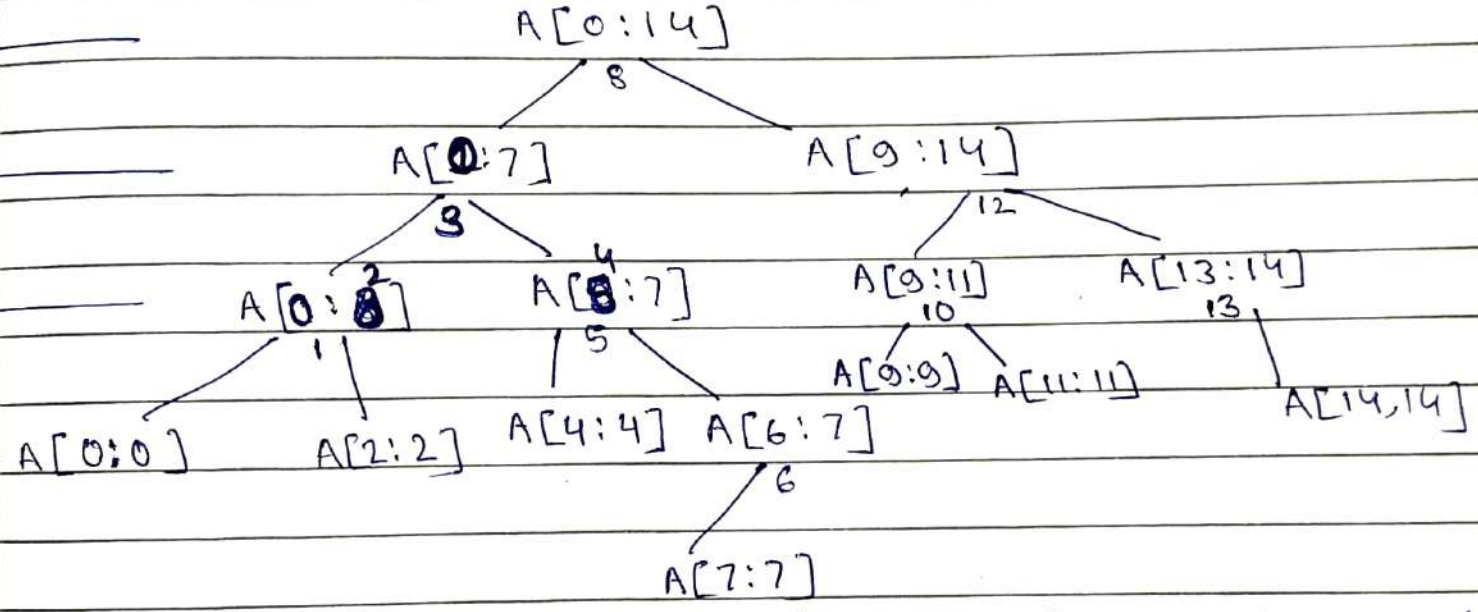
```
Algorithm Partition (low, high){
// low is the lower index and high is the higher
// index of the list.
        Pivot := A[low];
        i:= low, j:= high;
        while (i< j){
                do{
                  i++;
                }while(A[i] ≤ Pivot;
                do{
                  j--;
                }while (A[j] > Pivot);
                if(i<j)
                   swap(A[i], A[j]);
        }
        Swap(A[low], A[j]);
        return j;
}
Algorithm QuickSort(low, high){ ——— T(n)
        if (low < high){
n ———— j := Partition(low, high);
For best case only { T(n/2) ———— QuickSort(low, j);
                   { T(n/2) ———— QuickSort(j+1, high);
        }
}
```

$T(n) = 2T(n/2) + n = O(n \log n)$ [For best case]

Page No.
Date.

Analysis of Quick Sort

For **best case** schenario, the partitioning procedure is always done in the middle of a list, meaning the chosen pivot element is a median of all the elements which can randomly occur but we cannot achieve it. Suppose, we have a list of 15 elements say A[0:14]. The partitioning tree for best case is as below —



so, it basically dividing the list in half everytime until there is only one element left. If k times it is dividing the list then $n/2^k$ must be equal to 1 when the algorithm ends.

$$\therefore \frac{n}{2^k} = 1 \Rightarrow n = 2^k$$

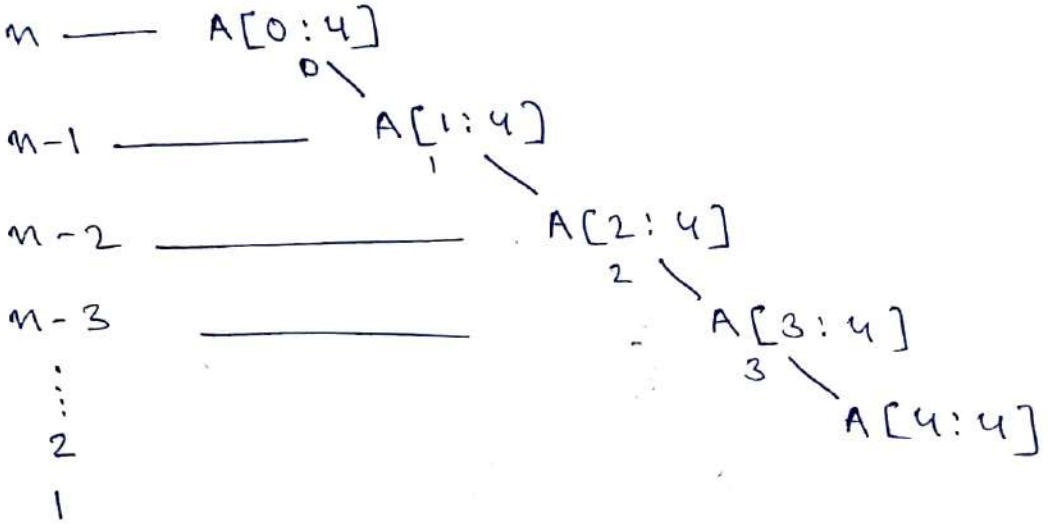
$$\Rightarrow k = \log_2 n$$

So, there are $\log n$ levels in the above partitioning tree and in each level n comparisons are made. So, the ~~time~~ best case time complexity for Quick sort is $O(n \log n)$

Teacher's Signature .....

For **worst case** schenario, say a list A[0:4] is already sorted.

Therefore, according to our partitioning method, partitioning will always be done at the beginning of the list. The partitioning tree will look like.



∴ In worst case, total time taken is $T(n) = 1+2+3+\cdots+n$

$$= \frac{n(n+1)}{2}$$

∴ $T(n) = O(n^2)$

∴ The worst case time complexity of quick sort is $O(n^2)$

Solution of worst case in quick sort

(i) If a given list is already sorted then the time take by quick sort is $O(n^2)$ as we ~~to~~ take the beginning element of the list as pivot. But if we always take the middle of the list as pivot then even if the list is already sorted, partitioning will always be done at the middle of the list and dividing the list always in two halves, converting the worst case into best case.

Recurrence Relation :

(1)

```
void test(int n){ ____ T(n)
    if(n>0){
        Printf("%d",n); ____ 1
        test(n-1); ____ T(n-1)
    }
}
```

∴ Total time taken by the above recursive function is

$$T(n) = T(n-1) + 1$$

This recurrence relation can be written as —

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1)+1 & n>0 \end{cases}$$

Now, we need to solve $T(n) = T(n-1) + 1$

$$\therefore T(n) = T(n-1) + 1$$
$$= [T(n-1-1)+1] + 1 = T(n-2) + 2$$
$$= [T(n-2-1)+1] + 2$$
$$= T(n-3) + 3$$
$$\vdots \text{ continuing for } k \text{ times}$$
$$\Rightarrow T(n) = T(n-k) + k$$

when $n-k = 0$, the algorithm stops.

$$\therefore n = k$$

$$\therefore T(n) = T(0) + n = 1 + n = \mathbf{\Theta(n)}$$

Trace :

test(3)
3
↓
test(2)
2
↓
test(1)
1
↓
test(0)
stop

So, when $n = 3$, we get $1+3 = 4$ calls.

(2)

$T(n)$	`void test(int n){`
1	`if (n>0){`
$(n+1)$	`for (i=0; i<n; i++){`
n	`printf("%d", n);`
	`}`
$T(n-1)$	`test(n-1);`
	`}`
	`}`

∴ Total time taken by this recursive function is

$$T(n) = T(n-1) + n + 1 + n + 1 = T(n-1) + 2n + 2$$

$$\Rightarrow T(n) \simeq T(n-1) + n$$

This recurrence relation can be written as —

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + n & n > 0 \end{cases}$$

Now, we need to solve $T(n) = T(n-1) + n$

$$\therefore T(n) = T(n-1) + n$$

$$= [T(n-1-1) + n-1] + n$$

$$= T(n-3) + 3n - 3$$

⋮ continuing k times

$$T(n) = T(n-k) + n \cdot k - (k-1)$$

$$= T(n-2) + (n-1) + n$$

$$= [T(n-3) + (n-2)] + (n-1) + n$$

$$= [T(n-4) + (n-3)] + (n-2) + (n-1) + n$$

⋮ continuing for k times

$$\Rightarrow T(n) = T(n-k) + (n-(k-1)) + (n-(k-2)) + (n-(k-3)) + \cdots + (n-1) + n$$

When $n-k=0 \Rightarrow n=k$, the algorithm stops,

$$\therefore T(n) = T(n-n) + (n-(n-1) + (n-(n-2)) + (n-(n-3)) + \cdots + n$$

$$= 1 + (1 + 2 + 3 + \cdots + n) = 1 + \frac{n(n+1)}{2}$$

$$\Rightarrow T(n) = \frac{n^2}{2} + \frac{n}{2} + 1 = \Theta(n^2)$$

(3)

```
T(n) ____ void test(int n){
           if(n>0){
                 for(i=0; i<n; i*=2){
log2n ____________ printf("%d", n);
                 }
T(n-1) ____ test(n-1);
           }
```

Total time taken by the above recursive function is —

$$T(n) = T(n-1) + \log_2 n$$

This recurrence relation can be written as —

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + \log_2 n & n > 0 \end{cases}$$

Here we need to solve $T(n) = T(n-1) + \log_2 n$

$$\therefore T(n) = T(n-1) + \log_2 n$$

$$= [T(n-1-1) + \log_2(n-1)] + \log_2 n$$

$$= T(n-3) + \log_2(n-2) + \log(n-1) + \log n$$

$$= T(n-4) + \log_2(n-3) + \log(n-1) + \log n$$

$$\vdots$$

$$\Rightarrow T(n) = T(n-k) + \log_2(n-(k-1)) + \log_2(n-(k-2)) + \cdots + \log_2(n-1) + \log_2 n$$

The algorithm stops when $n-k=0 \Rightarrow n=k$.

$$\therefore T(n) = T(n-n) + \log_2(n-n+1) + \log_2(n-n+2) + \cdots + \log_2 n$$

$$= 1 + \log_2 1 + \log_2 2 + \log_2 3 + \cdots + \log_2 n$$

$$= 1 + \log_2(1 \cdot 2 \cdot 3 \cdots n)$$

$$= 1 + \log_2 n!$$

$$\Rightarrow T(n) = 1 + \log_2 n! = O(n \log n)$$

(4)

```
Algorithm Test(int n){ ____ T(n)
    if(n>0){
        printf("%d", n); ____ 1
        Test(n-1); ____ T(n-1)
        Test(n-1); ____ T(n-1)
    }
}
```

Total time taken by the above recursive function is —

$$T(n) = 2T(n-1) + 1$$

This recurrence relation can be written as —

$$T(n) = \begin{cases} 1 & n = 0 \\ 2T(n-1) + 1 & n > 0 \end{cases}$$

Here, we need to solve $T(n) = 2T(n-1) + 1$

$\therefore T(n) = 2T(n-1) + 1$

$= 2[2T(n-1-1)+1] + 1$

$= 4T(n-2) + 2$

$= 4T[2T(n-2-1)+1] + 2$

$= 8T(n-3) + 3$

: continuing upto k times

$= 2^k T(n-k) + k$

Assuming $n-k = 0 \Rightarrow n = k$ we get

$T(n) = 2^n T(0) + n = 2^n \times 1 + n =$

$= 2[2T(n-1-1)+1] + 1$

$= 2^2 T(n-2) + 2 + 1$

$= 2^2[2T(n-2-1)+1] + 2 + 1$

$= 2^3 T(n-3) + 2^2 + 2 + 1$

$\vdots$

$= 2^k T(n-k) + 1 + 2^1 + 2^2 + \cdots + 2^{k-1}$

$= 2^k T(n-k) + 2^0 + 2^1 + 2^2 + \cdots + 2^{k-1}$

$= 2^k T(n-k) + 2^k - 1$

Assuming $n-k=0 \Rightarrow n=k$ —

$T(n) = 2^n T(0) + 2^n - 1$

$= 2^n \cdot 2^n - 1 = 2^{2n} - 1$

$= \Theta(2^n)$

[$\because a + ar + ar^2 + ar^3 + \cdots + ar^n$

$= \frac{a(r^{k+1}-1)}{r-1}$

$\therefore 2^0 + 2^1 + 2^2 + \cdots + 2^{k+1}$

$= \frac{2^{k-1+1}-1}{2-1} = 2^k - 1$]

Masters Theorem for decreasing functions

$T(n) = T(n-1) + 1$ —— $O(n)$.

$T(n) = T(n-1) + n$ —— $O(n^2)$

$T(n) = T(n-1) + \log n$ —— $O(n \log n)$

$T(n) = 2T(n-1) + 1$ —— $O(2^n)$

$T(n) = 3T(n-1) + 1$ —— $O(3^n)$

$T(n) = 2T(n-1) + n$ —— $O(n2^n)$

$T(n) = 3T(n-2) + \log n$ —— $O(3^{n/2} \cdot \log n)$

From the above observations we can say that the general recurrence relation for decreasing functions is —

$\boxed{T(n) = aT(n-b) + f(n)}$ where $a>0$, $b>0$ and $f(n) = O(n^k)$ where $k \geq 0$.

Here if $a=1$, $T(n) = O(n \cdot f(n))$

$= O(n \cdot n^k)$

$= O(n^{k+1})$

if $a>1$, $T(n) = O(a^{n/b} f(n))$

$= O(a^{n/b} \cdot n^k)$

if $a<1$, $T(n) = O(f(n)) = O(n^k)$

This is the Masters Theorem for decreasing functions.

(5)

```
Algorithm Test(int n){ ———— T(n)
    if(n>1){ ———— 1
        printf("%d", n); ———— 1
        Test(n/2); ———— T(n/2)
    }
}
```

Total time taken by the above algorithm is

$$T(n) = T(n/2) + 1$$

Now, the recurrance relation can be written as —

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n/2) + 1 & n > 1 \end{cases}$$

Here we need to solve $T(n) = T(n/2) + 1$

$$\therefore T(n) = T(n/2) + 1$$

$$= [T(n/2^2) + 1] + 1$$

$$= [T(n/2^3) + 1] + 1 + 1$$

$$= [T(n/2^4) + 1] + 1 + 1 + 1$$

$$\vdots \text{ continuing } k \text{ times}$$

$$= T(n/2^k) + k$$

When $n/2^k = 1$, the algorithm stops.

$$\therefore \frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$$

$$\therefore T(n) = \Theta(\log n)$$

(6)

```
Algorithm Test(int n){ ——— T(n)
    if(n>1){
    for(int i=0; i<n; i++){
        printf("%d", n); ——— n
    }
    Test(n/2); ——— T(n/2)
    }
}
```

Total time taken by the above algorithm is —

$$T(n) = T(n/2) + n$$

So, the recurrence relation can be written as follows —

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n/2) + n & n > 1 \end{cases}$$

$T(n) = T(n/2) + n$

$\Rightarrow T(n) = [T(n/2^2) + n/2] + n$

$= [T(n/2^3) + n/3] + n/2 + n$

$= [T(n/2^4) + n/4] + n/3 + n/2 + n$

⋮ continuing k times

$= T(n/2^k) + k \cdot n$ ——— (i)

The algorithm ends when $n/2^k = 1$

$\Rightarrow n = 2^k \Rightarrow k = \log_2 n$

$\therefore T(n) = T(n/2) + n$

$= \left[T(n/2^2) + n/2 \right] + n$

$= \left[T(n/2^3) + n/2^2 \right] + n/2 + n$

$= \left[T(n/2^4) + n/2^3 \right] + n/2^2 + n/2 + n$

$\vdots$

$\Rightarrow T(n) = T(n/2^k) + \frac{n}{2^{k-1}} + \frac{n}{2^{k-2}} + \cdots + \frac{n}{2^1} + n$

$= T(n/2^k) + n\left[1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right]$

when $n/2^k = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$, the algorithm ends

$\therefore T(n) = T(1) + n\left[1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}}\right]$

$\simeq 1 + n \cdot 1 = 1 + n$

$\Rightarrow T(n) = \Theta(n)$

⑦

```
Algorithm Test(int n){ ——— T(n)
    if(n>1){
        for(i=0; i<n; i++){
            stmt; ——— n
        }
        Test(n/2); ——— T(n/2)
        Test(n/2); ——— T(n/2)
    }
}
```

Total time taken by the algorithm is —

$$T(n) = 2T(n/2) + n$$

So, the recurrence relation can be written as —

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$

(6) $\therefore T(n) = 2T(n/2) + n$

$= 2\left[2T(n/2^2) + n/2\right] + n$

$= 2^2 T(n/2^2) + n + n$

$= 2^2\left[2T(n/2^3) + n/2^2\right] + n + n$

$= 2^3 T(n/2^3) + n + n + n$

$= 2^3\left[2T(n/2^4) + n/2^3\right] + n + n + n$

$= 2^4 T(n/2^4) + n + n + n + n$

$\vdots$

$\Rightarrow T(n) = 2^k T(n/2^k) + n\left[1 + \frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^{k-1}}\right]$

when $n/2^k = 1 \Rightarrow k = \log_2 n$, the algorithm ends.

$\therefore T(n) = nT(1) + n\left[1 + \frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^{k-1}}\right]$

$\simeq n + n$

$= 2n$

$\Rightarrow T(n) = 2^k T(n/2^k) + k \cdot n$

when $n/2^k = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$, the algorithm ends

$\therefore T(n) = nT(1) + \log n \cdot n$

$= n + n\log n$

$= \Theta(n\log n)$

Masters Theorem of division functions

The general recurrance relation for division functions is

$$\boxed{T(n) = aT(n/b) + f(n)}$$

where $a \geq 1$ and $b > 1$ and $f(n) = \Theta(n^k \log^p n)$

case 1 : If $\log_b a > k$ then $T(n) = \Theta(n^{\log_b a})$

Case 2: If $\log_b a = k$ then

If $p > (-1)$, $T(n) = \theta(n^k \log^{p+1} n)$

If $p = (-1)$, $T(n) = \theta(n^k \log \log n)$

If $p < (-1)$, $T(n) = \theta(n^k)$

Case 3: If $\log_b a < k$ then

If $p \geq 0$, $T(n) = \theta(n^k \log^p n)$

If $p < 0$, $T(n) = \theta(n^k)$

This is the master's Theorem for recurrence relation of division functions.

Ex ① $T(n) = 2T(n/2) + 1$

Ans: Here, $a = 2$, $b = 2$ and $f(n) = 1$

$= \theta(1)$

$= \theta(n^0 \log^0 n)$

So, $k = 0$ and $p = 0$

$\therefore \log_b a = \log_2 2 = 1$

$\therefore \log_b a > k$

$\therefore T(n) = \theta(n^1) = \theta(n)$

Ex ② $4T(n/2) + n$.

Ans. Here, $a = 4$, $b = 2$ and $f(n) = n = \theta(n)$

$= \theta(n^1 \log^0 n)$

$\therefore$ we can say from here that,

$k = 1$ and $p = 0$

$\therefore \log_b a = \log_2 4 = \log_2 2^2 = 2 \times 1 = 2$

$\therefore \log_b a > k$

$\therefore T(n) = \theta(n^2)$

Ex(3) $T(n) = 8T(n/2) + n$

Ans. Comparing $T(n) = 8T(n/2) + n$ with $T(n) = aT(n/b) + f(n)$ we get, $a = 8$, $b = 2$ and $f(n) = n = \Theta(n)$

$\therefore k = 1$ and $p = 0$ [$\because f(n) = \Theta(n) = \Theta(n^1 \log^0 n)$]

$\therefore \log_b a = \log_2 8 = \log_2 2^3 = 3$

$\therefore k < \log_b a$

$\therefore T(n) = \Theta(n^3)$ [If $k < \log_b a$, $T(n) = \Theta(n^{\log_b a})$]

Ex(4) $T(n) = 2T(n/2) + n$

Ans. $a = 2$, $b = 2$ and $f(n) = n = \Theta(n) = \Theta(n^1 \log^0 n)$

$\therefore k = 1$ and $p = 0$.

$\therefore \log_b a = \log_2 2 = 1$

$\therefore \log_b a = k$ and $p = 0$

$\therefore T(n) = \Theta(n \log n)$

Case 1:

$T(n) = 2T(n/2) + 1$ —— $\Theta(n)$

$T(n) = 4T(n/2) + 1$ —— $\Theta(n^2)$

$T(n) = 4T(n/2) + n$ —— $\Theta(n^2)$

$T(n) = 8T(n/2) + n^2$ —— $\Theta(n^3)$

$T(n) = 16T(n/2) + n^2$ —— $\Theta(n^4)$

Case 3:

$T(n) = T(n/2) + n$ —— $\Theta(n)$

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

$T(n) = 2T(n/2) + n^2 \log n$ —— $\Theta(n^2 \log n)$

$T(n) = 4T(n/2) + n^3 \log^2 n$ —— $\Theta(n^3 \log^2 n)$

$T(n) = 2T(n/2) + \frac{n^2}{\log n}$ —— $\Theta(n^2)$

Case 2:

$T(n) = T(n/2) + 1$ —— $\Theta(\log n)$

$T(n) = 2T(n/2) + n$ —— $\Theta(n \log n)$

$T(n) = 4T(n/2) + n^2$ —— $\Theta(n^2 \log n)$

$T(n) = 4T(n/2) + (n \log n)^2$ ——— $\Theta(n^2 \log^3 n)$

$T(n) = 2T(n/2) + \frac{n}{\log n}$ ——— $\Theta(n \log \log \log n)$

$T(n) = 2T(n/2) + \frac{n}{\log^2 n}$ ——— $\Theta(n)$

---

(8)

```
void Test(n){ ——— T(n)
    if(n>2){
        stmt; ——— 1
        Test(√n); ——— T(√n)
    }
}
```

∴ Total time taken by the algorithm is. $T(n) = T(\sqrt{n}) + 1$

we can write the recurrance relation as follows —

$$T(n) = \begin{cases} 1 & n = 2 \\ T(\sqrt{n}) + 1 & n > 2 \end{cases}$$

$\therefore T(n) = T(\sqrt{n}) + 1 = T(n^{1/2}) + 1$

$= [T(n^{1/2^2}) + 1] + 1 = T(n^{1/2^2}) + 2$

$= T(n^{1/2^3}) + 3$

$\vdots$

$\Rightarrow T(n) = T(n^{1/2^k}) + k$

when $n^{1/2^k} = 2$, the algorithm stops.

$\therefore n^{1/2^k} = 2 \Rightarrow \log n^{1/2^k} = \log 2 = 1$

$\Rightarrow \frac{1}{2^k} \log n = 1 \Rightarrow 2^k = \log n$

$\Rightarrow \log 2^k = \log \log n$

$\Rightarrow k \log 2 = \log \log n$

$\Rightarrow k = \log \log n$

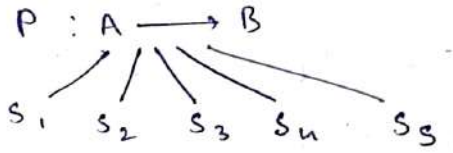
$\therefore T(n) = T(2) + \log \log n = 1 + \log \log n$

$\Rightarrow T(n) = \Theta(\log \log n)$

Greedy Method:

Optimization Problem:

Suppose, we have a problem P that is to travel from A to B. Now, there can be many solutions say S_1 : by foot, S_2: by cycle, S_3 : by bus, S_4: by train and S_5 : by flight.



So, there can be many solutions to a problem. Now if there is a constraint of completing the journey in 2hrs or less then only solution S_4 and S_4 is applicable.

The solutions to a problem which maintain some given constraints are called <u>feasible solutions</u> to the problem. There can be many feasible solutions to a problem.

But if there is another constraint of completing the journey at minimum cost then only solution S_4 is applicable.

A problem which requires either minimum or maximum result is known as <u>optimization problem</u> and the solution of the problem which is a feasible solution achieves the optimization objective of the optimization problem is called <u>optimal solution</u> of the optimization problem. There can only be one optimal solution.

Methods for solving Optimization Problem

- Greedy Method
- Dynamic Programming
- Branch and Bound

Greedy Method:

```
Algorithm Greedy(a, n){
//a[1:n] contains n inputs
    Solution := ∅  //Initialization of Solution set
    for i := 1 to n do {
        x := Select(a);
        if Feasible(Solution, x) then
            Solution := Union(Solution, x);
    }
    return Solution;
}
```

The function Select() selects an input from a[] and removes it. The selected input value is assigned to x. Feasible() is a boolean valued function that determines whether x can be included into Solution vector. The function Union() combines x with the solution and updates the objective function.

The above algorithm abstracts the subset paradigm of greedy approach. It suggests that one can devise an algorithm that works in stages considering one input at a time: At each stage, a decision is made regarding whether a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input into the partially constructed optimal solution results in an infeasible solution, the ~~result is~~ input is not added to the the partial solution else added. The above mentioned selection procedure itself is based one some optimization ~~prob~~ measure. This is basically the subset paradigm of the greedy approach.

[redacted] [a = [iPhone SE, one plus nord, K30 pro, note 10, note 9 pro, in note 1]

Suppose, I want a phone above ₹20,000 from a which is the best. Here my selection procedure gives me the set of feasible solutions as [iPhone SE, one plus nord, K30pro] I take each one of the inputs and based on my requirement

-t build up a selection procedure and the inputs that fills up my requirement add to the feasible solution set]

Knapsack Problem

Problem Statement ⇒ We are given n objects and a knapsack. Object i has weight w_i and the capacity of the knapsack is m. If a fraction of object i that is x_i where $0 \leq x_i \leq 1$ is put into the knapsack, a profit of $p_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. The contraint here is that the total weight of the objects put in the knapsack must be less or equal to m. So, formally, the problem can be stated as —

$$\text{maximize} \sum p_i x_i \quad \text{——— (1)}$$

$$\text{Subject to} \sum w_i x_i \leq m \quad \text{——— (2)}$$

$$\text{and } 0 \leq x_i \leq 1,\ 1 \leq i \leq n \quad \text{——— (3)}$$

A feasible solution of the problem stated above is any set $x = \{x_1, \ldots, x_n\}$ satisfying (2) and (3). A solution from the feasible solution set is called an optimal solution if for that solution (1) is satisfied.

Ex(1) Consider the following instance of knapsack problem: $n = 7$, $m = 15$, $(P_1, P_2, P_3, P_4, P_5, P_6, P_7) = (10, 5, 15, 7, 6, 18, 3)$ and $(w_1, w_2, w_3, w_4, w_5, w_6, w_7) = (2, 3, 5, 7, 1, 4, 1)$. Find out the optimal solution.

Ans:

Objects (O)	1	2	3	4	5	6	7
Profit (P)	10	5	15	7	6	18	3
weight (W)	2	3	5	7	1	4	1
P/W	5	1.66	3	1	6	4.5	3

The solution set x is written below —

x_i	x_1	x_2	x_3	x_4	x_5	x_6	x_7
$w_i x_i$	1				1		
$p_i x_i$					6		

$\therefore \sum w_i x_i = 1 +$

$\therefore \sum p_i x_i = 64$

$x = (x_1, x_2, x_3, x_4, x_5, x_6, x_7)$	$\sum_{i \in x} w_i x_i$	$\sum_{i \in x} p_i x_i$
$(1, 2/3, 1, 0, 1, 1, 1)$	$1+2+4+1+5+2 = 15$	$6+10+18+3+15+(2 \times 1.66) = 54.6$

Job Sequencing with Deadline

Problem Statement ⇒ We are given a set of n jobs. Associated with job i, there is an integer $d_i \geq 0$ and another integer $P_i > 0$ that is the deadline and the profit of the job respectively. For any job i, the profit P_i is earned iff the job is completed by its deadline. It takes one unit of time to complete a job and only one job can be done at any given time. A feasible solution is a subset J of jobs such that each job in subset J can be completed by its deadline. The value of a fasible solution J is the sum of the profits of the jobs in J or $\sum_{i \in j} P_i$. An optimal solution is a feasible solution with maximum value.

Ex ① Let $n = 4$, $\{P_1, P_2, P_3, P_4\} = \{100, 10, 15, 27\}$ and $\{d_1, d_2, d_3, d_4\} = \{2, 1, 2, 1\}$. Find the optimal solution.

0 – 1 – 2

Sol.

Job consider	Slot assign	Solution	Profit
–	–	ϕ	0
1	$\{(1,2)\}$	$\{1\}$	100
2	$\{(0,1),(1,2)\}$	$\{2,1\}$	100+10
3	$\{(0,1),(1,2)\}$	$\{2,1\}$	100+10

Job consider	Slot assign	Solution	Profit
4	{(0,1),(1,2)}	{4,1}	100+27

The optimal job sequence is $4 \longrightarrow 1$

and maximum profit is 127.

Ex ② Jobs : J_1 J_2 J_3 J_4 J_5

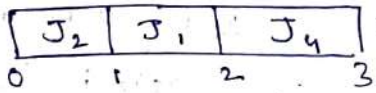
Profit : 20 15 10 5 1

Deadline : 2 2 1 3 3

Find maximum profit.

Ans.

Job considered	Slot assigned	Solution	Profit
–		ϕ	0
J_1	{(1,2)}	$\{J_1\}$	20
J_2	{(0,1),(1,2)}	$\{J_2, J_1\}$	20+15
J_3	{(0,1),(1,2)}	$\{J_2, J_1\}$	20+15
J_4	{(0,1),(1,2),(2,3)}	$\{J_2, J_1, J_4\}$	20+15+5
J_5	{(0,1),(1,2),(2,3)}	$\{J_2, J_1, J_4\}$	20+15+5



∴ The feasible job sequence is $J_2 \rightarrow J_1 \rightarrow J_4$

and maximum profit is 40

Ex ③ Let $n = 7$, $\{P_1, P_2, \ldots, P_7\} = \{35, 30, 25, 20, 15, 12, 2\}$ and $\{d_1, d_2, d_3, \ldots, d_7\} = \{3, 4, 4, 2, 3, 1, 2\}$. Find out optimal solution.

0 – 1 – 2 – 3 – 4

Ans.

J	Assigned slots	Job Considered	Action	Profit
ϕ	ϕ	–	–	0
$\{J_1\}$	{(2,3)}	J_1	Assign to (2,3)	35
$\{J_1, J_2\}$	{(2,3),(3,4)}	J_2	Assign to (3,4)	35+30
$\{J_1, J_2, J_3\}$	{(1,2),(2,3),(3,4)}	J_3	Assign to (1,2)	35+30+25
$\{J_1, J_2, J_3, J_4\}$	{(0,1),(1,2),(2,3),(3,4)}	J_4	Assign to (0,1)	35+30+25+2
$\{J_1, J_2, J_3, J_4\}$	{(0,1),(1,2),(2,3),(3,4)}	J_5	reject	35+30+25+2
$\{J_1, J_2, J_3, J_4\}$	{(0,1),(1,2),(2,3),(3,4)}	J_6	reject	35+30+25+2

$\{J_1, J_2, J_3, J_4$

∴ The opti

profit of

∴ The optim

Note

★ Follow the

exams

★★ If the

order then

then appl

Optimal M

If we ha

-tively

in O(n

merged

by repea

4 lists

We can

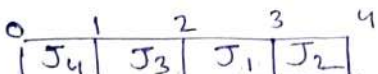
find t

are to

$\{J_1, J_2, J_3, J_4\}$	$\{(0,1),(1,2),(2,3),(3,4)\}$	J_7	reject	35+30+25+20

∴ The optimal solution is $J = \{J_1, J_2, J_3, J_4\}$ with a profit of 110.

∴ The optimal job sequence is



Note

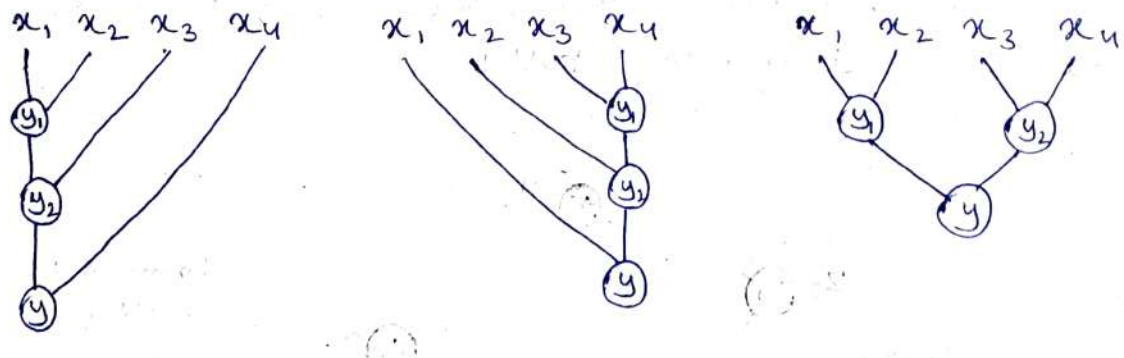
★ Follow the table shown in Ex ③ while doing math in exams

★★ If the profits in question are not given in descending order then arrange them in descending order first and then apply greedy approach.

```
Algorithm JobSequence(d, n){
// d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1 and n is total number of jobs. The jobs are ordered
// such that p[1] ≥ p[2] ≥ ... ≥ p[n].
// J[i] is the ith job in the optimal solution.
// At termination, d[J[i]] ≤ d[J[i+1]].
```

Optimal Merge Pattern (Used by Huffman Coding)

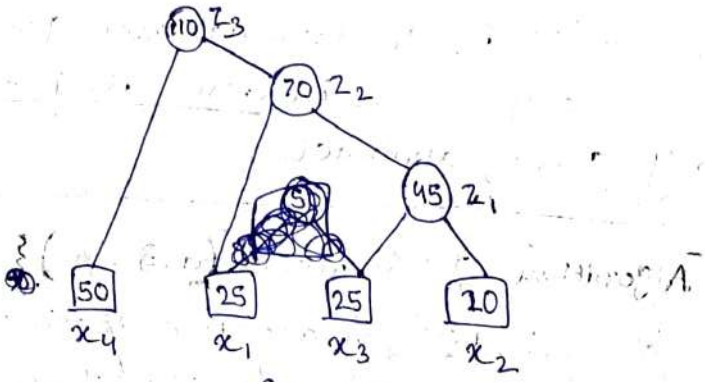
If we have two sorted lists with m and n elements respectively then we can merge them in a single sorted list in $O(n+m)$ time. When more than one lists are to be merged in a single sorted one then we can accomplish it by repeatedly merging sorted pair of lists. If we have 4 lists then we can merge them in following ways —



We can merge them in various ways but we want to find the one merge pattern in which minimum merges are to be made. So its an optimization problem which

requires the result to be minimized and we need ~~a sets~~ an ordering pair of the lists so that we can achieve the objective. Therefore, it fits in the ordering paradigm of greedy method.

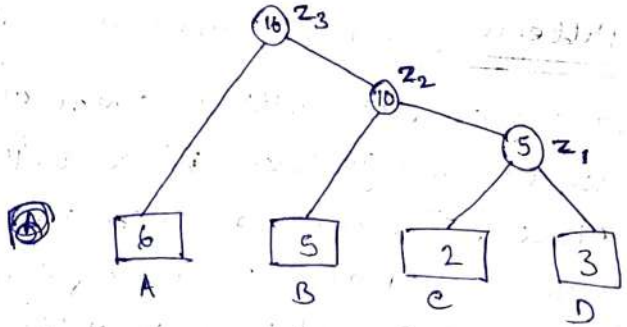
Since, merging of n elements list and m elements list requires possibly $(n+m)$ moves, the choice for selection criterion is : at each step merge two smallest lists together. Here is an example with four lists.



So, total 110 merges are performed in the above example which is the minimum one.

Ex① We have four lists $(A, B, C, D) = (6, 5, 2, 3)$. Find out the optimal merge pattern.

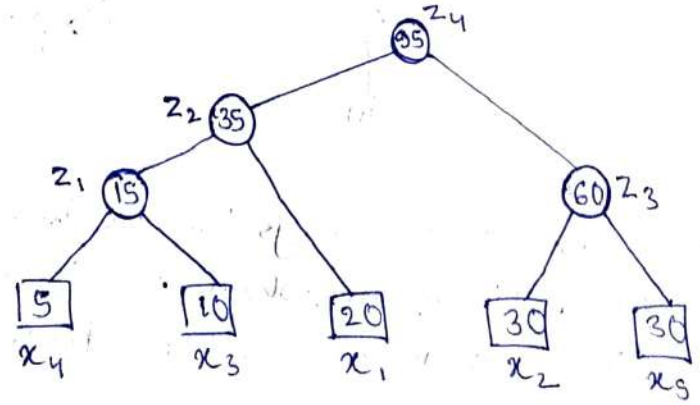
Ans.



∴ Total merging done = 5 + 10 + 16 = 31

Ex② We have five lists $(x_1, x_2, x_3, x_4, x_5) = (20, 30, 10, 5, 30)$. Find out the optimal merge pattern.

Ans.



∴ Total cost = 205

We can also find the cost by the formula given as

$$\sum_{i=1}^{n} d_i * x_i$$ [It is also called weighted external path length of the tree]

where d_i is the distance of the node from root node and x_i is the weight of the node [length of the file].

Huffman Coding

Message : BCCABBDDAECCBBAEDDCC

length of the message is 20 and each of the letters are represented in binary in 8 bits. So, to send a message of length 20, we need to send $20 \times 8 = 160$ bits.

But for the above message, we are not using all the characters and so we do not need 8 bits to represent each letter. For the above message —

character	count / frequency	code
A	3 / $\frac{3}{20}$	000
B	5 / $\frac{5}{20}$	001
C	6 / $\frac{6}{20}$	010
D	4 / $\frac{4}{20}$	011
E	2 / $\frac{2}{20}$	100

For the above message we need only 3 bits, meaning the size of the message reduces to $3 \times 20 = 60$ bits. But in the receiver side also we need to transfer a table containing the actual ASCII codes of the letters used associated with the reduced codes as shown below -

A (8 bits)	000
B (8 bits)	001
C (8 bits)	010
D (8 bits)	011
E (8 bits)	100

clearly it takes $8 \times 5 + 3 \times 5 = 40 + 15 = 55$ bits to represent the table in binary.

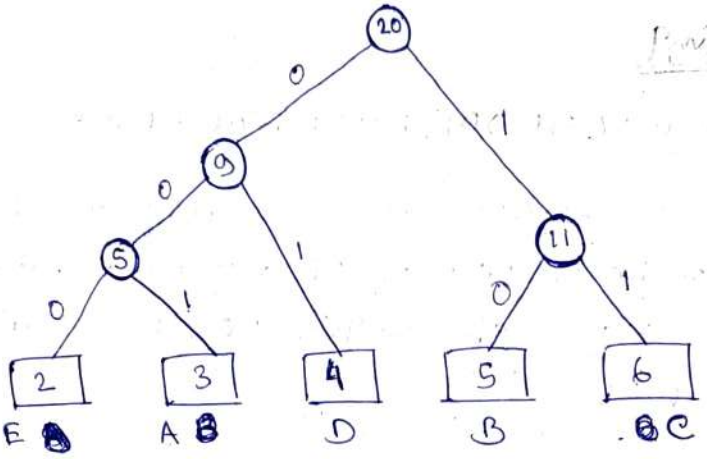
Therefore, total we need to send $60 + 55 = 115$ bits which is also 45 bits lesser than the message which uses 8 bits for each character.

This type of codes are called fixed size codes. Huffman code is variable size code which also reduces the size.

Huffman code follows optimal merge pattern

Message : BCCABBDDAECCBBAEDDCC

character	A	B	C	D	E
count	3	5	6	4	2



Optimal Merge Pattern Tree for given message

So, Huffman codes for the alphabets are given below —

character	Code
A	001
B	10
C	11
D	01
E	000

To represent this table in binary, we need $5\times8 + 3\times2 + 3\times2$ $= 40 + 6 + 6 = 52$ bits and to represent the whole message in binary, we need

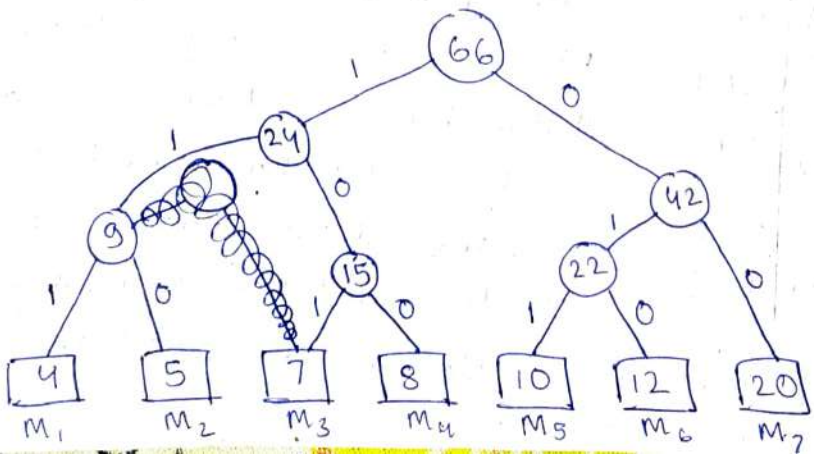
$3\times2 + 3\times3 + 2\times4 + 2\times5 + 2\times6$

$= 6 + 9 + 8 + 10 + 12 = 45$ bits.

So, total we need to send only $52 + 45 = 97$ bits.

Ex(1) Obtain a set of optimal Huffman codes for the messages $(M_1, \dots, M_7)$ with relative frequencies $(q_1, \dots, q_7) = (4, 5, 7, 8, 10, 12, 20)$. Draw the decode tree.

Ans.



Optimal Huffman codes for given message is given as below.

Character	Code
M_1	111
M_2	110
M_3	101
M_4	100
M_5	011
M_6	010
M_7	00

Minimum Cost Spanning Tree

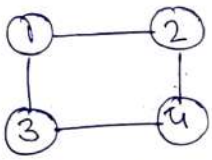
Spanning Tree : A connected subgraph S of graph G_2 (G_2 is defined as (V, E) where V is set of vertices and E is set of edges) is said to be a spanning tree iff

(i) S contains all the vertices in G_2

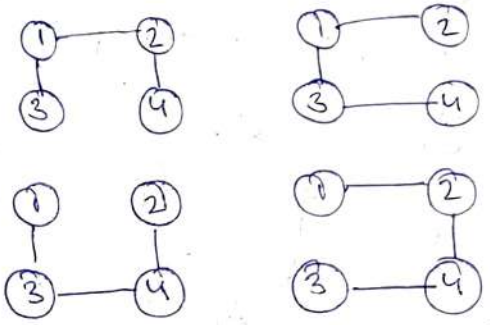
(ii) S contains $|V| - 1$ edges.

or in other words, S is defined as (V', E') where $V' = V$ and $|E'| = |V| - 1$

Example



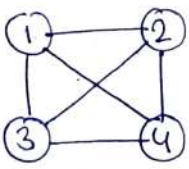
Graph G_2



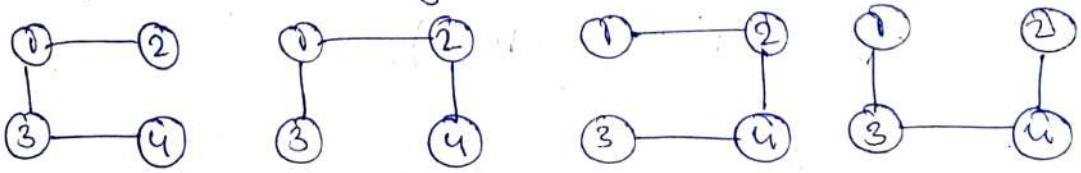
All possible spanning trees of G_2

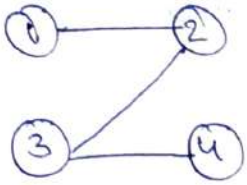
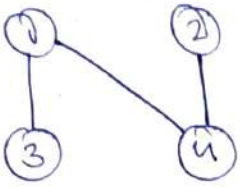
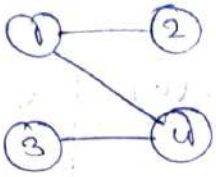
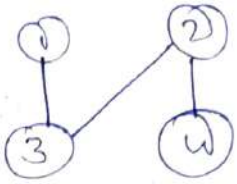
Spanning Trees of a complete graph

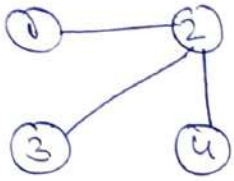
Suppose, G_2 is a complete graph as below —

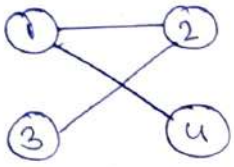


The all possible spanning trees of G_2 are



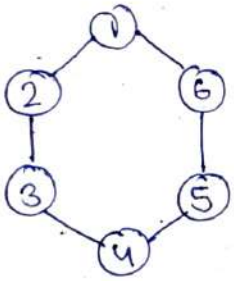

... 3 more of this kind


... 3 more of this kind

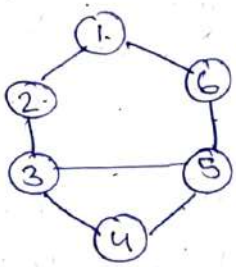
There are total 16 spanning trees of the above subgra-ph.

Formulae If a complete graph C_2 has n vertices then it has total n^{n-2} possible spanning trees.

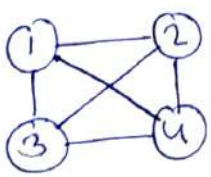
Spanning Trees of a Graph


Graph C_2

From C_2, we can form ${}^6C_5 = 6$ different spanning trees.


Graph C_1

But from C_1, we cannot form ${}^7C_5 = 21$ spanning trees. we need to calculate and subtract those cases where cycles are possible. In this case 2 cycles are possible. So, we can form $21-2=19$ spanning trees from C_1


Graph C_3

No. of possible spanning trees of any graph C_2 = no. of ~~verti~~ edges $C_{(\text{no. of vertices})-1}$ − no. of cycl

For graph C_3 we can have ${}^6C_3 - 4 = 16$ spanning trees (which is equal to the above formula for complete graphs).

Greedy Methods to find minimum cost spanning tree

- Prim's Algorithm
- Kruskal's Algorithm

Prim's Algorithm

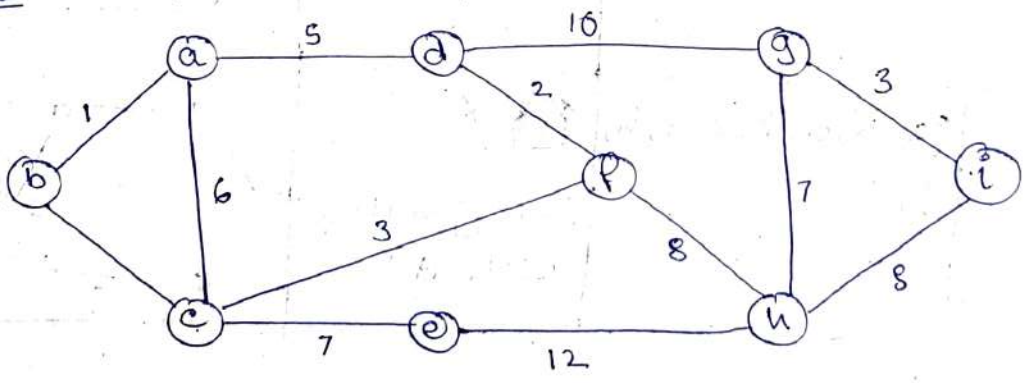
Here is the algorithm for Prim's Algorithm. Here **V** is the set of vertices ~~does~~ in the graph. U is the visited vertex set that contains the vertices which are included in the MST. T set contains the edges in the MST that is initially ϕ. The approach is to always find the minimum connected edge.

```
T = ϕ ;
U = {1} ; // The 1st vertex is visited.
while(U ≠ V) // V is set of vertices of the
{                    original graph.
    Find (u,v) , the lowest cost edge
    such that u∈U and v∈ V-U ;
    T = TU{(u,v)};
    U = U ∪ {v}
}
```

Union operation

~~It's~~

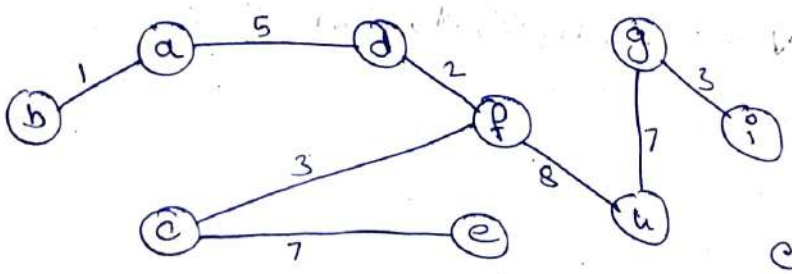
Example



$V = \{a, b, c, d, e, f, g, h, i\}$

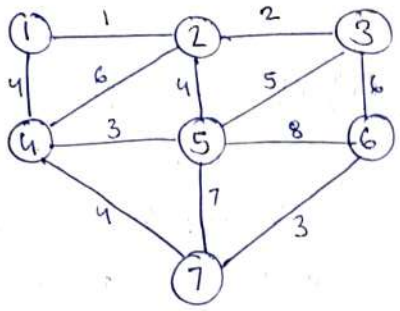
Step	Visited Vertex (U)	Unvisited Vertex	Minimum cost tree.
Initialize	{a}	{b,c,d,e,f,g,h,i}	a
1	{a,b}	{c,d,e,f,g,h,i}	a — b (1)
2	{a,b,d}	{c,e,f,g,h,i}	a — b (1), a — d (5)
3	{a,b,d,f}	{c,e,g,h,i}	a — b (1), a — d (5), d — f (2)
4	{a,b,d,f,c}	{e,g,h,i}	a — b (1), a — d (5), d — f (2), f — c (3)
5	{a,b,d,f,c,e}	{g,h,i}	a — b (1), a — d (5), d — f (2), f — c (3), c — e (7)
6	{a,b,d,f,c,e,h}	{g,i}	a — b (1), a — d (5), d — f (2), f — c (3), c — e (7), f — h (8)
7	{a,b,d,f,c,e,h,g}	{i}	a — b (1), a — d (5), d — f (2), f — c (3), c — e (7), f — h (8), g — h (7)
8	{a,b,d,f,c,e,h,g,i}	∅ (every vertex visited)	a — b (1), a — d (5), d — f (2), f — c (3), c — e (7), f — h (8), g — h (7), g — i (3)

The mst is —



Cost = 36

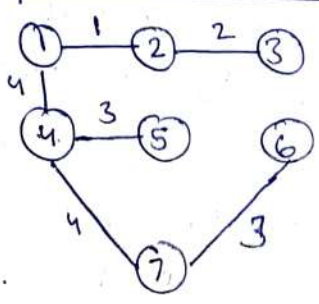
Example



$V = \{1, 2, 3, 4, 5, 6, 7\}$

Step	Visited vertex(U)	Unvisited vertex(U-V)	MST
Initialize	{1}	{2,3,4,5,6,7}	1
1	{1,2}	{3,4,5,6,7}	1 —1— 2
2	{1,2,3}	{4,5,6,7}	1 —1— 2 —2— 3
3	{1,2,3,4}	{5,6,7}	1 —1— 2 —2— 3 4 | 4
4	{1,2,3,4,5}	{6,7}	1 —1— 2 —2— 3 4 | 4 —3— 5
5	{1,2,3,4,5,7}	{6}	1 —1— 2 —2— 3 4 | 4 —3— 5 — 4 — 7
6	{1,2,3,4,5,7,6}	∅ (All vertices are visited)	1 —1— 2 —2— 3 4 | 4 —3— 5 — 4 — 7 — 3 — 6

The MST is —



cost = 17

Kruskal's Algorithm

Points on Prim's Algorithm : (i) If we have a disconnected graph then we cannot find the MST for this graph at all because we know that an MST should be connected. But the approach can be to find out the MSTs of the different components of the disconnected graph. Using Prim's, we can find the MST of one component of any disconnected graph.

(ii) At each step in Prim's Algorithms, the to be formed MST is always connected.

Time Complexity of Prim's Algorithm :

If the graph is represented using adjacency list, then using BFS algorithm, all the vertices can be traversed in $O(V+E)$ time where V = no. of vertices and E = no. of edges.

Using min heap data structure, we can get the minimum weight edge in $O(\log V)$ time.

Overall time complexity of Prim's algorithm is —

$$O(V+E) \cdot O(\log V)$$

$$= O(\log V \cdot (V+E)) = O(V \log V + E \log V).$$

$$\approx O(E \log V)$$

[Using **Fibonacci Heap**, we can reduce the complexity from $O(V \log V + E \log V)$ to $O(E + V \log V)$.]

Kruskal's Algorithm

Here is the Kruskal's algorithm —

Step 1: Sort E by increasing order of weight
// E is the set of edges in the graph

Step 2: $n \leftarrow$ Number of vertices in the graph

Step 3: $T \leftarrow \phi$ // T is the set to contain the edges of the MST.

Step 4: Initialize n sets containing a different element

of V // V is the set of vertices of the graph

Step 5: Repeat the following steps until T contains (n-1) edges.

i. e ← {u, v} ← shortest edge not yet considered from the sorted set of edges.

ii. Find a set from the previously formed sets of vertices that contains u, call it U_Component.

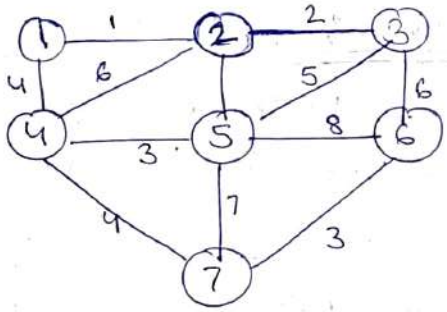
iii. Similarly find a set from those sets of vertices that contains v and call it V_Component.

iv. If U_Component and V_Component are the same set, it means that already a path between vertex u and v exists. So, do not include e into MST.

Else, Merge the sets, U_Component and V_Component and includ e into T.

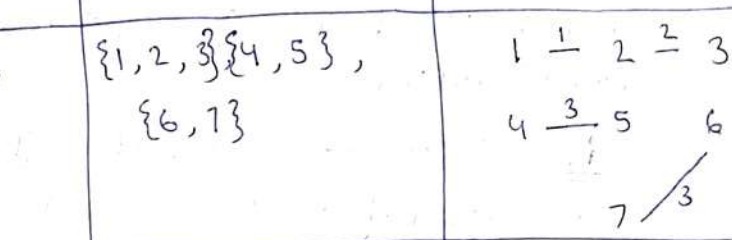
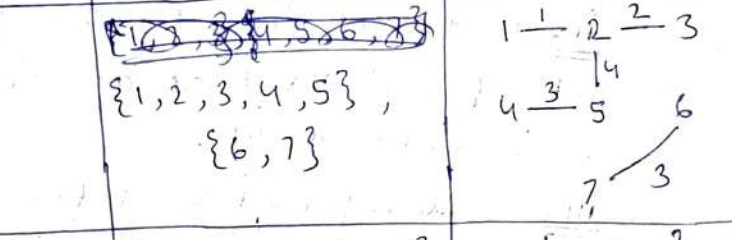
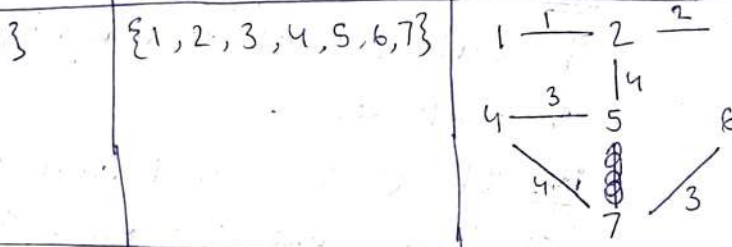
Step 6: Return T.

Example

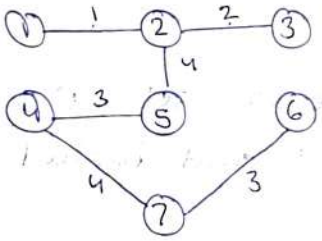


1 2 3 4 5 6 7

Step	Vertex Considered	Vertex Sets	MST
Initialize	∅	{1}, {2}, {3}, {4} {5}, {6}, {7}	
1	{1, 2}	{1, 2}, {3}, {4}, {5}, {6}, {7}	1 —1— 2
2	{2, 3}	{1, 2, 3}, {4}, {5}, {6}, {7}	1 —1— 2 —2— 3
3	{4, 5}	{1, 2, 3} {4, 5}, {7}, {6}	1 —1— 2 —2— 3 4 —3— 5

Step	Vertex considered	Vertex sets	MST
4	{6,7}	{1,2,3}{4,5}, {6,7}	
5	{2,5}	{1,2,3,4,5}, {6,7}	
6	{4,7}	{1,2,3,4,5,6,7}	

∴ The MST is —



Cost : 17

Points on Kruskal's Algorithm :

(i) If we have a disconnected graph then unlike Prim's Algo-rithm, Kruskal's Algorithms finds out MST for each of the components.

(ii) At each step of Kruskal's Algorithm, the to be formed MST may or maynot be connected.

(iii) If the given have all the edges having distinct weights then Prim's algorithm and Kruskal's algorithm both gives the exactly same MST. Otherwise both MSTs maynot be the same one.

Preferences

i) Kruskals algorithm is preferred when —
the graph is sparse, there are less no. of edges and the edges can be sorted in linear time.

) Prim's algorithm is preferred when —

the graph is dense, ~~the~~ and there are large number of ~~of~~ edges in the graph.

Time Complexity of Kruskal's Algorithm :

If we have $|V|$ vertices and $|E|$ edges then we need to select $|V|-1$ edges. So total time taken by this algorithm is $\theta(|V||E|) = \theta(ne)$ where n = no. of vertices and e = no. of edges.

By using min heap, this can be reduced to $\theta(e \log n)$.

Single Source Shortest Path

Dijkstra's Algorithm

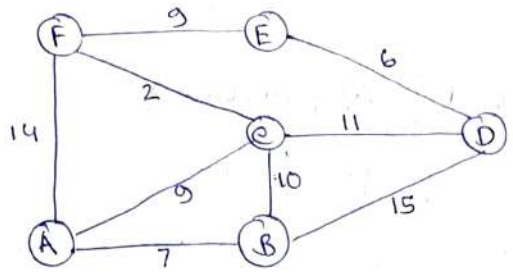
The Dijkstra's single source shortest path algorithm is given as below —

```
Algorithm Dijkstra(G, S){
// Input : The graph G and source vertex S is the
  input to the algorithm.
    for each vertex v in G do{
        distance[v] ← ∞
        previous[v] ← NULL
        If v ≠ S, add v to Q.
    }
    distance[S] ← 0
    while Q is not empty do{
        ~~Extract minimum node from Q~~
        U ← Extract v' from Q such that
          distance[v'] is the minimum one.
        For each unvisited neighbour V of U
        do{
            t ← distance[U] + weight of the
              edge (U, V).
            If t < distance[V] then{
                distance[V] ← t
                previous[V] ← U
    } }
}
```

return distance[], previous[]

3

Example

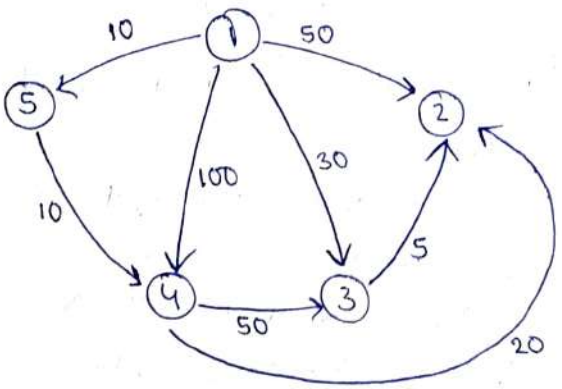


Take A as source.

Source Vertex

Vertex Considered	distance A ✓	B ✓	C ✓	D ✓	E ✓	F ✓	Graph
-	[0]	∞	∞	∞	∞	∞	
A	0	[7]	9	∞	∞	14	A
B	0	7	[9]	22	∞	14	A —7— B
C	0	7	9	20	∞	[11]	A —9— C, A —7— B
F	0	7	9	[20]	20	11	F —2— C, A —9— C, A —7— B
D	0	7	9	20	[20]	11	F —2— C —11— D, A —9— C, A —7— B
E	0	7	9	20	20	11	F —9— E, F —2— C —11— D, A —9— C, A —7— B

Example



Take 1 as source

Step	Vertex Considered	Vertex not visited	Visited vertex	distance 1	2	3	4	5
Init	-	1, 2, 3, 4, 5	-	[0]	∞	∞	∞	∞
1	1	2, 3, 4, 5	1	0	50	30	100	[10]
2	5	2, 3, 4	1, 5	0	50	30	[20]	10
3	4	2, 3	1, 5, 4	0	40	[30]	20	10
4	3	2	1, 5, 4, 3	0	[35]	30	20	10
5	2	-	1, 5, 4, 3, 2	0	35	30	20	10

Time complexity of Dijkstra's Algorithm:

Relaxation is the method used by Dijkstra's Algorithm to find out that from any vertex whether a vertex in the ~~graph~~ graph can be reached using lesser cost and if so then modify its cost to be the new lesser cost. Now, at each step, if there are a total of n vertices then Dijkstra's algorithm can be relaxing at most $(n-1)$ vertices for each of the n vertices. So, the worst case time of Dijkstra's algorithm is $\Theta(n^2)$.

Points on Dijkstra's Algorithm:

(i) ~~Dijkst~~ Dijkstra's algorithm works on directed as well as non directed graphs.

(ii) The drawback of dijkstra's algorithm is that in case where there are ~~edg~~ one or more edges having negative weight, it ~~may~~ does not always work. But it may work. Depends upon the graph.

Dynamic Programming :

The purpose of Dynamic Programming and greedy method is the same which is to solve optimization problems. But both have different approaches. The differences between both approach are —

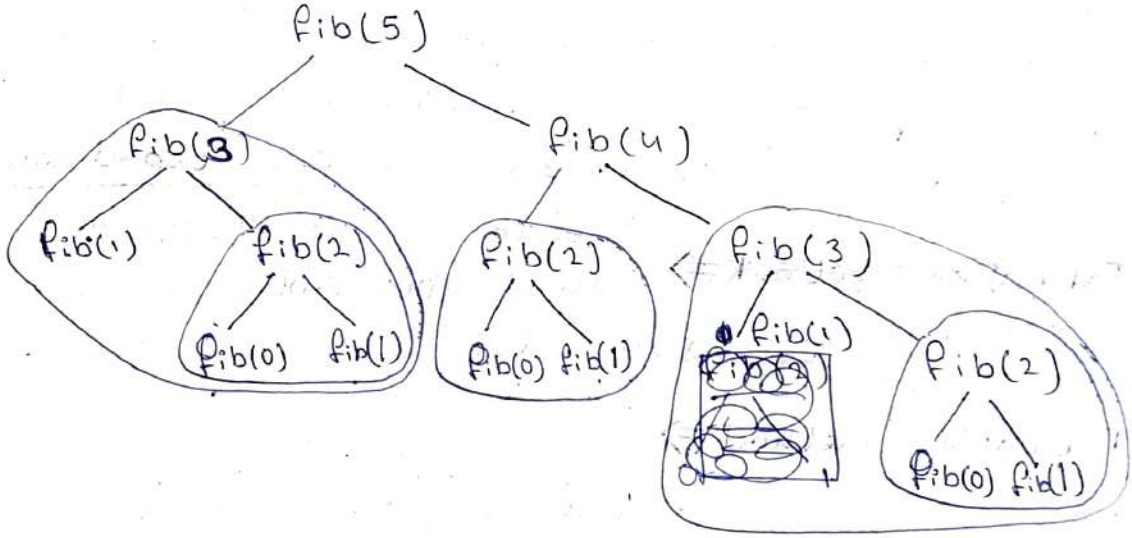
Greedy Method	Dynamic Programming
(i) In greed approach, we make ~~one~~ the decision of choosing optimal solution once and at each step, follow the predefined decision. Like in kruskal's, we always take the minimum edge.	(i) In dynamic programming, **principle of optimality** is used which says that a sequence of decisions is to be made to get to the optimal solution. At each step we make the decision which is not predefi-ned in dynamic programmin-g.
(ii) In greedy method, sometimes there is no such gurantee of getting optimal solution. As in case of dijkstra's algorithm if we have a graph containing negative edges then optimal solution is not guranteed.	(ii) In dynamic programming, we choose the optimal solu-tion by considering all the possible feasible solutions. So, optimal solution is guran-teed.
(iii) Greed method follows the problem solving heuristic of making the locally optimal choice at each stage.	(iii) Dynamic programming is an algorithmic technique that is usually based on a recurr-ent formula that uses some previously calculated states.
(iv) Greedy method is more effi-cient in terms of memory as it never looks back or revise previous choices.	(iv) Dynamic programming approa-ch requires a table for memoization and it increases its space complexity.
(v) Greedy approach is generally faster.	(v) Dynamic programming is slower than greedy approach

Here is the recursive definition and function of finding n^{th} fibonacci term.

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(n-2) + f(n-1) & \text{if } n > 1 \end{cases}$$

```
int fib(int n){
    if(n<=1){
        return n;
    return fib(n-2) + fib(n-1);
}
```
—— $T(n-2) + T(n-1) +$

Tracing tree of function fib(5) is given below —



For the above recurrence function, we can write

$$T(n) = T(n-2) + T(n-1) + 1 \approx 2T(n-1) + 1$$

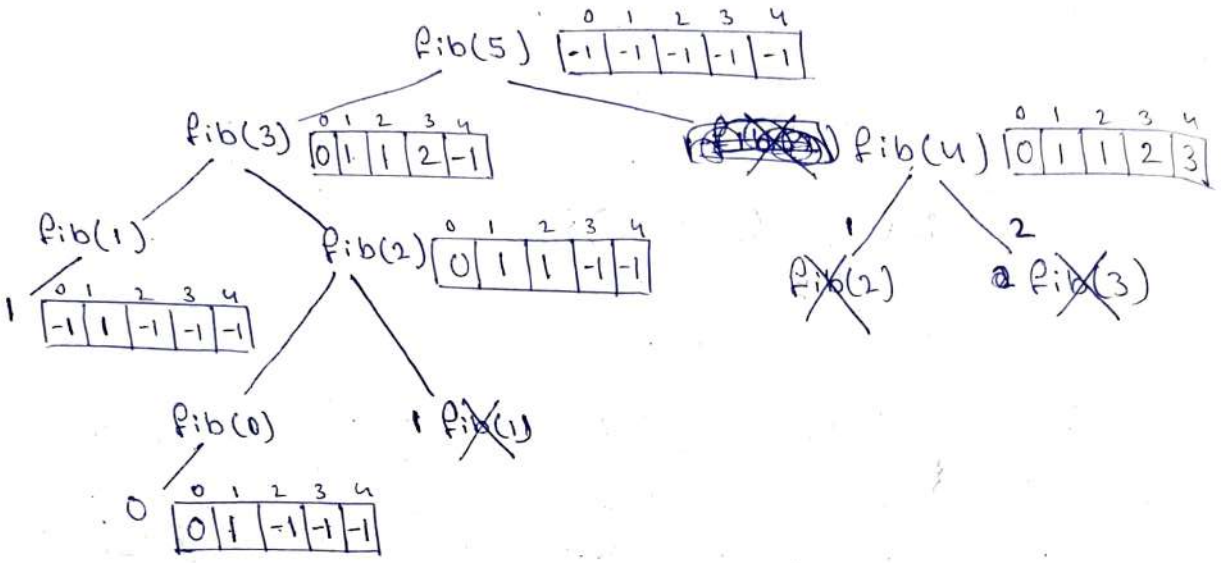
By Master's Theorem of decreasing function, $T(n) = \Theta(2^n)$

Clearly it is taking exponential time. We can reduce this by avoiding multiple same function calls like fib(3) has been called 2 times, fib(2) 3 times, fib(1) 5 times and fib(0) 3 times.

Memoization ⇒

Using the concept of **memoization**, we can use an array of length n to store the return values of corresponding function calls to avoid it to be called

multiple times. We initialize each value in the array say F with (-1). Lets have the recursion tree using memoization for fib(5).



So, for fib(n) we are now getting (n+1) function calls instead of 2^n. From exponential to linear, a huge reduce in time. Memoization follows **top-down approach** because as we can see that we are starting from fib(5), getting upto fib(0) and then again back upwards.

Tabulation Method ⇒

Here is a function to find nth term of fibonacci series in iterative method.

```
int fib(int n){
    if(n<=1)
        return n;
    F[0] = 0, F[1] = 1;
    for(i = 2; i <= n; i++)
        F[i] = F[i-2] + F[i-1];
    return F[n];
}
```

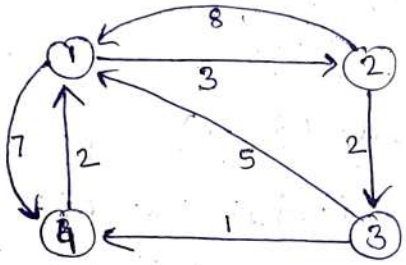
It is also the same approach as memoization but it uses iteration instead of recursion and **bottom-top approach** because here to find fib(5) we are going from 0 to 5.

All Pair Shortest Path

Floyd - Warshall Algorithm

All pair shortest path problem statement : Let $G_2(V, E)$ be a directed graph with n vertices. Let cost be a cost adjacency matrix for G_2 such that cost (i, j) is the cost of the edge $\langle i, j \rangle$ if $\langle i, j \rangle \in E(G_2)$ and cost $(i, j) = \infty$ if $i \neq j$ and $\langle i, j \rangle \notin E(G_2)$. The all pair shortest path problem is to determine a matrix A such that $A(i, j)$ is the cost of a shortest path from i to j.

Ex (1) Find shortest path between every pair of vertices in the following given graph.



Ans.

$A^0 =$

	1	2	3	4
1	0	3	∞	7
2	8	0	2	∞
3	5	∞	0	1
4	2	∞	∞	0

(Initial)

$A^1 =$

	1	2	3	4
1	0	3	∞	7
2	8	0	2	15
3	5	8	0	1
4	2	5	∞	0

(considering vertex 1 as sre)

$A^2 =$

	1	2	3	4
1	0	3	5	7
2	8	0	2	15
3	5	8	0	1
4	2	5	7	0

$A^3 =$

	1	2	3	4
1	0	3	5	6
2	7	0	2	3
3	5	8	0	1
4	2	5	7	0

$A^4 =$

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

(final ~~ans~~ ans)

Formula : $A^k(i,j) = \min\{[A^{k-1}(i,k) + A^{k-1}(k,j)], A^{k-1}(i,j)\}$

```
Algorithm Floyd_Warshall (G2, n){
// G2 is the graph and n is the number of vertices
// cost is a 2D array which is the cost adjacency
// matrix of G2 such that cost[i,j] = cost of the
// edge (i,j).
        A := cost   // A is initialized with cost
        k := 1
1+n ——— while (k≤n), do{
            i:= 1
(1+n)n ——— while(i ≤n),do{
  n.n ———————  j := 1
(1+n)n.n ——————— while (j ≤n),do{
  nn.n ———————— A[i,j]:= min(A[i,j],
                      A[i,k] + A[k,j])
                  j := j + 1
                }
            i := i + 1
          }
          k:= k+1
        }
        return A
}
```

Time complexity of Floyd Warshall Algorithm is $\underline{\theta(n^3)}$

0/1 Knapsack Problem

Problem Statement : Formally, 0/1 knapsack problem is defined as —

maximize $\sum p_i x_i$ [$\because p_i x_i$ = Profit earned for a fraction x_i of object i]

subject to $\sum w_i x_i \leq m$ [$w_i x_i$ = weight of a fraction x_i of object i]

and $x_i = 0$ or 1, $1 \leq i \leq j$ m = capacity of knapsack

Ex① Given the itemlist along with their profit and weights are $P = \{1, 2, 5, 6\}$ and $w = \{2, 3, 4, 5\}$. The capacity of the knapsack is 8.

Ans.

P	w	i \ w	0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0	0	1	2	5	6	6	7	8

Formula : $v[i, w] = \max\{v[i-1, w], v[i-1, w-w[i]] + p[i]\}$

x_1 x_2 x_3 x_4 $\quad 8-6=2$

0 1 0 1 $\quad 2-2=0$

Ex② $P = \{1, 4, 5, 7\}$, $w = \{1, 3, 4, 5\}$ and $m = 7$

Ans.

P	w	i \ w	0	1	2	3	4	5	6	7
		0	0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	1	1	1	1
4	3	2	0	1	1	4	5	5	5	5
5	4	3	0	1	1	4	5	6	6	9
7	5	4	0	1	1	4	5	7	8	9

x_1 x_2 x_3 x_4 $\quad 9-5=4$

0 1 1 0 $\quad 4-4=0$

∴ Including 2nd and 3rd item will be the optimal solution.

```
Algorithm ZeroOneKnapsack(P, wt, m, n){
// P is the array that stores profits of corresponding
// objects and wt is the array that stores weights.
// objects are like - 1st, 2nd, 3rd. and so on but
// the indexing of the arrays start from 0. so we
// have by default taken P[0] = wt[0] = 0.
// m is the knapsack capacity and n is the number of
// objects. We have used a matrix 'K[n+1][m+1].

    for i:=0 to i≤n, do{
        for w:=0 to w≤m, do{
            if (i=0 or w=0) then,
                K[i][w]:=0
            else if (wt[i]≤w) then,
                K[i][w]:= max(P[i]+K[i-1][w-wt[i]]
                              , K[i-1][w])
            else K[i][w]:= K[i-1][w]
        }
    }
    cost:= K[n][w]  // Optimal cost.
    included[n] := 0 for each i:=0 to i≤n
    // Initially no object is selected.

    i:=n, j:=m
    while (i>0 and j>0), do{
        if (K[i][j] = K[i-1][j]) then
            included[i]:=0, i:=i-1
        else{
            included[i] := 1
            j:= j - wt[i];
            i:= i-1
        }
    }
    return included.
}
```

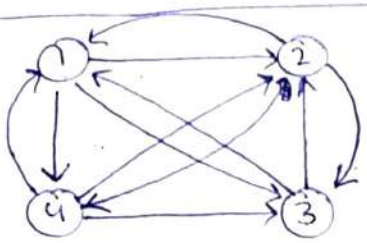
Travelling Salesman Problem

[Previous problem, 0/1 knapsack, was a subset problem. For n objects we could have 2^n different subsets as in for n bits we could have 2^n different combinations. But this problem is a permutation problem. Here we can have $n!$ different permutations for n objects]

Problem Statement : Let $G_2(V, E)$ be a directed graph with edge costs $c[i,j]$. The matrix $c[i,j]$ is defined such a way that $c[i,j] > 0 \ \forall \langle i,j \rangle \in E$ and $c[i,j] = \infty \ \forall \langle i,j \rangle \notin E$. Let $|V| = n$ and assume that $n > 1$. A tour of G_2 is a directed simple cycle that includes every vertex in V. The cost of a tour is the sum of the cost of the edges on the tour. The Travelling salesperson Problem is to find a tour with minimum cost.

Generalized Formula : $g(i, S) = \min_{k \in S} \{c_{ik} + g(k, S - \{k\})\}$

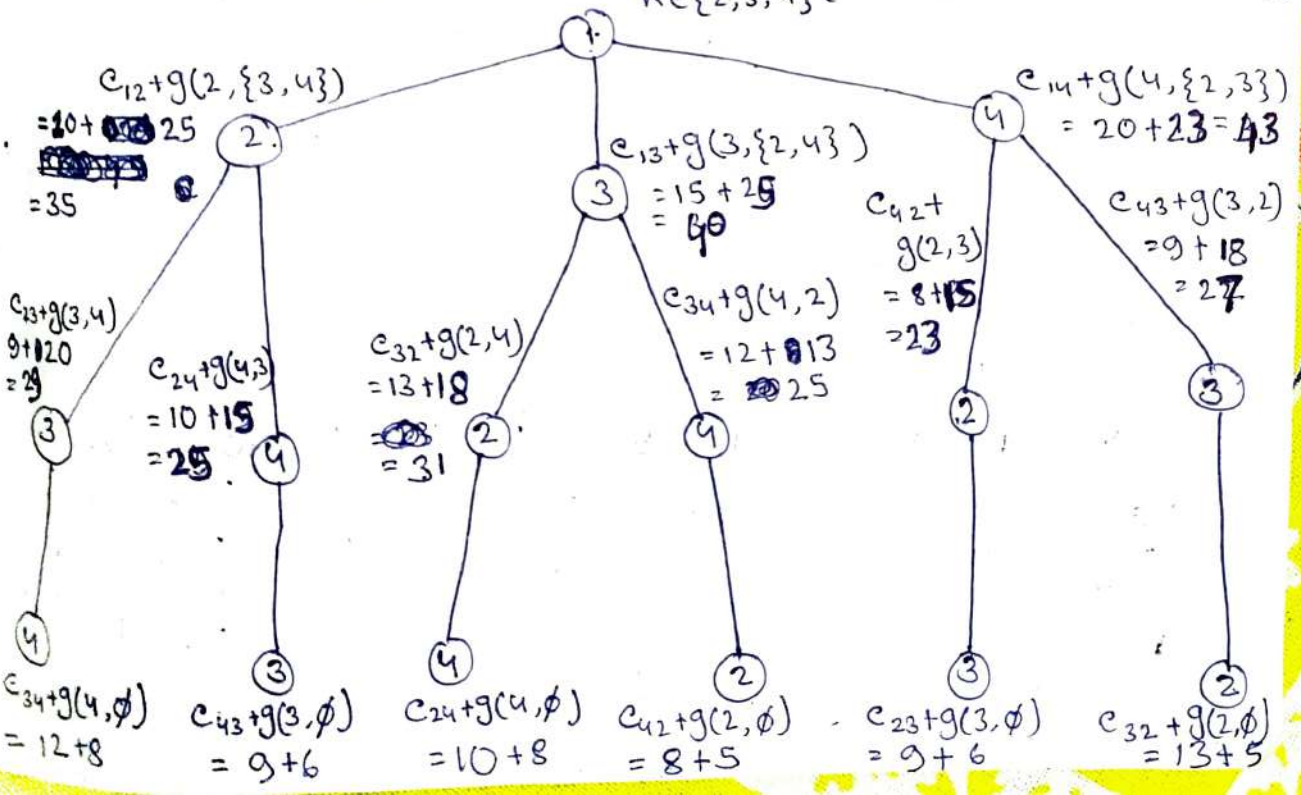
Ex (1)



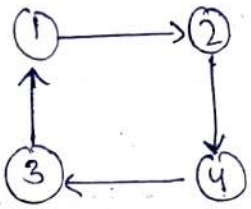
Cost =

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

$g(1, \{2,3,4\}) = \min_{k \in \{2,3,4\}} \{c_{1k} + g(k, \{2,3,4\} - \{k\})\}$



Therefore, the optimal route is →

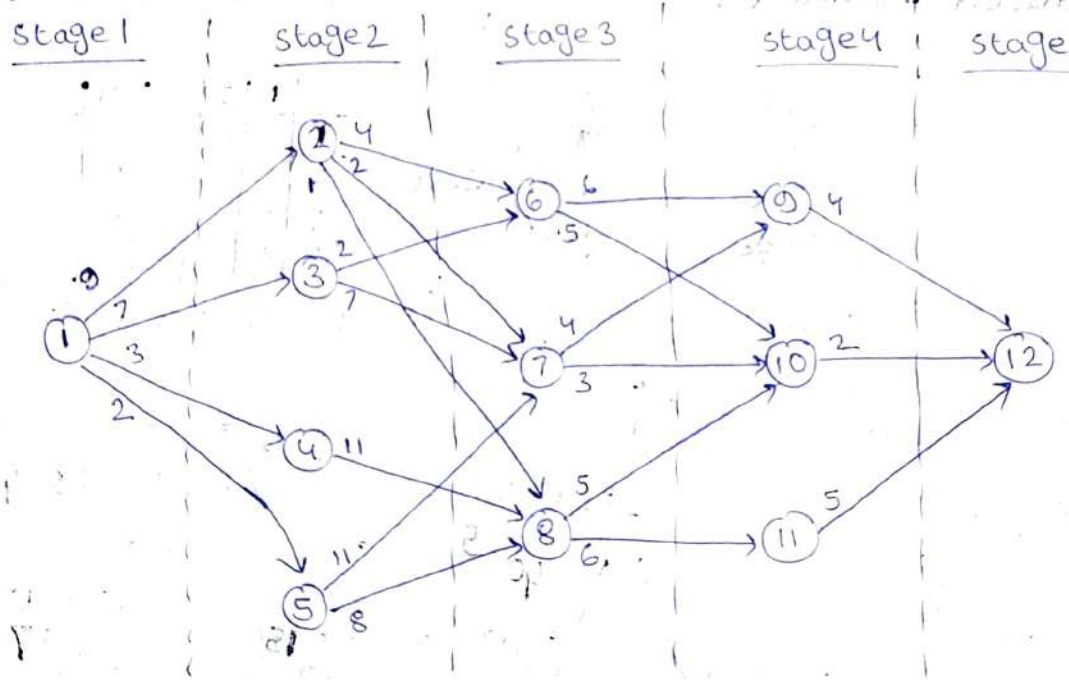


Multistage Graph

A multistage graph is a directed weighted graph where the vertices are divided in 2 stages such that the edged are connecting vertices from one stage to next stage. First stage and last stage both have single vertex to represent the ~~sink~~ source and sink of a graph.

Objective : There are several paths between source to sink but we have to choose the path that has minimum cost. So, its a minimization problem that is an optimization problem.

Forward Method



Cost of vertex V

V	1	2	3	4	5	6	7	8	9	10	11	12
Cost	16	7	9	18	15	7	5	7	4	2	5	0
d	2/3	7	6	8	8	10	10	10	12	12	12	12

The vertex for which the cost of vertex V is chosen

[The calculations are done from sink to source and not from source to sink]

stage no. ↙ ↘ vertex no.

$cost(5, 12) = 0$

stage 4 =>

(i) $cost(4,9) = 4$

(ii) $cost(4,10) = 2$

(iii) $cost(4,11) = 5$

stage 3 =>

(i) $cost(3,6) = \min\{(c(6,9) + cost(4,9)), (c(6,10) + cost(4,10))\}$

$= \min\{6+4, 5+2\} = 5+2 = 7$

[$\because c(a,b)$ = weight of edge a—b]

(ii) $cost(3,7) = \min\{(c(7,9) + cost(4,9)), (c(7,10) + cost(4,10))\}$

$= \min\{4+4, 3+2\} = 3+2 = 5$

(iii) $cost(3,8) = \min\{(c(8,10) + cost(4,10)), (c(8,11) + cost(4,11))\}$

$= \min\{5+2, 6+5\} = 5+2 = 7$

stage 2 =>

(i) $cost(2,2) = \min\{(c(2,6) + cost(3,6)), (c(2,7) + cost(3,7)), (c(2,8) + cost(3,8))\}$

$= \min\{4+7, 2+5, 1+7\}$

$= 5+2 = 7$

(ii) $Cost(2,3) = \min\{(c(3,6) + Cost(3,6)), (c(3,7) + Cost(3,7))\}$

$= \min\{2+7, 7+5\} = 2+7 = 9$

(iii) $Cost(2,4) = \min\{(c(4,8) + Cost(3,8))\}$

$= 11 + 7 = 18$

(iv) $Cost(2,5) = \min\{(c(5,7) + Cost(3,7)), (c(5,8) + Cost(3,8))\}$

$= \min\{11+5, 8+7\} = 8+7 = 15$

Stage 1 =>

(i) $Cost(1,1) = \min\{(c(1,2) + Cost(2,2)), (c(1,3) + Cost(2,3)), (c(1,4) + Cost(2,4)), (c(1,5) + Cost(2,5))\}$

$= \min\{9+7, 7+9, 3+18, 2+15\}$

$= \min\{16, 16, 21, 17\}$

$= 16$

General Formulae :

$$Cost(i,j) = \min\{c(j,l) + Cost(i+1,l)\}$$

(i = stage no., j = vertex no.)

$\langle j, l \rangle \in E$ (set of edges)

The method we are using to solve the problem is called forward method but till now we have filled the table in backward direction (sink → source) and also we did not follow Principle of Optimality that is taking

sequence of decisions at each step. But not as we have the data ready for the total graph, we will move towards sink from source by taking desicions at each step by using the data.

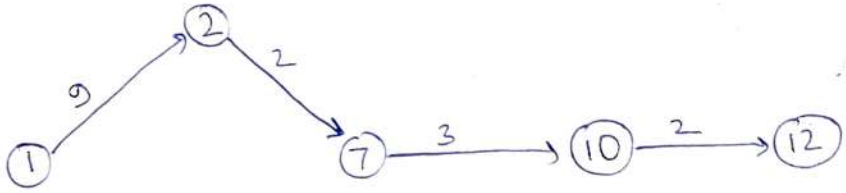
stage vertex

$d(1,1) = 2$

$d(2,2) = 7$

$d(3,7) = 10$

$d(4,10) = 12$



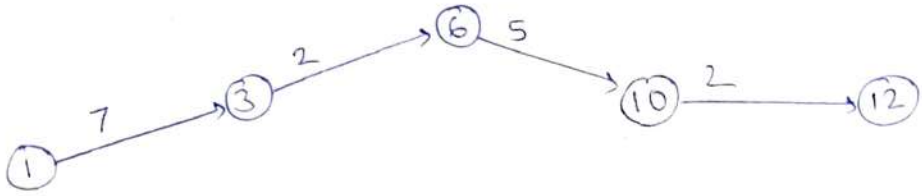
This is one of two shortest paths from source to sink. @ Total cost = 16

$d(1,1) = 3$

$d(2,3) = 6$

$d(3,6) = 10$

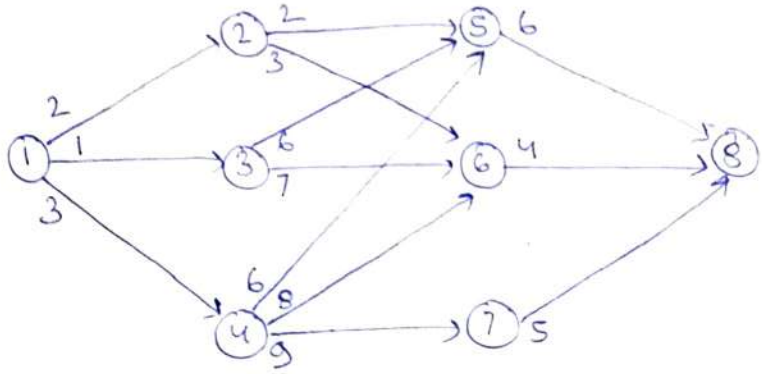
$d(4,10) = 12$



This is one of two shortest paths from source to sink. Total cost = 16.

Implementation

Let a given graph G_2 be —



Cost adjacency matrix for this graph is —

$C =$

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	2	1	3	0	0	0	0
2	0	0	0	0	0	2	3	0	0
3	0	0	0	0	0	6	7	0	0
4	0	0	0	0	0	6	8	9	0
5	0	0	0	0	0	0	0	0	6
6	0	0	0	0	0	0	0	0	4
7	0	0	0	0	0	0	0	0	5
8	0	0	0	0	0	0	0	0	0

Algorithm MultistageGraph (C, stage , n) {

Input : Cost adjacency matrix C , No. of stages stage and No. of vertices in the graph n.

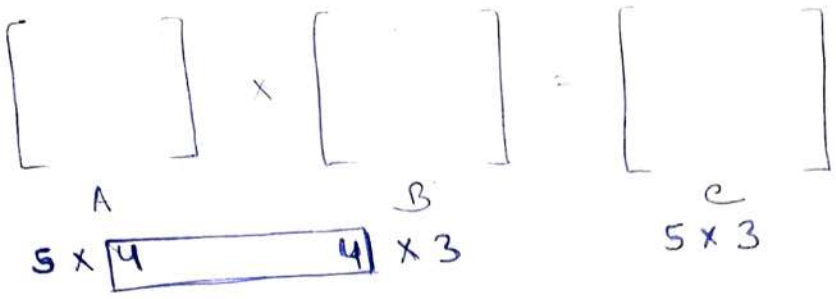
Data structure : 3 vectors of size n+1, cost , d and Path is used.

Output : The minimum cost path from source to sink.

```
Cost[n] ← 0
i ← n-1
while ( i ≥ 1 ) do, {
    min ← 9999999 //Some huge possible number.
    k ← i+1
    while (k ≤ n) do {
        if (c[i][k] ≠ 0 and
            c[i][k] + cost[k] < min) then {
            min ← c[i][k] + cost[k]
            d[i] ← k
        }
        k ← k+1
    }
    Cost[i] ← min, i ← i+1
}
Path[1] ← 1, Path[stage] ← n
From i = 2 to i < stage do
    Path[i] ← d[Path[i-1]]
}
```

~~Matrix Chain Multiplication~~

Matrix Chain Multiplication



This takes a total of $5 \times 4 \times 3 = 60$ multiplications

Ex①

$A_1 \times A_2 \times A_3 \times A_4$

$5 \times 4 \quad 4 \times 6 \quad 6 \times 2 \quad 2 \times 7$

$P_0 P_1 \quad P_1 P_2 \quad P_2 P_3 \quad P_3 P_4$

How these matrices should be multiplied to get minimum number of multiplications done.

Ans.

M:

	1	2	3	4
1	0	120	88	158
2		0	48	104
3			0	84
4				0

S:

	1	2	3	4
1	0	1	1	3
2		0	2	3
3			0	3
4				0

[M[1,2] represents min multiplications needed for $A_1 \times A_2$, M[1,3] represents min multiplications needed for $A_1 \times A_2 \times A_3$ and so on.]

(i) $M[1,2] = 5 \times 4 \times 6 = 120$

(ii) $M[2,3] = 4 \times 6 \times 2 = 48$

(iii) $M[3,4] = 6 \times 2 \times 7 = 84$

(iv) $M[1,3] = \min_{1 \le k < 3} \{ (M[1,1] + M[2,3] + P_0 P_1 P_3), (M[1,2] + M[3,3] + P_0 P_2 P_3) \}$

$= \min_{1 \le k < 3} \{ 88, 180 \} = 88$

(v) $M[2,4] = \min_{2 \le k < 4} \{ (M[2,2] + M[3,4] + P_1 P_2 P_4), (M[2,3] + M[4,4] + P_1 P_3 P_4) \}$

$= \min_{2 \le k < 4} \{ 252, 104 \} = 104$

(vi) $M[1,4] = \min_{1 \le k < 4} \{ (M[1,1] + M[2,4] + P_0 P_1 P_4),$
$(M[1,2] + M[3,4] + P_0 P_2 P_4),$
$(M[1,3] + M[4,4] + P_0 P_3 P_4) \}$

$= \min_{1 \le k < 4} \{ 244, 414, 158 \} = 158$

Therefore, the minimum no. of multiplications will be done if it is done in the following order —

$$(A_1 \times (A_2 \times A_3)) \times A_4$$

Some Points

(i) If we have n matrices, then total number of possible parenthesization required is —

$$\frac{2(n-1)C_{(n-1)}}{n}$$ (called **Catalan Number**)

For example, when $n = 3$

$A_1 \times A_2 \times A_3$

No. of possible parenthesization is $\frac{2\times2C_2}{3} = \frac{6}{3} = 2$

$(A_1 \times A_2) \times A_3$, $A_1 \times (A_2 \times A_3)$

When $n = 4$

$A_1 \times A_2 \times A_3 \times A_4$

∴ Catalan Number = $\frac{2\times3C_3}{4} = \frac{20}{4} = 5$

$((A_1 \times A_2) \times A_3) \times A_4$, $(A_1 \times (A_2 \times A_3)) \times A_4$, $A_1 \times (A_2 \times (A_3 \times A_4))$
$A_1 \times ((A_2 \times A_3) \times A_4)$, $(A_1 \times A_2) \times (A_3 \times A_4)$.

(ii) **Generalized Formulae** —

$$M[i,j] = \min_{i \le k < j} \{ M[i,k] + M[k+1,j] + P_{i-1} P_k P_j \}$$

Where $M[i,j]$ = Minimum number of multiplications required to multiply $A_i \times A_{i+1} \times \cdots \times A_j$ $(i < j)$

(iii) In the M table (in previous example), from bottom row, we have $1+2+3+4$ values calculated when number of matrices is 4.

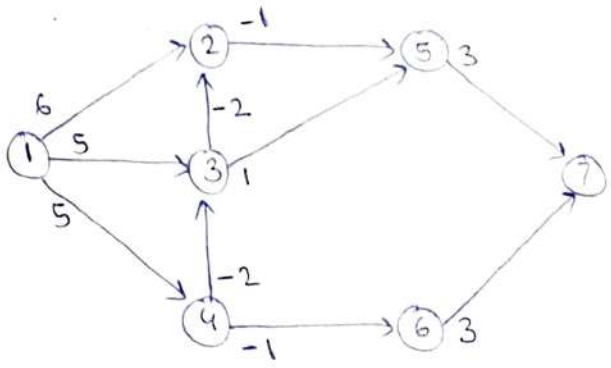
So, for n matrices it would be $1+2+\cdots+n$ that is $\frac{n(n+1)}{2}$

And for each value calculated, in the worst case scenario (top right corner value) we have different k values through n. So in the worst case scenario, time taken by matrix chain multiplication algorithm is $O(n^3)$.

Single Source Shortest Path

(Bellman Ford Algorithm)

In Dijkstra Algorithm, we were also finding single source shortest path but ~~it was~~ there was a problem that is if the graph contains any negative weighted edge then Dijkstra was not always working. This is solved in Bellman Ford algorithm that uses the dynamic programming approach.



The idea is to relax every edge in the graph for $(n-1)$ times where n is the number of edges in the graph.

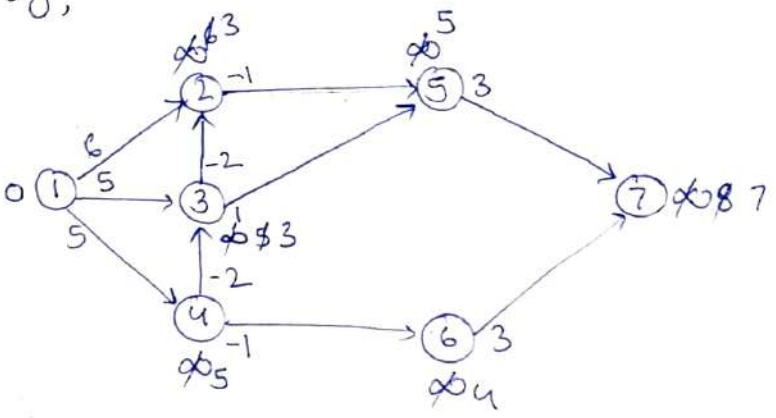
Relaxation => if $(d[u] + cost(u,v) < d[v])$ then,

$d[v] \leftarrow d[u] + cost(u,v)$

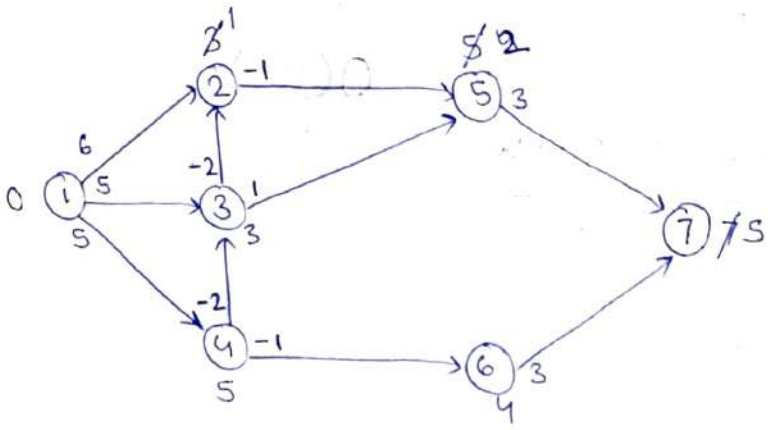
where $d[u]$ = the distance of the minimum path to vertex u and

$Cost(u,v)$ = the weight/cost of edge connecting vertex u and v.

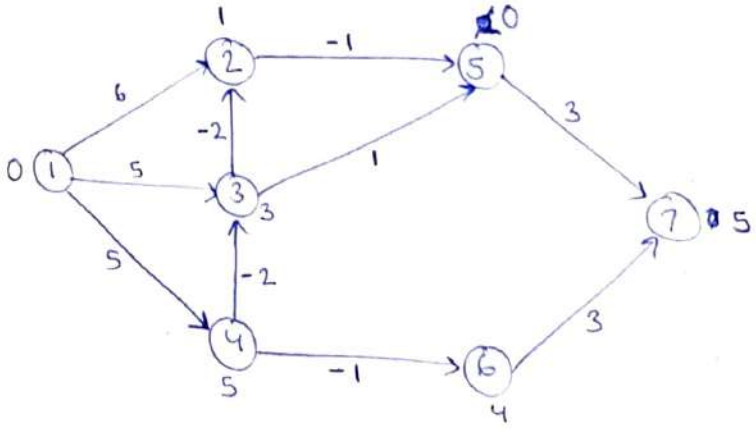
Initially,



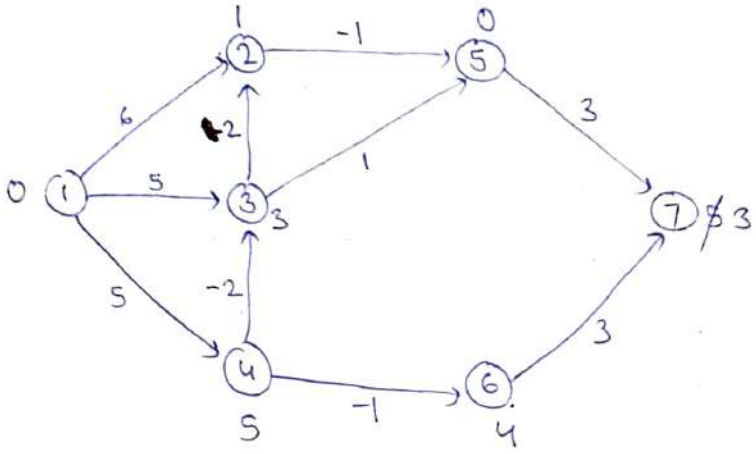
After pass I,



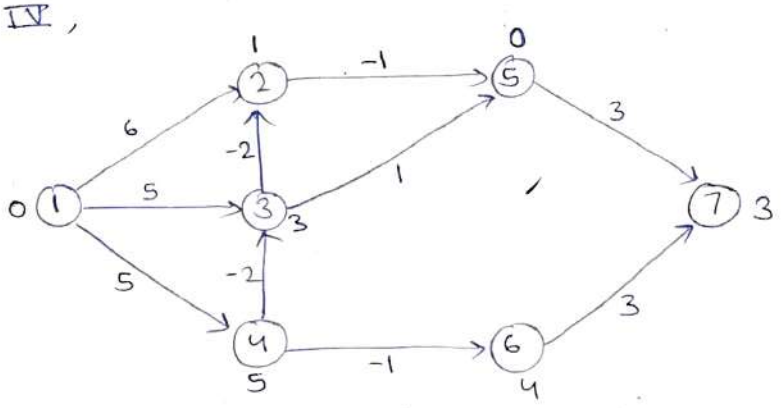
After pass II,



After pass III,



After IV,



so, we were going to relax every vertex of $n-1=5$ times but after pass IV, the distances stop changing. Therefore we can stop relaxation. No need for further relaxations. It is the shortest path from single source 1 to all other vertices.

Analysis

If we have $|V|$ vertices and $|E|$ edges then the Bellman Ford algorithm is relaxing every $|E|$ edges for $|V|-1$ times. So, the time complexity is $O(|V||E|)$ or, if $|V| \approx |E| \approx n$ then it is $O(n^2)$. that is the worst case time.

In case of complete graphs, if there are n vertices then there are $\frac{n(n-1)}{2}$ edges. In this case, total number of relaxations will be $\frac{n(n-1)}{2}(n-1)$ that is in case of complete graphs, Bellman Ford algorithm takes $O(n^3)$ time.

Longest Common Subsequence (LCS)

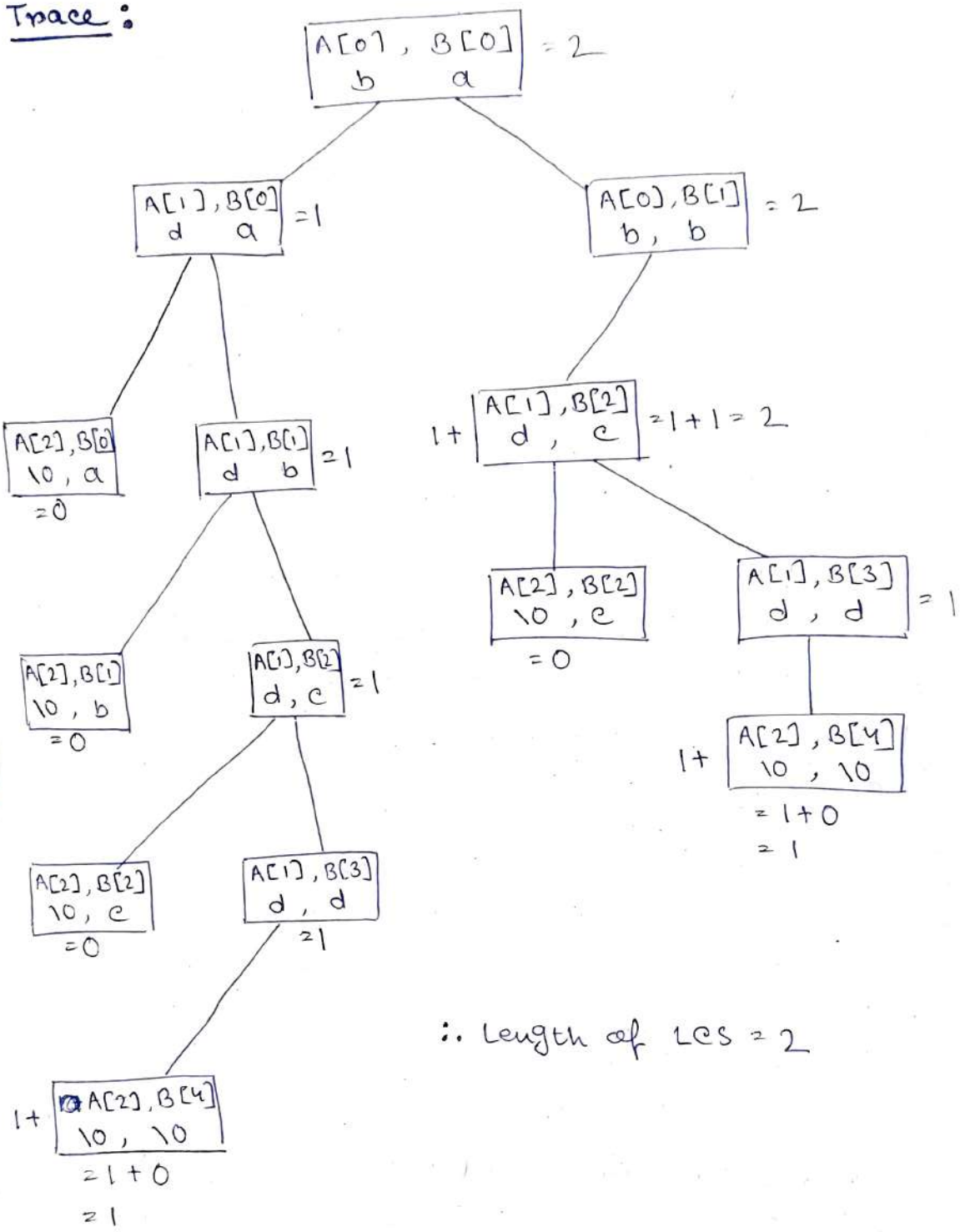
Recursive approach

```
Algorithm LCS (i, j){
    if (A[i] = NULL or B[i] = NULL)
        return 0
    else if (A[i] = B[i])
        return 1 + LCS(i+1, j+1)
    else
        return max(LCS(i+1, j), LCS(i, j+1))
```

	0	1	2
A	b	d	\0

	0	1	2	3	4
B	a	b	c	d	\0

Trace:



∴ Length of LCS = 2

Tracing Tree of LCS algorithm

This is the tracing tree of the recursive algorithm of LCS which uses top-down approach. Time taken by this recursive algorithm is $O(2^4)$. Here is the dynamic programming approach to solve LCS problem.

LCS algorithm using dynamic programming :

```
if (A[i] = B[j])
    LCS[i,j] = 1 + LCS[i-1, j-1]
else
    LCS[i,j] = max(LCS[i-1,j], LCS[i,j-1])
```

A

b	d
1	2

B

a	b	c	d
1	2	3	4

		a	b	c	d
	0	0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	1	1	1
d	2	0	0	1	1	2

→ size of LCS

and the LCS : bd

Ex ① Find ~~LS~~ LCS of given 2 strings

str 1 : STONE

str 2 : LONGEST

Ans.

		L	O	N	G	E	S	T
	0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0
S	1	0	↑0	↑0	↑0	↑0	↑0	↖1	↑0
T	2	0	↑0	↑0	↑0	↑0	↑0	↑1	↖2
O	3	0	↑0	↖1	↑0	↑0	↑0	↑1	↑2
N	4	0	↑0	↑1	↖2	↑0	↑0	↑1	↑2
E	5	0	↑0	↑1	↑2	↑0	↖1	↑1	↖2

		L	O	N	G	E	S	T
	0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0
S	1	0	↑0	↑0	↑0	↑0	↑0	↖1	↑1
T	2	0	↑0	↑0	↑0	↑0	↑0	↑1	↖2
O	3	0	↑0	↖1	←1	←1	←1	↑1	↑2
N	4	0	↑0	↑1	↖2	←2	←2	←2	↑2
E	5	0	↑0	↑1	↑2	↑2	↖3	←3	←3

∴ length(LCS) = 3

∴ LCS = "ONE"

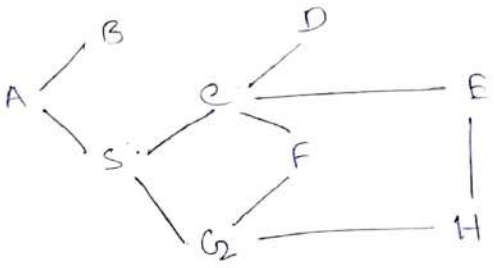
Analysis : The dynamic program approach of solving the longest common subsequence problems uses bottom up approach. It fills up a table i.e. matrix of size $m \times n$ where m and n are the sizes of the given strings. So, time taken by the dynamic programming approach of LCS algorithm is $O(m \times n) \approx O(n^2)$ that is much less than recursive approach which was taking exponential time.

Graph Traversal Algorithms

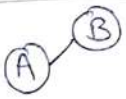
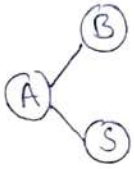
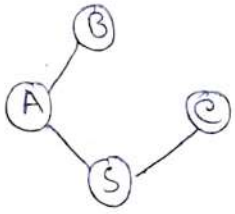
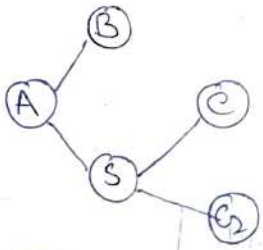
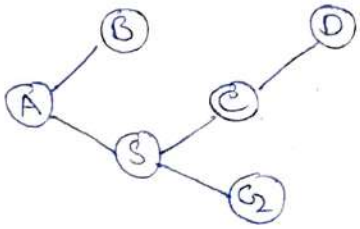
Breadth First Search (BFS)

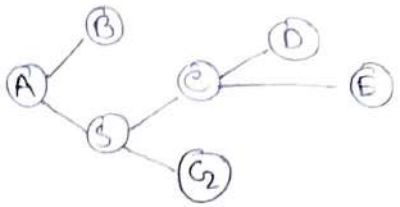
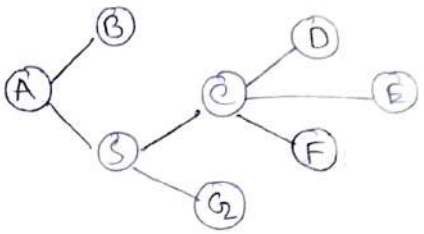
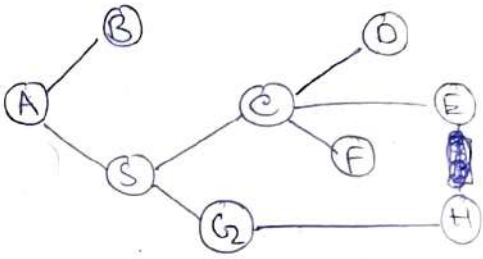
```
Algorithm BFS(G, s){
// Input : The graph G and source vertex is taken
// Data structure : Queue data structure is used
    for all v in V,    // V is set of vertices in G
        d[v] ← 0       // d is a vector of size |V|
    d[s] ← 1           // Source vertex marked
    Enqueue(Q, s)      // Enqueue() method
                          enqueues in queue Q
    While Q is not empty, do {
        b ← Dequeue(Q) // Dequeue() method
                          dequeues from queue
        for all u adjacent to b, do{
            if d[u] = 0, then{
                d[u] ← 1
                Enqueue(Q, u)
            }
        }
    }
}
```

Ex ①



Ans.

Operation	Queue	Graph
Enqueue(A)	A	—
Dequeue(A) Enqueue(B) Enqueue(S)	B, S	
Dequeue(B)	S	
Dequeue(S) Enqueue(C) Enqueue(G2)	C, G2	
Dequeue(C) Enqueue(D) Enqueue(E) Enqueue(F)	G2, D, E, F	
Dequeue(G2) Enqueue(H)	D, E, F, H	
~~Dequeue(D)~~	E, F, H	Same graph as above
Dequeue(E)	F, H	

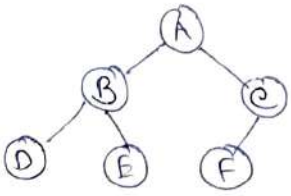
Operation	Queue	Graph
Dequeue(E)	F, H	
Dequeue(F)	H	
Dequeue(H)	Queue empty!	

NoTe

BFS & DFS algorithms are not about finding any shor-test ~~past~~ path or minimum cost tree but it is all ab-out traversing all the nodes in a graph. All of them should be traversed no matter what is the sequence is ~~unless~~ while the algorithm is properly followed.

The final graph after applying BFS algorithm is called BFS spanning tree. In case of DFS, ~~the~~ it is DFS spanni-ng tree.

For knowledge



Traversing the graph in BFS : A, B, C, D, E, F

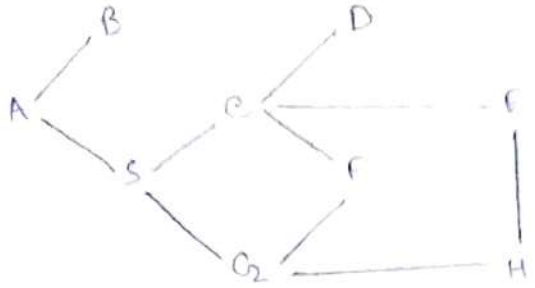
Traversing the graph in DFS : A, B, D, E, C, F

So, BFS traversal is basically level-order traversal and DFS traversal is basically pre-order traversal of a graph.

Depth First Search (DFS)

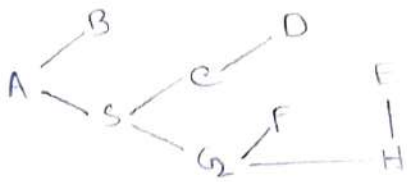
```
Algorithm DFS_iterative(G2, s){
// G2 is the graph and s is the source vertex.
// Data structure used is stack
        Let S be the stack
        Push(S, s)  // Push() method pushes into
                       given stack, the given element

        mark s as visited
        while (S is not empty), do{
                v ← Pop(S) // Pop() method pops from
                              given stack.

                for all adjacent w of v in G2, do{
                        if w is not visited, then{
                                Push(S, w)
                                mark w as visited
                        }
                }
        }
}

Algorithm DFS_recursive(G2, s){
        mark s as visited
        for all adjacent w of s in G2, do{
                if w is not visited, then{
                        DFS_recursive(G2, w)
                }
        }
}
```

Ex (1)



Ans source : A

Operation	stack	Graph
[illegible] –	–	A –
[illegible] PUSH(A)	A	A –
POP(A) PUSH(B) PUSH(S)	B, S	A
POP() PUSH(C) PUSH(G2)	B, C, G2	A — S
POP() PUSH(F) PUSH(H)	B, C, F, H	A — S — G2
POP() PUSH(E)	B, C, F, E	A — S — G2 — H
POP() [illegible]	B, C, F	A — S — G2 — H, E
POP()	B, C	A — S — G2 (F) — H, E
POP() PUSH(D)	B, D	A — S (C) — G2 (F) — H, E
POP()	B	A — S (C, D) — G2 (F) — H, E

Operation	Stack	Graph
POP()	—	

when adjacency list is used

Analysis : Both BFS and DFS takes $O(|V|+|E|)$ time where V = set of vertices and E = set of edges. Approximately it is $O(n^2)$ for a graph of n vertices when adjacency matrix is used.

For Knowledge

Previously I said that BFS and DFS algorithms were not meant to find shortest path between given pairs or any minimum cost tree but only to search the whole graph. It is true that BFS and DFS algorithms are traditional graph searching algorithms but they can also be used to find shortest paths between ~~the~~ source and destination if implemented correctly.

Things will get more exposure if it is compared with the single source shortest path algorithm that is the Dijkstra algorithm. Dijkstra can be think of a generalized version for finding shortest paths of BFS algorithm. Dijkstra uses priority queue, ~~we~~ where based on the cost to get to a node we prioritize the adjacent nodes of a certain node where BFS uses normal FIFO queue.

Therefore it can be said that Dijkstra modifies the BFS algorithm to find shortest paths efficiently and BFS algorithm can also be used to find shortest paths but on unweighted graphs.

Preferences

When the destination is far away from source, DFS is preferred over BFS. Otherwise BFS works faster when destination closer to source.

Backtracking —

Important points —

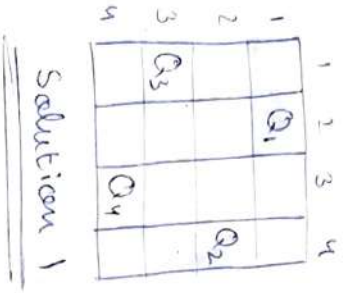
1. Backtracking is a problem solving strategy that uses brute force method.
2. Brute force approach finds all the possible solutions and selects desired solution per given constraints.
3. Now, dynamic programming uses brute force approach but it is to find only the optimum solution to a maximization or minimization problem. In other words, dynamic programming uses brute force to find feasable solutions and among them it picks up the optimal one.
4. The main difference with dynamic programming of backtracking is here that backtracking picks up all the solutions.
5. Backtracking chooses solutions by applying some constraints that is called bounding function.
6. Solutions to the backtracking problems can be represented as state space trees.
7. Backtracking follows depth first search (DFS) method.
8. Branch and bound that is a strategy to find optimal solution also uses brute force approach but follows breadth first search (BFS) method.

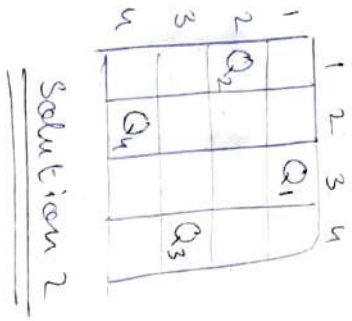
N-Queen Problem

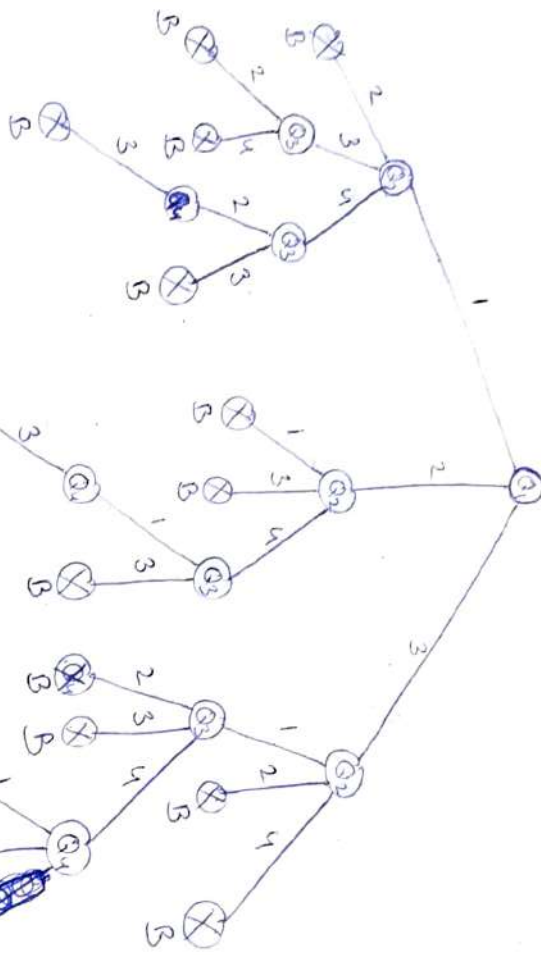
The N-Queen problem is that we need to place n queens on an nxn chessboard so that no two are under attack, that is no two queens are on the same row or column or diagonal.

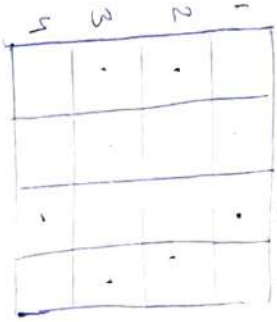
Here is an example of 4-Queen problem where there are 4 queens, Q_1, Q_2, Q_3 and Q_4 and we need to arrange them in a 4x4 chessboard where the bounding function is defined as no two queens are on same row, column or diagonal. We want all possible solutions so backtracking can be used.

Here we assume Q_1, Q_2, Q_3 and Q_4 can only be placed in row 1, 2, 3 and 4 respectively thus reducing the no. of possible solutions as well as the same now problem is solved. The state space tree is shown below



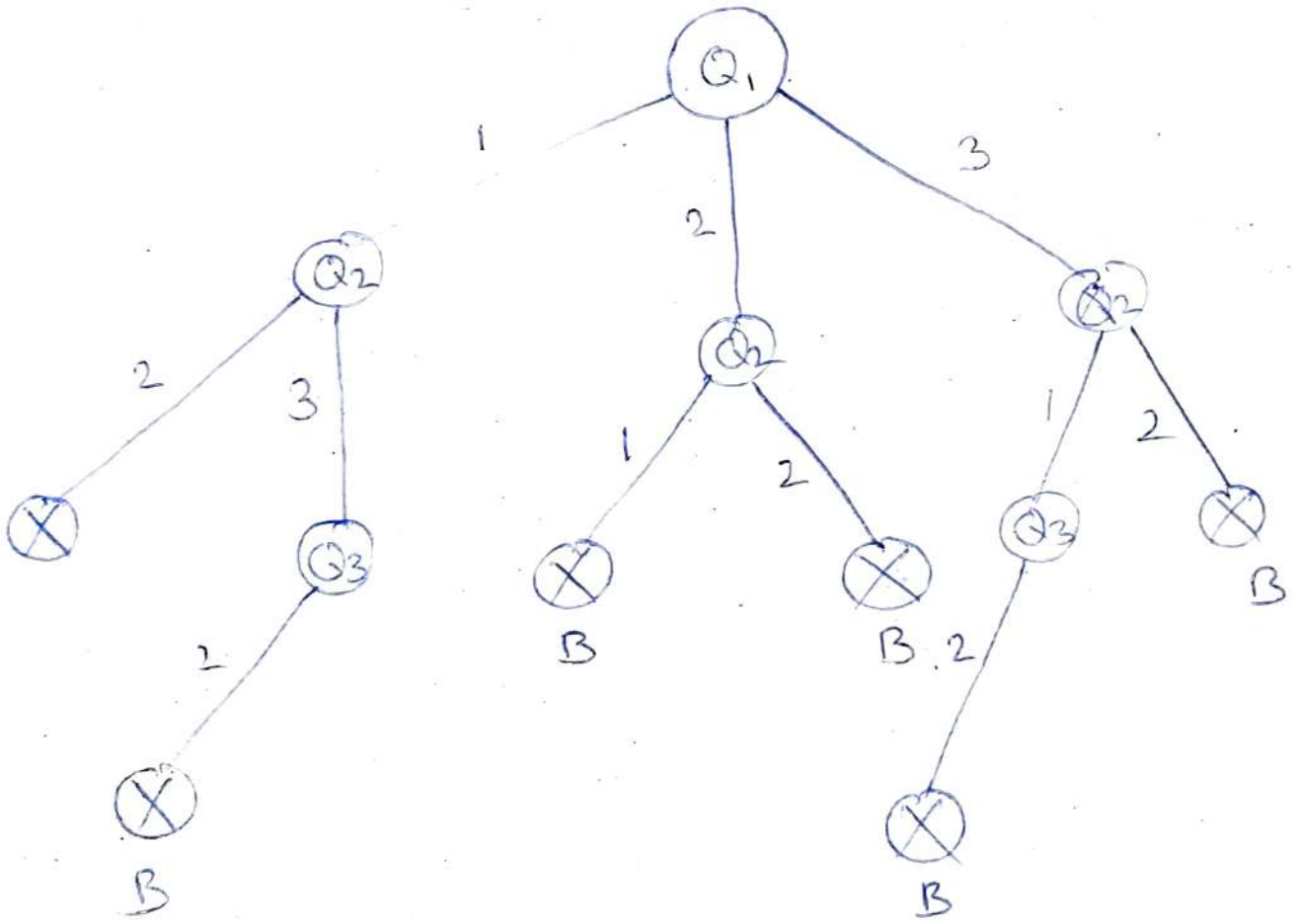






Ex ① Find solutions for 3-Queen problem.

Ans:

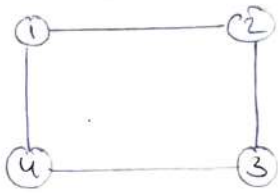


No solutions for 3-Queen problem

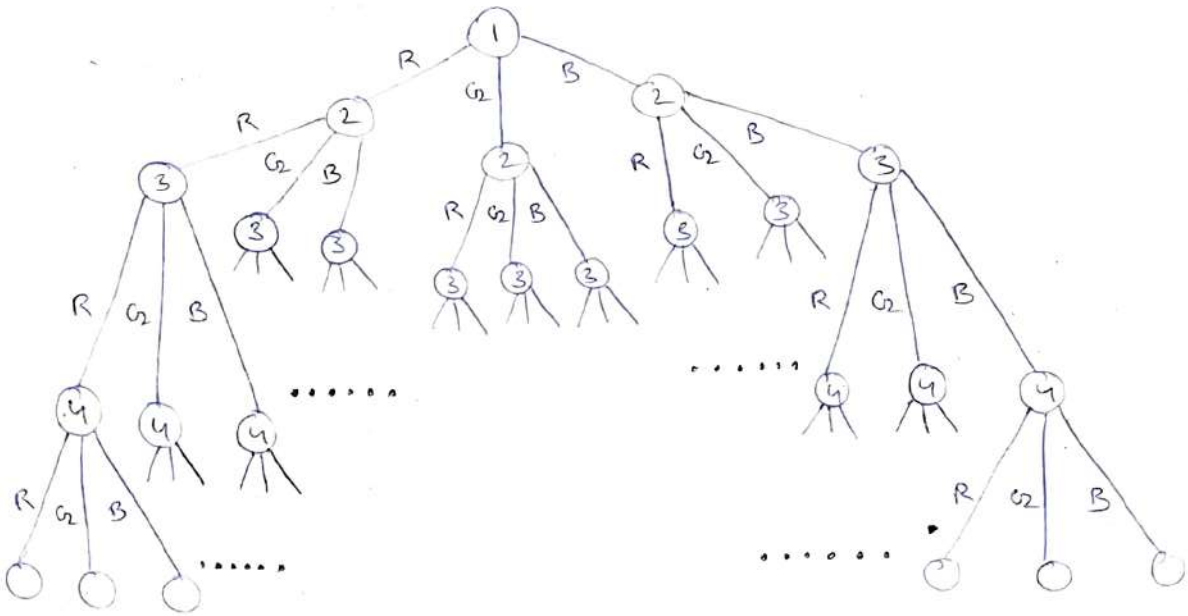
Graph Coloring Problem

Let G be a graph and m be a given positive integer. We want to discover whether nodes of G can be colored in such a way that no two adjacent nodes have the same color yet only m colors are to be used. This is called the m-colorability decision problem. The m-colorability optimization problem asks for the smallest integer m for which the graph G can be colored. The integer i.e. the solution of the m colorability optimization problem is called the chromatic number of the graph G.

Assume the following graph to be colored with three colors — {R, G, B}

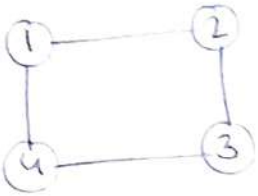


Without applying any bounding function, the state space tree is as below —

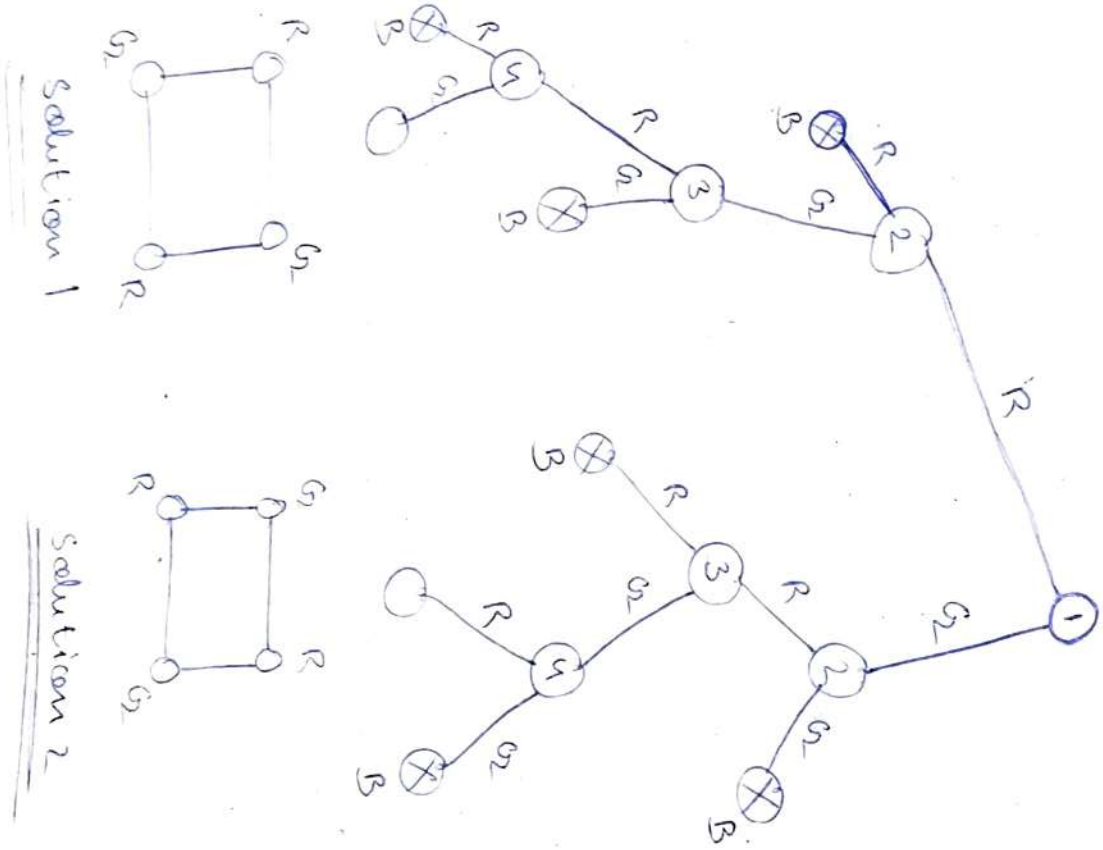


There are total $3^0 + 3^1 + 3^2 + 3^3 + 3^4 = \frac{3^{4+1} - 1}{3 - 1}$ no. of nodes in the state space tree above. So, the **worst case** time of graph coloring problem is $O(m^n)$ where m is the number of colors and n is the number of ~~edges~~ vertices in the graph.

Now, ~~as~~ applying bounded function, the state space tree is as given below —



$m_2 = \{R, G_2, B\}$



NP-Complete Problems

1. **Deterministic Algorithm :** Let A be an algorithm to solve any problem say Π. We say that A is deterministic if, when presented with an instance of the problem Π, A has only one choice in each step throughout its execution. Thus if A is run again and again on the same input instance, the output never changes.

2. **P class :** The class of decision problems P consists of those decision problems whose yes/no solution can be obtained using a deterministic algorithm that runs in polynomial time. Some examples of the problems that comes under P class are —

~~Sorting :~~

(i) Sorting : Given n integers, are they sorted in descending or ascending order?

(ii) Set disjointness : Given 2 sets of integers, ~~are~~ is their intersection set empty?

(iii) 2-coloring : Given an undirected graph G_2, is it 2-colorable? i.e. can its vertices be colored using only 2 colors given that no adjacent vertices should have same color.

3. **Theorem** Class P is closed under complementation.

Proof : We say that a class of problems say C is closed under complementation if for any problem $\Pi \in C$, the complement of Π also belongs to C.

We know that the 2-coloring problem lies under P class. The complement of the 2-coloring problem can be stated as — Given an undirected graph G_2, is it not 2-colorable? Now since 2-coloring is in P, there is a deterministic algorithm A which, when presented to 2-coloring problem yields answer yes if ~~the gp~~ G_2 is 2-colorable and when presented to the complement of 2-coloring problem, yields no if G_2 is 2-colorable that can easily be done by interchanging yes and no.

Proved

3. **NP Class :** Class NP consists of those problems Π for which there exists a deterministic algorithm A which, when presented to with a claimed solution to an instance of Π, will be able to verify its correctness in polynomial time. That is if the claimed solution is yes there is a way to verify it in polynomial time.

In simple terms, the class of decision problems NP consists of those decision problems for which there exists a non-deterministic algorithm that runs in polynomial time.

4. **Non-deterministic algorithm :** Let A be an algorithm to solve any problem say Π. We say A is nondeterministic if, when presented with an instance of Π, may or maynot produce the yes/no answer in the correct format corresponding to the problem and if produced then it may or maynot be the correct answer to the problem, there is required a verification process to verify the answer but producing the answer as well as the verification process should be in polynomial time i.e. $O(n^i)$ where n = size of input and i = a non-negative integer. Also on several runs with the same input, it may produce ~~dis~~ different answers.

So, the running time of non-deterministic algorithm is the sum of the time taken in the guessing phase & i.e. producing the answer and the time taken by the verification phase that is, $O(n^i) + O(n^j) = O(n^k)$ where i, j and k are non-negative integers.

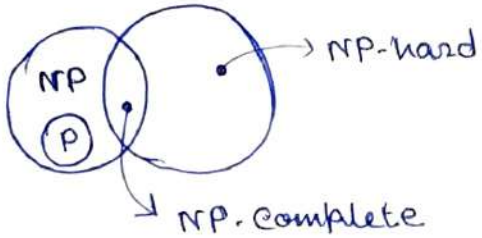
5. **NP-Complete :** NP-complete is a subclass of decision problems in NP that are the hardest in the sense that if one of them is proven to be solvable by a polynomial time deterministic algorithm, then all problems in NP are solvable by a polynomial time deterministic algorithm.

Def^{n} 1 => Let Π and Π' be 2 decision problems. We say that Π reduces to Π' in polynomial time, symbolized as $\Pi \propto_{poly} \Pi'$ if there exists a deterministic algorithm A

that behaves as follows. When A is presented with an instance I of problem Π, it transforms it into an instance I' of problem Π' such that the answer to I is yes iff the answer to I' is yes and this transformation must be achieved in polynomial time.

Def^{n} 2 A decision problem Π is said to be **NP-hard** if for every problem Π' in NP, $\Pi' \propto_{poly} \Pi$

Def^{n} 3 A decision problem Π is said to be **NP-complete** if (i) $\Pi \in NP$ and (ii) for every problem Π' in NP, $\Pi' \propto_{poly} \Pi$ that it belongs to NP-hard.



Analogy

6. The satisfiability Problem:

Given a boolean formula f, we say that f is in **conjunctive normal form (CNF)** if it is the conjunction of **clauses** where a clause is the disjunction of **literals** where literals are nothing but boolean variables or its negation. An example of such formula is —

$$f = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_3 \vee x_4 \vee \bar{x}_5) \wedge (x_1 \vee \bar{x}_3 \vee x_4)$$

Literal — clause — negation of boolean vars.

A formula f is said to be satisfiable if there is a truth assignment to its variables that makes the whole formula true. In the above example, if x_1 and x_3 are set to true, always f is true.

7. Theorem Let Π, Π' and Π'' be 3 decision problems such that $\Pi \propto_{poly} \Pi'$ and $\Pi' \propto_{poly} \Pi''$. Then $\Pi \propto_{poly} \Pi''$ (Transitive Property)

Proof : Let A be an algorithm that reduces $\Pi \propto_{poly} \Pi'$ in $p(n)$ steps and B that reduces $\Pi' \propto_{poly} \Pi''$ in $q(n)$ steps where p and q are some polynomial.

Let x be an input to A of size n. The size of the output of the algorithm A when presented with input x cannot exceed $cp(n)$ as it can only output at-most c symbols in each step of its execution where c is some +ve integer and $c>0$. If B is presented with an input size $p(n)$ or less, it running time is $O(q(cp(n))) = O(r(n))$ for some polynomial r. It follows that reduction from Π to Π' followed by the reduction from Π' to Π'' is a polynomial time reduction from Π to Π''

8. **Corollary :** If Π and Π' are 2 problems in NP such that $\Pi' \propto_{poly} \Pi$ and Π' is NP-complete then Π is also NP-complete.

Proved

Proof : Π' is NP-complete means every problem in NP reduces to Π' in polynomial time and it is given that Π' is reduces to Π in polynomial time. So, by transitive property, we can say that every problem in NP reduces to Π that in turn means that Π is NP complete

Proved