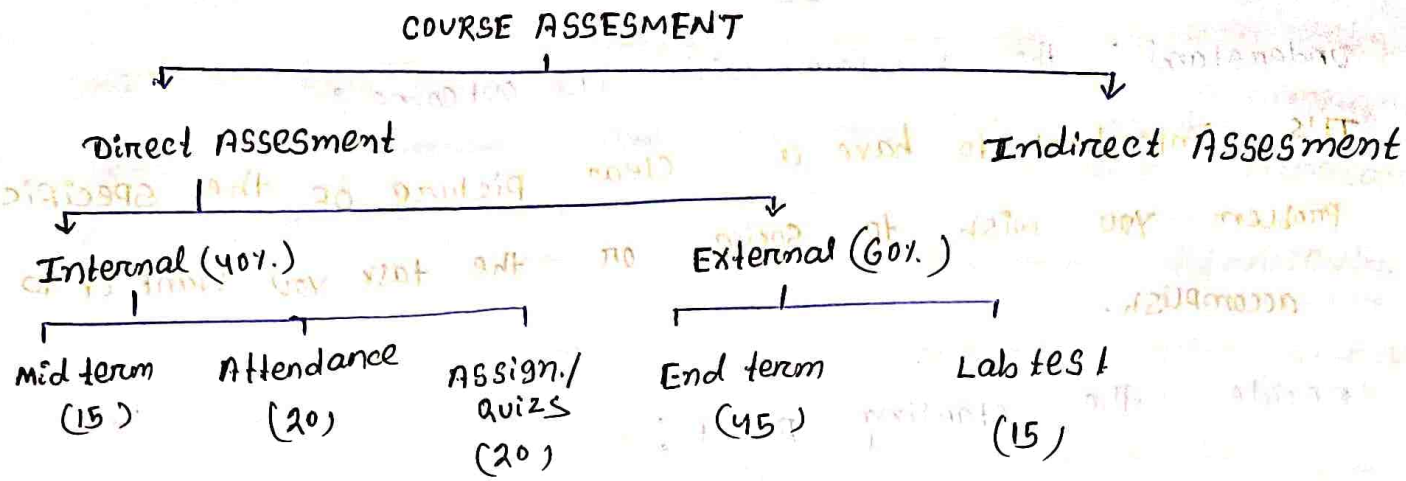
AD-1 (CSE 3131)



- CH-1 → $\overline{CO_1}$ → Introduction, correctness, Time & Space comp.
- CH-2 → $\overline{CO_2}$ → Sorting & searching
- CH-3 → $\overline{CO_3}$ → Graph
- CH-4 → $\overline{CO_4}$ → Greedy
- CH-5 → $\overline{CO_5}$ → DAC
- CH-6 → $\overline{CO_6}$ → DP

01/10/2022

ALGORITHM

Finite no. of steps needed to solve a particular problem with correct / desired output / soln is called Algorithm

ALGORITHM [ANALYZER]	PROGRAM [PROGRAMMER]
→ No syntax (only pseudo code)	→ Syntax is needed
→ Design phase	→ Implementation phase
→ Natural language	→ Computer language
→ No H/W or S/W	→ Compiler is needed

STEPS TO WRITE AN ALGORITHM

1. Understand the problem with its outcome:

It's important to have a clear picture of the specific problem you wish to solve or the task you want it to accomplish.

2. Decide the starting point:

It is challenging to identify the start and the end point which are crucial to list the steps of any process. To determine the starting point, try to answer the following questions:

What data is available?

What is the location of the data?

What are the constraints / rule to work with the available data?

How are the values of the data related to each other?

3. Decide the end point:

As we identified the starting point we can det. the end point of the algorithm by finding ans to the following questions

What are the changes coming in from start to end?

What would be the additions or what would be removed?

4 Think and determine how each step would be achieved:

As we have the step by step outline, its imp. to think about coding of each step. The language we would use, the resources available identifying robust & efficient way to acomplish each step are some of the imp aspects we need to consider.

5. Review & algorithm:

As we comple writting the algorithm, it is eavally imp. to evaluate it. The algorithm has been designed to achieve a certain goal so that we can start Writing a program using it. Try to answer the following

CHARACTERISTIC FEATURE OF AN ALGORITHM

1 INPUT

An algorithm has zero or more inputs, i.e quantities which are given to it initially before the algorithm begins

2 OUTPUT

An algorithm has one or more outputs i.e quantities which have a specified relation to the inputs.

3 Fitness

An algorithm must always terminate after a finite no. of steps.

4. Definiteness :

Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case

5. Effectiveness :

An algorithm is also generally expected to be effective This means that all of the operations to be performed in an algorithm must be sufficiently basic that they can in principle be done exactly & in a finite length of time.

- Algorithm Correctness

Analyze if the algorithm produces the desired output after a finite no. of operations.

- Algorithm efficiency

Estimate the amount of resources needed to execute the algorithm on a machine

PSEUDO CODE:

Pseudo code is a term which is often used in programming and algorithm based fields. It is a methodology that allows the programmer to represent the implementation of an algorithm. simply we can say that the cooked-up representation of an algorithm often at times, algorithms are represented with the help of pseudo codes as they can be interpreted by programmers no matter what their programming back-ground of knowledge is. Pseudo code, as the name suggests, is a false code or a representation of code which can be understood by every layman with some school level programming knowledge.

ALGORITHM VS PSEUDO CODE

- **Algorithm:**

It's an organized logical seavence of the actions or the approach towards a particular problem. A programmer implements an algorithm to solve a problem. Algorithms are expressed using natural verbal but somewhat technica annotations.

- **Pseudo code:**

It's simply an implementation of an algorithm in the form of annotations and informative text written in plain eng. It has no syntax like any of the programming language and thus can't be compiled or interpreted by the same

Other classifications

- Randomized Algorithms :

CLASSIFICATION BY IMPLEMENTATION METHOD

-> Recursion or Iteration

Ex: The tower of Hanoi is implemented in a recursive fashion while stock span problem is implemented iteratively.

-> Exact or approximate

Ex:- For NP-Hard problems, approximation algorithms are used. Sorting algorithms are the exact algorithms.

-> series or parallel or Distributed algorithm

-> classification by Design method

-> Greedy method :-

Ex:- Fractional knap sack, Activity selection

-> Divide & conquer

Ex:- Merge sort, quick sort

-> Dynamic programming

Ex - 0-1 knapsack, subset-sum problem

- Linear programming

Ex- Maximum flow of directed graph

Other classifications

- Randomized Algorithms :-

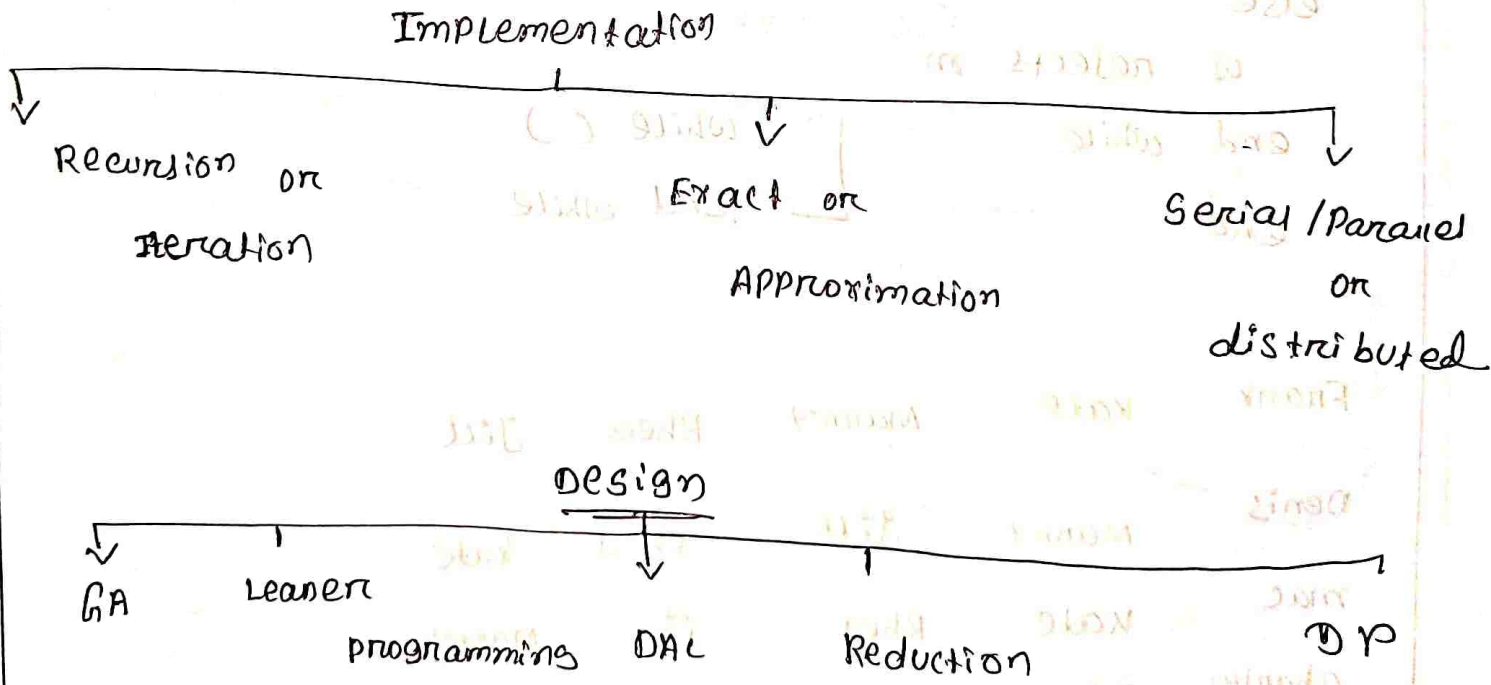
Ex :- Randomized quicksort Algorithm

- classification by complexity :-

Ex :- some algorithms take O(n), while some take expo. time

- classification by research area

- Branch & Bound Enumeration & Backtracking :-



STABLE MATCHING :

Initialize each person to be free

While (some man is free & hasn't proposed to every women)

choose such a man m

w = 1st women on m's list whome m has not yet proposed

if (w is free)

assign m & w to be engaged.

else if (w prefers m to he fiance m')

assign m & w to be engaged, and m' to be free

else

w rejects m

end while

end

[while ( )
end while

Frank	Kate	Marry	Rhea	Jill
Denis	Marry	Jill	Rhea	Kate
mac	Kate	Rhea	Jill	Marry
Charles	Rhea	Marry	Kate	Jill

BOYS	High	Low

Rhea	Frank	Mac	Denis	Charlie
Marry	Mac	Charlie	Denic	Frank
Kate	Denis	Mac	Charlie	Frank
Jill	Charlie	Denis	Frank	Mac

Step-I

Frank : ~~Kate~~
Denis : Marry
Mac = Kate
Charlie: Rhea

Step-II

Frank:
Denis = Marry
Mac = Kate
Charlie: Rhea

Step-3

Frank - Rhea
Denis - Marry
Mac - Kate
Charlie -

Step-4

Frank - Rhea
Denis -
Mac - Kate
Charlie - Marry

Step-5

Frank - Rhea
Denis - Jill
Mac - Kate
Charlie - Marry

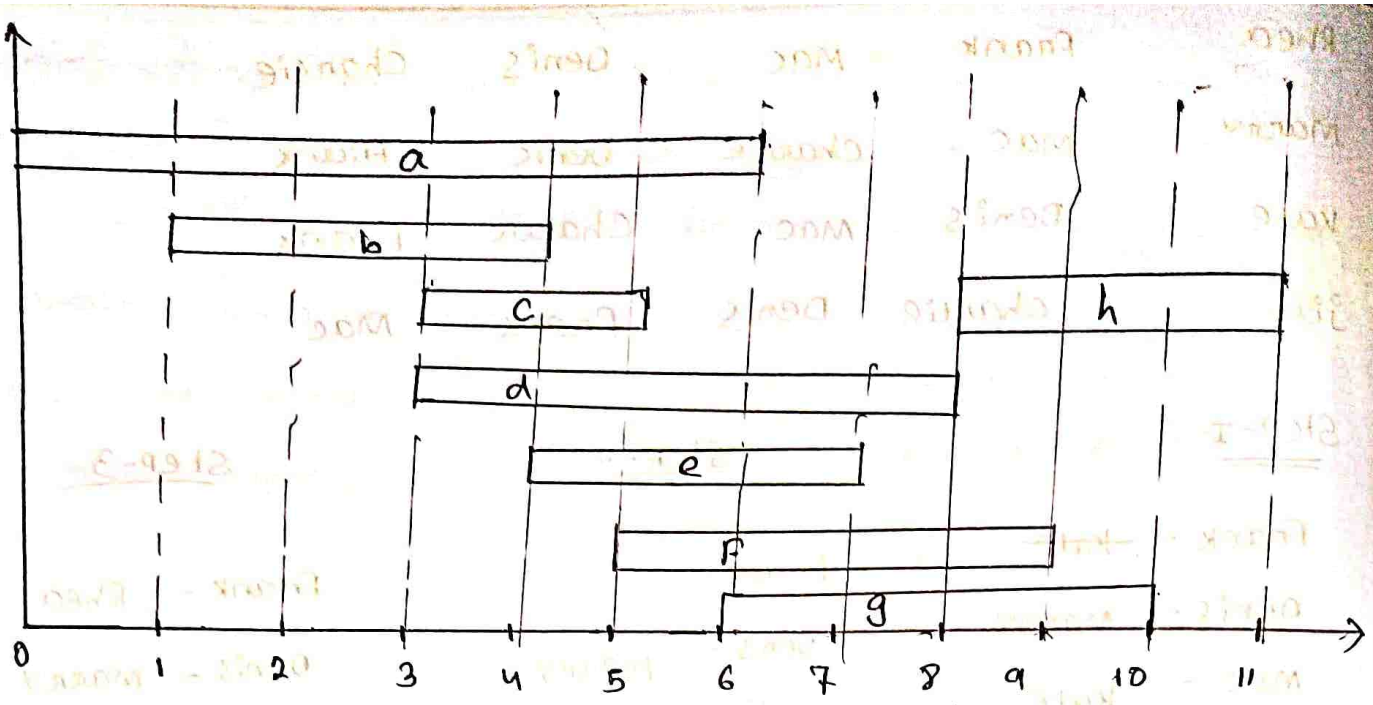
H.W

Using for loop implement this algorithm

Interval Scheduling

Input: Set of jobs with start times & finish times

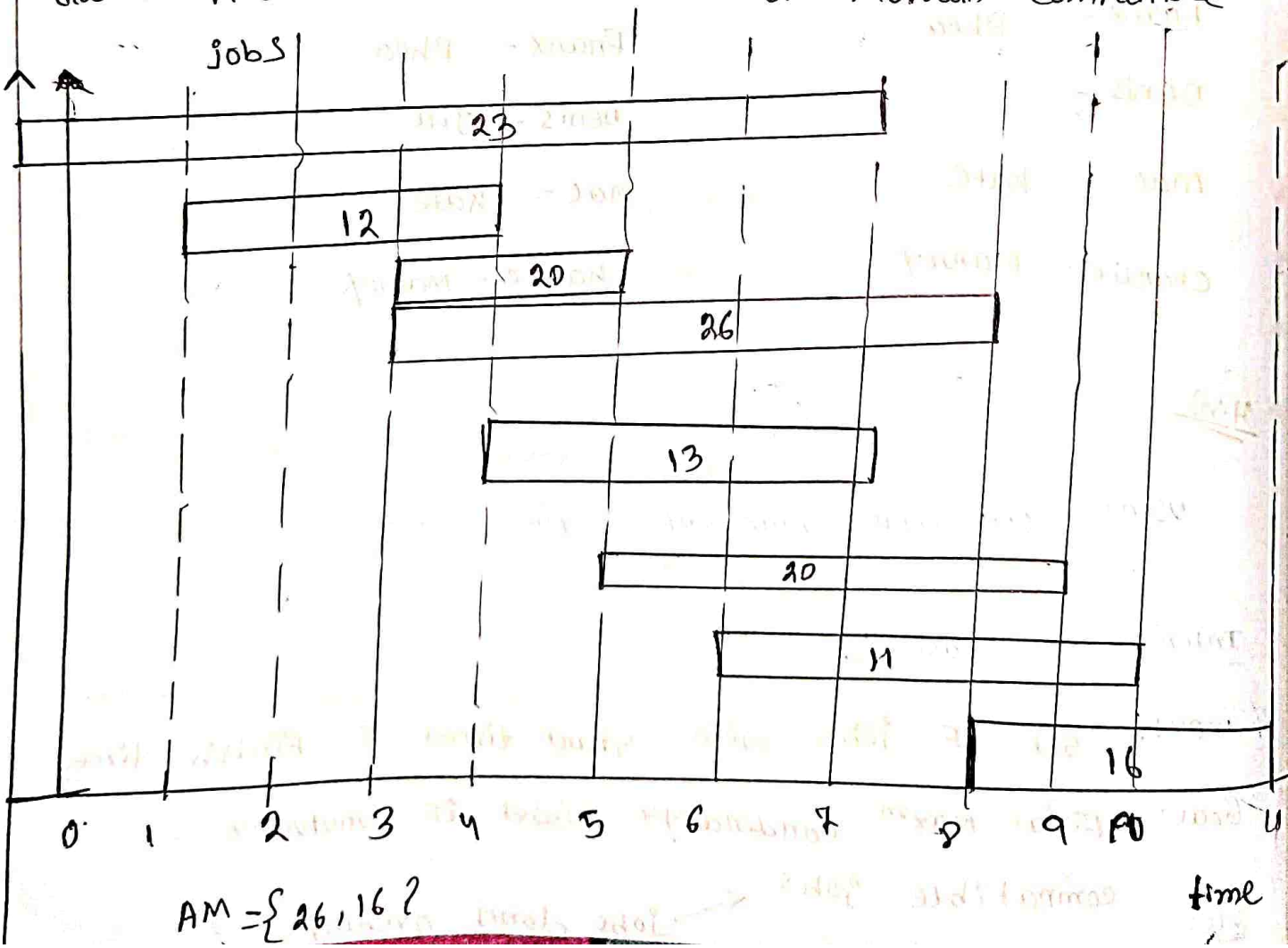
Goal: Find maxm cardinality subset of mutually compatible jobs



Weight interval scheduling :

Input: set of jobs with start time, finish time & weights

Goal: find maxm weight subset of mutually compatible jobs



AM = {26, 16}

For (each person to be free ; till all men are engaged

For (first man in the list ; till last man in list)

w = 1st woman on m's list whom m has not yet proposed.

if (w is free)

assign m & w to be engaged.

else if (w prefers m to her fiance m')

assign m and w to be engaged, and m' to be free

else

w rejects m

end fix

end for

ARTIFICAL RESOURCES

① Time → also called natural resources

② space.

Algorithm Correctness

Partial correctness

Total correctness

Partial correctness

```
int i=1
int j, k;
For (i=1 ; i<n ; i++)
  {
    Print (i);
  }
```

Loop invariant tool

Def:- If the algorithm is receiving correct data on i/p stops, then it's result is correct.

Total = correctness

Partial correctness + Stp property (termination property)

↑ if condn is false

```
For (int i=1 ; i<n ; i++)
    print i
```

if n=5 & we need to print 1 to 5 but we get

So, it's partial correctness

if we use $i \leq n$ then it's total correctness

Proof by

① counter example (indirect proof)

② Induction (direct proof)

③ Loop invariant.

Ans

counter example

① Prove or disprove $\lceil x+y \rceil = \lceil x \rceil + \lceil y \rceil$

② Prove or disprove $\forall x \forall y (xy \geq x)$

Ans ① $x = 1/2$, $y = 1/2$

$\lceil x+y \rceil = \lceil 0.5+0.5 \rceil = \lceil 1 \rceil = 1$

R.H.S

$\lceil x \rceil + \lceil y \rceil = \lceil 0.5 \rceil + \lceil 0.5 \rceil = 1+1 = 2$

L.H.S ≠ R.H.S —— This is disprove

2

Let $x = -1$

$y = 2$

L.H.S = $xy = -2 \neq -1$

$\Rightarrow xy \neq x$ (disprove)

Proof by induction

① $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \Rightarrow 1+2+\cdots+n = \frac{n(n+1)}{2}$

Ans

Step 1: (Small i/p value)

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

$\therefore n=1$ is true or $P(1)$ is true

Soln

Let us consider $P(n): 1+2+3+\cdots+n = \frac{n(n+1)}{2}$

Step-1

For $n=1$

L.H.S = 1 R.H.S = $\frac{1(1+1)}{2} = 1$

So L.H.S = R.H.S Here $P(n)$ is true for $n=1$

Step-2

Let us consider P(n) be true for $n = K$

$$\rightarrow 1+2+\cdots+K = \frac{K(K+1)}{2}$$

Step:

Let $n = K+1$

we need to prove P(n) is true for $n = K+1$

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

L.H.S

$$1+2+\cdots+K+K+1$$

$$= 1+2+\cdots+[(K+1)-1]+(K+1)$$

$$= (1+2+\cdots+K)+(K+1)$$

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

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

$$= \frac{K^2+3K+2}{2}$$

$$= \frac{K^2+K(2+1)+2}{2} = \frac{K^2+2K+K+2}{2}$$

$$= \frac{K(K+2)+1(K+2)}{2} = \frac{(K+1)(K+2)}{2} = R.H.S$$

Quiz - 1

Prove (n^2+n) is even for all integer value using Inducter

Ans

let $P(n)$ is n^2+n is even for all integer i.e n^2+n is divisible by 2.

Step-1

For $n=1$

$n^2+n = 1+1 = 2$ is divisible by 2

$\rightarrow$ $P(1)$ is true

Step-2

Let $P(n)$ is true for $n=K$

is (K^2+K) is divisible by 2

$\rightarrow$ $K^2+K = 2C$ $(C = \text{any const.})$

Step-3

we need to prove $P(K+1)$ is true

i.e $(K+1)^2+(K+1)$ is divisible by 2

$= (K+1)^2+(K+1) = 2C$

$= (K+1)^2 + K+1$

$= K^2+2K+1+K+1$

$= k^2+k+2k+2$

$= (k^2+k)+2k+2$

$=$ k^2+k is divisible by 2

and $2k, 2$ also divisible by 2

So, $P(k+1)$ is true.

goln

Step-4

$n = k+1$

$(k+1)^2 + k+1$

$\rightarrow$ $k^2+2k+1+k+1$

$= k^2+k+2k+2$

$= 2c + 2k+2$

$= 2(c+k+1)$

which is divisible by 2

Loop invariant

> It holds 3 steps:

① Initializing → Before starting loop

② Maintainance → within the body of loop

③ Termination → Based on condition

Dt - 11.10.2022

Quiz-2

using principle of mathematical induction, Prove that

$n < 2^n \; / \; \forall n \in N$

Ans-> Let $P(n)$ be the statement that $n < 2^n \; (\forall n \in N)$

Step-1

For $n = 1$

$1 < 2^1$

$\rightarrow$ $P(1)$ is true.

Step-2

Let $P(n)$ is true for $n = k$

$k < 2^k, \; (\forall k \in N)$

Step-3

we need to prove $P(k+1)$ is true

i.e $k+1 < 2^{k+1}$

$= k+1 < 2^k \cdot 2^1$

$=$ This is true because we consume that $k < 2^k$, so $k+1 < 2^{k+1}$

So, $P(k+1)$ is true

Hence Proved that $n < 2^n \; (\forall n \in N)$

Soln

Step-1

For $n=1$

$\Rightarrow 1 < 2^1 \Rightarrow 1 < 2$ (True)

Step-2

$n = k$

$k < 2^k$ (Assume this as true)

Step-3.1

$n = k+1$

$(k+1) < 2$

From Step-2

$$k < 2^k$$

$$\Rightarrow 2k < 2^k \cdot 2$$

$$\Rightarrow 2k < 2^{k+1} \quad (k+1 \leq 2k)$$

$$\Rightarrow (k+1) \leq 2k < 2^{k+1}$$

$$\Rightarrow (k+1) < 2^{k+1} \text{ (True)}$$

Hence we proved that $n < 2^n, (\forall n \in N)$

Proof by loop invariant

ex - Linear search

```
Linear search (A,V)
1. for j=1 to A.length;
2.     if A[j] == V;
3.         return j
4.     return NIL
```

Loop invariant → At the start of each iteration of the for loop on line 1. the subarray A [1 : j-1] doesn't contain the value v.

Initialization- prior to the first iteration, the array A [1 : j-1] is empty.

(j == 1). That (empty) subarray doesn't contain the value v.

Maintanance :

Line 2 checks whether A[j] is the desired value (v), if it is, the algorithm will returns j, thereby terminating and producing the correct behaviour (the index of value v is returned if v is permanent) if A[j] ≠ v then the loop variant holds at the end of the loop (the subarray A [1 : j] doesn't contain the value v)

Termination

The For loop on line 1 terminates when $j > A.length$ (that is n), Because each iteration of a For loop increements j by 1, then $j = n+1$. The loop invariant states that the value is not present in the subarray of $A[1:j-1]$ substrating $n+1$ For j, we have $A[1:n]$. Therefore the value is not present in the original array A & the algorithm returns NIL

H.W

Sum of n numbers

Input :- a non-negative Integer @ n.

Output :- the Sum $1+2+3+\cdots+n$

```
Sum → 0
i ← 1
while i ≤ n
    Invariant. Sum = 1+2+---+(i-1)
        Sum ← Sum + i
        i ← i+1
return Sum
```

Initialization - The loop invariant holds initially

Since sum = 0 & i = 1, at this point (the sum is zero)

Maintenance - Assuming the invariant holds before the ith iteration, it will be true also after this iteration since the loop adds i to the sum, & increments i by one.

Termination - When the loop is just about to terminate the invariant states that sum = 1+2+----+n, just what is needed for the algorithm to be correct.

Example : Insertion Sort

```
Insertion-Sort (A)
1.  For  j = 2 to A.length
2.      key = A[j]
3.      // Insert A[j] into the sorted sequence A[1....j-1].
4.      i = j-1
5.      While i>0 and A[i] > key
6.          A[i+1] = A[i]
7.          i = i-1
8.      A[i+1] = key
```

At the start of each iteration of the for loop of lines 1-8, the subarray A[1... j-1] consists of the elements originally in A[1.... j-1], but in sorted order.

Initialization : We start by showing the loop invariant holds before the first loop iteration, when j = 2. The subarray A[1.... j-1], therefore, consists of just the single element A[1]. Which is the fact the original element in A[1]. Moreover, the subarray is sorted (trivially, of course), which shows that the loop invariant holds prior to the first iteration of the loop.

Maintenance: Next, we tackle the second priority: showing that each iteration maintains the loop invariant. Informally, the body of the 'for' loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on one by one position to the right until it finds the proper position for $A[j]$ (lines 4-7) at which point it inserts the value of $A[j]$ (line 8). The subarray $A[1 \ldots j]$ then consists of the elements originally in $A[1 \ldots j]$, but in sorted order. Incrementing j for the next iteration of the for loop then preserves the loop invariant.

Termination: Finally, we examine what happens when the loop terminates. The condition causing for loop to terminate is that $j > A.length = n$. Because each loop iteration increases j by 1, we must have $j = n+1$ at that time. Substituting $n+1$ for j in the wording of loop invariant, we have that the subarray $A[1..n]$ consists of the elements originally in $A[1..n]$, but in sorted order. Observing that the subarray $A[1..n]$ is the entire array, we concluded that the entire array is sorted. Hence the algorithm is correct.

Example : Bubble Sort

```
BubbleSort(A)
1. For i = 1 to A.length - 1
2.     For j = A.length to i + 1
3.         if A[j] < A[j - 1]
4.             SWAP(A[j], A[j - 1])
```

Invariant : At the start of each iteration of the For loop on line 1, the subarray $A[1 : i-1]$ is sorted.

Initialization : prior to the first iteration, the array $A[1 : i-1]$ is empty ($i = 1$), That (empty) subarray is sorted by definition

Maintenance : Given the guarantees of the inner loop at the end of each iteration of the For loop at line 1, the value at $A[i]$ is the smallest value in the range $A[i : A.range]$. Since the value in $A[i:i]$ were sorted and were less than the value in $A[i]$ the values in the range $A[1 : i]$ are sorted.

Termination: The for loop at line 1 ends when i equals A.length - 1. Based on the maintenance proof, this means that all values in A[1: A.length - 1] are sorted and less than the value at A[length]. So, by definition, the values in A[1: A.length] are sorted.

Invariant: At the start of each iteration of the for loop on line 2, the value at location A[j] is the smallest value in the subrange from A[j: A.length]

Initialization: Prior to the first iteration, j = A.length. The subarray A[j: A.length] contains a single value (A[j]) and the value at A[j] is (trivially) the smallest value in the range from A[j: A.length])

Maintenance: The if statement on line 3 compares the elements at A[j] and A[j-1], swapping A[j] into A[j-1] if it is the lower value and leaving them in place. If not, Given the initial condition that the value in A[j] was the smallest value in the range A[j: A.length], this means the value in A[j-1] is now the smallest value in the range A[j-1: A.length] This also means that every value in the subarray A[j: A.length] is greater than the value at A[j-1]

Termination 2: The for loop on line 2 terminates when $j = i+1$ and given the maintenance property, this means that the value at $A[i]$ (which is $A[j-1]$) will be the smallest value in the range $A[i : A.range]$ ($A[j-1 : A.range]$)

Example : SQUARE Function : SQ(n)

1. $S \leftarrow 0$
2. $i \leftarrow 0$
3. while $i < n$
4. $S \leftarrow S+n$
5. $i \leftarrow i+1$
6. returns S

To prove : $S_i = n * n$ for any $i = n$

Basic step : For $n = 1$, $S = 0+1 = 1$

Induction Hypothesis : For an arbitary value K, $S_K = K * n$ and $i = K$ hold after going through the loop K times

Inductive step : when the loop is entered $(K+1)$ th time $S = K * n$ and $i = K$ at the beginning of the loop.

Inside the loop ; $S_{K+1} \leftarrow K * n + n$ and $i \leftarrow K+1$ producing $S_{K+1} = (K+1) * n$ and $i = K+1$. Thus $S = K * n$ and $i = K$ hold for any natural no. K.

Example : Factorial Function : FAC (n)

Prove that algorithm Fac (n) returns $n!$ for all non-negative integers $n \geq 0$ procedure Fac (n: nonnegative integer)

$n \geq 0$ if $(n = 0)$ the return 1

else return $n *$ Fac $(n-1)$

Basic Step : For $n = 0$

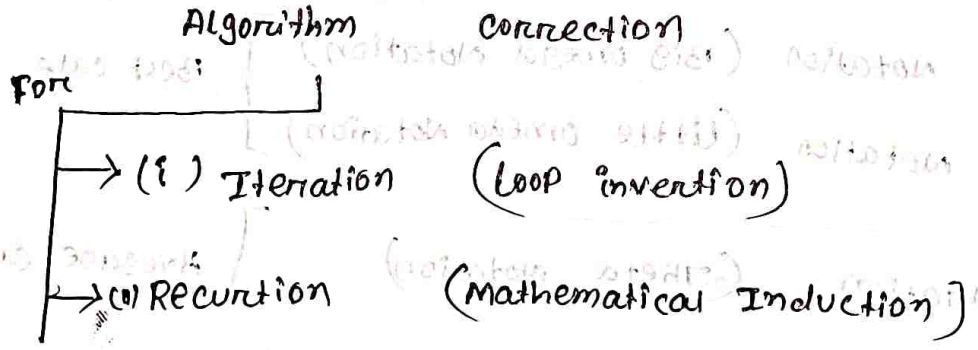
The proof is trival. Factorial $(0) = 1$.

Inductive Hypothesis : Fac (n) produces $k!$ For any $0 \leq k < n$

Inductive Steps : For $n = k+1$

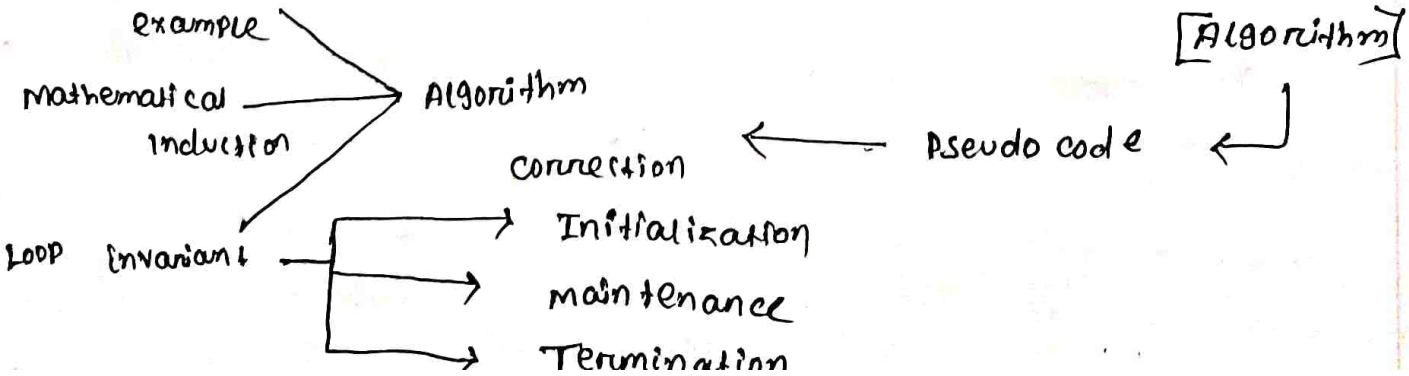
Fac (n) returns $(k+1) * Fac(k) = k+1 * k! = (k+1)!$

→



Till Now : [We have Finished]

Problem → Solution Steps must include '6' Features



Dt. 15.10.2022

TIME AND SPACE COMPLEXITY

> Time . Complexity

It is the time taken by the algorithm to execute each set of instructions. It is always better to select the most efficient algorithm when a simple problem can solve with different methods

> Space. Complexity

It is usuary reffered to as the amount of memory consumed by the algorithm. It is composed of two different spaces; Auxiliary space & input space.

> ASYMPTOTIC NOTATION

1) Ω Notation (Big omega Notation)] Best case,
2) ω Notation (little omega Notation)]
3) Θ Notation (Theta Notation)] Avegage case
4) o Notation (little oh Notation)] Worst case
5) O Notation (Big oh Notation)]

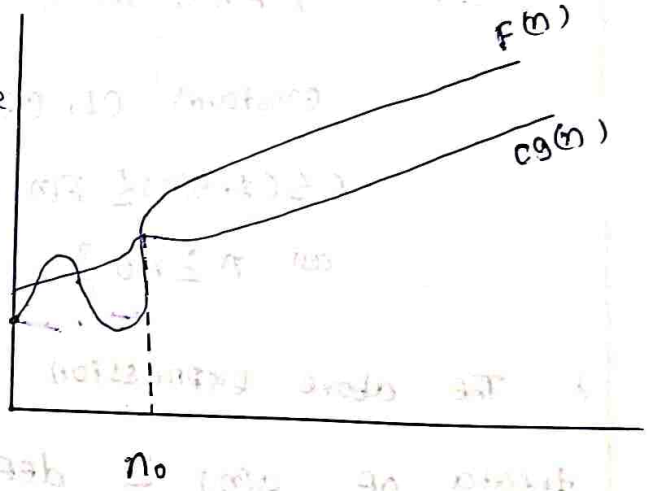
(M)
Low
↓
High
(KM)

Ω Notation (Big omega Notation)

Lower Bound (At least)

- The Ω notation denotes the lower bound of an algorithm i.e, the time taken by the algorithm can't be lower than this. In other words, this is the fastest time in which the algorithm will return a result. Its the time taken by the algorithm when provided with the best-case input. So, if a function is g(n), then the omega representation is shown as Ω(g(n)) and the relation is shown as:

$\Omega(g(n)) = \{ f(n) :$ there exist positive constants c and n_0 such that $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0 \}$



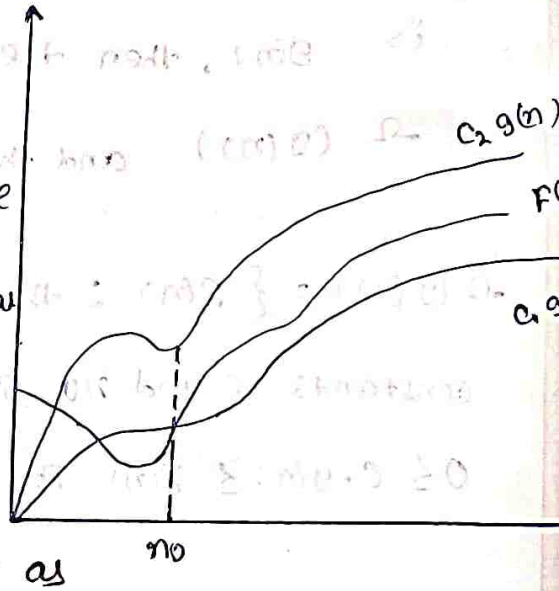
The above expression can be read as omega of g(n) is defined as set of all the functions f(n) for which there exist some constant c and n0 such that c * g(n) is less than or equal to f(n), for all n greater than or equal to n0

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

θ NOTATION (Theata Notation)

(Exact time)

- The θ Notation is used to find the average bound of an algorithm i.e, it defines an upper bound and a lower bound, and your algorithm will lie in between these levels, so, if the function is g(n), then the theta representation is shown as θ(g(n)) and the relation is shown as:



$\theta(g(n)) = \{ f(n)$: there exist positive constants $c1, c2 \in n0$ such that $0 \leq c1 \cdot g(n) \leq f(n) \leq c2 \cdot g(n)$ for all $n \geq n0 \}$

- The above expression can be read as $f(n) = \theta(g(n))$ theata of g(n) is defined as set of all the functions f(n) for which there exists some positive constants c1, c2, and n0 such that c1 * g(n) is less than or equal to f(n) and f(n) is less than or equal to c2 * g(n) for all n that is greater than or equal to n0

O NOTATION (Big O Notation)

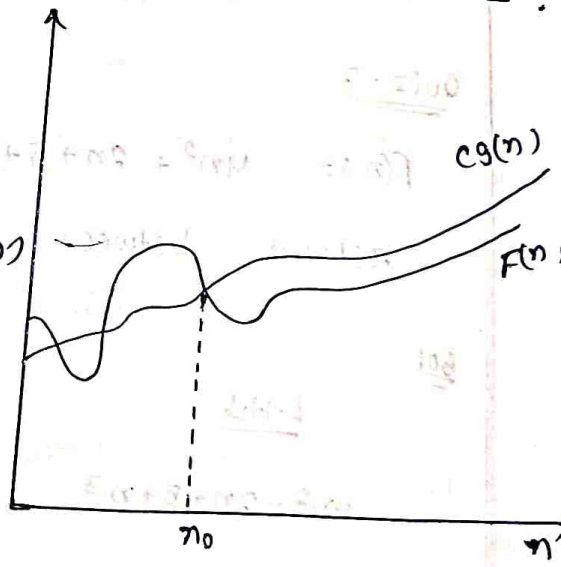
[Upper Bound (At most)]

> The Big O notation defines the upper bound of any algorithm i.e, your algorithm can't take more time than this time.

In other words, we can say that the Big O notation denotes the maximum time taken by an algorithm or the worst-case time complexity of an algorithm. So, big O notation is the most used notation for the time complexity of an algorithm.

So, if a function is $g(n)$, then the big O representation of $g(n)$ is shown as $O(g(n))$ and the relation is shown as:

> $O(g(n)) = \{F(n) :$ there exist positive constants c and $n0$ such that $0 \leq F(n) \leq c \cdot g(n)$ for all $n \geq n0\}$



$F(n) = O(g(n))$

> The above expression can be read as Big O of $g(n)$ is defined as a set of functions $F(n)$, for which there exist some constants c and $n0$ such that $F(n)$ is greater than or equal to 0 and $F(n)$ is smaller than or equal to $c * g(n)$, for all n greater than or equal to $n0$.

Ex:- Find out which one is greater if

> $f(n) = 2n + 3$ $g(n) = 3n$?

> change it to Big O Notation

Sol:

$2n + 3 \leq 3n$

$\Rightarrow f(n) \leq c \cdot g(n)$

$\Rightarrow 2(n) + 3 \leq 3(n)$

$\Rightarrow f(n) \leq O\ g(n)$

$\Rightarrow f(n) = O(n)$

Quiz:3

$f(n) = 4n^2 + 2n + 5 + n^3$ and $g(n) = 6n^2 + n^3$. Find the proper relation between $f(n)$ with $g(n)$ in terms of complexity.

Sol

L.H.S

$4n^2 + 2n + 5 + n^3$

R.H.S

$6n^2 + n^3$

$\Rightarrow f(n) \leq c \cdot g(n)$

$\Rightarrow n^3 + 4(n^2) + 2(n) + 5 \leq n^3 \cdot 6(n^2)$

$\Rightarrow F(n) = O\ g(n^3)$

$\Rightarrow F(n) = O(n^3)$

SPACE COMPLEXITY

Defination

→ Space complexity of an algorithm is the total space taken by the algorithm w.r.t the input size.

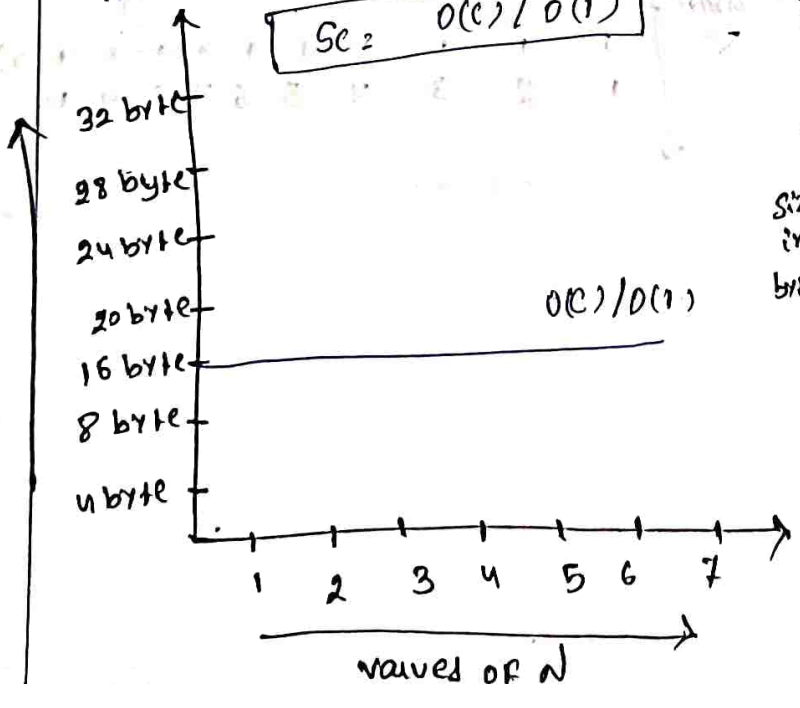
Space complexity : Space used by variable(s) + Auxiliary space
(Size of data types) used (Extra space / temporary space)

Algorithm-1 // sum of two variable n_1 & n_2

```
Function add (n1, n2)
{
    Sum = n1+n2;
    return sum;
}
```

n1 : 4 bytes
n2 : 4 bytes
Sum : 4 bytes
Aux : 4 bytes
Total : 16 bytes

SC : O(C) / O(1)

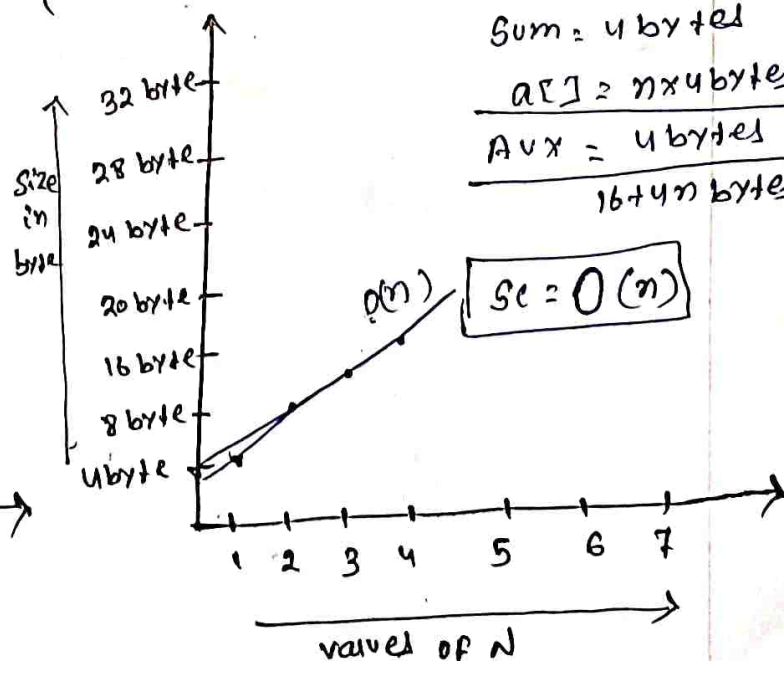


Algorithm-2 // sum of elements of an array

```
Function sum( a[], i )
{ int sum = 0
  for( i = 0 ; i <= n ; i++ )
  {
    sum = a[i] + sum;
  }
  return sum;
}
```

i : 4 bytes
n : 4 bytes
Sum : 4 bytes
a[] : n x 4 bytes
Aux : 4 bytes
16 + 4n bytes

SC : O(n)

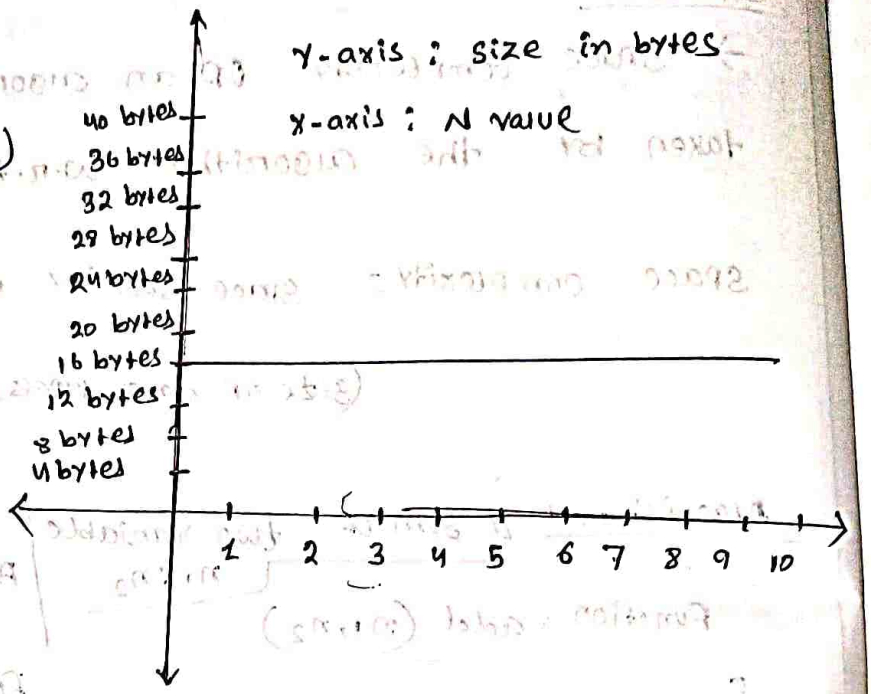


Dt - 17.10.2022

Algorithm-3 - Factorial of a number (iterative)

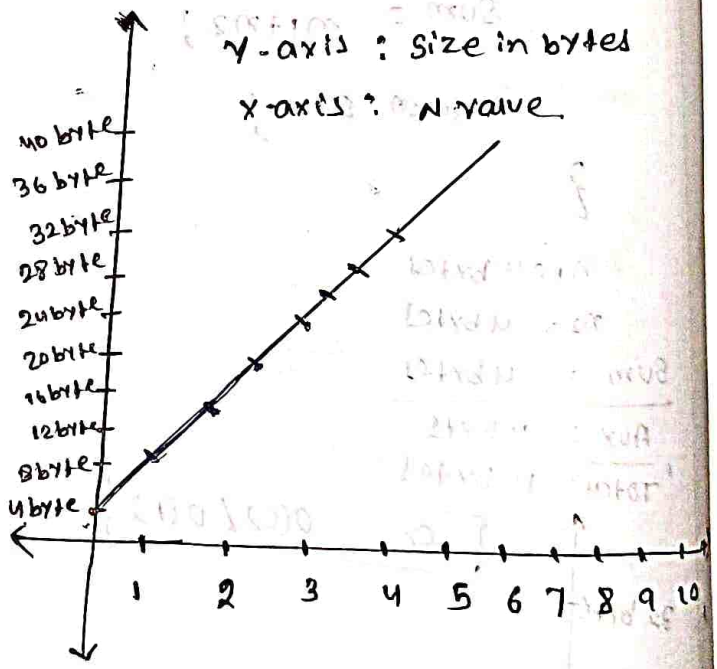
```
int Fact = 1;
For (int i = 1; i <= n; ++i)
{
  Fact *= i;
}
return Fact;
```

fact = 4b
i = 4b
n = 4b
Aux = 4b ⟶ Total = 16b



Algorithm-4 - Factorial of a number (recursive)

```
if (n <= 1)
{
  return 1;
}
else
{
  return (n * factorial 2(n-1));
}
```



Algorithm-3

Variables -

Fact - 4 bytes

i : 4 bytes

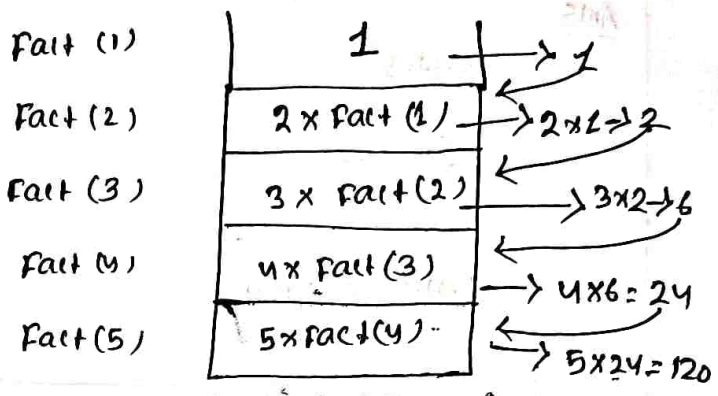
n : 4 bytes

Auxiliary space - 4 bytes

Space complexity - 16 bytes

$O(1)$

Algorithm-4



n : 4 bytes

Aux = n x 4 bytes

Total : 4n + 4 bytes

Space complexity : $O(n)$

Time complexity

$$\underbrace{O(1)/O(c)}_{\text{const.}} < \underbrace{O(\log\log\cdots\log n)}_{\text{logarithmic}} < O(\log n) < \underbrace{O(n) < O(n\log n)}_{\text{linear}} < \underbrace{O(n^2)}_{\text{linear arithm}}$$

$$\text{polynomial} < O(n^3) < \cdots < O(n^k) < \underbrace{O(2^n)}_{\text{exponential}} < O(3^n) < \cdots$$

$$\cdots < O(k^n) < \underbrace{O(n!)}_{\text{Factorial}} < \underbrace{O(n^n)}_{\text{Super polynomial}}$$

Dt-18.10.2022

Quiz-4

$F(n) = 3n^2+4n$ and $g(n) = n^2 \log n + n\log n + \log n$. Find the proper relationship between $F(n)$ & $g(n)$ in terms of complexity.

ANS

L.H.S : $3n^2+4n$

R.H.S : $n^2\log n + n\log n + \log n$

L.H.S $\geq$ R.H.S

- ~~$F(n) \geq c \cdot g(n)$~~ $\Rightarrow F(n) \leq g(n)$
- ~~$F(n) \leq c \cdot g(n)$~~ $\Rightarrow F(n) \leq c \cdot g(n)$
- ~~$F(n) = \Omega(n^2)$~~ $\Rightarrow F(n) = O\, g(n^2 \log n)$
- ~~$F(n) = \Theta(n^2)$~~ $\Rightarrow F(n) = O\, g(n^2 \log n)$

TIME COMPLEXITY

```
For (i=1; i<=n; i++)        ---> It will run for 1,2,3,.....,n-1,n times
{
    print i;                ---> It will run for n times
}
```

```
For (i=1; i<=n; i++)        ---> n+1
{
    print i;                ---> n
}
```

Time complexity -> $2n+1$.

Ex-2

```
for (i=1; i≤n; i=i+2)        ⟶  n/2 + 1
{
    print i;                 ⟶  n/2
}
```

Time complexity: $n+1$ $O(n)$

```
for (j=1; j≤n; j=j+400)      ⟶  n/400 + 1
{
    print j;                 ⟶  n/400
}
```

time complexity = $n+1$

Total Time complexity = $n+1+n+1$ (1 & 2)

$= 2n+2$ $O(n)$

Ex-3

```
for (i=1; i≤n; i=i+2)        ⟶  n/2 + 1
{
    for (j=1; j≤n; j=j+2)    ⟶  n/2 (n/2 + 1) =
    {
        print j;             ⟶  n2/4
    }
}
```

Time complexity = $\underset{1}{\left(\frac{n}{2}+1\right)} + \underset{2}{\left(\frac{n^2}{4}+\frac{n}{2}\right)} + \underset{3}{\left(\frac{n^2}{4}\right)}$

$= O(n^2)$

Ex-5

```
for (i=1; i<=n; i=i+1)          ---->  1
{
   for (k=1; k<n; k=k)          ---->  1(n^n)
   {
      print k;                  ---->  n^n
   }
}
```

Time complexity = $O(n^n)$

TC = $2n^n + 1$.

6

```
for (i=1 ; i<=n ; i=i*4)
{
    print i;
}
```

7

```
for (j=n ; j>=1 ; j=j/4)
{
    print j;
}
```

ANS-6

Assume $i > n$

$\Rightarrow 4^k > n$

$\Rightarrow \log 4^k > \log n$

$\Rightarrow k \log 4 > \log n$

$\Rightarrow k > \log_4 n$

Time comp. $= O(\log_4 n)$

J

$n \rightarrow \frac{n}{4^0}$

$\frac{n}{4} \rightarrow \frac{n}{4^1}$

$\frac{n}{16} \rightarrow \frac{n}{4^2}$

$\frac{n}{64} \rightarrow \frac{n}{4^3}$

$\frac{n}{4^k}$

ANS-7

Assume $j \leq 1$

$\Rightarrow \frac{n}{4^k} = 1 \Rightarrow n = 4^k$

$\Rightarrow k = \log_4 n$

Time comp. : $O \log_4 n$

Note: $a^k = 1$

$\Rightarrow k = \log_a 1$

8)

```
for( i=1 ; i*i<n ; i++)
{
    print i;
}
```

Assume - $i \times i \geq n$

→ $i^2 = n$

→ $i = \sqrt{n}$

Time complexity : $O(\sqrt{n})$

9

```
for ( j=1 ; j<=n ; j+=1)        ⟶ n+1
{
  for ( i=0 ; i<=j ; i++)       ⟶ n(n+1)
  {
    print i;                    ⟶ n2
  }
}
```

$n+1+n^2+1+n^2$

$= 2n^2+n+2$

Time complexity: $O(n^2)$

10)

```
for ( j=1 ; j<=n ; j=j*2)       ⟶ log2 n
{
  print x;                      ⟶ log2 n
}
for (i=1 ; i<=k ; i=i*2)        ⟶ log2 k
{
  print i;                      ⟶ log2 k
}
```

T.C = $O(\log(\log n))$

•RECURRENCE RELATION• •CH-3.1•

-> A recurrance is an equality and inequality that describes a function is terms of its values on smaller inputs. To solve a recurrance relation means to obtain a function defined on the natural number that satisfy the reccurance.

There are three methods used to solve the recurrance relation.

i) Substitution Method
ii) Recursion tree method
iii) Master Method.

a

```
int print (n)  ———————— T(n)
{
   i=1
   if (n>0)  ———————— 1
   {
      print i;  ———————— 1
      print (n-1);  ———————— T(n-1)
   }
}
```

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

-> Recurrence Relation

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

Using substitution method

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

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

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

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

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

$= T(n-3) + 1$

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

$\downarrow$

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

$\downarrow$

$T(n-2) + 2$

$\downarrow$

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

$\downarrow$

$T(n-3) + 3$

$\vdots$

K times $T(n) = T(n-K) + K$

Assume $n \leq 0 \Rightarrow$ $n=0$ / $n=1$ $\Rightarrow T(0) = 1$

$T(n) \leq T(0)$

$T(n-k) \leq T(0) \Rightarrow n-k \leq 0 \Rightarrow \boxed{n=k \;/\; k=n}$

Put the value of k as n

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

$= T(0) + n$

$= 1+n$

$\boxed{T(n) = n} \longrightarrow$ Time complexity $= O(n)$

ii RECURSSION TREE METHOD

Solution steps: (Division Function)

1) Find the cost included in the equation
2) Follow the order of function (i.e $T(\frac{n}{2})$)
3) Left side mention the number of nodes created / generated
4) Right side, mention the cost after adding the generated cost of each node, at each level.
5) Total cost = cost of leaf node(s) + cost of Intermediate node(s)

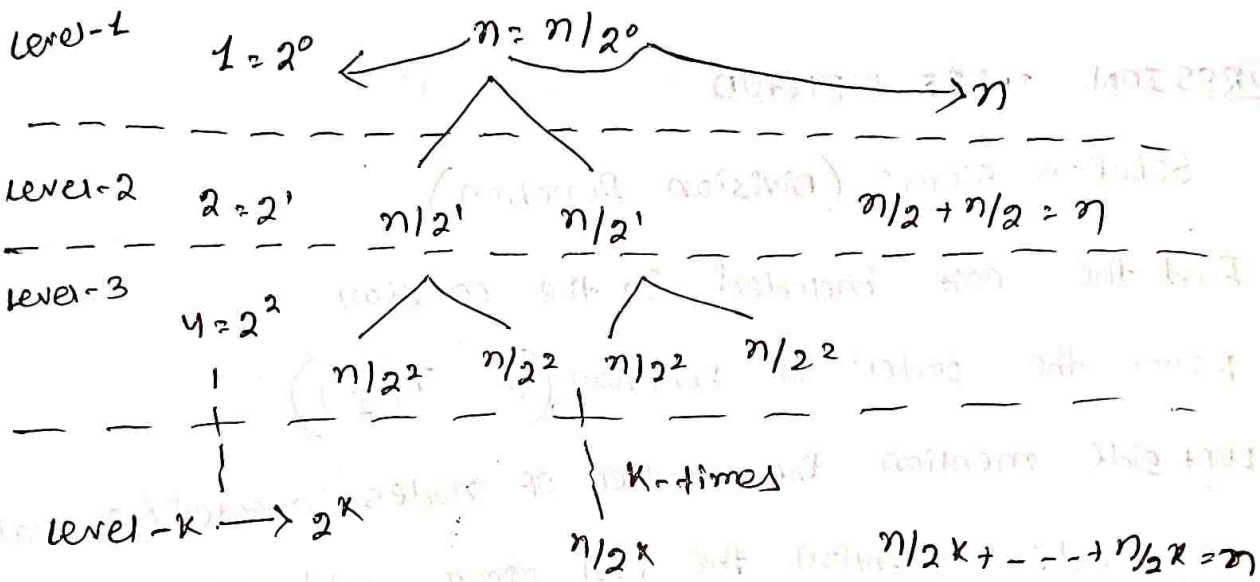
Time complexity (cost of each leaf node × number of leaf nodes) (number of intermediate nodes at each level × Cost of each node at each level)

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

$\boxed{T(n) = 2T(n/2) + n}$ → cost

cost involved (at each level)

no. of nodes (at each level)



Total cost = $C_L + C_I$ → Intermediate node (cost)

↳ Leaf node

Assume

$n \le 1 \Rightarrow T(n) \le T(1) \Rightarrow T(n/2^K) \le T(1)$

$\Rightarrow n/2^K = 1 \Rightarrow n = 2^K$

$\Rightarrow \boxed{K = \log_2 n}$

$\therefore \quad C_L : 2^k = 2^{\log_2 n} = n^{\log_2 2} = n^1 = n$

$C_I = k * n = \log_2 n * n = n \log n$

NOTE :

$$a^{\log_2 b} = b^{\log_2 a}$$

Total cost = $C_L + C_I = n + n \log_2 n$

Time complexity = $O(n \log_2 n)$ $\begin{cases} n = 8 \\ n \log_2 n \rightarrow 8 \log 8 \end{cases}$

$= 8 \times 3 = 24$

iii) DIVISION FUNCTION [MASTER METHOD]

$$T(n) = a\, T(n/b)^{(k(n/b))} + \Theta(n^k \log^p n)$$

where, $a \geq 1$; $b \geq 1$; p is any real number

case 1 :- if $a > b^k$, then $T(n) = \Theta(n \log_b a)$

case-2 :- if $a = b^k$, then

a) if $p > -1$ then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$

or $T(n) = \Theta(n^k \log^{p+1} n)$

b) if $p = -1$, then $T(n) = \Theta\left(\log \log_b^a \log \log n\right)$

or $T(n) = \Theta\left(n^x \log \log n\right)$

c) if $p < -1$, then $T(n) = \Theta(n \log_b^a)$

or $\Theta(n^x)$

Case-3:

a) $p \geq 0$, then $T(n) = \Theta\left(n^x \log^p n\right)$

b) $p < 0$, then $T(n) = \Theta(n^x)$

Dt-31.10.2022

Quiz-8

using master theorem solve the below Recurrance relations:

a) $T(n) = 44T(n-4) + n2$

b) $T(n) = 0.2T(n-1) + n$

c) $T(n) = T(n-5) + 5^5$

Ans

a

$T(n) = 44T(n-4) + n^2$

$a = 44 \quad b = 4 \quad k = 2 \quad p = 0$

$a > 1 \Rightarrow O(n^k a^{n/b})$

$= O(n^2 @ 44^{n/4})$

b

$T(n) = 0.2T(n-1) + n$

$a = 0.2 \quad b = 1 \quad k = 1$

$a < 1 \Rightarrow O(n^k) = O(n)$

c

$T(n) = T(n-5) + 5^5$

$a = 1, b = 5, k = 0,$

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

$= O(n^{0+1}) = O(n^1) = O(n)$

HEAP

→ Heap tree is Almost complete Binary tree.

→ Heap tree follows the two property :

a) Structural property

b) Ordering property

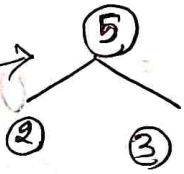
→ Building Heaptree, we have to follow two methods

1) Insert the elements one by one and swap if needed

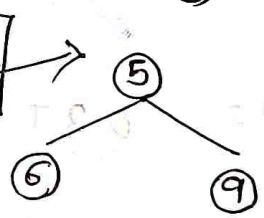
2) Heapify Method

› There are two type of Heap :

1. Max Heap [Parent Node > Child node(s)]



2. Min Heap [Parent Node < Child node(s)]



BUILD HEAP TREE

1) Using the first Method (Insert the elements one by one and swap if need

Solution steps :

i) Insert one element at a time.

ii) Check the tree whether it follows any property if Min Heap / Max Heap . If not , then Swap.

Example

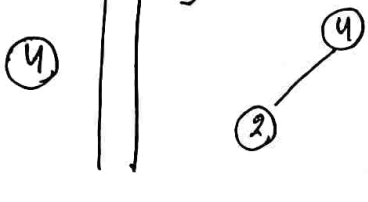
Using the swap property, build the max heap.

Sol^{n} $\{4, 2, 1, 7, 8, 10\}$

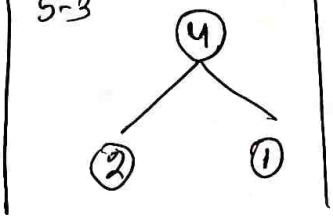
S-1

4	2	1	7	8	10
0	1	2	3	4	5

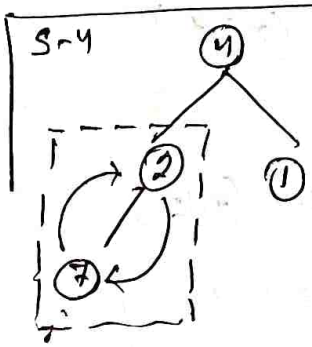
S-2



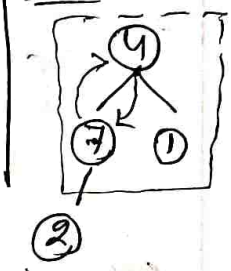
S-3



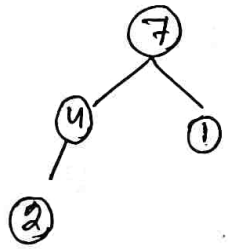
S-4



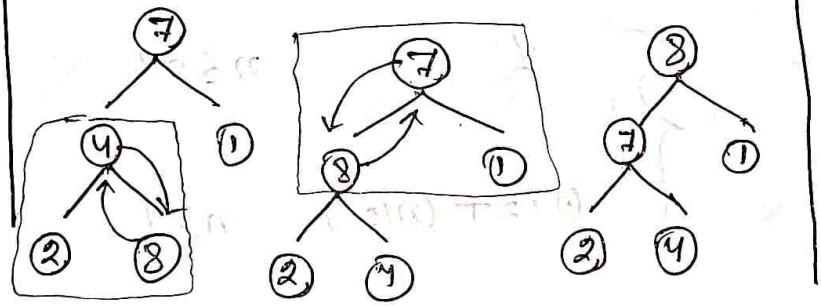
S-4.1



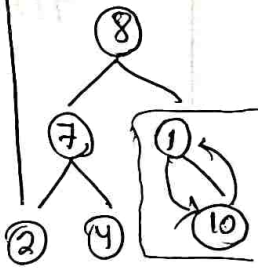
S-4.2



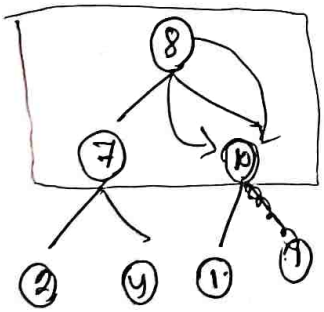
S-5

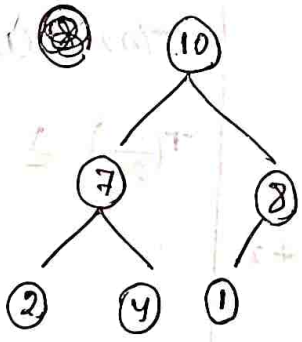


S-6



S-6.1





Q-9

Derive the Recurrance relation of the given algorithm. Find the time complexity using substitution method

```
void icon(n)
{
  if (n>1)
  {
    icon(n/2)
```

Ans

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

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

$$T = \left[T\left(\frac{n/2}{2}\right) + 1\right] + 1$$

$$T = T\left(\frac{n}{4}\right) + 2$$

$$= T\left(\frac{n}{8}\right) + 3$$

$$= T\left(\frac{n}{16}\right) + 4$$

$$\vdots$$

$$T(n) = T\left(\frac{n}{2^k} + k\right)$$

$T(n) \geq (1) T$

$T\left(\frac{n}{2^k}\right) \geq$

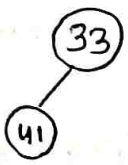
Quiz - 10

Build min heap by considering the element setting given below

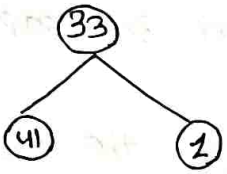
$E = \{33, 41, 1, 5, 6, 0, 3\}$

S-1

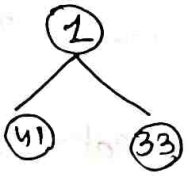
S-2



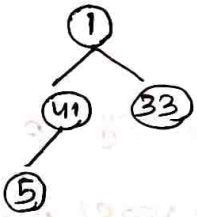
S-3



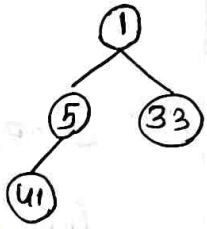
S-3.1



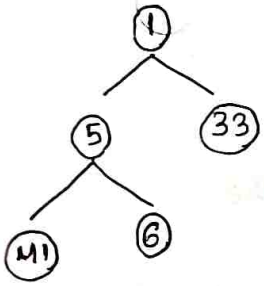
S-4



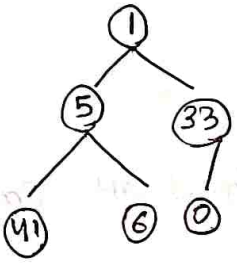
S-4.1



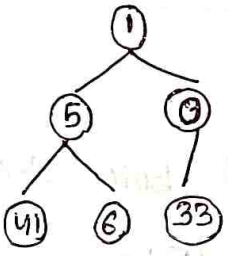
S-5



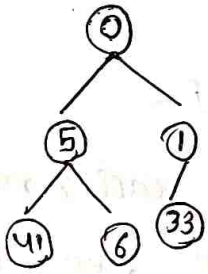
S-6



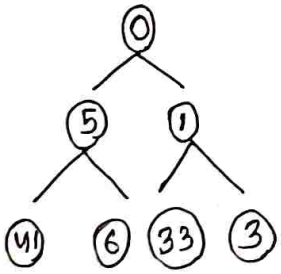
S-6.1



S-6.2



S-7



0	5	1	41	6	33	3

BUILD HEAP TREE

2) Using heaping method :

Solution steps :

1) Convert the array to heap tree format. (i.e, arrange the array in heap tree format)

2) Swap / Heapify if needed as per the requirement (i.e max Heap / min Heap)

Time complexity - $O(n)$

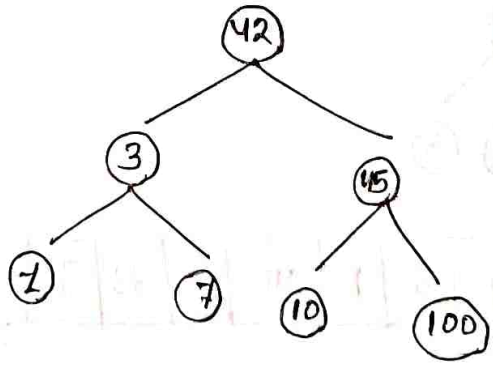
EXAMPLE :

Using heapify method build the max-heap from the given element set (E).

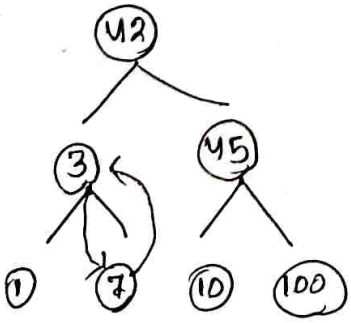
$E = \{42, 3, 45, 1, 7, 10, 100\}$

Solution :

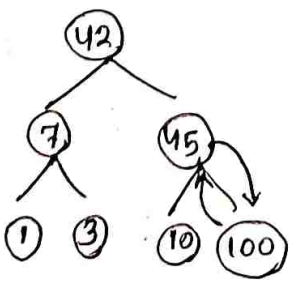
42	3	45	1	7	10	100
0	1	2	3	4	5	6



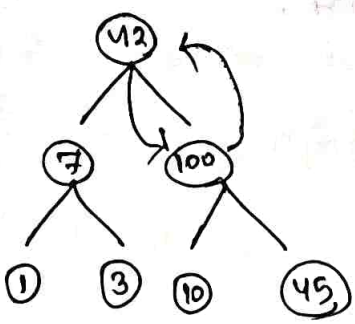
S-1



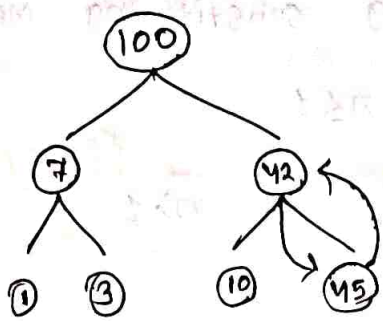
S-1.2



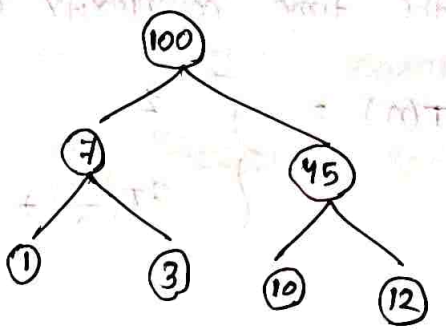
S-2



S-3



S-4



100	7	45	1	3	10	42

DELETING OF ELEMENT IN HEAP TREE

Single Node :

Best Case → $O(1)$ / $\Omega(1)$

(Last node delete)

Worst time complexity → $\log_2 n$

n nodes WTC → $n \log_2 n$

Quiz - 1)

Find the time complexity using substituting method.

$$T(n) = \begin{cases} 1 & \text{if } n \le 1 \\ 3T(\frac{n}{3}) + \log_3 n & \text{if } n > 1 \end{cases}$$

Ans

$$T(n) = 3T(n/3) + \log_3 n$$

$$\therefore T(n/3) = 3T(n/9) + \log_3^{(n/3)}$$

$$\therefore T(n/9) = 3T(n/27) + \log_3^{n/9}$$

$$T(n) = 3T(n/3) + \log_3 n$$

$$= 9T(n/9) + 3\log_3^{(n/3)} + \log_3 n$$

$$= 27T(n/27) + 9\log_3^{n/9} + 3\log_3^{n/3} + \log_3 n$$

(k times)

$$T(n) = 3^k T(n/3^k) + 3^{n-1}\log_3 n + 3^n \log_3 n \; ---- \log_3 n$$

Assume, $n \le 1 \Rightarrow T(1) = 1 \Rightarrow T(n) \le T(1) \Rightarrow \frac{n}{3^k} \le 1$

$\Rightarrow n = 3^k$ or $n < 3^k$

$\Rightarrow k = \log_3 n$

putting the value of k,

$$T(n) = 3^{\log_3 n} T(n/n) + 3^{\log_3 n - 1} \log n/3 \log_3 n - 1 + ---- \log$$

$$\Rightarrow T(n) = n + \frac{3^{\log_3 n}}{3} \log_3 n/3 \; \frac{\log_3 n}{3} + --- (\log_3 n$$

$$\Rightarrow T(n) = n + \frac{n}{3} + ---- \log_3 n$$

$$\therefore T(n) = O(n) \quad (\text{Ans})$$

HEAP SORT

→ Heap Sort is a sorting-technique whose time complexity is less (i.e $O(n \log_2 n)$) as compared to insertion sort i.e $(O(n^2))$ bubble sort i.e, $(O(n^2))$

Solution Step: (Sorting is ascending order using Heapify Method)

1) convert the element set to array or Rearrange the Heap tree From the array
2) Follow the heapify to build the MinHeap.
3) Delete the root element by directly swapping with last node of the Min heap.
4) Store the deleted elements in an array From Left to Right Manner
5) Repeat step (2), (3) & (4.) until all elemends / nodes are deleted From the Min heap.

$O(n) + O(n \log n) = O(n \log n)$

NOTE

$O(n \log n) + O(n \log n) = 2n \log_2 n = O(n \log_2 n)$

HEAP SORT = Build heap + Delete the Heap

one by one + Swap → (For n nodes) $O(n \log n)$

Heapify method → $O(n)$

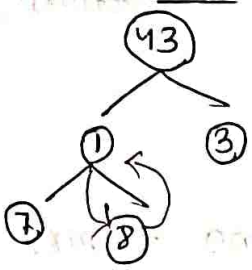
$O(n \log n)$

Ex:- using maxheap :-

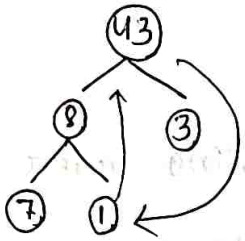
$E = \{43, 1, 3, 7, 8\}$

Soln:-

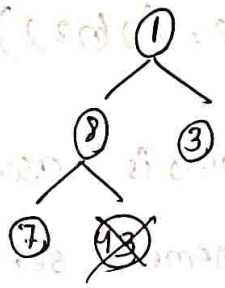
Step-1

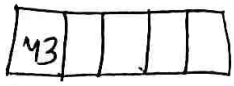


Step-2

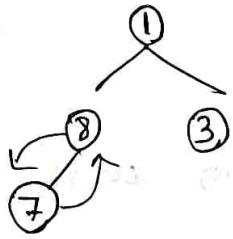


Step-3

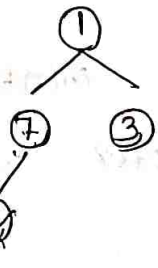




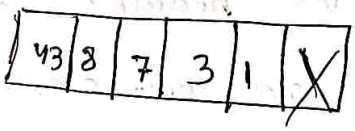
Step-4



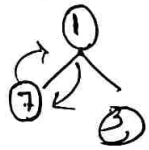
Step-5



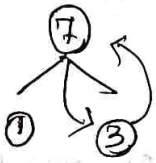
Step-6



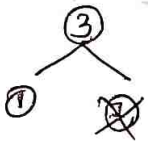
Step-7



Step-8

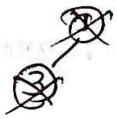


Step-9



Step-10

Step-11



BY COR-MAN | NOTE FOR EXAM

Heap Sort

MAX HEAP -> Ascending order

Min Heap -> Descending order.

quiz -12

$$T(n) = \begin{cases} 1, & \text{if } n \le 1 \\ 4T(n/4) + \log_4 n, & \text{if } n > 1 \end{cases}$$

Solve using substitution method....

Sol

$T(n) = 4T(n/4) + \log_4 n$

$\therefore T(n/4) = 4T(n/16) + \log_4 (n/16)$

$\therefore T(n/16) = 4T(n/64) + \log_4 n/64$

$T(n) = 4T(n/4) + \log_4 n$

$= 16T(n/16) + 4\log_4 n/4 + \log_4 n$

$= 64T(n/64) + 16\log_4 n/16 + 4\log_4 n/4 + \log_4 n$

⋮

k times

$T(n) = 4^k T(n/4^k) + 4^{k-1}\log_4 n/4^{k-1} + 4^{k-2}\log_4 n/4^{k-2} + \cdots + \log_4 n$

Assume $n \le 1 \Rightarrow T(1) = 1$

$\Rightarrow \frac{n}{4^k} \le 1 \Rightarrow 4^k = n \Rightarrow k = \log_4 n$

Wrong

putting the values of k,

$T(n) = 4^{\log_4 n} T(n/n) + 4^{\log_4 n - 1}\log n/4^{k-1} + 4^{\log_4 n - 2}\log n/4$

$\cdots + \log_4 n$

$= n + \frac{n}{4} + \frac{2n}{4^2} + \cdots + \log_4 n$

$= n + n\left(\frac{1}{4} + \frac{2}{4^2} + \frac{3}{4^3} + \cdots \frac{k}{4^k}\right)$

Let, $S = \frac{1}{4} + \frac{2}{4^2} + \frac{3}{4^3} + \cdots + \frac{k}{4^k}$

$\frac{S}{4} = \frac{1}{4^2} + \frac{2}{4^3}$

$T(n) = O(n)$

CH-4 CLASSIFICATION OF SORTING TECHNIQUE :

Classified by

1) Number of comparisions

2) Number of swaps

3) Memory usage < In place : The auxiliary space needed From
Out of place : The auxiliary space needed more From in place.

4) Recursion

5) Stability : order of element remain same after sorting

6) Adaptibility : same complexity in condition (best, AVG & worst)

7) Internal Sorting / External Sorting

↓ required primary or internal memory (RAM) during starting.

↘ Required secondary Storage (SSD/HDD) during Sorting

Sorting technique	Stable	Inplace	Adaptive
BUBBLE SORT	yes	yes	yes
INSERTION SORT	yes	yes	yes
SELECTION SORT	NO	yes	NO
HEAP SORT	NO	yes	NO

• GRAPH •

Graph is a non-linear data structure which include set of vertices (V) & set of edges (E)

Graph (G) - (V, E), where V is set of vertices & E is set of edges.

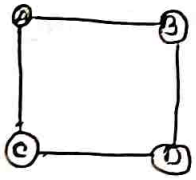
K- regular graph

- All vertices must have same degree, where k-degree
- → Maximum no. of edges,

$$E = \frac{V \times K}{2}$$

- → Finite or infinite

Ex:-



K = 2

[2-regular graph]

- → Joint or disjoint graph
- connected or not-connected

complete graph (K_n / K_v)

- → All vertices must have (n-1) or (V-1) degree, where,

n = no. of nodes

V = no. of vertices

- → Maxm no. of edges

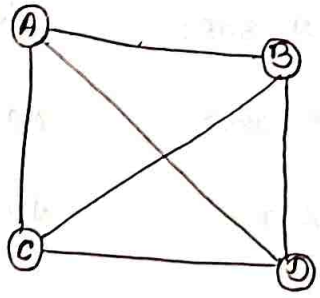
$$E = {}^{n}C_2 = \frac{n(n-1)}{2}$$

or

$$E = {}^{V}C_2 = \frac{V(V-1)}{2}$$

- → Finite graph.

Ex:-



[K_4 - complete graph]

- → Joint graph / connected-graph

NOTE:

1) Every regular graph is complete graph c.

2) Every complete graph is regular graph

Graph representation:

There are two methods:

1) Adjacency List / Array

2) Adjacency matrix

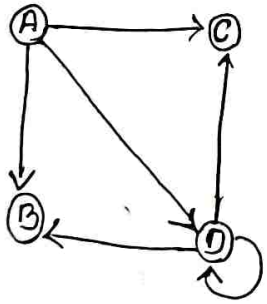
↳ 2D array a[][]

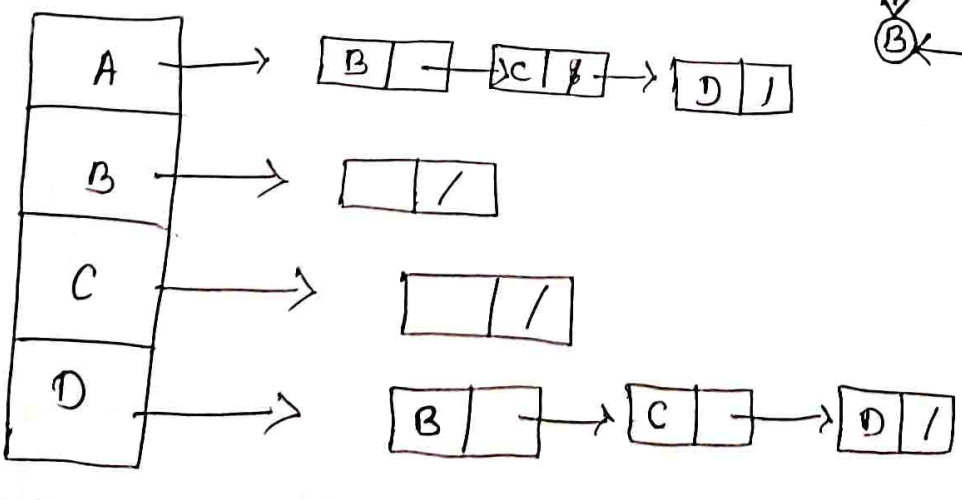
3D array a[][][]

ND array a[][]---n

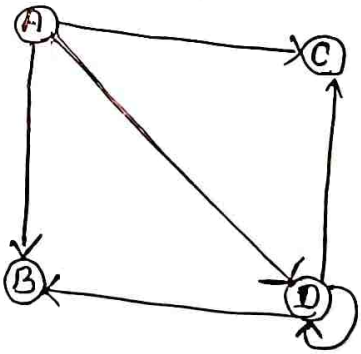
Adjacency List / Array

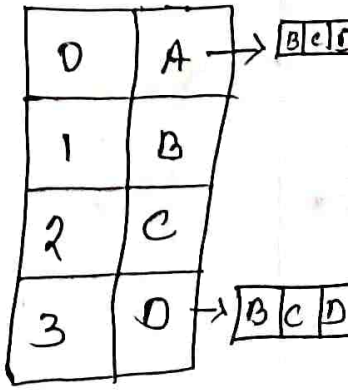
Direct graph (using linked list)



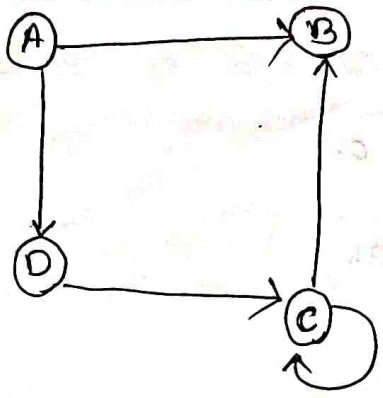


direct graph (using array)





Time complexity: O(vertex + Edge)

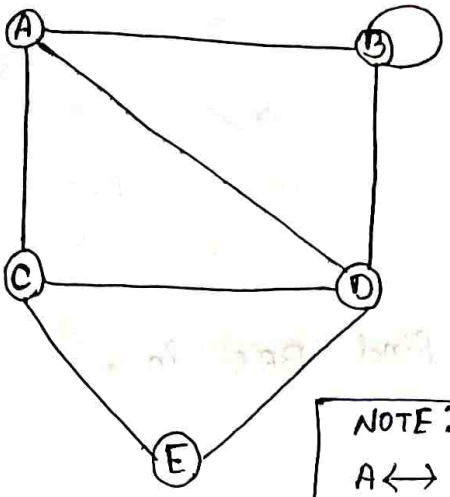


using (linked list)

Dt - 11.11.2022

ADJACENCY MATRIX :

undirected graph representation

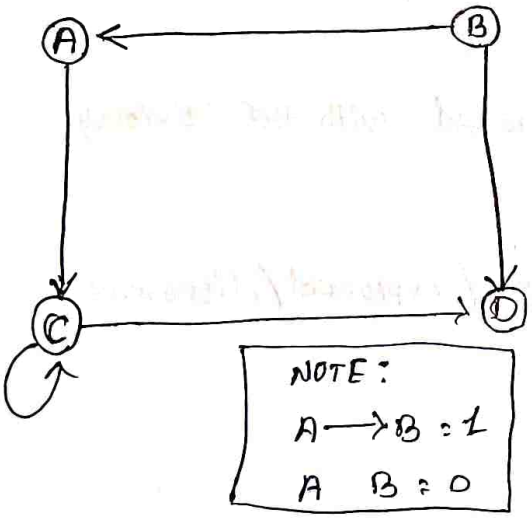


	A	B	C	D	E
A	0	1	1	1	0
B	1	1	0	1	0
C	1	0	0	1	0
D	1	1	1	0	1
E	0	0	1	1	0

5×5

NOTE :
A ↔ B = 1
A B = 0

Directed graph representation

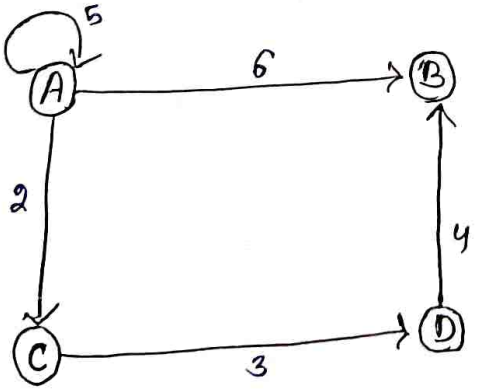


	A	B	C	D
A	0	0	1	0
B	1	0	0	1
C	0	0	1	1
D	0	0	0	0

4×4

NOTE :
A → B = 1
A B = 0

weighted Directed graph representation



	A	B	C	D
A	5	6	2	0
B	0	0	0	0
C	0	0	0	3
D	0	4	0	0

NOTE :
-> IF weighted directed edge => put the value
-> other wise = 0

Graph Traversal :

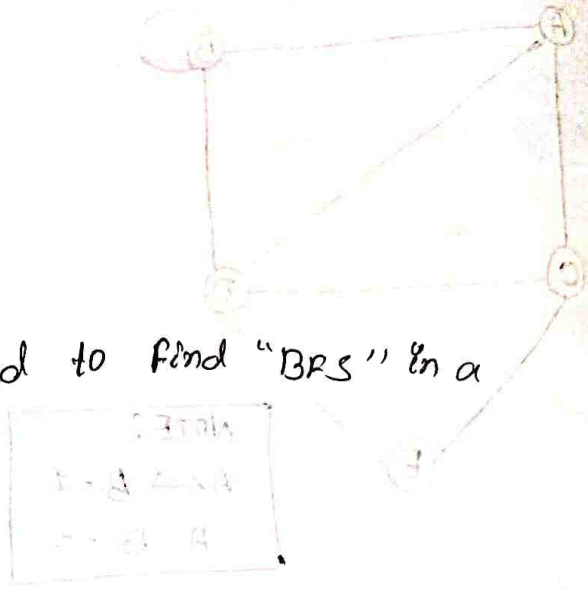
There are two methods :

1) BFS (Breadth First search)

2) DFS (Depth First Search)

BFS (Breadth First Search)

→ "Queue" data structure is used to find "BFS" in a graph



Solution steps :

① Start the traversal from a vertex whose indegree is "0"

② Explore the neighbour nodes attached with the previously selected

③ Stop until all vertices are visited / explored / discovered

Algorithm (BFS)

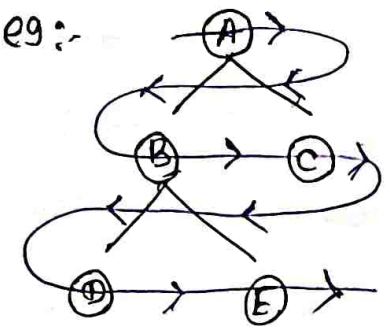
Input : "S" is the source vertex

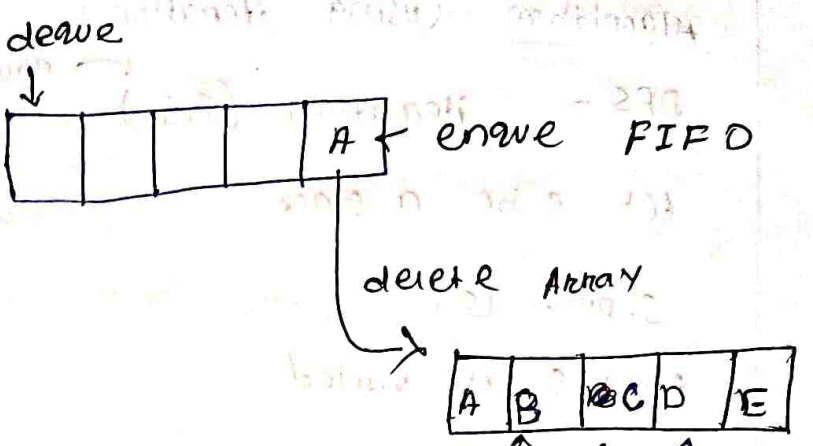
```
BFS (G,S)
  Let "Q" be the Queue
  Q.enqueue (S)
  Mark S as visited
  while (Q is not empty)
    V = Q.dequeue ()
  For all neighbours "w" of "v" in graph 'G' if "w" is not visited
```

Q. enqueue (w)

Mark "w" as visited.

eg :-





V = A = Q. dequeue

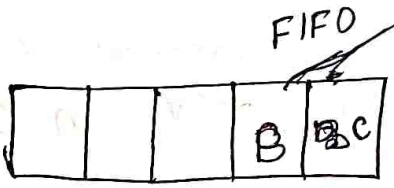
W = neighbours = B, C

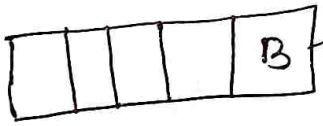
V = C

W = {Q}

↙

aka neighbours



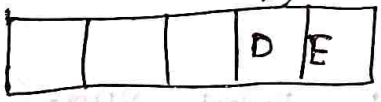


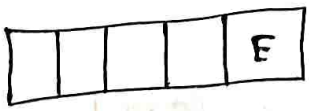
V = B

W = {A, D, E}

↙

already in array





NOTE

-> Traversal is any direction

-> Adjecency list / Adjacency array / Adjgacency matrix

T.C = O(V+E)

↓ T.C = O(V2)

Dt - 12/11/22

a

DFS (Depth First search)

Algorithm (using iteration)

DFS - iteration (G, s.) ← source vertex

let s be a stack

s.push (s)

mark s as visited

while (s is not empty)

v = s.top()

s.pop()

For all neighbours "w" of "v" in graph "G"

if w is not visited.

s.push (w)

Mark w as visited.

Algorithm (using recursion)

DFS : recursive (G, s)

Mark s as ~~vertex~~ visited.

For all neighbours "w" of s in graph G if w is not visited.

DFS : recursive (G, w)

∴ "STACK" Data structure is used while finding the DFS

(LIFO).

Solution Step:

1. Start the traversing of the vertex where indegree is 0.
2. Follow any direction (Left / right)

   From the previously selected vertex to the immediately eliminated vertex / vertices and store the vertex in a stack.
3. Stop the traversal if all the vertices have successfully visited / explored
4. Print the vertices present within the stack in LIFO order

TIME COMPLEXITY

BFS

$O(V+E)$

$O(V^2)$

due to adjacency list / Array & Matrix

DFS

$O(V+E)$

Adjacency list / Array.

TESTING BIPARTITENESS:

A Bipartite Graph is one whose vertices can be divided into disjoint and independent sets, say U & V such that every edge has one vertex is 'U' and the other is 'V'

ALGORITHM

Bipartite (G, s)

let q be an empty queue and s as the source

color v as Red.

q.enque (s)

While !q.empty()

u = q.dequeue()

For each v in u.adj.list

if v.color is nil

v.color : (u.color == RED) ? BLUE : RED

v.enqueue (v)

else if (v.color == u.color)

return "Not Bipartite"

return "Bipartite".

Solution step

1) Start the testing of Bipartiteness from any of the vertex present in the graph 'G' and color the vertex is "RED"

2) Then color the neighbour vertex / vertices of the previously selected vertex as 'BLUE'

(3) Repeat Step 1 & 2 until all vertex have successfully colored

(4) If the same color occur consecutively, then we can say the graph is not bipartite.

NOTE

- Testing of bipartiteness on a graph is the applicating of "BFS"
- Queue linear data structure is used while finding the Bipartitness.
- Time complexity: $O(MHE1)$ / $O(V2)$
  - $O(MHE1)$ → AL / AP
  - $O(V2)$ (Amatrix)

Topological Society / Topological ordering.

- Topological Sorting → It is a linear ordering of its vertices such that for every directed for every edge uv for vertex u to v, u comes before vertex v.
- Topological sort / order should be possible in order directed acyclic graph (DAG).
- Every DAG must have atleast one or unique Topological ordering / sorting.

Algorithm <TARDOS>
(Text book)

To compute a topological sorting / ordering of "G" :-

Find a node / vertex "v" with no incoming edges and order if finite delet "v" from G.

Recursively compute a topological ordering of G-{v}, and append this order after "v"

Algorithm <CORMEN>

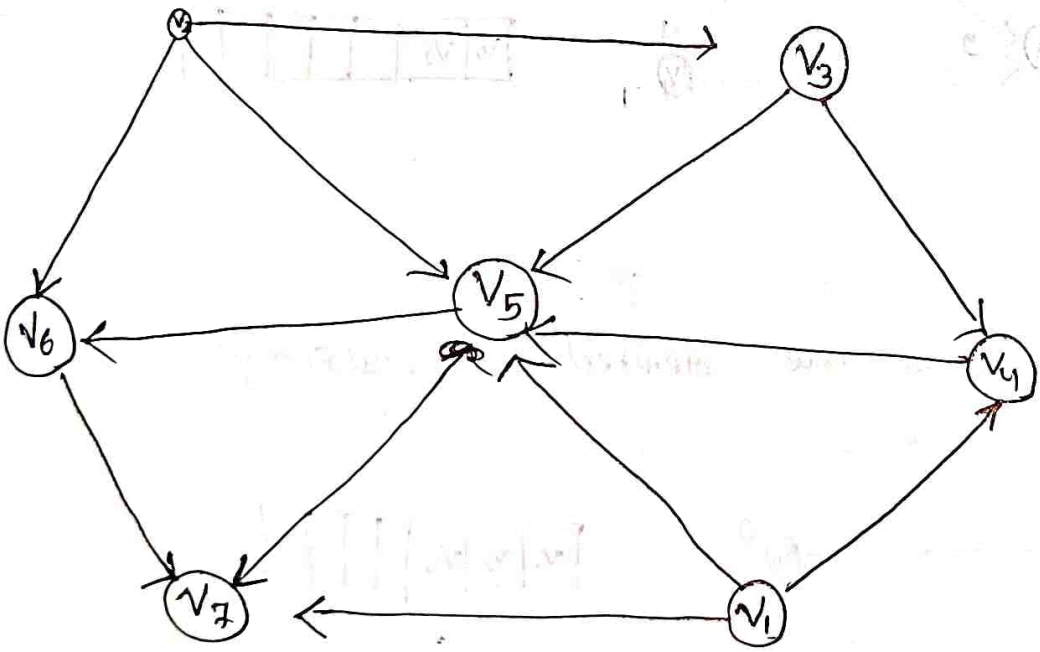
Topological sort (G)

1. Call DFS (G) to compute finishing time v.f for each vertex "v"
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices.

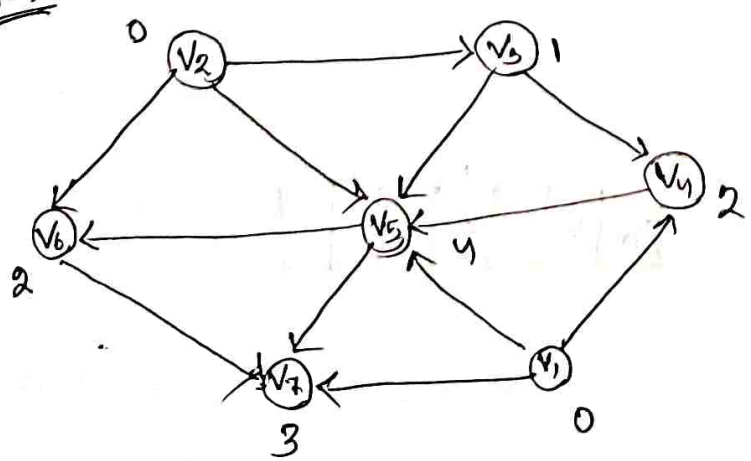
Solution steps :- (using DFS / stack data structure)

1. Start from the vertex whose outdegree is "0"
2. put the vertex in stack and mark as visited.
3. Than select the next vertex whose out degree is "1" & then follow "step-2"
4. Stop exploring the vertex once all vertices have visited / explored successfully.
5. Print the vertices in LIFO order to get the correct Topological Sorting / ordering of the Directed Acyclic Graph (DAG)

EXAMPLE :- (TARDOS)

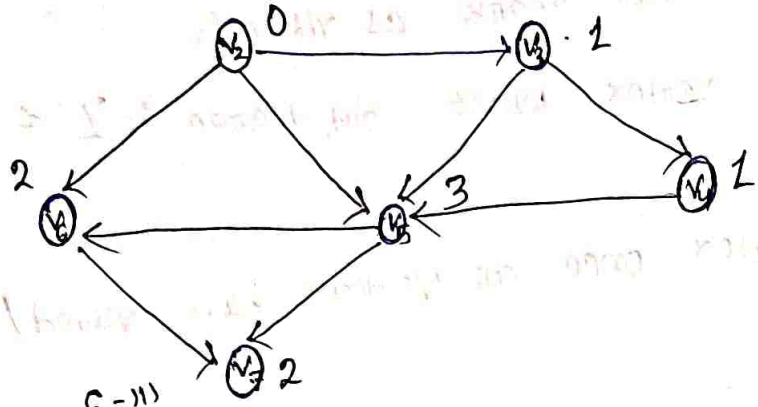


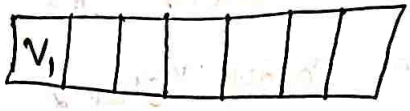
S-1



S-II)

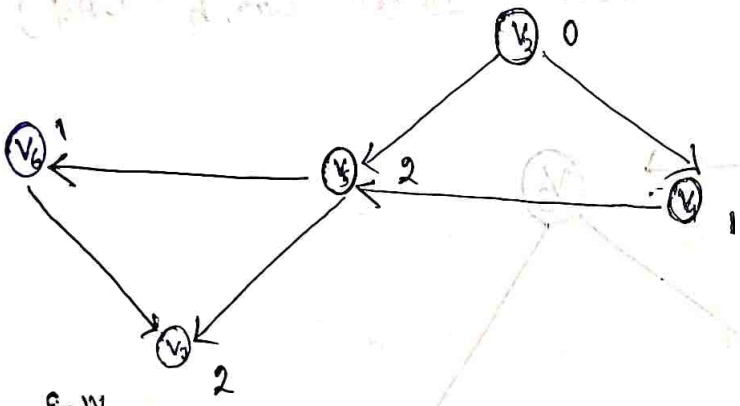
Remove or delete the previously selected source and store in an array.

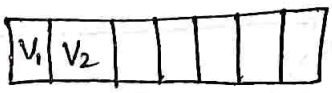




S-III)

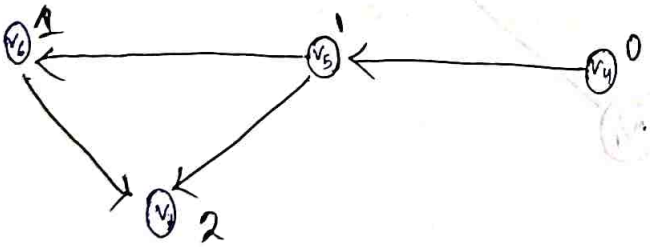
Update the indegree of all the remaining vertices.

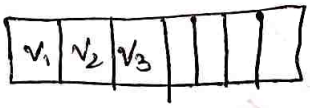


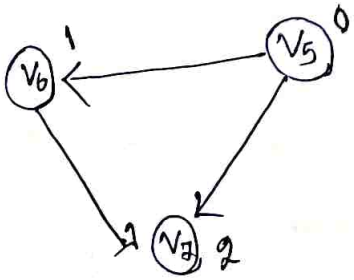


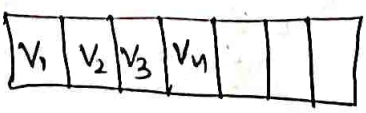
S-IV)

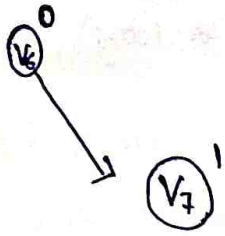
Stop when all vertices are deleted successfully.

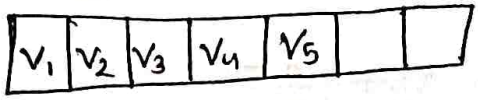


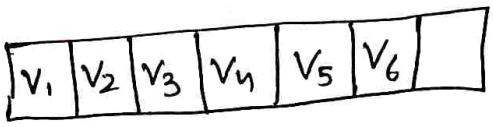


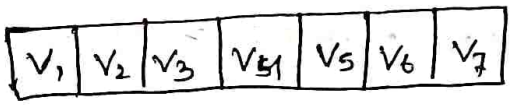






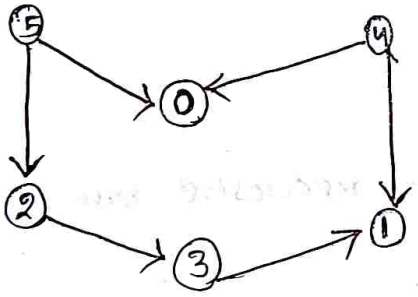


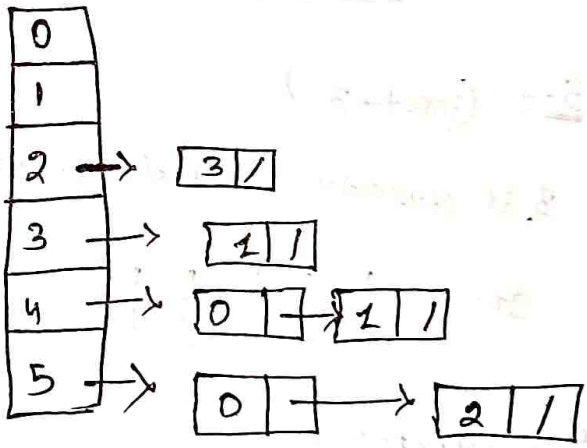




EXAMPLE (COREMEN)

Find the TS/TO using DFS / Stack





[Adjacency list]

S-1 (intvp : 0)

Topological Sort(0)

visited(0) = TRUE

↓

List is empty. No more recursive case

Stack : | 0 |

S-2 (input - 1)

Topological Sort(1), visited(1)=TRUE

↓

list is empty. No more recursive call.

Stack | 0 | 1 |

S-3 (input - 2)

Topological Sort(2), visited(2) = TRUE

↓

Topological Sort(3), visited(3) = TRUE

↓

"1" is already visited, so no more recursive call.

Stack | 0 | 1 | 3 | 2 |

S-4 (input - 3)

3 is already visited, No more recursive call

Stack | 0 | 1 | 3 | 2 |

S-5 (input - 4)

Topological Sort(4), visited(4) = TRUE

↓

"0" & "1" are already visited. No more recursive call

Stack | 0 | 1 | 3 | 2 | 4 |

S-6 (input-5)

TOPOLOGICAL SORT(5), visited(5) = TRUE

"0" & "2" are already visited, No more recursive call.

Stack

0	1	3	2	4	5

S-7

Print the visited in LIFO order.

Array

5	4	2	3	1	0

Dt - 16.11.2022

NOTE :-

→ DFS is used in TOPOLOGICAL Sorting / TOPOLOGICAL ordering (It is the application of "DFS")

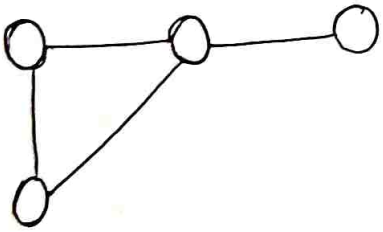
→ "Stack" linear data structure is used while finding the TOPOLOGICAL Sorting / ordering.

→ Time complexity : O (Vertex + edges) → (AL / AA)

SPANNING TREE :

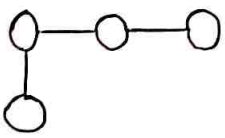
Given a connected graph, a spanning tree of that graph is a sub graph that is a tree and joined all vertices. A single graph can have many spanning trees.

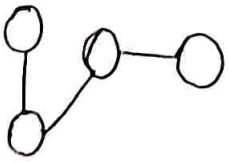
For example :

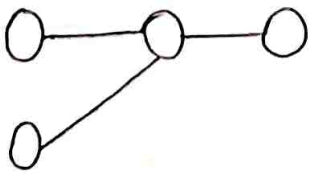


Form the graph. There can be multiple spanning trees like.

Spanning tree







PROPORTIES OF SPANNING TREE :

1. There may be several minimum spanning trees of the same weight having the minimum number of edges

2. If all the weights of a given graph are the same, then every spanning tree of that graph is minimum.

3. If each edge has a distinct weight, then there will be only one, unique minimum spanning tree.

4. A connected graph G can have more than one spanning trees.

5. A disconnected graph can't have to span the tree, or it can't span all the vertices.

6. Spanning tree don't contain cycles.

7. Spanning tree has $(n-1)$ edges where n is the number of vertices.

Addition of even one single edge results in the spanning tree loosing its property of Acyclicity and elimination of one single edge results in it's losing the property of connectivity.

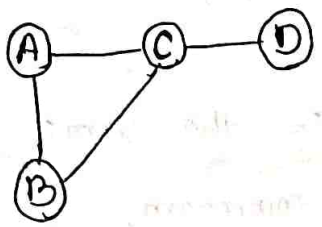
NOTE :-

→ Maximum number of spanning Trees (STs) possible in a complete undirected graph is $n(n-2)$ / $v(v-2)$

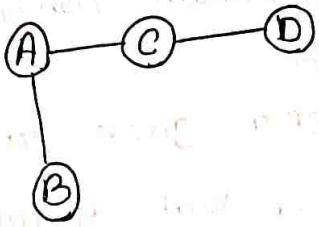
Where, n/v stands for vertices.

2) Spanning Tree is subset of a Graph.

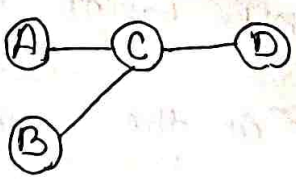
Example



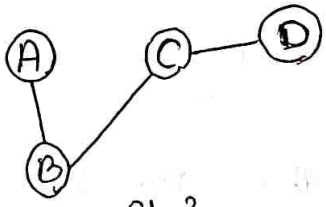
= Graph =



St-1



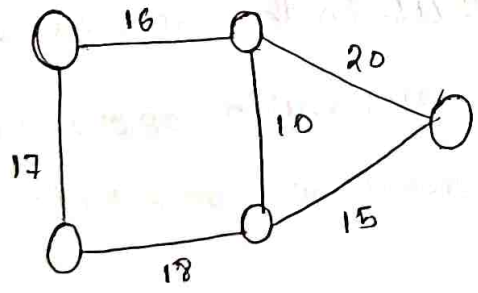
St-2



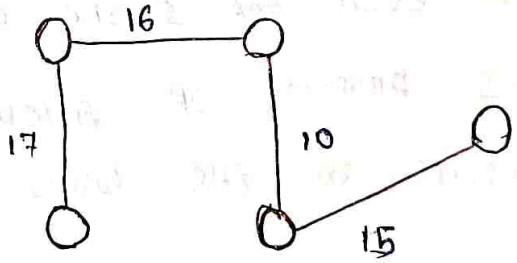
St-3

MINIMUM SPANNING TREE (MST)

Minimum Spanning Tree is a Spanning Tree which has minim total cost. If we have a linked undirected graph with a weigh (or cost) combine with each edge. Then the cost of Spanning th would be the sum of the cost of it's edge.



[Connected, undirected Graph]



[Minimum cost Spanning Tree
Total cost = 17+16+10+15 =

MINIMUM SPANNING TREE (MST)

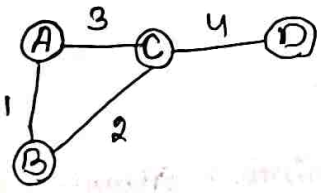
NOTE

-> If more than one minimum spanning trees (MSTs) are generated, than the cost of all MST's will be same.

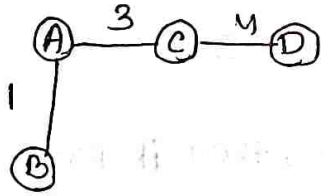
> There must be atleast one MST possible in a Directed weighted Graph (DWG) / Undirected weighted graph (UWG)

> We can find MST IFF the graph is Directed / undirected weighted Graph.

Example

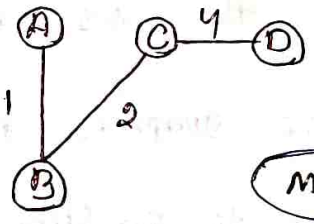


[Graph]



[ST-1]

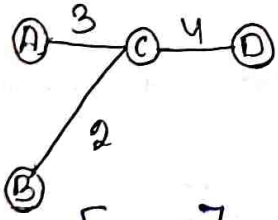
Cost - 1+3+4 = (8)



MST

[ST-3]

Cost - 1+2+4 = (7)



[ST-2]

Cost - 3+2+4 = (9)

There are two methods / Algorithms exist to Find Minimum Spanning Tree (MST):

a) Kruskal's Algorithm.

b) Prim's Algorithm.

KRUSKAL'S ALGORITHM

An algorithm to construct a minimum spanning tree for a connected weighted graph. It is a Greedy Algorithm. The Greedy choice is to put the smallest weighted edge that does not beause a cycle in the MST constructed so far.

If the graph is not linked, then it finds a minimum spanning Tree.

Steps to Finding MST using Kruskal's Algorithm:

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

MST-KRUSKAL (G, W)

1. A = Ø
2. For each vertex V ∈ G.V
3. Make-SET (V)
4. Sort of edges of G.E into non decreasing order by weight w.

5. For each edge $(u,v) \in G.E$, taken in nondecreasing order by weight.

6. if FIND-SET(u) ≠ FIND-SET(v)

7. A = A ∪ {(u,v)}

8. UNION (u,v)

9. ~~UNION (A,V)~~ return A

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

- E is at most V^2 and $\log V^2 = 2 \times \log V$ is $O(\log V)$
- If we ignore isolated vertices, which will each their components of the minimum spanning tree, $V \leq 2E$, so $\log V$ is $O(\log E)$

Thus, the total time is,

1. $O(E \log E) = O(E \log V)$

Solution Steps :

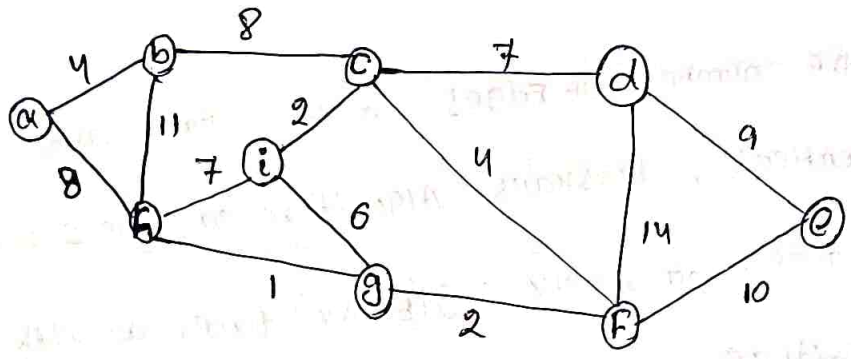
1) Keep all the vertices excluding the edges
2) Start selecting the edge in increasing order (i.e, selected, the edge whose cost / weight is minimum)

3) If the newly selected edge creats cycle in the MST, then we have to discard that edge and start selecting the next minimum edges

4) Stop, if all vertices have successfully covered / connected

EXAMPLE:

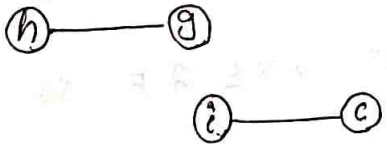
Find the MST from the given graph using Kruskal's Algorithm



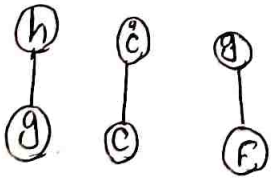
Increasing order: 1, 2, 2, 4, 4, 6, 7, 7, 8, 8, 9, 10, 11, 14
(× × × × × × ×)

Step-1

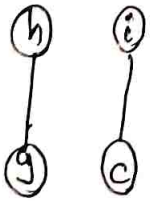
Step-2



Step-3



Step-4



⇒ Prim's Algorithm

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

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

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

Steps for finding MST using Prim's Algorithm

1→ Create MST set that keeps track of vertices already included in MST.

2→ Assign key values to all vertices in the input graph. Initialize all key values as INFINITE (∞). Assign key values like 0 for the first vertex so that it is picked first.

3→ While MST set doesn't include all vertices.

(i) Pick vertex u which is not is MST set and has minimum key value. Include 'u' to MST set.

(ii) Update the key value of all adjacent vertices of u. To update, iterate through all adjacent vertices. For every adjacent vertex v, if the weight of edge u,v less than the previous key value of v, update key value as a weight of u,v.

```
MST-PRIM (G, w, r)
1. for each u ∈ V[G]
2. do key[u] ← ∞
3. π[u] ← NIL
```

4→ key [r] ← 0
5→ Q ← V [G]
6→ While Q ! Φ
7→ ~~while~~ do u ← EXTRACT ← MIN (Q)
8. for each v ∈ Adj [u]
9. do if v ∈ Q and w (u,v) < key [v]
10. then π [v] ← u
11. key [v] ← w (u,v)

MST- PRIM (G, w, r)

1. for each u ∈ G.V
2. u. key = ∞
3. u. π = NIL
4. r. key = 0
5. Q = G.V
6. while Q ≠ Φ
7. u = EXTRACT -MIN (Q)
8. for each v ∈ G. Adj [u]
9. if v ∈ Q and w (u,v) < v. key
10. v. π = u
11. V. key = w (u,v)

Solution-Steps.

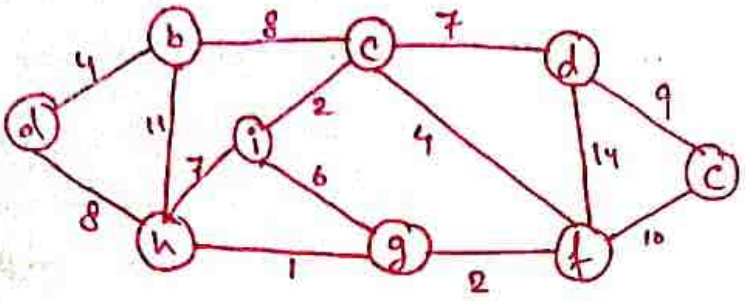
1. Keep the vertices without considering the edges.
2. Start selecting the edge where weight / cost is Minimum or any vertex.
3. Select the next minimum edge which is connected with the previously selected edges vertices / vertex.
4. Stop selecting the new edge if it makes any cycle in the MST and select the next minimum edge is connected

with the previously explored vertices

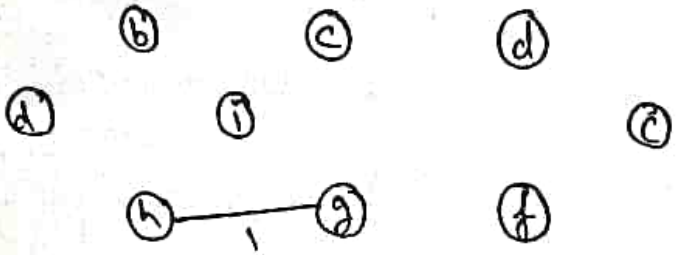
5. Stop, if all vertices have successfully connected

Example [CORMEN]

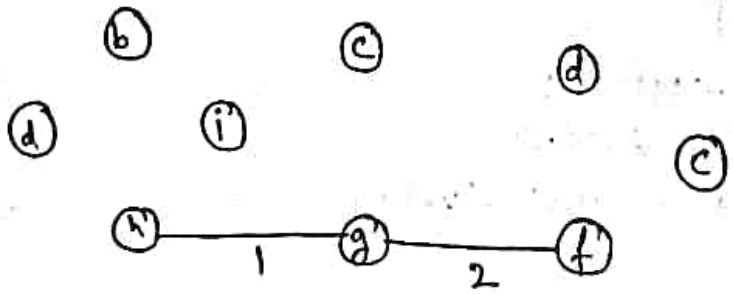
Find the MST from the given graph using Prim's Algorithm



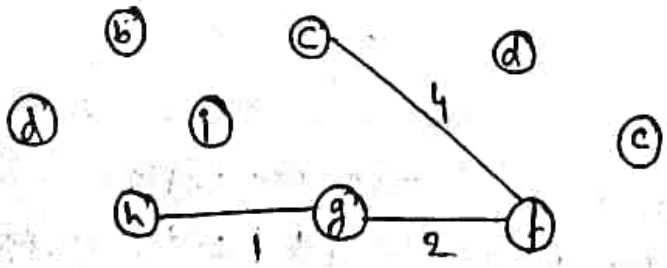
Step 1



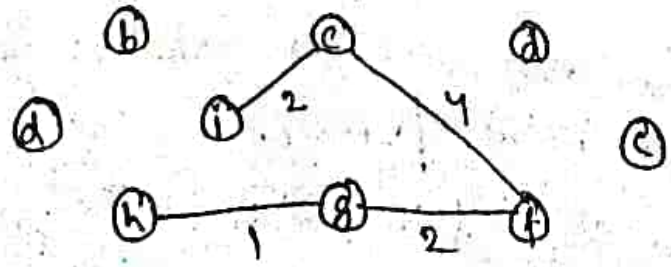
Step 2



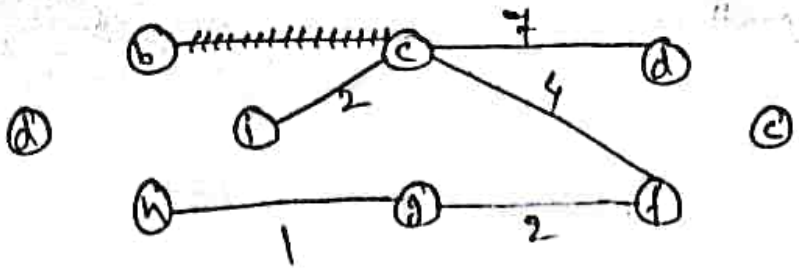
Step 3



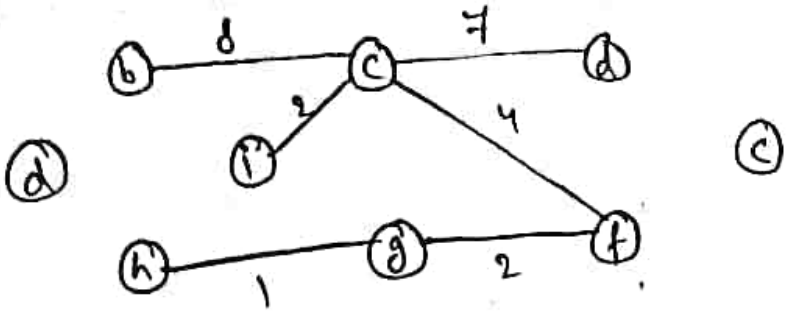
Step 4



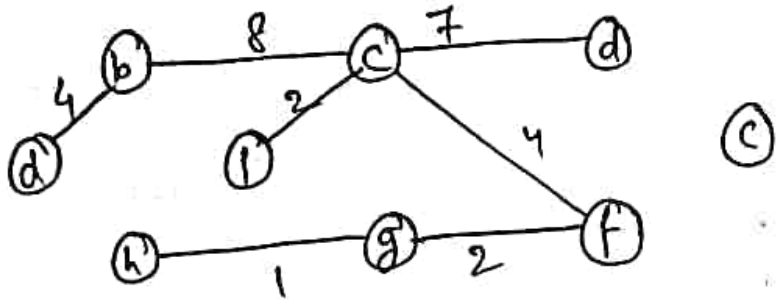
Step 5



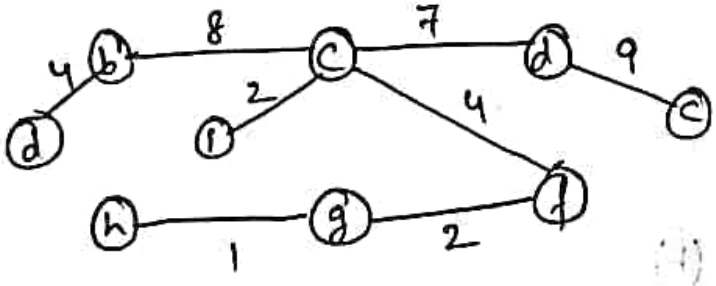
Step 6



Step 7



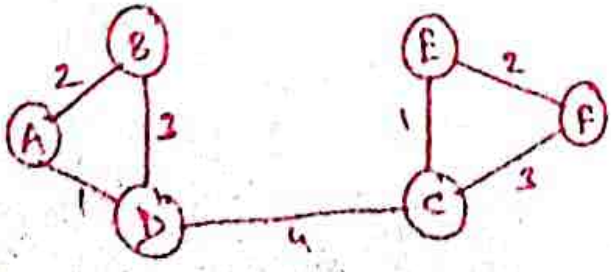
Step 8



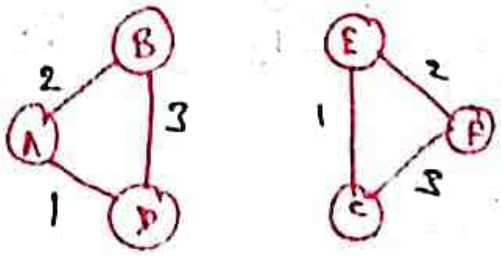
Total cost = 4+8+7+9+2+4+1+2 = 37

Q Find the MST of the given graph using Prim's algorithm and Kruskal's algorithm.

(a)



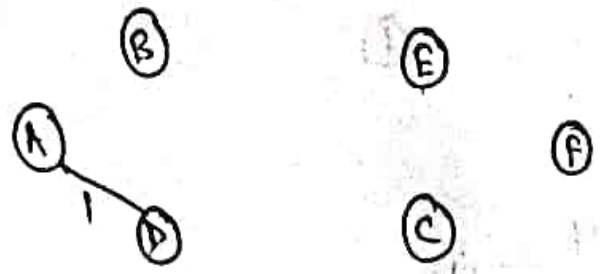
(b)



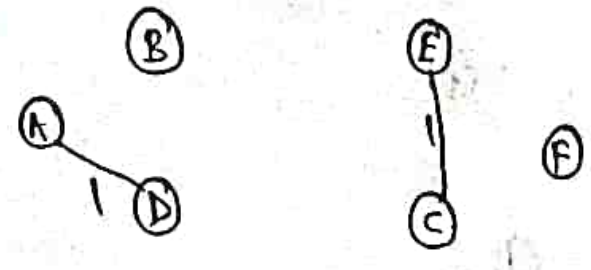
Kruskal's Algorithm

(a)

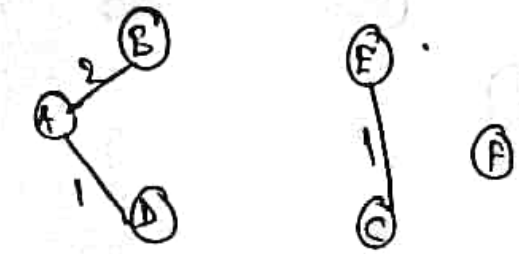
Step 1



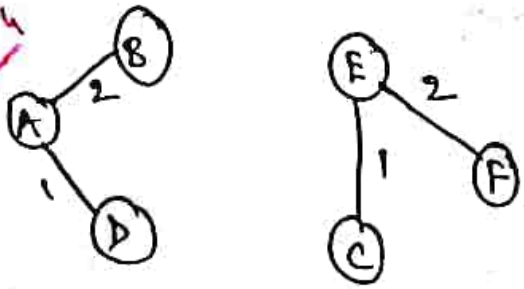
Step 2



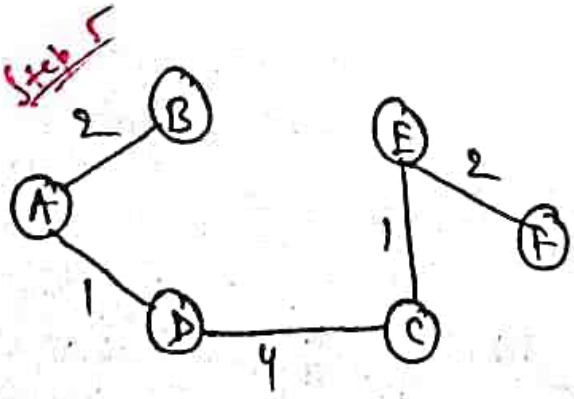
Step 3



Step 4



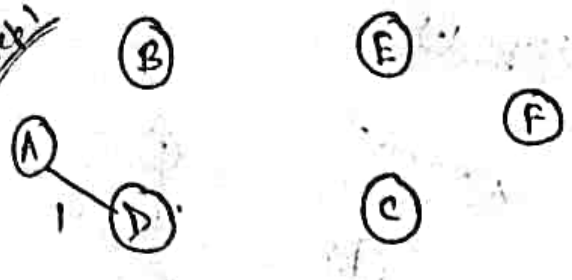
Step 5



Total cost = 1+1+2+2+4
= 10

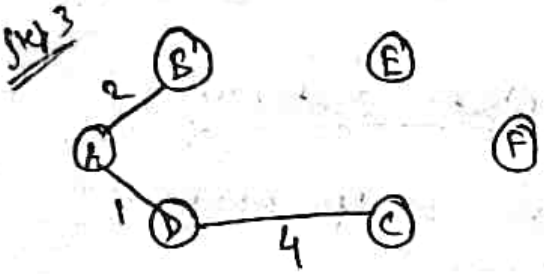
Prim's Algorithm (a)

Step 1

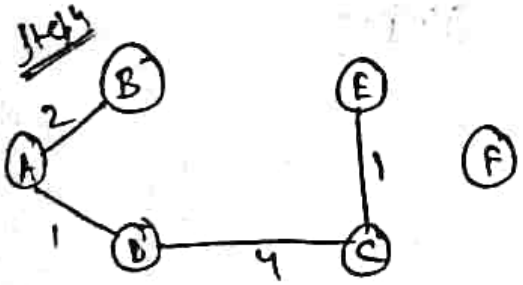


Step 2

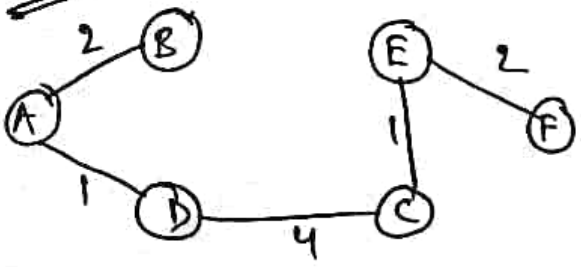
Step 3



Step 4

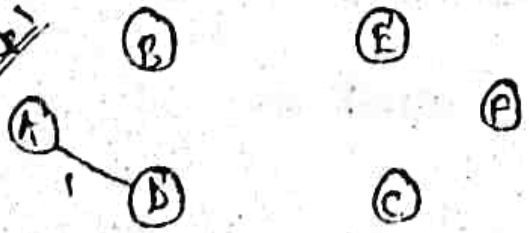


Step 5

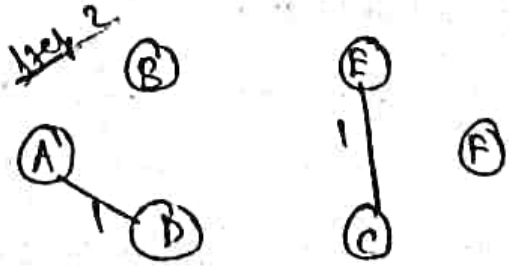


(b) Kruskal's Algorithm

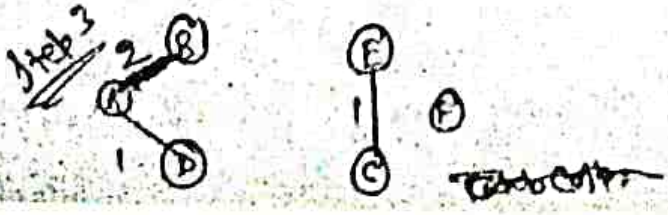
Step 1



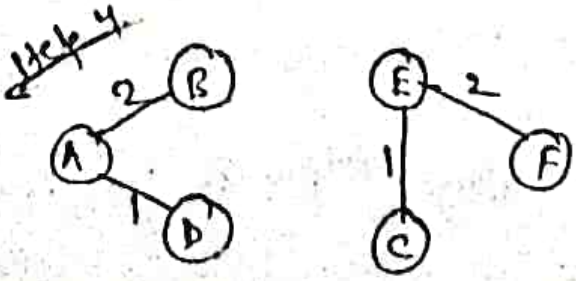
Step 2



Step 3



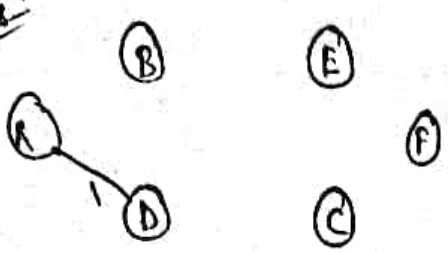
Step 4



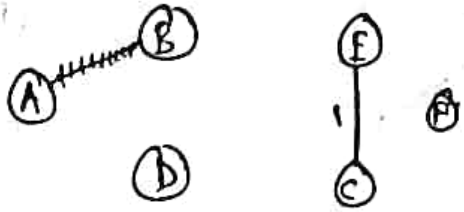
Total cost = 1+2+1+2 = 6

(b) Prim's Algorithm

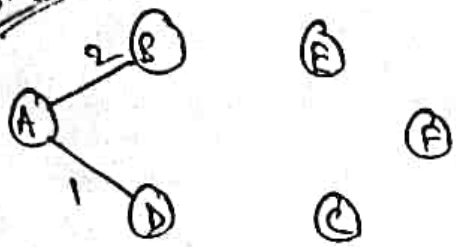
Step 1:

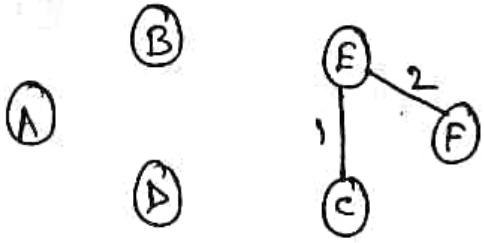


Step 1.1:



Step 2





Cont continue

Cont continue

So Prim's Algorithm won't work on disconnected graph.

Note:

→ Prim's Algorithm will not work if the graph is Disconnected graph

→ Prim's Algorithm can start from a minimum edge or any vertex, but Kruskal's Algorithm will work only on edges.

→ Time complexity: Using Adjacency Matrix : $O(V^2)$.

Using Adjacency list and binary heap: $O($

last = v_{next}

known = known $\cup$ $\{v_{next}\}$

Analysis: The running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of |E| and |V| using the Big-O notation. The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and operation Extract-Min(Q) is simply a linear search through all vertices in Q. In this case, the running time is $O(|V|^2 + |E|) = O(V^2)$. But with Min-Priority Queue it drops down to $O(V + E \log V)$.

9

Solution Steps:

1) Select any vertex as the source vertex.
2) Put the value / distance of the selected source vertex as "0" and the rest of the vertices as "∞".
3) Start "Relaxing" the vertices which are connected with the previously selected vertex.
4) Stop if all vertices have successfully ~~visited~~ "Relaxed".

Relax:

if $(d(v) > d(u) + c(u,v))$
then $d(v) = d(u) + c(u,v)$

where-
- d(u) is the value/distance of u (sender/source)
- d(v) is the distance value of v (Receiver/Destination)
- c(u,v) is the cost/weight of the edge connected the vertices 'u' to 'v'.

DIJKSTRA (G, w, s)

1. INITIALIZE-SINGLE-SOURCE (G, s)
2. S = Φ
3. Q = G.V
4. while Q ≠ Φ
5. u = EXTRACT-MIN (Q)
6. S = S ∪ {u}
7. for each vertex v ∈ G. Adj [u]
8. RELAX (u, v, w)

Dijkstra's Algorithm (G, l)

Let S be the set of explored nodes
for each u ∈ S, we store a distance d(u)
Initially S = {s} and d(s) = 0
while S ≠ V
Select a node v ∉ S with at least one edge from S for which d'(v) = min$_e$ = (u, v) : u ∈ s d(u) + le is as small as possible.
Add v to S and define d(v) = d'(v)
EndWhile

Shortest Path - Dijkstra (G, s, t)

known = {s}
for i = 1 to n, dist [i] = ∞
for each edge (s, v), dist [v] = w (s, v)
last = s
while (last ≠ t)
select v_{next} the unknown vertex minimizing dist [v]
for each edge (v_{next}, x), dist [x] = min [dist [x], dist [v_{next}] + w (v_{next}, x)]

Application (Kruskal's Algorithm)

- In order to layout electrical wires
- In computer network (LAN connection)

Dijkstra's Algorithm:

→ It is a greedy algorithm that solves the single-source shortest path problem for a directed graph $G = (V, E)$ with nonnegative edge weights, i.e., $w(u,v) \geq 0$ each edge $(u,v) \in E$

Dijkstra's Algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. That's for all vertices, $v \in S$; we have $d[v] = \delta(s,v)$. The algorithm repeatedly selects the vertex $u \in V-S$ with the minimum shortest-path estimate, insert u into S and relaxes all edges leaving u.

Because it always chooses the "lightest" or "closest" vertex in $V-S$ to insert into set S, it is called as the Greedy strategy.

INITIALIZE-SINGLE-SOURCE (G,s)

1. for each vertex $v \in G.V$
2. $v.d = \infty$
3. $v.\pi = NIL$
4. $s.d = 0$

RELAX (u,v,w)

1. if $v.d > u.d + w(u,v)$
2. $v.d = u.d + w(u,v)$
3. $v.\pi = u$

Step-3

Considering A as source and B as destination

$\infty < \infty + 1$

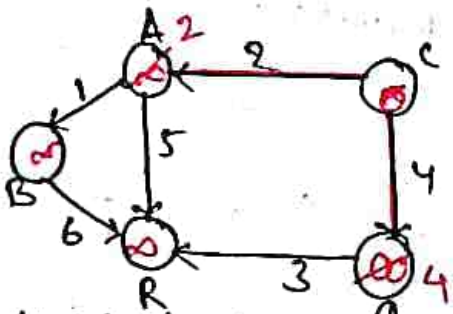
Hence B cannot be relaxed

Considering A as source and R as destination

∞

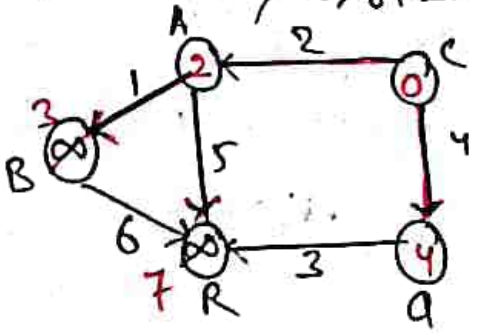
Q Source = C

Step 1 →

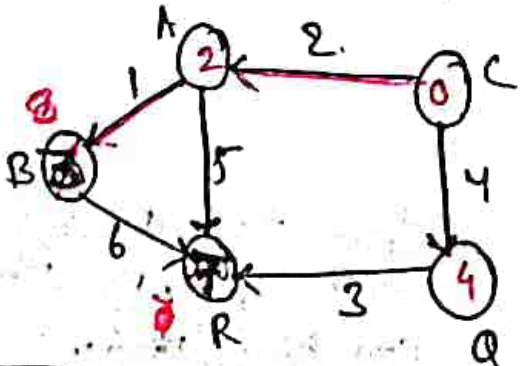


$d(v) > d(u) + c(u,v) \Rightarrow \infty > 0+2 = 2$

Step 2 →



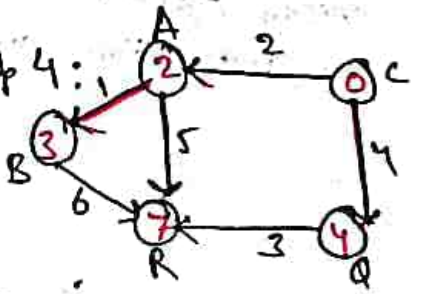
Step 3 →



Taking source as R

No other vertices can be relaxed

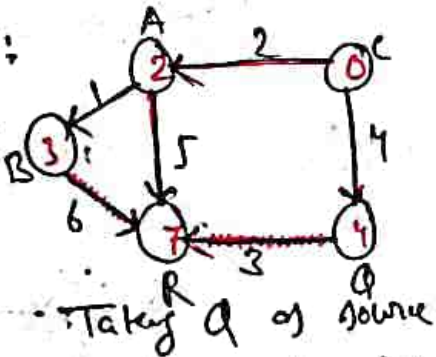
Step 4:



Taking source as B

No other ~~source~~ vertices can be relaxed

Step 5:



Taking Q as source

No other vertices can be relaxed

Visited	Vertices s	x	y	z
	0	∞	∞	∞
s	0	10	∞	5 ∞
s, y	0	8	14	5 7
s, y, z	0	8	13	5 7
s, y, z, t	0	8	9	5 7

CO-4

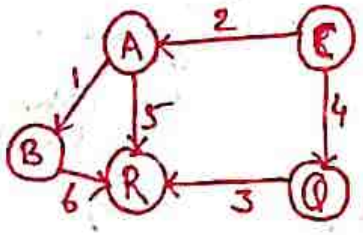
Greedy Algorithm

Greedy Algorithm is an approach for solving a problem by selecting the best option avilable at the moment.

Q Find the shortest path from 'Q' to other vertices using Dijkstra's Algorithm

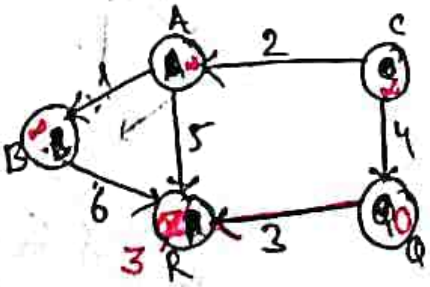
Relax: if $(d(v) > d(u) + c(u,v)$ then $d(v) = d(u) + c(u,v)$

$d(v)$ = value/distance of destination/receiver
$d(u)$ = " source/sender
$c(u,v)$ = cost/weight of edge.

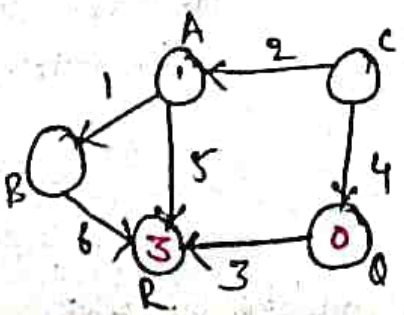


~~Step 3~~

Step 1



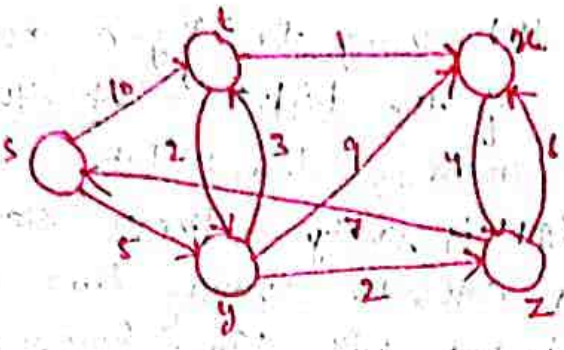
Step 2



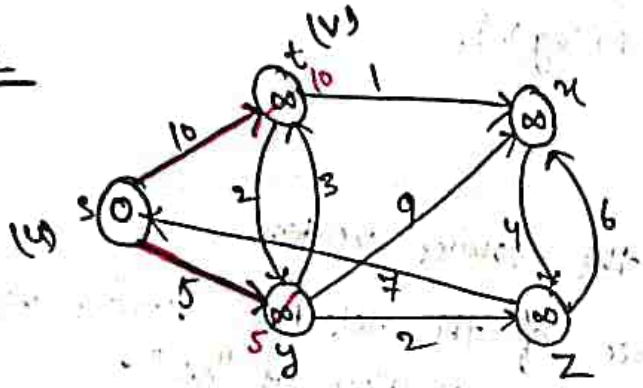
∴ Since the outdegree is zero for source R, no other vertices can be relaxed.

Example [CORMEN]

Q → Find the shortest path from "s" to other vertices using Dijkstra's Algorithm.



Step 1



$d(v) > d(u) + c(u,v)$

$\Rightarrow \infty \quad 0 + 10$

$\Rightarrow \infty > 10$

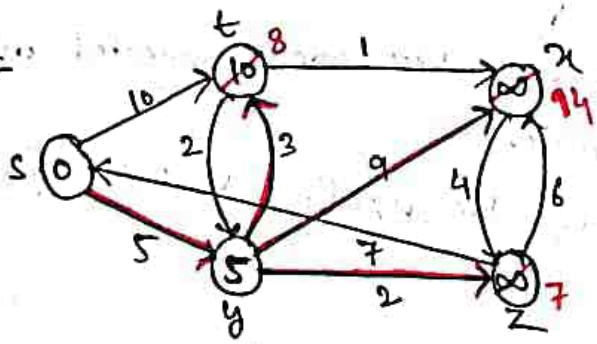
$d(v) = d(u) + c(u,v)$

$= 0 + 10$

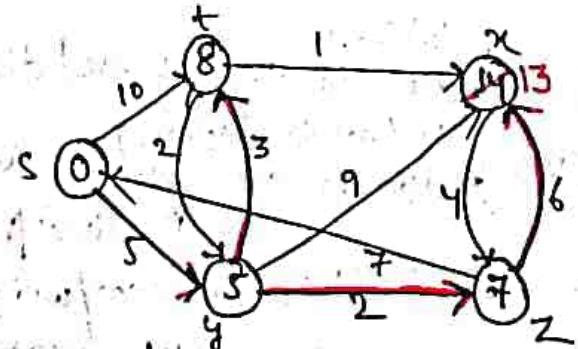
$d(v) = 10$ Similarly when s is u and y is v

$d(v) = 5$

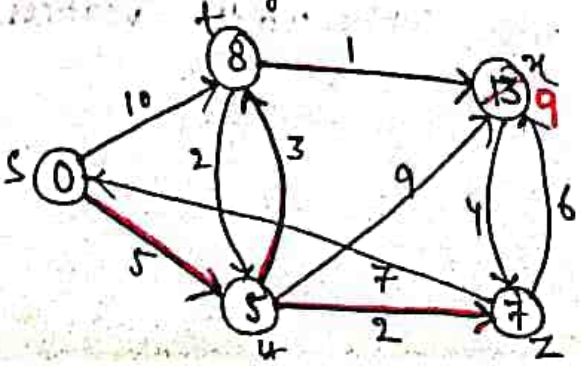
Step-2



Step 3



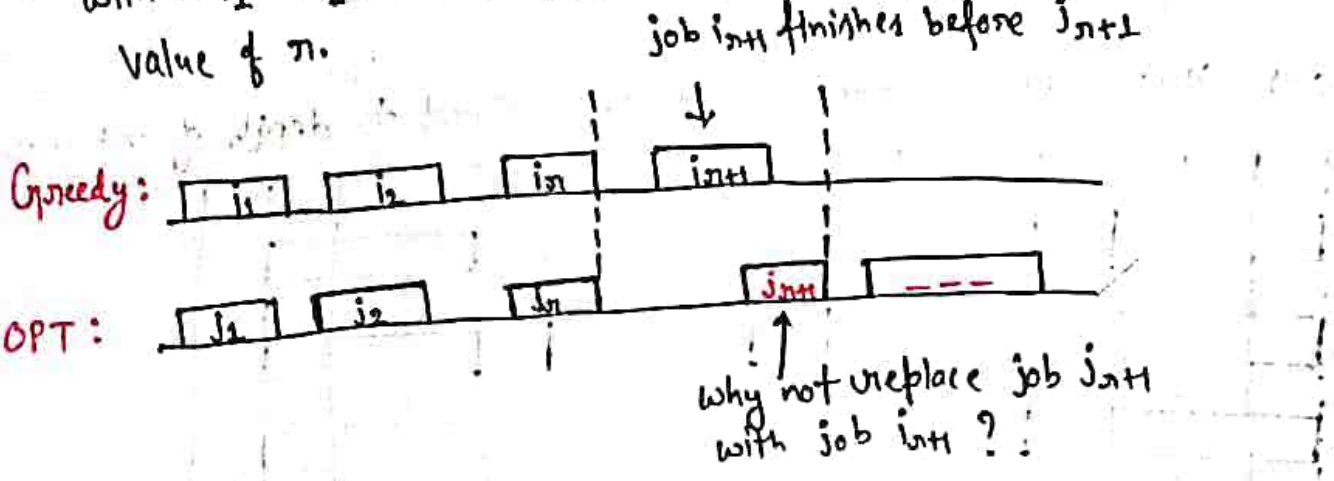
Step 4



Theorem: Greedy algorithm is optimal.

Pf. (by contradiction)

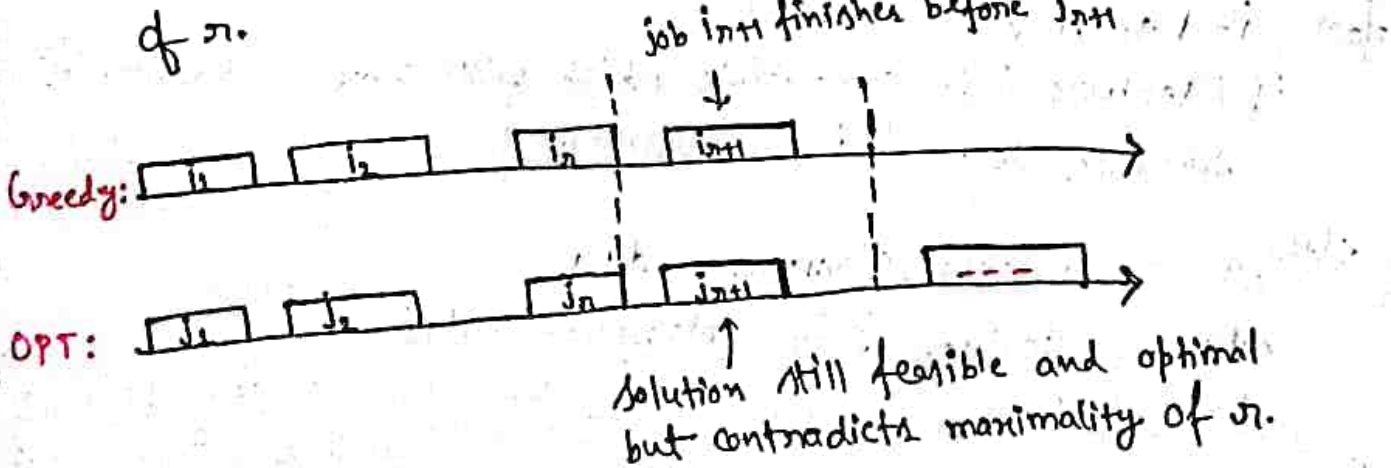
- Assume greedy is not optimal, and let's see ~~that~~ what happens.
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots, j_m$ denote set of jobs in the optimal solution with $i_1 = j_1$, & $i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.



Theorem: Greedy algorithm is optimal.

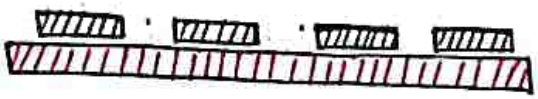
Pf. (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution with $i_1 = j_1$, $i_2 = j_2, \ldots i_r = j_r$ for the largest possible value of r.



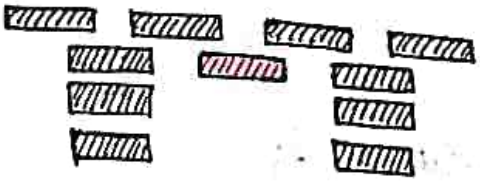
- [Earliest start time] Consider jobs in ascending order of s_j.
- [Earliest finish time] Consider jobs in ascending order of f_j.
- [Shortest interval] Consider jobs in ascending order of $f_j - s_j$.
- [Fewest conflicts] for each job j, count the number of conflicting jobs c_j. Schedule in ascending order of c_j.

> Greedy template: Consider jobs in some natural order. Take each job provided it's compatible with the ones already taken.



Counterexample for earliest start time

Counterexample for shortest interval



Counterexample for fewest conflic

> Greedy Algorithm: Consider jobs in increasing order of finish times. Take each job provided it's compatible with the ones already taken.

Sort jobs by finish times so that $f_1 \le f_2 \le \dots \le f_n$.

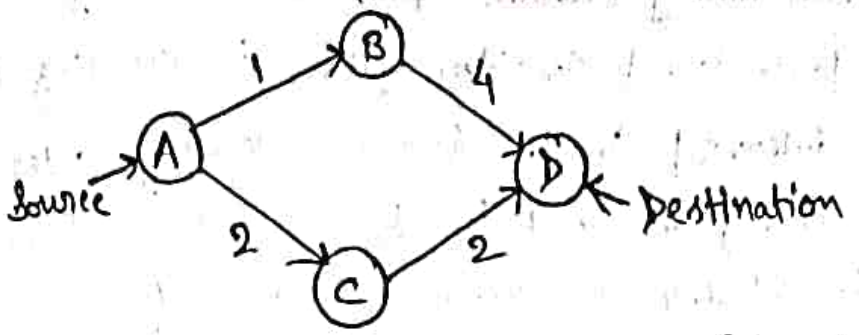
```
A ← φ        → Set of jobs selected.
for j = 1 to n {
    if (job j compatible with A)
        A ← A ∪ {j}
}
return A
```

Implementation $O(n \log n)$.

- Remember job j^* that was added last to A.
- Job j is compatible with A if $s_j \ge f_{j^*}$.

CO-4

Example:



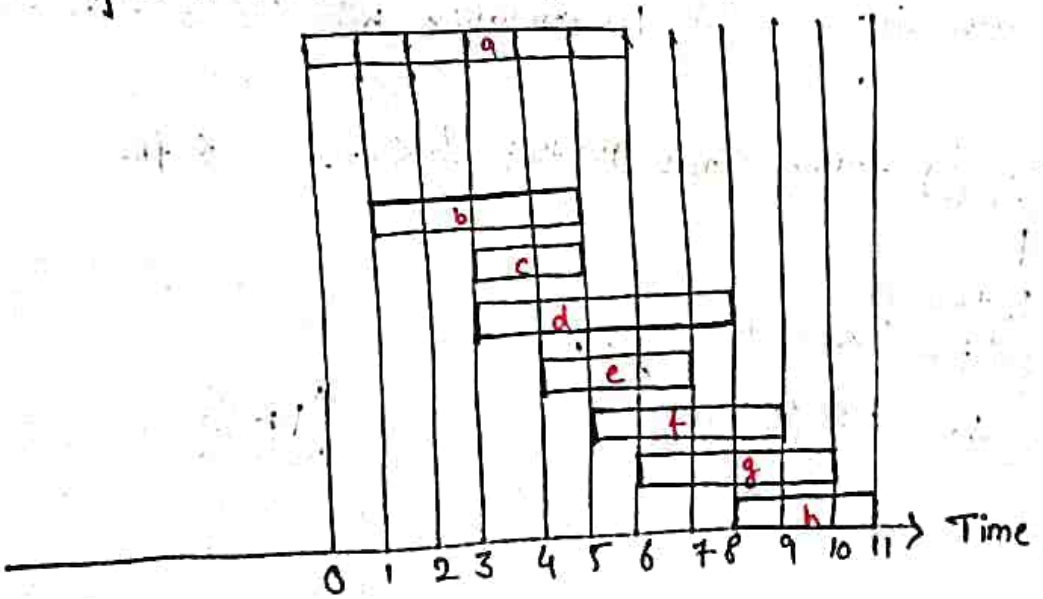
$A \to D$: $\frac{A-B}{1} + \frac{B-D}{4} = 5$ Ans

Note:-

1) No Extra memory is used to find the optimal solution using Greedy Algorithm
2) We may or may not get the optimal solution using Greedy Algorithm.

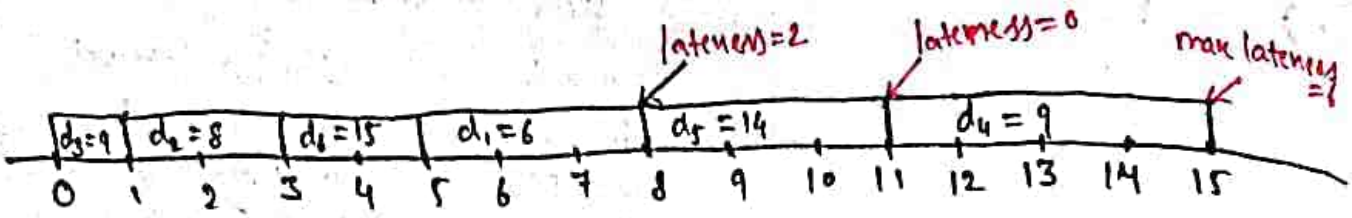
⇒ Interval scheduling

- Job j starts at s_j and finishes at f_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs



⇒ Interval scheduling: Greedy Algorithms:

Greedy template. Consider jobs in some natural order. Take each job provided it's compatible with the ones already taken.



Greedy template: Consider jobs in some order

- [Shortest processing time first] Consider jobs in ascending order of processing time t_j.
- [Earliest deadline first] Consider jobs in ascending order of deadline d_j.
- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

Greedy template: Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time t_j.

	1	2
t_j	1	10
d_j	100	10

Counter example.

- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

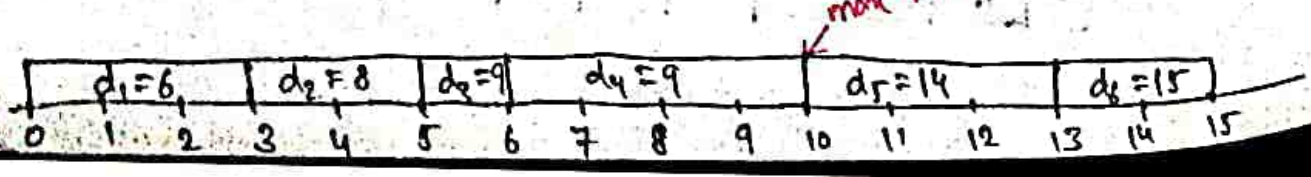
	1	2
t_j	1	10
d_j	2	10

Counter example

Greedy Algorithm: Earliest deadline first.

Sort n jobs by deadline so that $d_1 \le d_2 \le \cdots \le d_n$

```
t ← 0
for j = 1 to n
    Assign job j to interval [t, t + t_j]
    s_j ← t, f_j ← t + t_j
    t ← t + t_j
Output intervals [s_j, f_j]
```



Earliest starting time
EST

a → 0
b → 1
c → 3
d → 3
e → 4
f → 5
g → 6
h → 8

Earliest finish time
EFT

a → 6
b → 4
c → 5
d → 8
e → 7
f → 9
g → 10
h → 11

Shortest interval

a → 6-0 = 6
b → 4-1 = 3
c → 5-3 = 2
d → 8-3 = 5
e → 7-4 = 3
f → 9-5 = 4
g → 10-6 = 4
h → 11-8 = 3

$f_i \leq s_j$

For EST → {a, g}
EFT → {b, e, h}
SI → {c,
FC → {h, b, e}

Fewest conflict

a → 5 (b, c, d, e, f)
b → 3 (a, c, d)
c → 4 (a, b, d, e)
d → 6 (a, b, c, e, f, g)
e → 5 (a, c, d, f, g)
f → 5 (a, d, e, g, h)
g → 4 (d, e, f, h)
h → 2 (f, g)

Scheduling to Minimize Lateness

Minimizing lateness problem.

- Single resource processes one job at a time.
- Job j requires t_j units of processing time and is due at time d_j.
- If j starts at time s_j, it finishes at time $f_j = s_j + t_j$
- Lateness: $l_j = \max\{0, f_j - d_j\}$
- Goal: Schedule all jobs to minimize maximum lateness $L = \max l_j$.

Eg

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15

Interval Partitioning: Lower Bound on Optimal Solution.

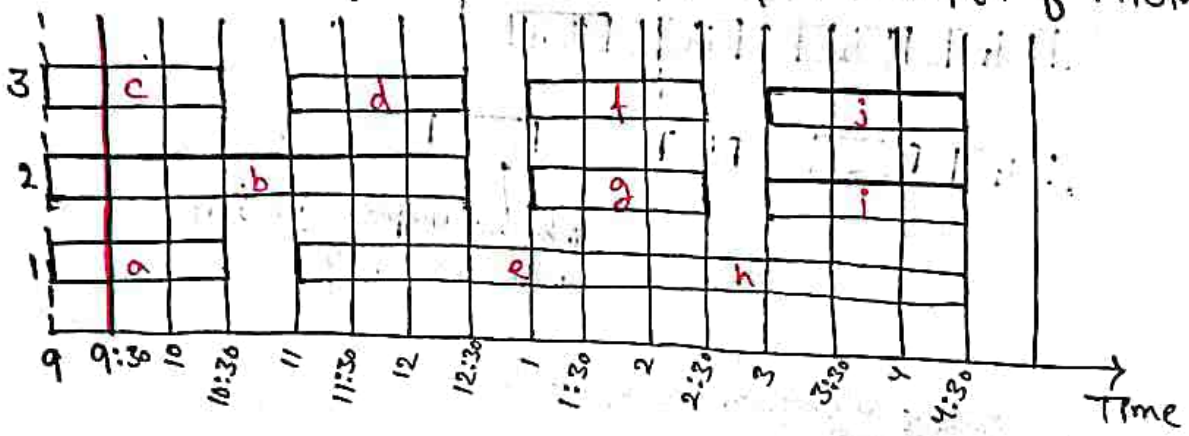
Def. The depth of a set of open intervals is the maximum number that contain any given time.

Key observation: Number of classrooms needed ≥ depth.

Ex. Depth of schedule below = 3 ⇒ schedule below is optimal.

a, b, c all contain 9:30

Q. Does there always exist a schedule equal to depth of intervals?



> Greedy algorithm:- Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

Sort intervals by starting time so that $s_1 \le s_2 \le \cdots \le s_n$.

d ← 0 ← number of allocated classrooms

for j = 1 to n {

  if (lecture j is compatible ~~which~~ with some classroom k)

    schedule lecture j in classroom k.

  else

    allocate a new classroom d+1

    schedule lecture j in classroom d+1

    d ← d+1

}

Implementation $O(n \log n)$.

- For each classroom k, maintain the finish time of the last job added.
- Keep the classrooms in a priority queue.

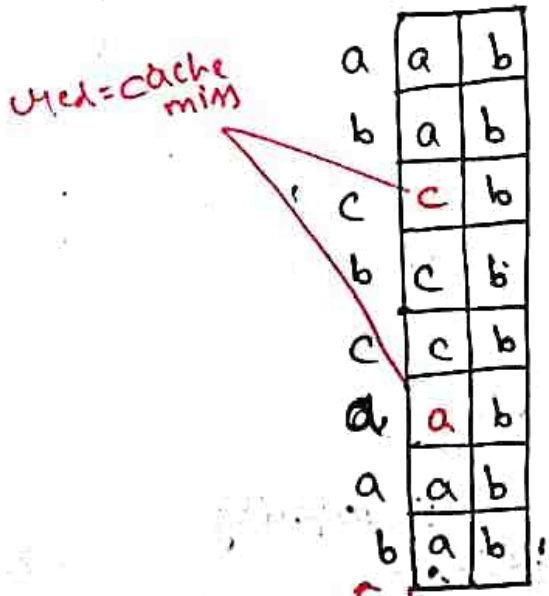
Caching.

- Cache with capacity to store k items.
- Sequence of m item requests $d_1, d_2, \ldots, d_m$.
- Cache hit: item already in cache when requested.
- Cache miss: item not already in cache when requested: must bring requested item into cache, and evict some existing item, if full.

Goal. Eviction schedule that minimizes number of cache misses.

Ex: k=2, initial cache = ab,
requests: a, b, c, b, c, a, a, b.

Optimal eviction Schedule: 2 cache misses



→ Farthest-in-future: Evict item in the cache that is not requested until farthest in the future.

Current cache | a | b | c | d | e | f |

future queries g a b c e d a bb a c d e a f a d e f g h...

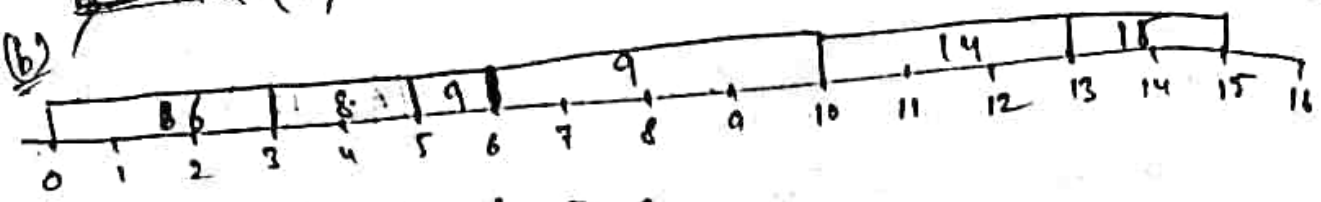
↑ Cache miss

↑ eject this one

Theorem: [Belladay, 1960s] FF is optimal eviction schedule.

If: Algorithm and theorem are intuitive; proof is subtle

(b)



$\max\{0, 3-6\} = 0$

$\max\{0, 5-8\} = 0$

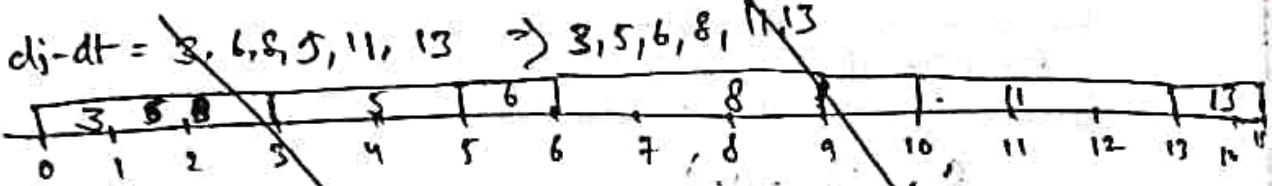
$\max\{0, 6-9\} = 0$

$\max\{0, 10-9\} = 0$

$\max\{0, 13-14\} = 0$

$\max\{0, 15-15\} = 0$

(c) $d_j - d_t$ = 3, 6, 8, 5, 11, 13 → 3, 5, 6, 8, 11, 13



$\max(0, 0) = 0$

$\max(0, 0) = 0$

$\max(0, 0) = 0$

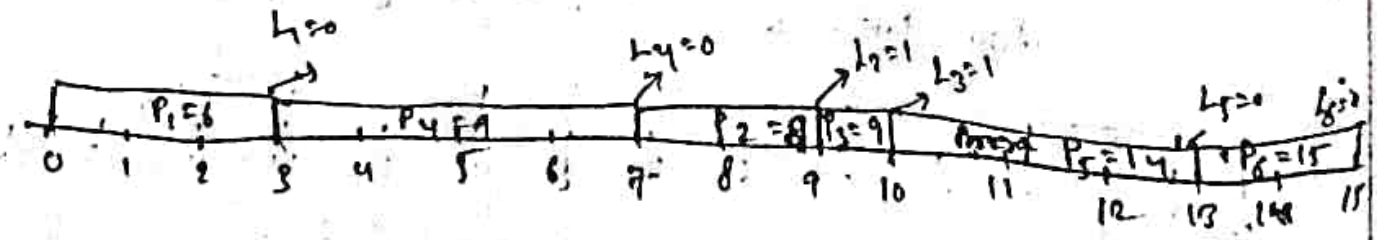
$\max(0, 2) = 2$

$\max(0, 2) = 2$

$\max(0, 2) = 2$

(e). $d_j - d_t$ = Slack

	P_1	P_2	P_3	P_4	P_5	P_6
P_j	1	2	3	4	5	6
t_i	3	2	1	4	3	2
d_j	6	8	9	9	14	15
Slack	3	6	8	5	11	13



Total lateness = 1+1+2

⇒ There are three methods / approaches used to find the lateness:

a) Shortest Processing Time (t_i) first
b) Earliest Deadline (d_i) first
c) Smallest Slack ($d_i - t_i$) first

a) SPTF (t_i)

Q1 Find the maximum lateness using the approaches

	P_1	P_2	P_3	P_4	P_5	P_6
Process Number (P_i)	1	2	3	4	5	6
Process Time (t_i)	3	2	1	4	3	2
Deadline (d_i)	6	8	9	9	14	15

Sol → $\max\{0, (f_3 - d_3)\}$
$= \max\{0, (1-9)\} = \max\{0, -8\} = 0$

→ $\max\{0, (f_2 - d_2)\} = \max\{0, 3-8\}$
$= \max\{0, -5\} = 0$

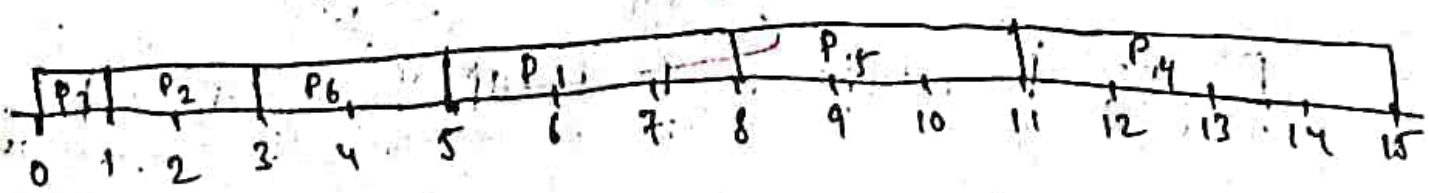
→ $\max\{0, (f_6 - d_6)\} = \max\{0, 5-15\}$
$= \max\{0, -10\} = 0$

→ $\max\{0, f_1 - d_1\} = \max\{0, 8-6\}$
$= 2$

→ $\max\{0, f_5 - d_5\} = \max\{0, 11-14\}$
$= 0$

→ $\max\{0, f_4 - d_4\} = \max\{0, 15-9\}$
$= 6$

Max late = 6 Total lateness = 6+2 = 8



Single-link k-clustering algorithm

- Form a graph on the vertex set U, corresponding to n clusters.
- Find the closest pair of objects such that each objects is in a different cluster, and add an edge between them.
- Repeat n-k times until there are exactly k clusters.

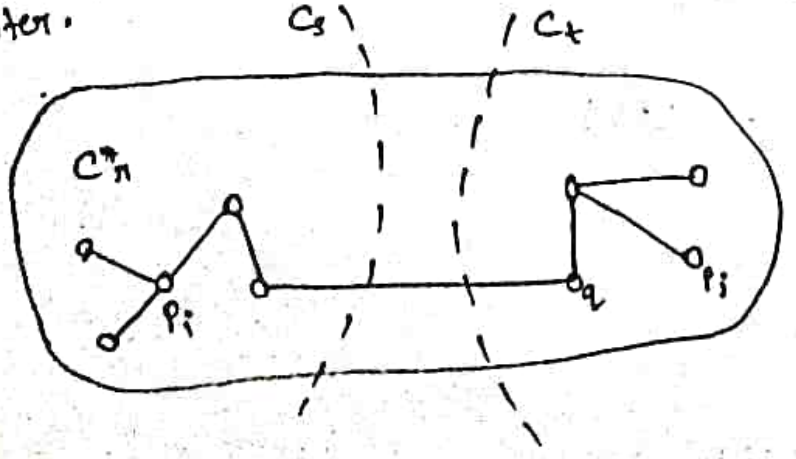
Key observation. This procedure is precisely Kruskal's algorithm (except we stop when there are k connected components).

Remark. Equivalent to finding an MST and deleting the k-1 most expensive edges.

Theorem. Let C^* denote the clustering $C^*_1, \ldots, C^*_k$ formed by deleting the k-1 most expensive edges of a MST. C^* is a k-clustering of max spacing.

Pf. Let C denote some other clustering $C_1, \ldots, C_k$.

- The spacing of C^* is the length d^* of the $(k-1)^{st}$ most expensive edge.
- Let p_i, p_j be in the same cluster in C^*, say C^*_r, but different clusters in C, say C_s and C_t.
- Some edge (p, q) on $p_i - p_j$ path in C^*_r spans two different clusters in C.
- All edges on $p_i - p_j$ path have length $\le d^*$ since Kruskal chose them.
- Spacing of C is $\le d^*$ since p and q are in different cluster.



Clustering. Given a set U of n objects labeled $p_1, \ldots, p_n$, classify into coherent groups.

↑ photos, documents, micro-organisms

Distance function – Numeric value specifying "closeness" of two objects.

↑ number of corresponding pixels whose intensities differ by some threshold

Fundamental problem – Divide into clusters so that points in different clusters are far apart.

- Routing in mobile ad hoc networks.
- Identify patterns in gene expression.
- Document categorization for web search.
- Similarity searching in medical image database
- ~~of~~ Skycat: cluster 10^9 sky objects into stars, quasars, galaxies.

k-clustering Divide objects into k non-empty groups.

Distance function Assume it satisfies several natural properties.

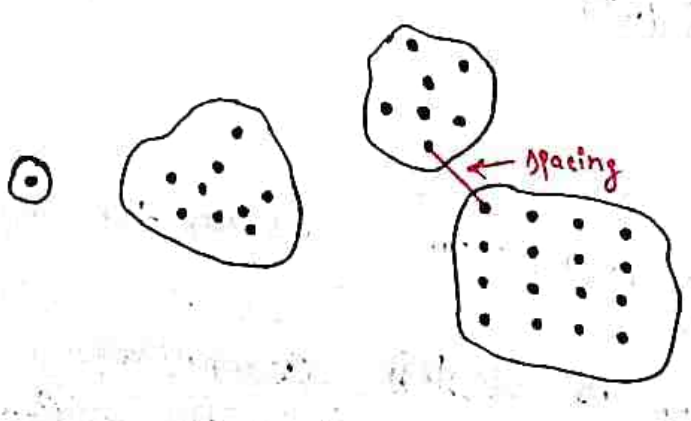
- $d(p_i, p_j) = 0$ iff $p_i = p_j$ (identity of indiscernibles)
- $d(p_i, p_j) \geq 0$ (nonnegativity)
- $d(p_i, p_j) = d(p_j, p_i)$ (symmetry)

Spacing

Min distance between any pair of points in different clusters.

Clustering of maximum spacing

Given an integer k, find a k-clustering of maximum spacing.



Lec-11

(Huffman coding) Algorithm

Time complexity

```
Huffman(S) {
    if |S| = 2 {
        return tree with root and 2 leaves
    } else {
        let y and z be lowest-frequency letters
        in S
        S' = S
        remove y and z from S'
        insert new letter w in S' with fw = fy + fz
        T' = Huffman(S')
        T = add two children y and z to leaf w
        from T'
        return T.
    }
}
```

Time Complexity -

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

So $O(n^2)$

Note:

i) Huffman coding is used to compress the size of the data / message

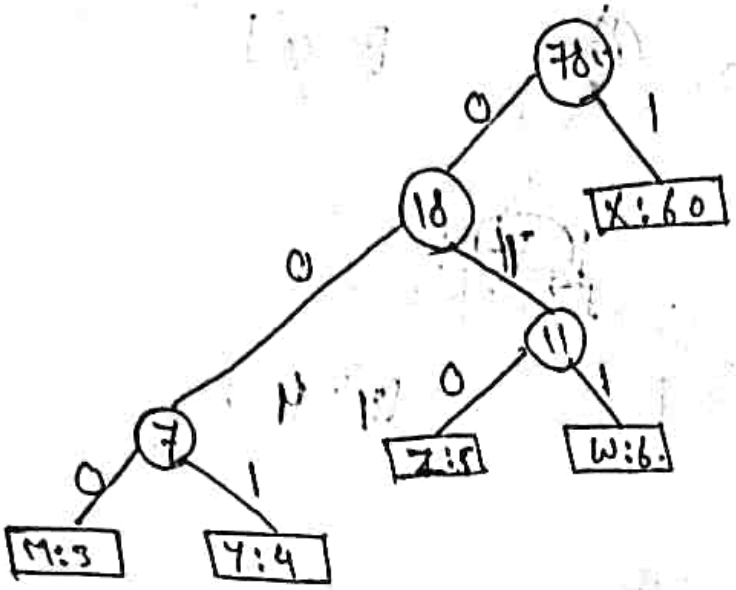
ii) Huffman coding is lossless compression

iii) Recurrence Relation of Huffman Coding is

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

iv) Time complexity of Huffman Coding $\theta(n^2)$

Note:
→ current value
(i) CV ≥ NV (LHS)
(ii) CV < NV (RHS)
↓ New value



Note
LHS – 0
RHS – 1

Character	Code	
W	011	(3 bits)
X	1	(1 bit)
Y	001	(3 bits)
Z	010	(3 bits)
M	000	(3 bits)

Total no of bits without compression

$7 \times 78 = 546$

" " 4 " with comparision

$6 \times 3 + 60 \times 1 + 4 \times 3 + 5 \times 3 + 3 \times 3$

$= 114$ bits

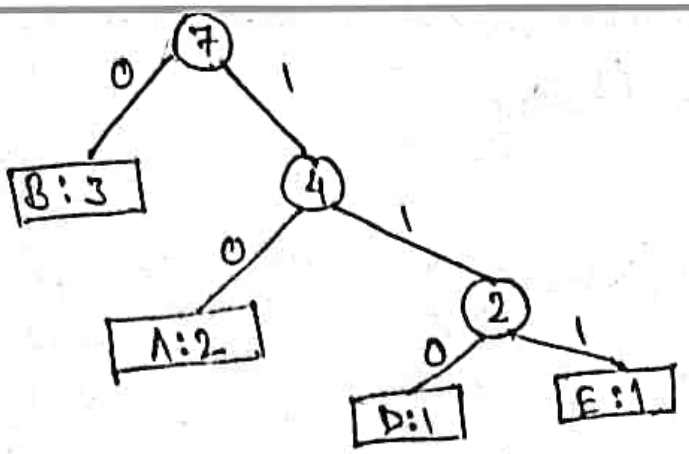
Avg bits.

$\frac{114}{78} = 1.461$ bits/ch.

No of bits saved $= 546 - 114$

$= 432$

A bits saved / ch $= 7 - 1.461 =$

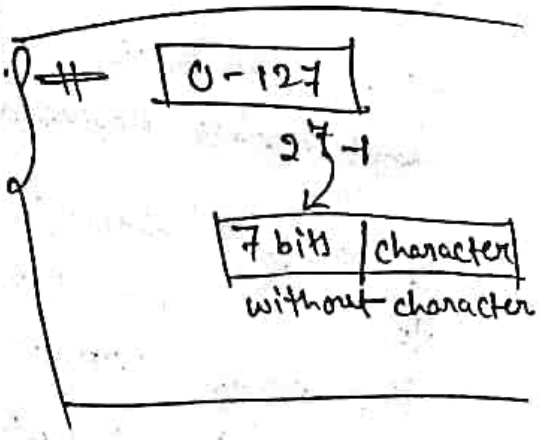


Huffman Tree

Character	Code	
A	10	(2 bits)
B	0	(1 bit)
D	110	(3 bits)
E	111	(3 bits)

Note:—
Left → 0
Right → 1

$7 \times 7 = 49$ bits without compression



Total bits after compression :—

A → 2x2 = 4
B → 3x1 = 3
C → 1x3 = 3
D → 1x3 = 3

= 13 bits

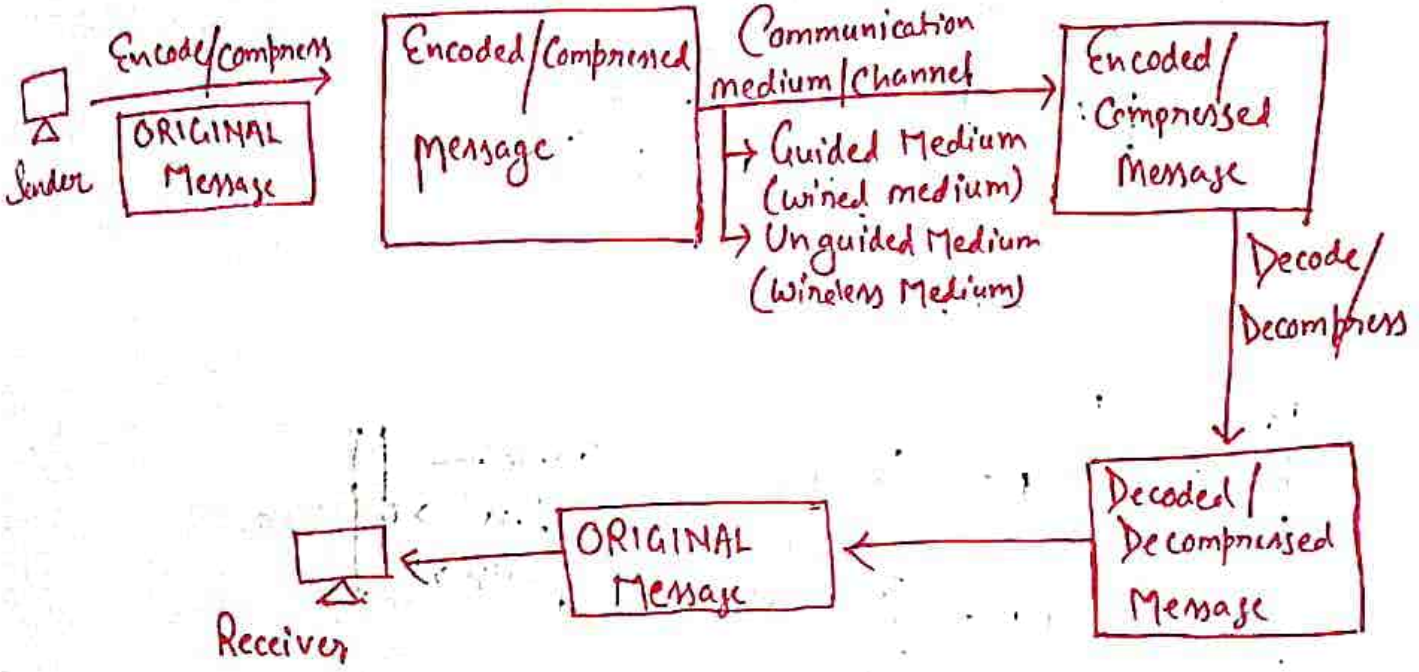
No. of bits saved = 49 − 13 = 36 bits

bits saved per character = $\frac{13}{7}$ = 1.85 bit/ch.

Q What is the total number of bits needed to represent the message using Huffman coding and find the average number of bits required per character.

Character	Frequency
W	6
X	60
Y	4
Z	5
M	3

Huffman Coding :



Lossy Compression – Some data is/are missing

~~Lossless~~ Lossless Compression – No data " missing.

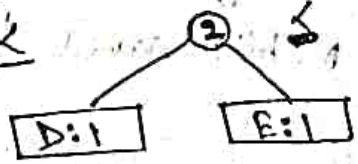
Q ~~Qn~~ ~~Which~~ What is the total number of bits needed to transfer the message using Huffman Coding and find the average number of bits needed for each character

Message / Data : AABBBDE.

Soln →

Character	Frequency
A	2
B	3
D	1
E	1

LHS value ≤ RHS value

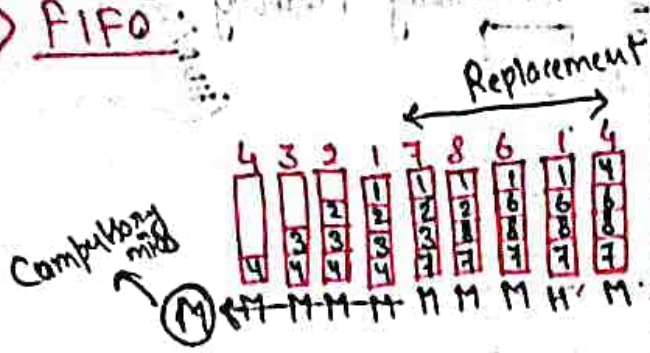


Note:

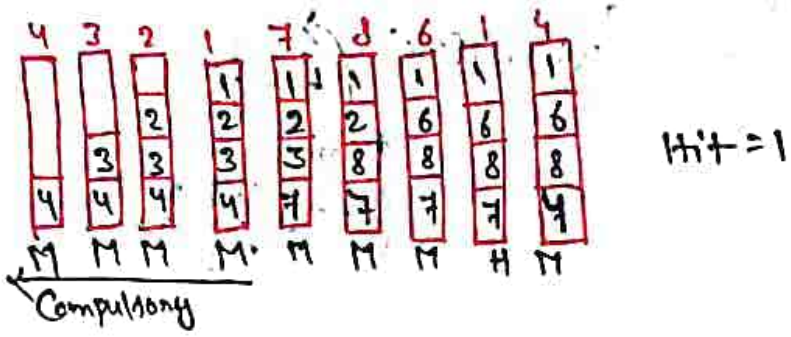
Left side → Current value ≥ value

Right side → Current value < next value

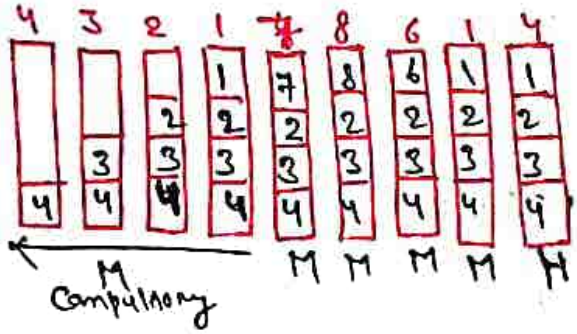
a) FIFO :



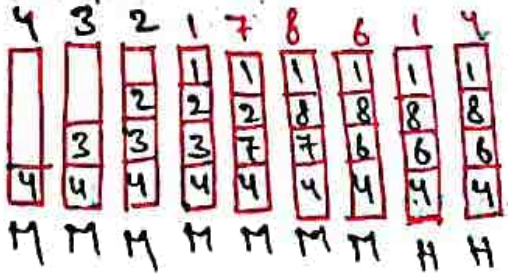
b) LRU : (4, 3, 2, 1, 7, 8, 6, 1, 4)



c) MRU :

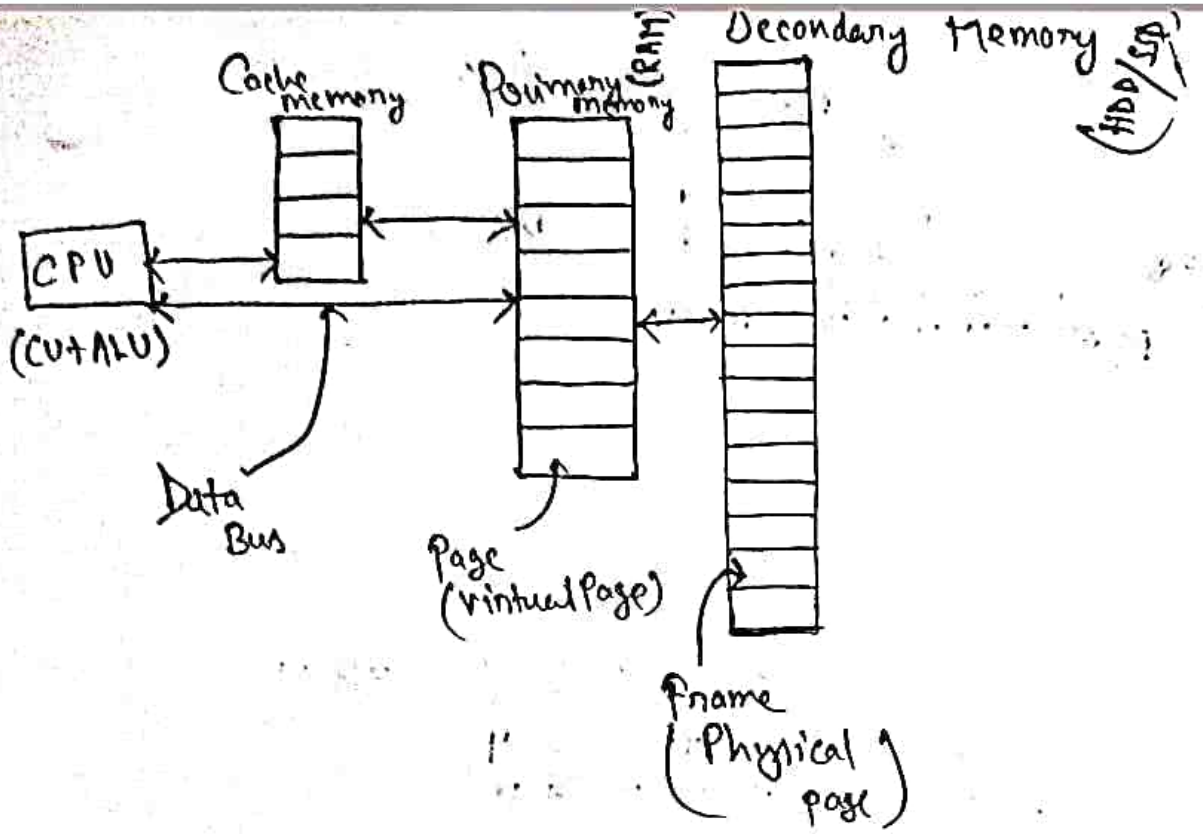


d) OC :



Note :-

i) Optimal Caching will not be practically implemented
ii) Bellady has proposed this theorem. in 1960.
iii) If Hit count will increase, then the prefarance will be increased.



There are 4 methods / techniques used to do the cache Replacement —

1) First In First Out (FIFO)
2) Least Recently used (LRU)
3) Most Recently Used (MRU)
4) Optimal Caching (OC) / → Optimal Cache Replacement (OCR)

[farthest-In-Future]

Q. Why the page replacement techniques find what is the number of hit(s) / miss(es), where page size is 4.

Data Streams: 4,3,2,1,7,8,6,1,4

Def. A reduced schedule is a schedule that only inserts an item into the cache in a step in which that item is requested.

Intuition: Can transform an unreduced schedule into a reduced one with no more cache misses.

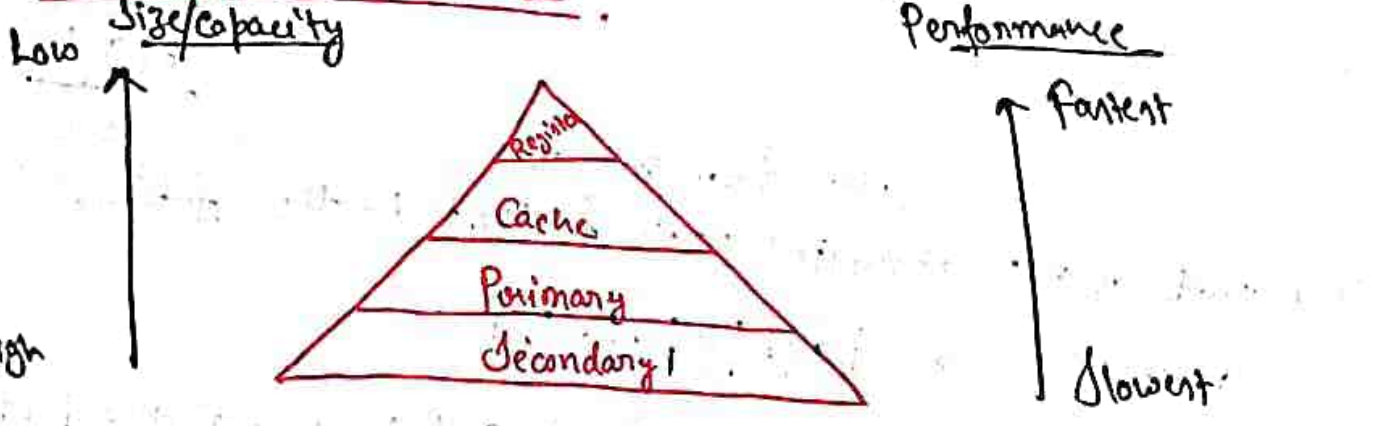
a	a	b	c
a	a	x	c
c	a	d	c
d	a	d	b
a	a	c	b
b	a	x	b
c	a	c	b
a	a	b	c
a	a	b	c

an unreduced schedule

a	a	b	c
a	a	b	c
c	a	b	c
d	a	d	c
a	a	d	c
b	a	d	b
c	a	c	b
a	a	c	b
a	a	c	b

a reduced schedule

Memory Hierarchy.



The Divide and conquer approach.

Divide :

The problem into a number of subproblems are smaller instances of the same problem

conquer :

The subproblems by solving them recursivily. If the subproblem sizes are small enough, however just solve the subproblems in the straight forward manner

combine :

The solutions to the subroblems into the solutions for the original problem.

The merge sort algorithm closely follows the divide & conquer paradigm. Intutively, it opperates as follows.

Divide

Divide the n-elements seqvence to be sorted into two subseqvences of n/2 elements each.

conquer

Sort the two subseqvences recursively using merge sort.

combine

Merge the two sorted subseqvences to produce the sorted answer.

MERGE (A, p, q, r)

1. $n_1 = q - p + 1$
2. $n_2 = r - q$
3. Let $L[1 \ldots n_1+1]$ and $R[1 \ldots n_2+1]$ be new arrays

```
4.  for  i = 1 to n1
5.      L[i] = A[p+i-1]
6.  for j = 1 to n2
7.      R[j] = A[q+j]
8.      L[n1+1] = ∞
9.      R[n2+1] = ∞
10.     i = 1
11.     j = 1
12.     for k = p to r
13.         if L[i] ≤ R[j]
14.             A[k] = L[i]
15.             i = i+1
16.         else  A[k] = R[j]
17.             j = j+1
```

MERGE-SORT (A,p,r)

1. if p<r — O(1)
2. q = ⌊(p+r)/2⌋ — O(1)

(1 & 2 together) Divide — O(1)

3. MERGE-SORT (A,p,q) → T(⌈n/2⌉)
4. MERGE-SORT (A,q+1,r) → T(⌊n/2⌋)
5. MERGE (A,p,q,r)

EXAMPLE:

Sort the elements present in the array using merge-sort

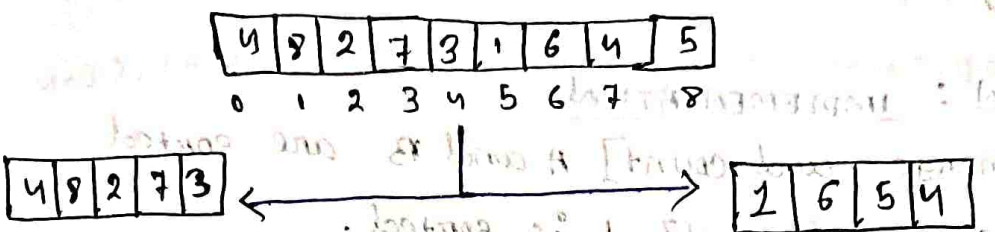
Array

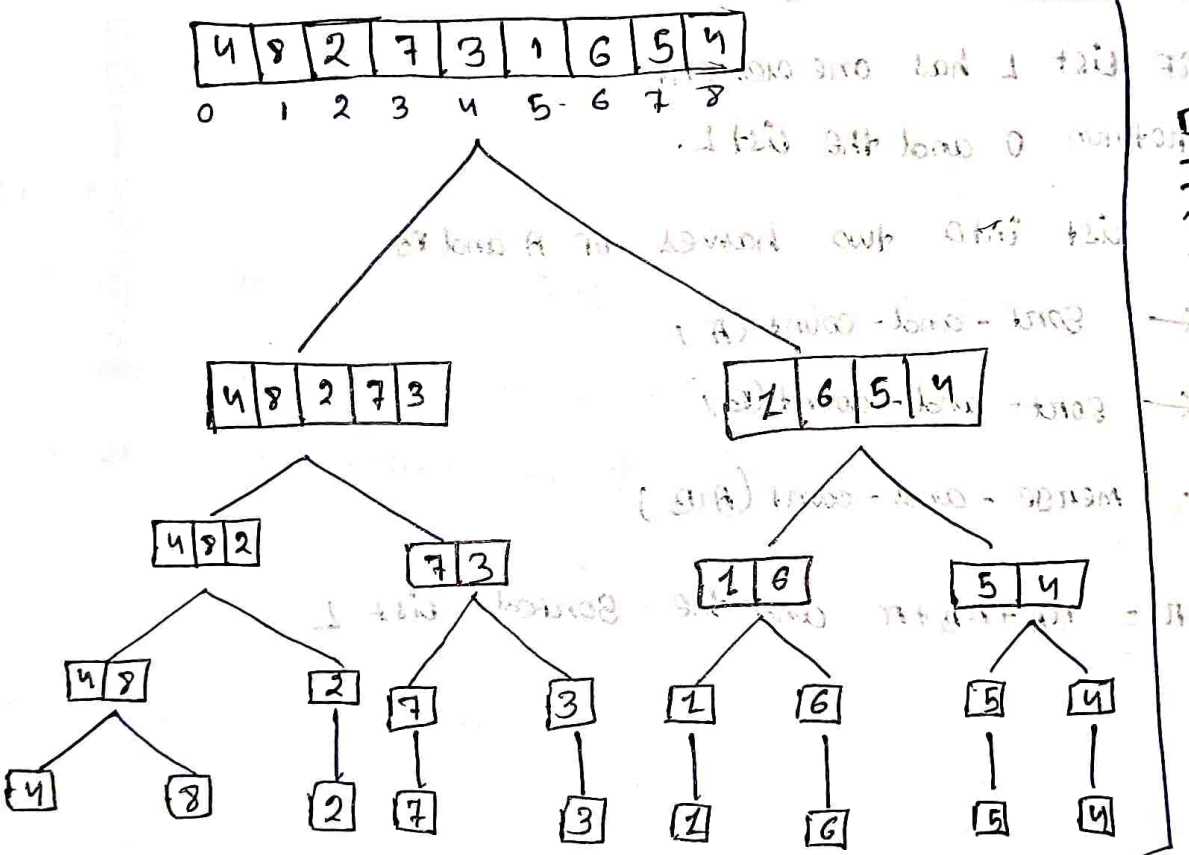
4	8	2	7	3	1	6	5	4
0	1	2	3	4	5	6	7	8

P, q, r

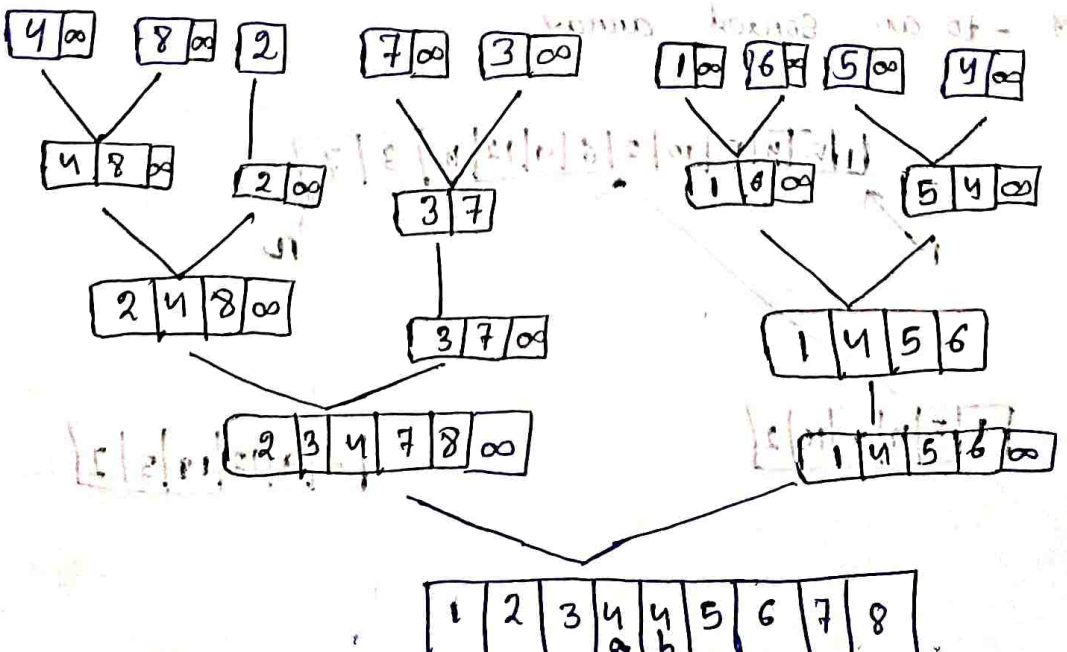
Soln

$q = \left\lfloor \frac{(p+r)}{2} \right\rfloor = \left\lfloor \frac{0+8}{2} \right\rfloor = \left\lfloor \frac{8}{2} \right\rfloor = 4$





#



RECURRANCE RELATION

$$T(n) = T(\lceil n/2 \rceil) + T\left\lfloor \frac{n}{-2} \right\rfloor + O(c \cdot n)$$

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

COUNTING. INVERSION : IMPLEMENTATION

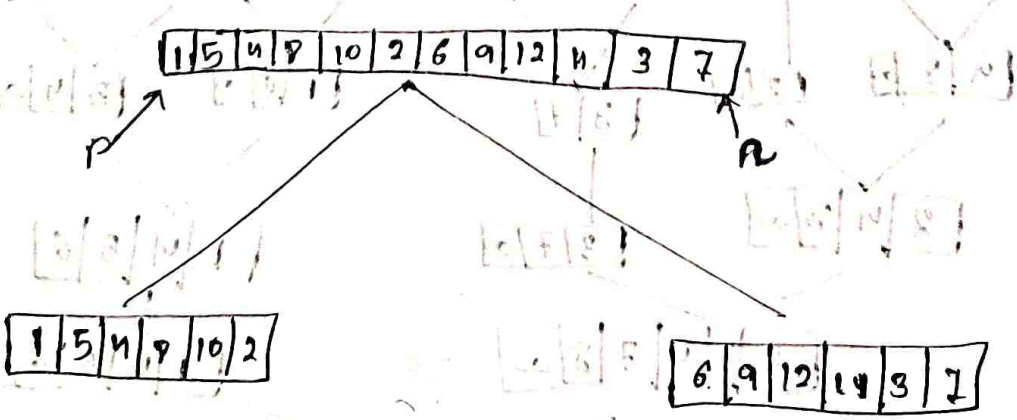
Pre-condition [merge-and-count] A and B are sorted

post-condition [sort-and-count] L is sorted.

```
Sort-and-count (L) {
if list L has one element
return 0 and the list L.
divide the list into two haves of A and B
(rA, A) ← Sort-and-count (A)
(rB, B) ← sort-and-count (B)
(r, L) ← merge-and-count (A, B)
return r = rA + rB + r and the sorted list L
}
```

EXAMPLE

Find the total number of inversions need to make the unsorted array - to an sorted array.



5| (5-4), (5-2), (4-2), (8-2) (10-2)

8| (6-3), (9-3), (9-7), (12-11), (12-3), (12-7), (11-3), (11-7),

9| (4-3), (5-3), (8-3), (8-6), (8-7), (10-3), (10-6), (10-7), (10-9)

5+8+9 = 22

Recurrance relation : $T(n) = 2T(n/2) + n$

$TC = O(n \log_2 n)$

NOTE

* $TC = O(n \log_2 n)$ (using DAC (divide and conquer))
* Implementation of merge sort.
* Brute force method the TC = $O(n^2)$

$$\begin{cases} n(n-1) \\ n^2 - n \\ O(n^2) \end{cases}$$

$O(n^2) > O(n \log_2 n)$

Hence we use DAC.

-> QUICK SORT (A,p,r)

1. if p<r
2. q = PARTITION (A,p,r)
3. QUICKSORT (A,p,q-1)
4. QUICKSORT (A,q+1,r)

-> Partitioning the array

PARTITION (A,p,r)

1. x = A[r]

2. i = p-1

3. for j = p to r-1

4. if A[j] ≤ x

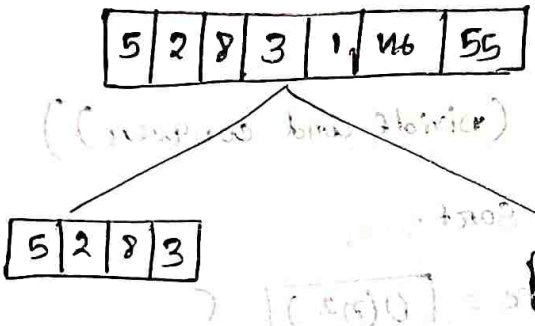
5. i = i+1

6. exchange A[i] with A[j]

7. exchange A[i+1] with A[r]

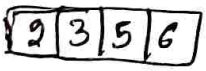
8. return i+1

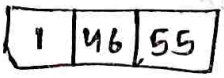
Q. FIND the total no. of inversion from the given array.



$\left\lfloor \frac{0+6}{2} \right\rfloor = \lfloor 3 \rfloor = 3$

(5-2) (5-3) (8-3) → 3



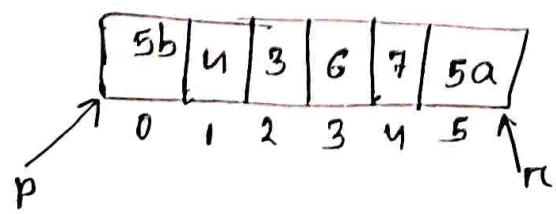


(2-1), (3-1), (5-1), (8-1) → 4

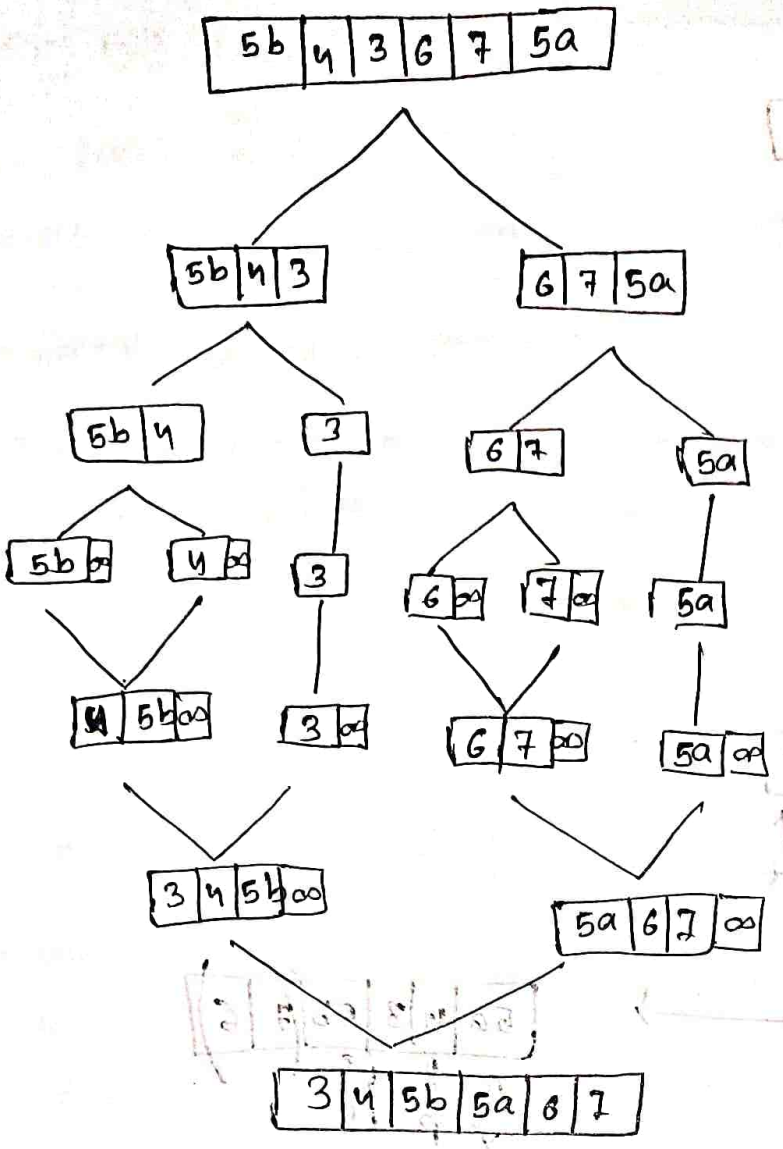
Total no. of inversions = 7

Q Sort the array using Quick sort and merge sort

merge sort



$\left\lfloor \frac{p+r}{2} \right\rfloor = \left\lfloor \frac{0+5}{2} \right\rfloor = 2$

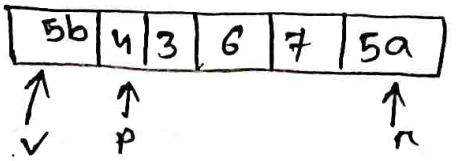


alphabatic order - 5a 5b

Solution_steps:

- select the first element as pivot element (v)
- Denote the next element of pivot element as "p" and run that "p" to the last element by following the condition if (p>v) then stop.
- Denote the last element of the array as "π" and run that "π" to the first element by following condition if (π ≤ v) then stop.
- Exchange the value of "π" and "v" if P and V crossed each other. otherwise exchange the value of "p" & "π"

Solution



p > r

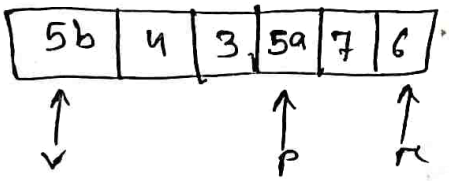
4 > 5a (false)

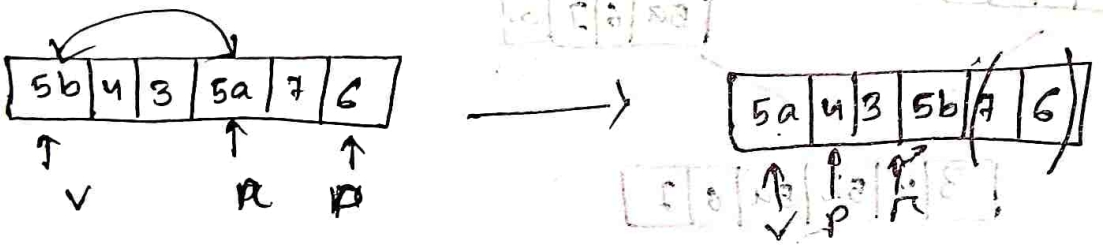
3 > 5a (false)

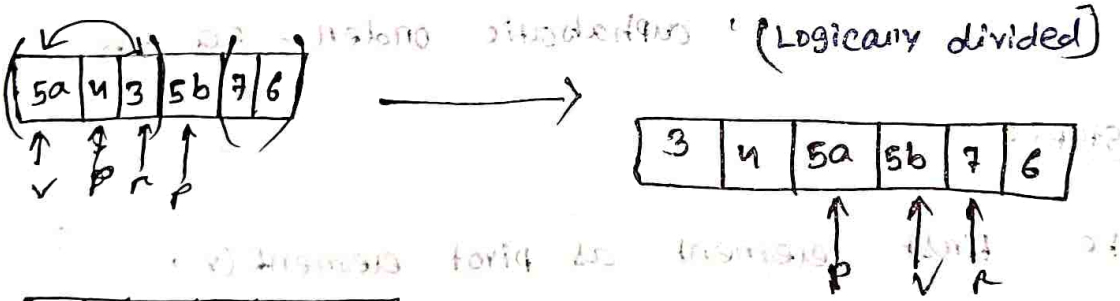
6 > 5a (true)

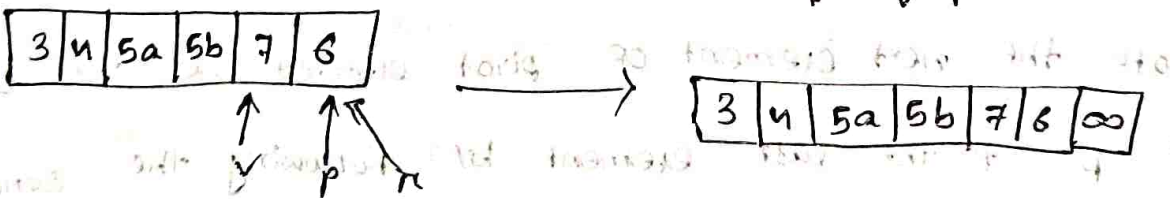
r ≤ v

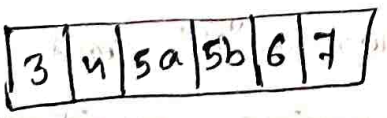
5a ≤ 5b true



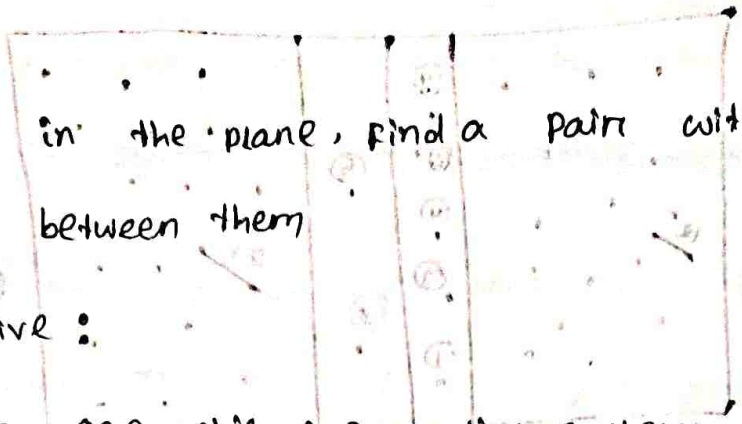








CLOSEST PAIR OF POINTS



Closest pair: Given n points in the plane, find a pair with smallest Euclidian distance between them

Fundamental geometric primitive:

- Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.
- Special case of nearest neighbour, Euclidean MST, Voronoi.

Brute force: Check all pairs of points p & q with $\theta(n^2)$

1-D version: $O(n \log n)$ easy if points are on a line

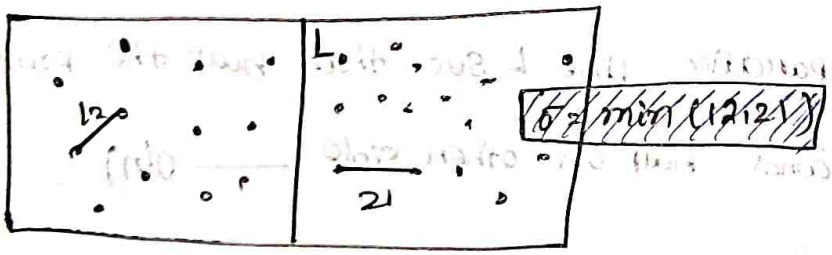
Assumption: No two points have same x co-ordinate

↓

to make presentation clear

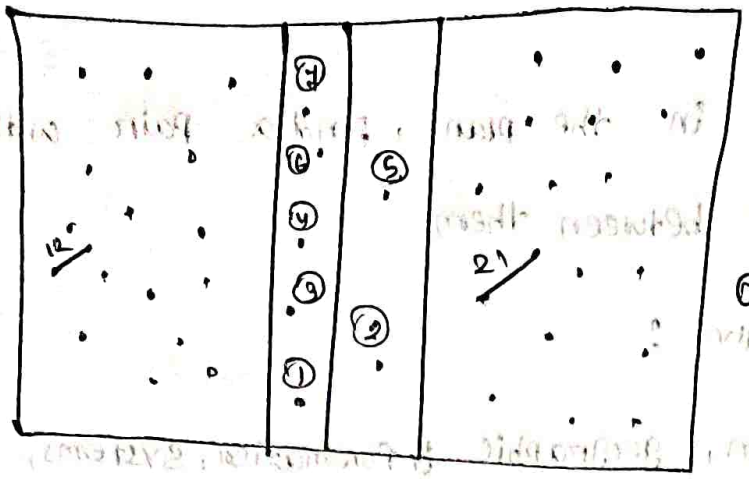
ALGORITHM

- Divide: Draw vertical line L so that roughly $\frac{1}{2}n$ points on each side



Find closest pair with one point in each side, assuming that distance $< \delta$

- Observation: only need to consider points within δ of line L
- Sort points in 2δ-strip by their y co-ordinate.
- only check distances of those within 11 positions in sorted list



$\delta = \min(12, 21)$

Def: Let s_i be the point in the 2δ-strip, with the i^{th} smallest y-coordinate

Claim: If $|i-j| \geq 12$, then the distance betn s_i & s_j is at least δ.

Pf:

- No two points lie in same $1/2\delta$-by-$1/2\delta$-box
- Two points at least 2 rows apart have distance $\geq 2(1/2\delta)$

Fact: Still true if we replace 12 with 7

closest-pair $(p_1, \ldots, p_n)$ {

compute separation line L such that half the points are one on side and half on other side ——— $O(n)$

$\delta_1 =$ closest-pair (left half)
$\delta_2 =$ closest-pair (right half) ——— $2T(n/2)$

$\delta = \min(\delta_1, \delta_2)$

Delete all points further than δ from separation line.

L $\longrightarrow O(n)$ sort remaining points by y-co-ordinate $\longrightarrow O(n \log n)$

Scan points in y-order and compute distance between each point and next 11 neighbours. If any one of these distance is less than δ, update δ. $\longrightarrow O(n)$

return δ.

}

Running time: $T(n) \leq 2T(n/2) + O(n \log n)$

$\Rightarrow T(n) = O(n \log^2 n)$

KARSTUBA'S ALGORITHM

Solution steps:

① Both the number must be have even digits - If odd digits then place 0 in the L.H.S of the number to make even.

② Divide the digits of the number in equal halves

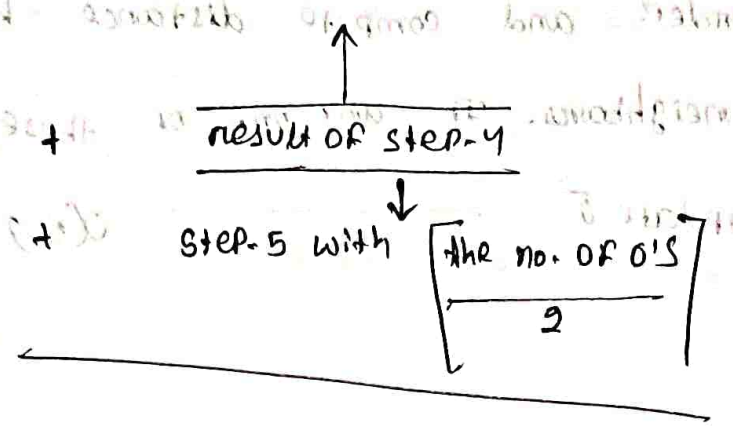
③ Da, (1st half of 1st no.) x (1st half of 2nd no.)

④ Da, (2nd half of 1st no.) x (2nd half of 2nd no.)

⑤ Da (1st half of 1st no. + 2nd half of 1st no.) x (1st half of 2nd no. + 2nd half of 2nd no.)

⑥ (Step-5) → (Step-4) → (Step-3)

⑦ Step 3 with the no. of 0's (i.e as per the digits of step-4)



Result ← (product of two numbers)

Ex:-

X = 14312

Y = 142

S-3 = 014 × 000 = 0

S-4 = 312 × 142 = 44304

S-5 = (014 + 312) (000 + 142) = 326 × 142 = 46292

S-6 = 46292 − 44304 − 0 = 1988

S-7 = From S-3 ← 00000

44304

1988000 ← (n/2)

2032304

CLUSTERING

Theorum-3 : cluster

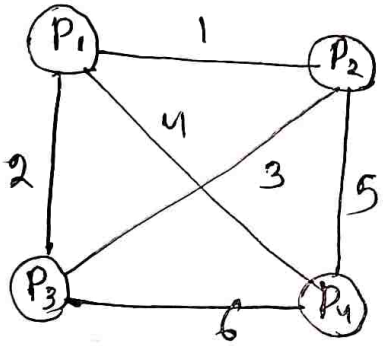
2 approaches

	P_1	P_2	P_3	P_4
P_1	0	1	2	4
P_2	1	0	3	5
P_3	2	3	0	6
P_4	4	5	6	0

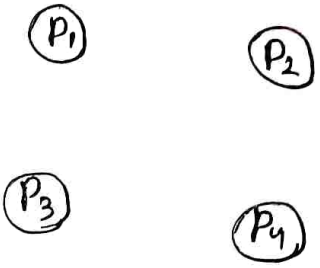
(i) $|V| \Rightarrow$ using kruskal's algo.

(ii) $|E| \Rightarrow$ Removing $(k-1)^{th}$ largest weight from the MST

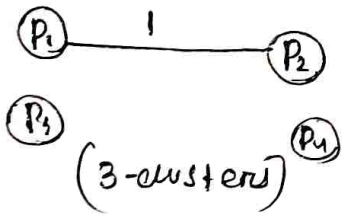
Soln :-



kruskal's →

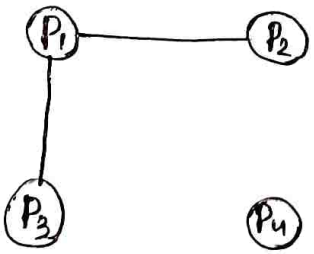


→



(3-clusters)

→



(2 clusters)

BINOMIAL CO-EFFICIENT

A Binomial co-efficient ${}^nC_K$ or $C(n,K)$ can be defined as the coefficient of x^K in expansion of $(1+x)^n$.

EX: The Binomial co-efficient is given as follows:

$$(a+b)^n = {}^nC_0 a^n b^0 + {}^nC_1 a^{n-1} b^1 + {}^nC_2 a^{n-2} b^2 + \cdots + {}^nC_n a^{n-n} b^n$$

where ${}^nC_0, {}^nC_1, \ldots\ldots {}^nC_n$ are called as Binomial coefficient

Another Defination:

A Binomial co-efficient ${}^nC_K$ or $C(n,K)$ also gives the number of ways, disregarding order, in which "K" objects can be choosen from "n" objects: more formally, the number of K-elements subsets / K-combinations of an n-element set.

For example, let's say we have 2 objects (i.e, $n=2$)

1) ${}^2C_0$ / $C(2,0)$ = Number of ways of choosing "0" objects out of 2 objects. $\left[\frac{2!}{0!(2-0)!} = \frac{2!}{2!} = 1\right]$

2) ${}^2C_1$ / $C(2,1)$ = Number of ways of choosing '1' object out of 2 objects $\left[\frac{2!}{1!(2-1)!} = \frac{2!}{1!} = 2\right]$

3) ${}^2C_2$ / $C(2,2)$ = Number of ways of choosing "2" obj, out of 2 objects. $\left[\frac{2!}{(2-2)!2!} = \frac{2}{2} = 1\right]$

* ^nC_k / $C(n,k) = \dfrac{n!}{k!(n-k)!} \Big/ \dfrac{n!}{(n-k)!\,k!}$

ALGORITHM FOR BINOMIAL co-efficient

```
int Binomial_coefficient (n,k)
{
{
for (i=0 ; i<=n ; i++)
do
   for(j=0; j<=k; j++)
   do
      if (j=0 or i=j)
then C [i][j] = 1
else C [i][j] = C[(i-1),(j-1)] + C [(i-1), j]
   end for
end for
   return C [n,k];
}
```

RECURRANCE RELATION OF Binomial co-efficient.

$$^nC_k = \begin{cases} 1, & \text{if } k=0 \text{ or } n=k \\ {}^{n-1}C_{k-1}, & \text{for } n>k>0 \end{cases}$$

or

$$C(n,k) = \begin{cases} 1, & \text{if } k=0 \text{ or } n=k \\ C((n-1),(k-1)) + C((n-1),k), & \text{for } n>k>0 \end{cases}$$

Solution for Binomial co-efficient

n \ k	0	1	2	3	4	5	6	7
0	1	X						
1	1	1	X					
2	1	2	1	X				
3	1	3	3	1				
4	1	4	6	4	1			
5	1	5	10	10	5	1		
6	1	6	15	20	15	6	1	
7	1	7	21	35	35	21	7	1

K → n

$C(n,K) = C((n-1),(K-1)) + C(n-1,K)$

$^2C_1 \;/\; C(2,1)$

$= C(1,0) + C(1,1)$

$= 1+1$

$= 2$

^nC_n

$^nC_n = 1$

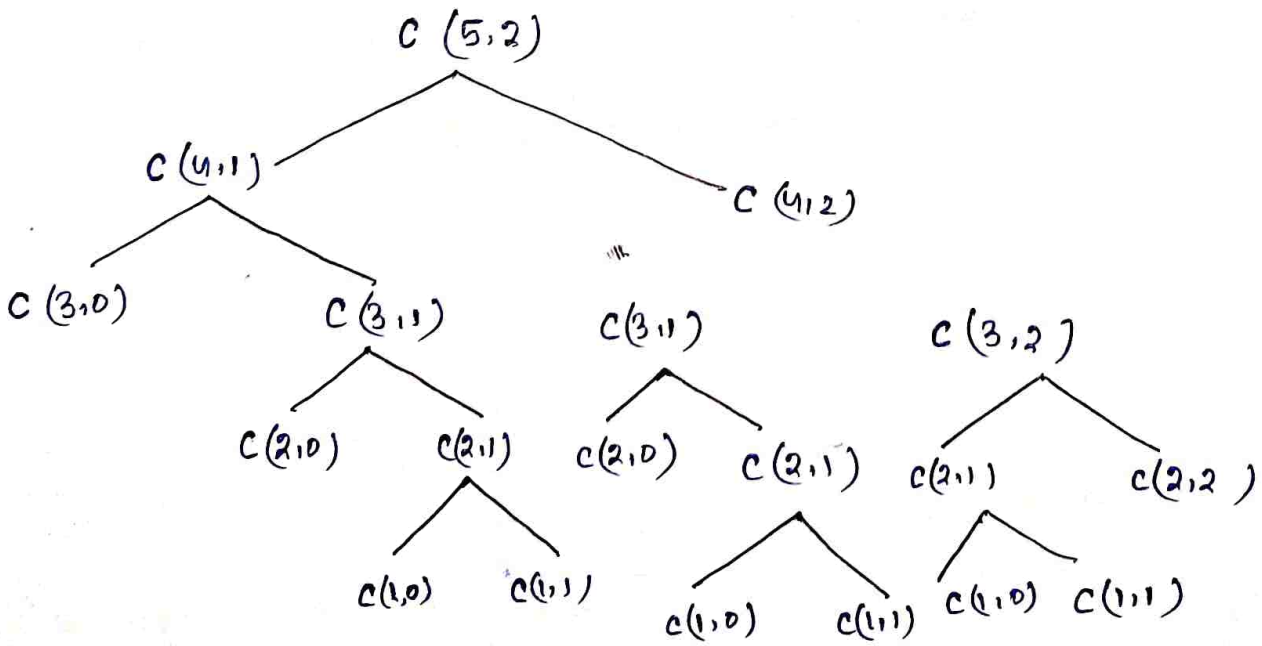
Base condition

$^nC_0 \rightarrow {}^nC_0 = 1$

Base condition

OVERLAPPING SUBPROBLEM

For $^5C_2$ or $C(5,2)$ the recursion tree will be as follows.



→ Since same sub problems are called again and again. SO this problem has "overlapping subproblem".

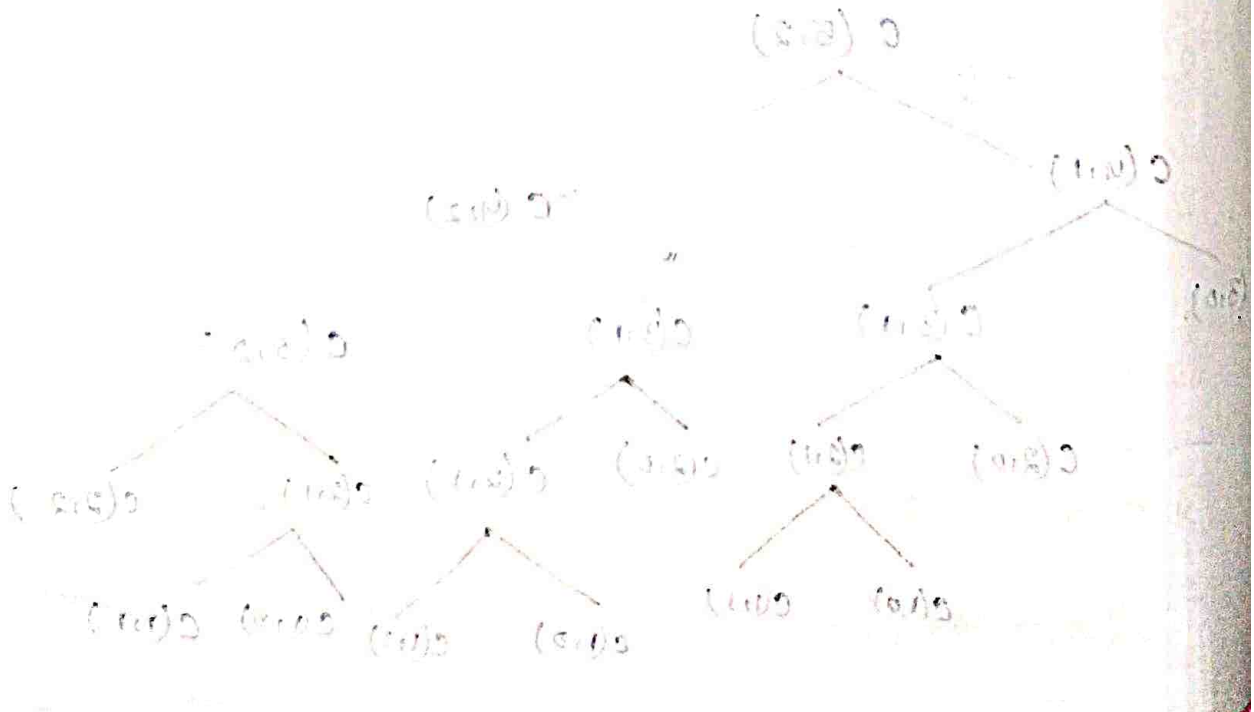
→ Binomial coefficients problem has both porperties OR Dynamic Programming (i.e optimal substructure & overlapping subproblem)

MATRIX CHAIN MULTIPLICATION

CORMAN (read) P9. - 370 - 390

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}_{2\times 2} \qquad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix}_{2\times 3}$$

$$AB = \begin{bmatrix} a_{11}\cdot b_{11} + a_{12}\cdot b_{21} & a_{11}\cdot b_{12} + a_{12}\cdot b_{22} & a_{11}\cdot b_{13} + a_{12}\cdot b_{23} \\ a_{21}\cdot b_{11} + a_{22} b_{21} & a_{21}\cdot b_{12} + a_{22} b_{22} & a_{21}\cdot b_{13} + a_{22} b_{23} \end{bmatrix}_{2\times 3}$$



MATRIX-MULTIPLY (A,B)

```
1. If A.columns ≠ B.rows
2.     error "incompatible dimensions"
3. else, let C be a new A.rows × B.column matrix
4.     for i = 1 to A.rows
5.         for j = 1 to B.columns
6.             cij = 0
7.             for k = 1 to A.columns
8.                 cij = cij + aik . bkj
9.     return C.
```

MATRIX-CHAIN-ORDER (p)

```
1.  n = p.length - 1
2.  let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables
3.  for i = 1 to n
4.      m[i,i] = 0
5.  for l = 2 to n                    // l is the chain length
6.      for i = 1 to n-l+1
7.          j = i+l-1
8.          m[i,j] = ∞
9.          for k = i to j-1
10.             q = m[i,k] + m[k+1,j] + pi-1 pk pj
11.             if q < m[i,j]
12.                 m[i,j] = q
13.                 s[i,j] = k
14.     return m and s
```

RECURSIVE SOLUTION

$$M[i,j] = \begin{cases} 0, & \text{if } i=j \\ \min\limits_{i \le k < j} \{ M[i,k] + M[(k+1),j] + P_{(i-1)} P_k P_j \}, & \text{if } i<j \end{cases}$$

where $M[i,j]$ is the minimum number of scalar multiplications needed to compute a matrix $A_{i..j}$

P_i, P_j, P_k are the dimensions of the matrices.

COUNTING the NUMBER OF PARENTHESIZATION

$$P(n) = \begin{cases} 1, & \text{if } n=1 \\ \sum\limits_{k=1}^{n-1} P(k), P(n-k), & \text{if } n \ge 2 \end{cases}$$

where $P(n)$ is the number of alternative parenthesization of a sequence of "n" matrices.

Note:

$$\frac{2(n-1)\,C_{(n-1)}}{n}$$

No. of ways parenthesization can be placed.

EXAMPLE

Find the optimum / minimum number of multiplications needed using the following matrices and also mention the proper parenthesization

$M_{2\times1}$ $M_{1\times3}$ $M_{3\times1}$

QUIZ-19

Find the optimum / minimum no. of multiplication needed using the following matrices and also mention the proper parenthesization

$$\underset{P_0}{(M_1)_{3\times 2}} \quad \underset{P_1}{(M_2)_{2\times 4}} \quad \underset{P_2}{(M_3)_{4\times 1}} \quad \underset{P_3}{(M_4)_{1\times 2}} \quad \underset{P_4}{}$$

	1	1	(3)
		2	3
			3

0	24	14	
	0	8	12
		0	8
			0

$$M[1,2] = \min_{1\le k<2} \{ k=1,\ M[1,1] + M[2,2] + P_0P_1P_2 \}$$

$$= 0 + 0 + (3\times 2\times 4)$$

$$= 24$$

$$M(2,3) = \min_{2\le k<3} \{ M[2,2] + M[3,3] + P_1P_2P_3 \}$$

$$= \min_{2\le k<3} \{ 0 + 0 + 2\times 4\times 1 \}$$

$$= 8$$

$$M(3,4) = \min_{3\le k<4} \{ M[3,3] + M[4,4] + P_2P_3P_4 \}$$

$$= \min_{3\le k<4} \{ 0 + 0 + 4\times 1\times 2 \}$$

$$= 8$$

$$M[1,3] = \min_{1\le k<3} \begin{cases} k=1;\ M[1,1] + M[2,3] + P_0P_1P_3 \\ k=2;\ M[1,2] + M[3,3] + P_0P_2P_3 \end{cases}$$

$$= \min_{1\le k<3} \begin{cases} k=1;\ 0 + 8 + (3\times 2\times 1) = 14 \\ k=2;\ 24 + 0 + (3\times 4\times 1) = 36 \end{cases}$$

• Weight Interval Scheduling •

Given: A list of jobs/tasks which have a start time, finish ti and a weight associate with it

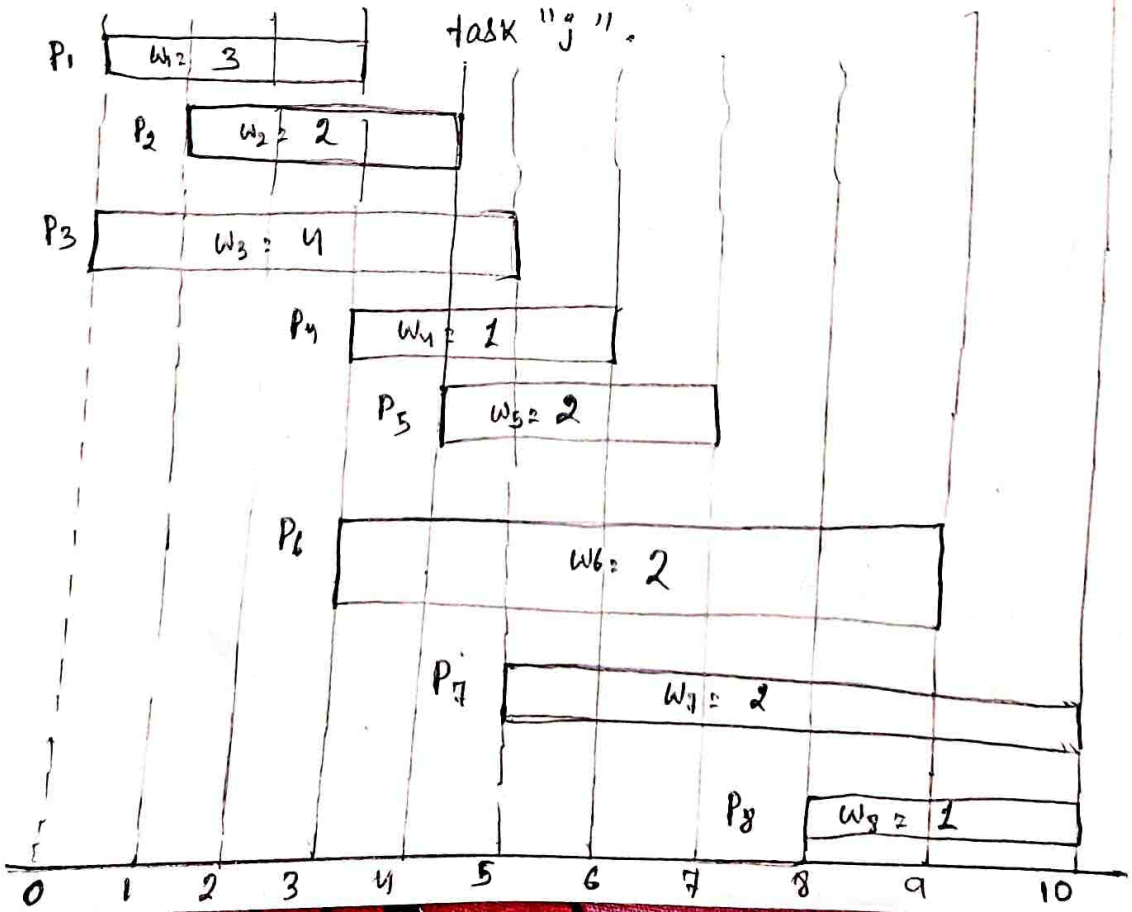
Aim: We want to find the subset of non overlapping/compati jobs/tasks with the maximum weight.

Notation: For a job "j", the start time is denoted as "S_j", th Finish time is denoted as "f_j" and the weight assigned to the job/tasks "j" is denoted as "w_j"

Formal to find the maximum weight of the compatible set of jobs considering the task/jobs "j" [OPT(j)] is as follows.

$$OPT(j) = \max \left\{ w_j + OPT(P(j)),\ OPT(j-1) \right\}$$

Where P(j) is the set of compatible jobs/task which have considers/observed before the job/task "j".



i	1	2	3	4	5	6	7	8
w_i	3	2	4	1	2	5	2	1
p(i)	0	0	0	1	2	1	3	5
OPT(i)	3	3	4	4	5	8	8	8

$$OPT(i) = \max \begin{cases} w_i + OPT(p(i)) \\ OPT(i-1) \end{cases}$$

Base condition

$$OPT(0) = 0$$

$OPT(1) = \max \{ w_1 + OPT\ p(1),\ OPT\ p(0) \}$

$= \max \{ w_1 + OPT(p(1), p(0)) \}$

$= \max \{ 3+0 \}$

$\Rightarrow \max \{3, 0\} = (3)$

$OPT(2) = \max \{ w_2 + OPT(p(2)), OPT(1) \}$

$OPT(0)$

$= \max \{ 2+0, 3 \}$

$\Rightarrow \max(2,3) = 3$

$OPT(3) = \max \{ w_3 + OPT(p(3)), OPT(2) \}$

$= \max \{ 4+0, 3 \}$

$= \max(4,3) = 4$

$OPT(4) = \max\{w_4 + OPT\ P(4),\ OPT(3)\}$

$= \max\{1+1, 4\}$

$= \max\{2, 4\} = (4)$

$OPT(5) = \max\{w_5 + OPT(P(5)),\ OPT(4)\}$

$= \max\{2+3, 4\}$

$= \max\{5, 4\} = (5)$

$OPT(6) = \max\{w_6 + OPT(P(6)),\ OPT(5)\}$

$= \max\{5+3, 5\}$

$= \max\{8, 4\} = (8)$

$OPT(7) = \max\{w_7 + OPT\ P(7),\ OPT(6)\}$

$= \max\{2+04, 6\}$

$= \max\{6, 8\} = (8)$

$OPT(8) = \max\{w_8 + OPT(P(8)),\ OPT(7)\}$

$= \max\{1+5, 08\}$

$= \max\{6, 8\} = (8)$

FINDING OPTIMAL SOLUTION FOR PICKING / SELECTING COMPATIBLE SET OF JOBS / TASKS HAVING MAXIMUM WEIGHT.

The solution for finding which items we picked is by backtracking, simply if $w_i + OPT(p(i)) \geq OPT(i-1)$, then we add item "i" to optimal solution and look at $OPT(p(i))$, else we ignore this item and look at subproblem $OPT(i-1)$.

```
Compute-Opt (i)
  if i=0 then
  Return 0
ELse
    Return max (v_i + compute-opt (p(i)), compute-opt (i-1))
end if
```

```
M- compute - Opt (j)
  if j=0 then
      Return 0
  Else if M(j) is not empty then
      Return M[j]
      Else
Define M[j] = max (v_j + M-compute-opt (p(j)), M- compute
                                        -opt (j-1))
```

Find Solution (j)

If j=0 then

output nothing

ELSE

If $V_j + M[p(j)] \geq M[j-1]$ then

output j together with the result of Find-Solution

ELSE

output the result of Find-Solution (i-1)

End if

End if

Iterative-compute-opt

$M[0] = 0$

for $j = 1, 2, \ldots, n$

$M[j] = \max(V_j + M[p(j)], M[j-1])$

End for

TIME COMPLEXITY OF WEIGHTED INTERVAL SCHEDULING (WIS)

1. Sort the jobs/tasks $[O(n \log n)]$
2. Finding p(i), where $i = 1, 2, 3, \ldots, n$ (clever use of binary search)
3. To find OPT(i) after getting the value of p(i) $[O(n)]$
4. Back track to ~~Start~~ select optimal set of jobs/tasks $[O(n)]$

Total time complexity = $O(n \log n)$

Quiz-20

Q) Find the set of compatible jobs / tasks with maximum weight using Dynamic Programming. From the below table.

i	1	2	3	4	5	6	7	8	9
W(i)	2	11	3	4	2	6	10	9	8
P(i)	0	0	1	2	2	2	2	3	4
OPT(i)	2	11	11	11	11	11	15	15	15

TW W_7 W_3 W_1

$15 - 10 = 5 - 3 = 2 - 2 = 0$

7 5 3

TW W_7 W_5 ✗ WRONG W_3

$15 - 10 = 5 - 2 = 3 - 3 = 0$

BELLMAN FORD ALGORITHM:

```
Shortest-Path (G, S, t)
    n = number of nodes in G
    Array M[0 ... n-1, V]
    Define M[0, t] = 0 and M[0, v] = ∞ for all other v ∈ V
    for i = 1, ....., n-1
        for v ∈ V in any order
        compute M[i, v] using the recurrence (6.23)
        End for
    EndFor
    Return M[n-1, S]
```

(6.23) if $i > 0$ then

$$OPT(i, v) = \min\left(OPT(i-1, v), \min_{w \in V}\left(OPT(i-1, w) + C_{vw}\right)\right)$$

* push-Based-Shortest-Path (G, s, t)

n = number of nodes in G

Array M[V]

Initialize M[t] = 0 and M[v] = ∞ for all other v ∈ V

For i = 1, ..., n-1

for w ∈ V in any order

IF M[w] has been updated in the previous iteration then

for all edges (v, w) in any order

M[v] = min(M[v], c_{vw} + M[w])

If this changes the value of M[v], then first[v] = w

End for.

End for

If no value changed in this iteration, then End the algorithm

End for

Return M[s]

Solution Step:

1) Initially put the distance of source vertex as "0" and other as "∞"

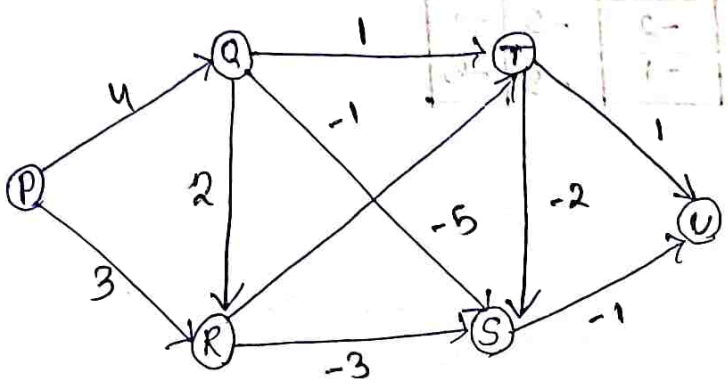
2) Relax the edge for (|V|-1)/(n-1) times by following formula:

Relax formula : if (d(v) > d(u) + c(u,v)) then d(v) = d(u) + c(u,v)

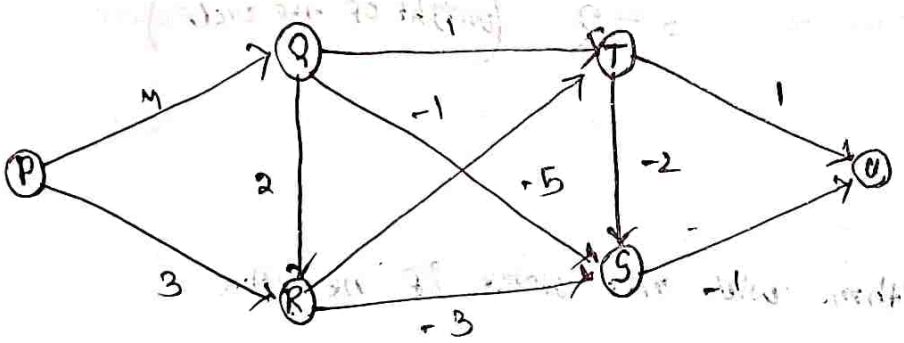
3) If the distance of the vertices is getting changed after running for $(|V|)/(n)$ times, then we can conclude that negative weight cycle is present in that graph.

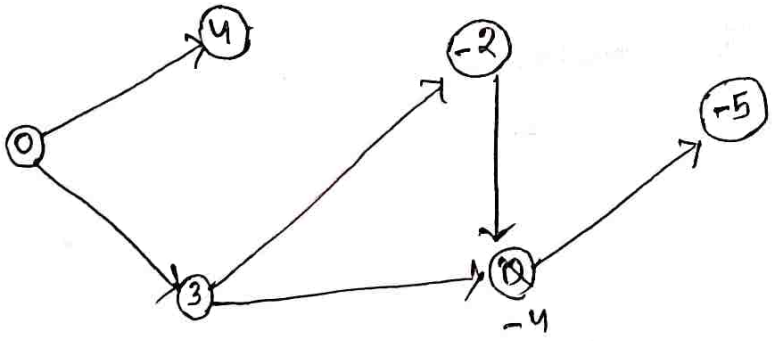
EXAMPLE

Find the single source shortest path using Bellman-Ford Algorithm



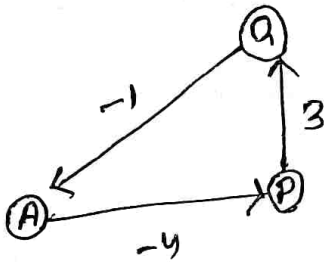
Solution





	0	1	2	3	4	5
P	0	0	0	0	0	0
Q	∞	4	4	M	4	4
R	∞	3	3	3	3	3
S	∞	∞	0	-4	-4	-4
T	∞	∞	-2	-2	-2	-2
V	∞	∞	∞	-1	-5	-5

Negative weighted cycle



$(-1) + (-4) + 3$

$= -5 + 3$

$= -2$ (weight of the cycle)

NOTE:

Bellman-ford algorithm will not work if negative weighted cycle is present.

Knapsack Problem

> Knapsack is a container or bag, Suppose, we have given some itemes which has some weight/ profits. we have to put some itemes in the knapsack in such a way that total values / profits produces a "maximum profit"

> There are two type of knapsack problems:

① fractional knapsack (Greedy Approach)

② 0/1 knapsack (Dynamic Programming)

0/1 knapsack

In 0/1 knapsack, we can consider an item/ object (i.e "1") or we can't consider an item/ object (i.e "0")

Formula to find maximum profit by considering the items

$$p(i,w) = \begin{cases} 0, & \text{if } i=0 \text{ or } w=0 \\ \max\{ p((i-1),w), p((i-1),(w-w_i)) + p_i \}, & \text{if } i>0 \text{ and } w \geq 0 \end{cases}$$

where "p" is the maximum profit by considering the item

"p_i" is the profit of the item "i"

"W" is the weight/ capacity of knapsack

"w_i" is the weight of the item "i"

"i" is the item.

Ex:

Find the maximum profit with selected items using 0/1 knapsack where the capacity of the knapsack is "8".

weight (w_i) : $\{3, 4, 6, 5\}$

profit (P_i) : $\{2, 3, 1, 4\}$

Solution

i↓	0	1	2	3	4	5	6	7	8 ← W
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	3	3	5	5
4	0	0	0	2	3	4	4	5	6

i	W_i	P_i
1 → 3 → 2	(w_1)	(P_1)
2 → 4 → 3	(w_2)	(P_2)
3 → 6 → 1	(w_3)	(P_3)
4 → 5 → 4	(w_4)	(P_4)

P(1,3)

$$\max\{ P(0,3),\ P(0,(3-3)+2) \}$$

$$= \max\{ 0,\ P(0,0)+2 \}$$

$$= \max\{0, 2\}$$

$$= \max\{0, 2\}$$

$$= 2$$

i →	1	2	3	4
	1	0	0	1

Quiz-21

Find the maximum profit using 0/1 knapsack where the maximum capacity of the knapsack is "12"

weights (w_i): {5, 4, 8, 1, 6}

profits (p_i): {2, 7, 3, 4, 10}

i ↓ / w_i →	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	2	2	2	2	2	2	2	2
2	0	0	0	0	7	7	7	7	7	9	9	9	9
3	0	0	0	0	7	7	7	7	7	9	9	9	10
4	0	4	4	4	7	11	11	11	11	11	13	13	13
5	0	4	4	4	7	11	11	14	14	14	17	17	21

i	w_i	p_i
1	5	2
2	4	7
3	8	3
4	1	4
5	6	10

$P(i,w) = \max\{P(i-1,w),\ P((i-1),(w-w_i)) + p_i\}$

$= \max\{P(0,1),\ P(0,-4)+2\}$

$= \max\{0,\ \text{invalid}\}$

$= 0$

$P(1,5) = \{P(0,5),\ P(0,5-5)+2\}$

$= \{0, 2\}$

$= 2$

$P(1,6) = 2$

$P(2,4) = \{P(1,4),\ P(1,4-4)+7\}$

$= \{0, 7\}$

$= 7$

$\{P(2,4),\ P(2,-4)+3\}$

$= 7$

$P(3,5)$

$= \{P(2,5),\ P(2,-3)+3\}$

$=$

$P(5,12) =$

$= P(4,12),\ P(4,12-1)+4\}$

$\therefore P(4,12)$

$= 13+4 = 17$

Fractional Knapsack :

→ The items / objects which will give us profit after getting divided / fractioned are considered in this approach.

→ Fractional knapsack is using "Greedy Approach"

→ Here, $\frac{\text{profit } (P_i)}{\text{weight } (w_i)}$ is providing the priority to select the item which will give the maximum profit.

Example

Find the maximum profit using fractional knapsack where the capacity of the knapsack is "9".

profit $(P_i) = \{2, 4, 6, 1\}$

weights $(w_i) = \{8, 4, 7, 3\}$

Soln

	1	2	3	4
profit (P_i)	2	4	6	1
weight (w_i)	8	4	7	3
$\frac{P_i}{w_i}$	$\frac{2}{8}$ = 0.25	$\frac{4}{4}$ = 1	$\frac{6}{7}$ = 0.85	$\frac{1}{3}$ = 0.33

P_2 $P_2 + P_3 = 4 + 4.25 = 8.25$ $5 < 7$ w (w_3)

Time complexity: $O(n*w)$ / $O(i*w)$

Space complexity: $O(n*w)$ / $O(i*w)$

Time complexity: $O(n \log n)$ / $O(i \log i)$

POLYNOMIAL and THE FFT

The most common use for Fourier transform, and hence the FFT, is in signal processing. A signal is given in the time domain: as a function mapping time to amplitude Fourier analysis allows us to express the signal as a weighted sum of phase-shifted sinusoids of varying frequencies The weights and phases associated with the frequencies characterize the signal in the frequency domain. Among the many everyday applications of FFT's are compression techniques used to encode digital video and audio information, including MP3 files. several fine books delve into the rich area of signal processing; the chapter notes reference a few of them.

POLYNOMIALS

A polynomial in the variable x over an algebraic field F represents a function $A(x)$ as a formal sum:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

we call the values $a_0, a_1, \ldots, a_{n-1}$ the coefficients of the polynomial. The co-efficients are drawn from a field F. typically the set C of complex numbers. A polynomial $A(x)$ has degree k if its highest nonzero coefficient is a_k; we write that degree $(A) = k$. Any integer strictly greater than the degree of a polynomial is a degree-bound of that polynomial. Therefore the degree of a polynomial of degree-bound n may be any integer between 0 and $n-1$, inclusive.

we can define a variety of operations on polynomials, for polynomial addition: if $A(x)$ and $B(x)$ are polynomials of degree bound

n, there sum is a polynomial C(x), also of degree-bound n, such that C(x) = A(x) + B(x), for all x in the underlying field. That is, if

$$A(x) = \sum_{j=0}^{n-1} a_j x^j,$$

and

$$B(x) = \sum_{j=0}^{n-1} b_j x^j,$$

then

$$C(x) = \sum_{j=0}^{n-1} c_j x^j,$$

where $c_j = a_j + b_j$ for $j = 0, 1, \ldots, n-1$. For example if we have the polynomials $A(x) = 6x^3 + 7x^2 - 10x + 9$ and $B(x) = -2x^3 + 4x - 5$, then $C(x) = 4x^3 + 7x^2 + 6x + 4$

For polynomial multiplication, if A(x) and B(x) are polynomials of degree-bound n, their product C(x) is

polynomial and the FFT

polynomials

A polynomial in the variable x over an ~~algebraic~~ algebraic field F represents a function $A(x)$ as a formal sum

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

The most common use for Fourier transforms, and hence the FFT, is in signal processing. A signal is given in the time domain: as a function mapping time to amplitude. Fourier analysis allows us to express the signal as a weighted sum of phase-shifted sinusoids of varying frequencies. The weights are phases associated with the frequencies characterize the signal in the frequency domain.

Among the many everyday applications of FFTs are compression techniques used to encode digital video and audio information, including MP3 files. Several fine books delved in to rich area of signal processing the chapter notes reference a few of them.

←

We call the values $a_0, a_1, \ldots a_{n-1}$ the coefficients of the polynomial. The coefficients are drawn from a field F, typically the set C of complex numbers.

A polynomial $A(x)$ has degree k if its highest nonzero coefficient is a_k: we write that degree$(A) = k$. Any integer strictly greater than the degree of a polynomial

of degree bound of that polynomial. Therefore, the degree of a polynomial of degree bound n may be any integer between 0 and n-1, inclusive.

We can define a variety of operations on polynomials. For polynomial addition, if A(x) and B(x) are polynomials of degree-bound n, their sum is a polynomial C(x), also of degree bound n. such that $C(x) = A(x) + B(x)$ for all x in the underlying field. That is, if

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

and

$$B(x) = \sum_{j=0}^{n-1} b_j x^j$$

then

$$C(x) = \sum_{j=0}^{n-1} c_j x^j$$

where $c_j = a_j + b_j$ for $j = 0, 1, \ldots, n-1$, for example, if we have the polynomials $A(x) = 6x^3 + 7x^2 - 10x + 9$ and $B(x) = -2x^3 + 4x - 5$, then $C(x) = 4x^3 + 7x^2 - 6x + 4$

For polynomial multiplication, if A(x) and B(x) are polynomials of degree bound n, their product C(x) is a polynomial of degree bound $2n-1$ such that $C(x) = A(x)B(x)$ for all x in the underlying field. You probably have multiplied polynomials before by multiplying each item in A(x) by each term in B(x) and then combining terms with equal equal powers. For example we can multiply A(x) =>

$A(x) = 6x^3 + 7x^2 - 10x + 9$

and $B(x) = -2x^3 + 4x - 5$ then.

$C(x) = 4x^3 + 7x^2 - 6x + 4$

For polynomial multiplication, if $A(x)$ and $B(x)$ are polynomials of degree-bound n, their product $C(x)$ is a polynomial of degree-bound $2n-1$ such that $C(x) = A(x)\,B(x)$ for all x in the underlying field. You probably have multiplied polynomials before, by multiplying each term in $A(x)$ by each term in $B(x)$ and then combining terms with equal powers. For example, we can multiply $A(x) = 6x^3 + 7x^2 - 10x + 9$ and $B(x) = -2x^3 + 4x - 5$ as follows.

$$\begin{array}{r} 6x^3 + 7x^2 - 10x + 9 \\ -2x^3 \qquad + 4x - 5 \\ \hline -30x^3 - 35x^2 + 50x - 45 \\ 24x^4 + 28x^3 - 40x^2 + 36x \\ -12x^6 - 14x^5 + 20x^4 - 18x^3 \\ \hline -12x^6 - 14x^5 + 44x^4 - 20x^3 - 75x^2 + 86x - 45 \end{array}$$

Another way to express the product $C(x)$ is

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j .$$

where

$$c_j = \sum_{k=0}^{j} a_k b_{j-k}$$

Note that degree(C) = degree(A) + degree(B), implying that if A is a polynomial of degree-bound n_a and B is a polynomial of degree-bound n_b, then C is a polynomial of degree-bound $n_a + n_b - 1$, since a

polynomial of degree-bound k is also a polynomial of degree bound k+1, we will normally say that the product polynomial c is a polynomial of degree bound $n_a + n_b$.

Coefficient representation

A coefficient representation of a polynomial $A(x) = \sum_{j=0}^{n-1} a_j x^j$ of degree-bound n is a vector $a = (a_0, a_1, \ldots, a_{n-1})$, in matrix of coefficients. For equations in this chapter, we shall generally treat vectors as column vectors.

The coefficient representation is convenient for certain operations on polynomials. For example the operation of ~~evaluat~~ evaluating the polynomial $A(x)$ at a given point x_0 consists of computing the value of $A(x_0)$; we can evaluate a polynomial in $\Theta(n)$ times using Horner's rule

$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-2} + x_0(a_{n-1}))\ldots)$

Similarly adding two polynomials represented by the coefficient vectors $a = (a_0, a_1, \ldots a_{n-1})$ and $b = (b_0, b_1, \ldots b_{n-1})$ takes $\Theta(n)$ times, we just produce the coefficient vector $c = (c_0, c_1, \ldots c_{n-1})$ where $c_j = a_j + b_j$ for $j = 0, 1, \ldots n-1$

example

$A(x) = x^2 + 3x + 2$

$B(x) = 2x^2 + 1$

$C(x) = A(x) \times B(x)$

$= (x^2 + 3x + 2) \times (2x^2 + 1)$

$= 2x^2(x^2 + 3x + 2) + 1(x^2 + 3x + 2)$

$= 2x^4 + 6x^3 + 4x^2 + x^2 + 3x + 2$

$= 2x^4 + 6x^3 + 5x^2 + 3x + 2.$

Coefficient representation $\Rightarrow$ $O(n^2)$ $\rightarrow$ reduced to $O(n \log n)$

$A(x) = 2(x^0) + 3(x^1) + 1(x^2)$

$A \rightarrow [2, 3, 1]$.

$B(x) = 1(x^0) + 0(x^1) + 2(x^2)$

$B \rightarrow [1, 0, 2]$.

$C(x) = 2(x^0) + 3(x^1) + 5(x^2) + 6(x^3) + 2(x^4)$.

$C \rightarrow [2, 3, 5, 6, 2]$.

point value representation

A point value representation of a polynomial $A(x)$ of degree-bound n is a set of n point-value pairs

$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$

such that all of the x_k are distinct and $y_k = A(x_k)$ for $k = 0, 1, \ldots, n-1$. A polynomial has many different point-value representations, since we can use any set of n distinct point $x_0, x_1, \ldots, x_{n-1}$ as a basis for the representation.

quires to compute the inverse DFT.

Steps ①, ② ③ ⇒ $\Theta(n)$

Steps (2), (4) ⇒ $\Theta(n \log n)$

The FFT ⇒

By using a method known as the fast Fourier transform (FFT), which takes advantage of the special properties of the complex roots of unity, we can compute $DFT_n(a)$ in time $\Theta(n \log n)$, as opposed to the $\Theta(n^2)$ time of the straight forward method. We assume throughout that n is an exact power of 2. Although strategies for dealing with non-power-of-2 sizes are known, they are beyond the scope of this book.

The FFT method employs a divide-and-conquer strategy, using the even indexed and odd indexed coefficients of $A(x)$ separately to define the two new polynomials $A^{[0]}(x)$ and $A^{[1]}(x)$ of degree-bound $\frac{n}{2}$

$$A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1}$$

$$A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1}$$

Note that $A^{[0]}$ contains all the even indexed coefficients of A (the binary representation of the index ends in 0) and $A^{[1]}$ contains all the odd indexed coefficients (the binary representation of the index ends in 1).

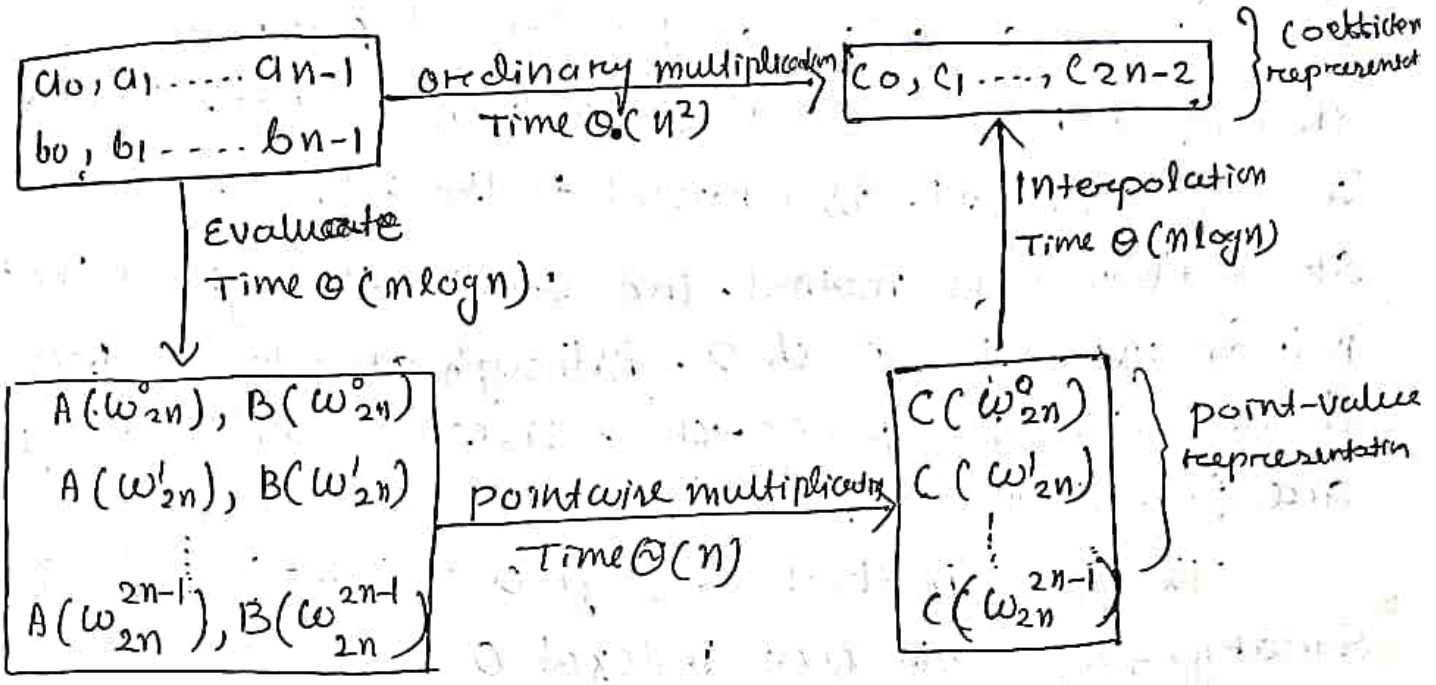
It follows that $A(x) = A^{[0]}(x^2) + x A^{[1]}(x^2)$

So that the problem of evaluating $A(x)$ at $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ reduces to

(1) evaluating the degree bound $n/2$ polynomials $A^{[0]}(x)$ and

A faster algorithm for n-point interpolation is based on Lagrange's formula:

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}$$



1) Double degree-bound: create coefficient representations of A(x) and B(x) as degree-bound 2n polynomials by adding n high-order zero coefficients to each.

2) Evaluate: compute point-value representations of A(x) and B(x) of length 2n by applying the FFT of order 2n on each polynomial. These representations contain the values of the two polynomials at the (2n)-th roots of unity.

3) pointwise multiply: compute a point-value representation for the polynomial $C(x) = A(x) \cdot B(x)$ by multiplying these value together pointwise. This representation contains the value of C(x) at each (2n)-th root of unity.

4) Interpolate: create the coefficient representation of the polynomial C(x) by applying the FFT on 2n point-value

computing a point-value representation for a polynomial given in coefficient form is in principle straightforward since all we have to do is select n distinct points $x_0, x_1, \ldots, x_{n-1}$ and then evaluate $A(x_k)$ for $k = 0, 1, \ldots, n-1$, with Horner's method, evaluating a polynomial at n points take time $\Theta(n^2)$. we shall see later that if we choose the points x_k cleverly we can accelerate this computation to run in time $\Theta(n \log n)$.

Theorem 30.1

(Uniqueness of an interpolating polynomial)

For any set $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$ of n point-value pairs such that all the x_k values are distinct, there is a unique polynomial $A(x)$ of degree bound n such that $y_k = A(x_k)$ for $k = 0, 1, \ldots, n-1$.

Proof The proof relies on the existence of the inverse of a certain matrix. equation (30-3) is equivalent to the matrix eqn.

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

The matrix on the left is denoted $V(x_0, x_1, \ldots, x_{n-1})$ and is known as a vandermonde matrix. By problem D-1, this matrix has determinant.

$$\prod_{0 \le j < k \le n-1} (x_k - x_j).$$

The proof of Theorem 30.1 describes an algorithm for interpolation based on solving the set (30.4) of linear equations in time $O(n^3)$.

For $y_{n/2}, y_{n/2+1}, \ldots, y_{n-1}$, letting $k = 0, 1, \ldots, n/2-1$ lines.

12 yields $y_{k+(n/2)} = y_k^{[0]} - \omega_n^k y_k^{[1]}$.

$$= y_k^{[0]} + \omega_n^{k+(n/2)} y_k^{[1]} \quad (\text{since } \omega_n^{k+(n/2)} = -\omega_n^k)$$

$$= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k})$$

$$= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n})$$

$$= A(\omega_n^{k+(n/2)}) \quad (\text{since } \omega_n^{2k+n} = \omega_n^{2k})$$

Thus, the vector y returned by RECURSIVE-FFT is indeed the DFT of the input vector a.

Lines 11 and 12 multiply each value $y_k^{[1]}$ by ω_n^k, for $k = 0, 1, \ldots, n/2-1$.

Line 11 adds this product to $y_k^{[0]}$, and line 12 subtracts it. Because we use each factor ω_n^k in both its positive and negative forms, we call the factors ω_n^k *twiddle factors*.

To determine the running time of procedure RECURSIVE-FFT, we note that exclusive of the recursive calls, each invocation takes time $\Theta(n)$, where n is the length of the input vector. The recurrence for the running time is therefore

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

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

Thus, we can evaluate a polynomial of degree-bound n at the complex nth roots of unity in time $\Theta(n \log n)$ using the fast Fourier transform.

12 $y_{k+(n/2)} = y_k^{[0]} - w y_k^{[1]}$

13 $w = w w_n$

14 return y // y is assumed to be a column vector

The RECURSIVE-FFT procedure works as follows. Lines 2-3 represents the basis of the recursion, the DFT of one element is the element itself. since in this case. $y_0 = a_0 w_1^0$

$= a_0 \cdot 1$

$= a_0$

Line 6-7 define the coefficient vectors for the polynomials $A^{[0]}$ and $A^{[1]}$. Lines 4, 5, and 13 guarantee that w is updated properly so that whenever line 11-12 are executed, we have $w = w_n^k$.

8-9 perform the recursive DFT$_{n/2}$ computations. Setting for $k = 0, 1, \ldots, n/2-1$

$y_k^{[0]} = A^{[0]}(w_{n/2}^k)$

$y_k^{[1]} = A^{[1]}(w_{n/2}^k)$

or, since $w_{n/2}^k = w_n^{2k}$ by the cancellation lemma

$y_k^{[0]} = A^{[0]}(w_n^{2k})$.

$y_k^{[1]} = A^{[1]}(w_n^{2k})$

Lines 11-12 combine the results of the recursive DFT$_{n/2}$ calculations. For $y_0, y_1, \ldots, y_{n/2-1}$, line 11 yields

$y_k = y_k^{[0]} + w_n^k y_k^{[1]}$

$= A^{[0]}(w_n^{2k}) + w_n^k A^{[1]}(w_n^{2k})$

$= A(w_n^k)$

$A^{[1]}(x)$ at the points $(\omega_n^0)^2, (\omega_n^1)^2, \ldots (\omega_n^{n-1})^2$.
and then.

(2) combining the results according to equation (30.9).

By the halving lemma, the list of values (30.10) consists not of n distinct values but only of the $n/2$ complex $(n/2)^{th}$ roots of unity, with each root occurring exactly twice. Therefore, we recursively evaluate the polynomials $A^{[0]}$ and $A^{[1]}$ of degree-bound $n/2$ at the $n/2$ complex $(\frac{n}{2})^{th}$ roots of unity. These subproblems have exactly the same form as the original problem, but are half the size. We have now successfully divided an n-element DFT n computation in to two $n/2$-element DFT $n/2$ computations. This decomposition is the basis for the following recursive FFT algorithm, which computes the DFT of an n-element vector $a = (a_0, a_1, \ldots, a_{n-1})$, where n is a power of 2

algorithm

```
RECURSIVE-FFT(a)
1  n = a.length // n is a power of 2
2  if n == 1
3        return a
4  ω_n = e^(2πi/n)
5  ω = 1
6  a^[0] = (a_0, a_2, ..., a_{n-2})
7  a^[1] = (a_1, a_3, ..., a_{n-1})
8  y^[0] = RECURSIVE-FFT(a^[0])
9  y^[1] = RECURSIVE-FFT(a^[1])
10 for k = 0 to n/2 - 1
11       y_k = y_k^[0] + ω y_k^[1]
```

Example

$A(x) = x^2 + 2x + 1$

$[(-2,1), (-1,0), (0,1), (1,4), (2,9)]$

$B(x) = x^2 - 2x + 1$

$[(-2,9), (-1,4), (0,1), (1,0), (2,1)]$

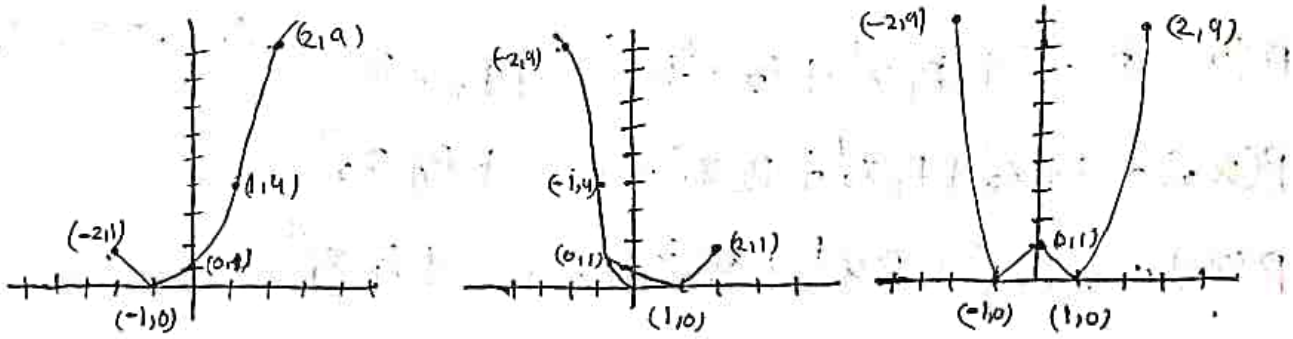
$C(x) = A(x) \times B(x)$

$[(-2,1), (-1,0), (0,1), (1,4), (2,9)]$

$\times [(-2,9), (-1,4), (0,1), (1,0), (2,1)]$

$[(-2, 1\times9), (-1, 0\times4), (0, 1\times1), (1, 4\times0), (2, 9\times1)]$

$\therefore C(x) = [(-2,9), (-1,0), (0,1), (1,0), (2,9)]$



$A(x) = a_0 x^0 + a_1 x^1 + \cdots + a_n x^n$

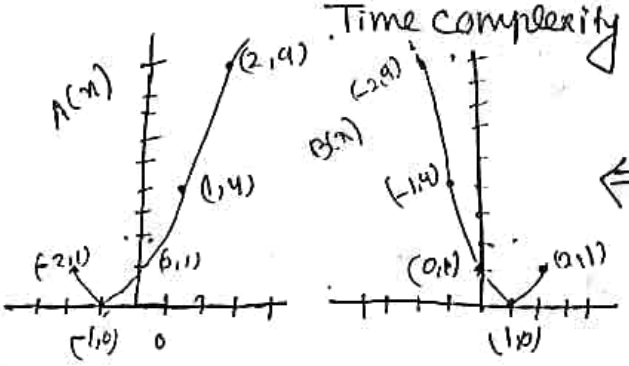
$B(x) = b_0 x^0 + b_1 x^1 + \cdots + b_n x^n$

$C(x) = c_0 x^0 + c_1 x^1 + \cdots + c_n$

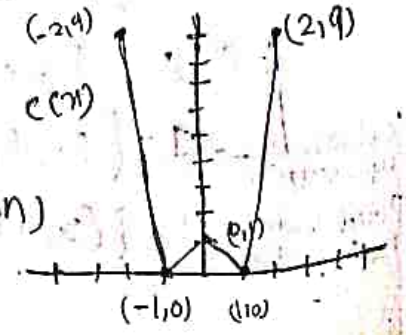
Co-efficient Representation $\Rightarrow$ Point value Representation (Evaluation) — Time complexity: $O(n^2)$ — Point value representation $\Rightarrow$ Co-efficient Representation (Interpolation)

(Evaluation) ⇓ FFT — Time complexity: $O(n \log)$

(Interpolation) ⇑ Time complexity:



⇐ point value multiplication ⇒ Time complexity: $O(n)$



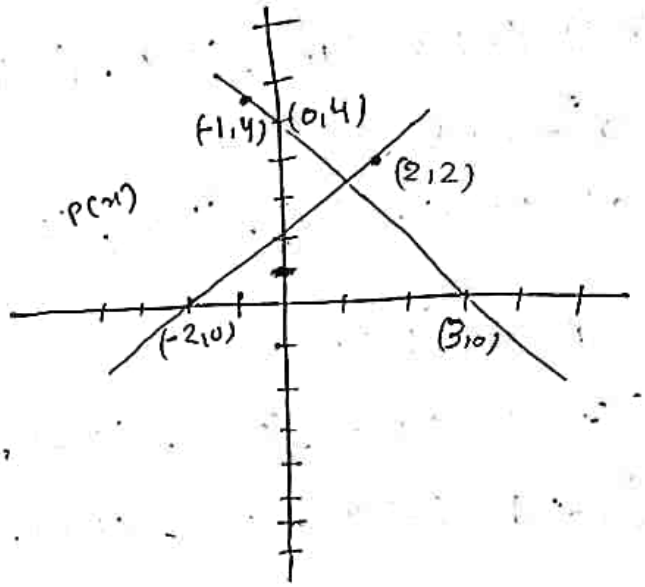
point value Representation $T(n) = O(n^2)$

$P(x) = P_0 + P_1 x$

$(-2,0)$, $(2,2)$.

$P(x) = 1 + 0.5x$.

$(3,0)$, $(-1,4)$



→ Atleast 2 points needed to represent a line on only plane

(n+1) arbitrary points uniquely define a degree n polynomial: $\{(x_0, P(x_0)), (x_1, P(x_1)), (x_3, P(x_3)), \ldots, (x_n, P(x_n))\}$.

$$P(x) = P_0 x^0 + P_1 x^1 + P_2 x^2 + \cdots + P_n x^n$$

$$P(x_0) = P_0 x_0^0 + P_1 x_0^1 + P_2 x_0^2 + \cdots + P_n x_0^n$$

$$P(x_1) = P_0 x_1^0 + P_1 x_1^1 + P_2 x_1^2 + \cdots + P_n x_1^n$$

$$\vdots$$

$$P(x_n) = P_0 x_n^0 + P_1 x_1^1 + P_2 x_n^2 + \cdots + P_n x_n^n$$

$$\begin{bmatrix} P(x_0) \\ P(x_1) \\ \vdots \\ P(x_n) \end{bmatrix} = \begin{bmatrix} x_0^0 & x_0^1 & x_0^2 & \cdots & x_0^n \\ x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^n \\ \vdots & & & & \\ x_n^0 & x_n^1 & x_n^2 & \cdots & x_n^n \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ \vdots \\ P_n \end{bmatrix}$$

Two unique Representation of polynomials

$$P(x) = P_0 x^0 + P_1 x^1 + P_2 x^2 + \cdots + P_n x^n$$

Coefficient representation → 1 - $\{P_0, P_1, \ldots, P_n\}$

Point value representation → 2 - $\{(x_0, P(x_0)), (x_1, P(x_1)), \ldots, (x_n, P(x_n))\}$

Interpolation at the complex roots of unity

We know complete the polynomial multiplication sche by showing how to interpolate the complex roots of unity by a polynomial, which enables us to convert from point-value form back to coefficient form. We interpolate by writing the DFT as a matrix equation and then looking at the form of the matrix inverse.

From equation (30.4), we can write the DFT as the matrix product $y = V_n a$, where V_n is a vandermonde matrix containing the appropriate powers of ω_n:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

The (k, j) entry of V_n is ω_n^{kj} for $j, k = 0, 1, \ldots, n-1$. The exponents of the entries of V_n form a multiplication table.

For the inverse operation, which we write as $a = DFT_n^{-1}(y)$, we proceed by multiplying y by the matrix V_n^{-1}, the inverse of V_n.

Evaluation

$P(x) = 3x^5 + 2x^4 + x^3 + 7x^2 + 5x + 1$

Evaluate at n points $\pm x_1, \pm x_2 \ldots \pm x_{\frac{n}{2}}$

$$P(x) = (2x^4 + 7x^2 + 1x^0) + (3x^5 + x^3 + 5x)$$

$$= \underbrace{(2x^4 + 7x^2 + 1x^0)}_{P_e(x^2)} + x\underbrace{(3x^4 + x^2 + 5x^0)}_{P_o(x^2)}$$

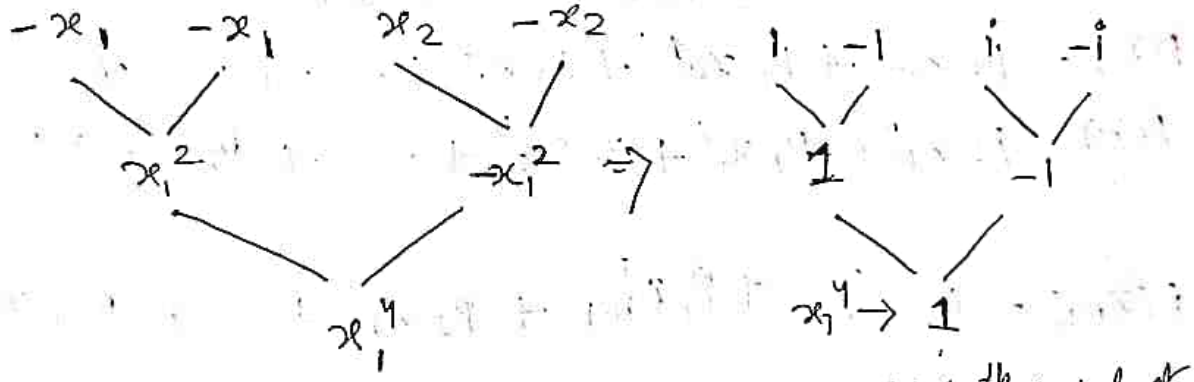
$$P(x) = P_e(x^2) + x\,P_o(x^2)$$

$$\left.\begin{aligned} P(x_1) &= P_e(x_1^2) + x_1 P_o(x_1^2) \\ P(x_j) &= P_e(x_j^2) - x_j P_o(x_j^2) \end{aligned}\right\}$$ lot of overlap.

where $P_e(x^2)$ and $P_o(x^2)$ have degree "n/2".

example

$P(x) = x^3 + x^2 + x + 1$



points are 4th root of unity.

Nth root of unity

$e^{i\theta} = \cos(\theta) + i\sin\theta$

$x^n = 1$

$\omega = e^{\frac{2\pi i}{n}}$

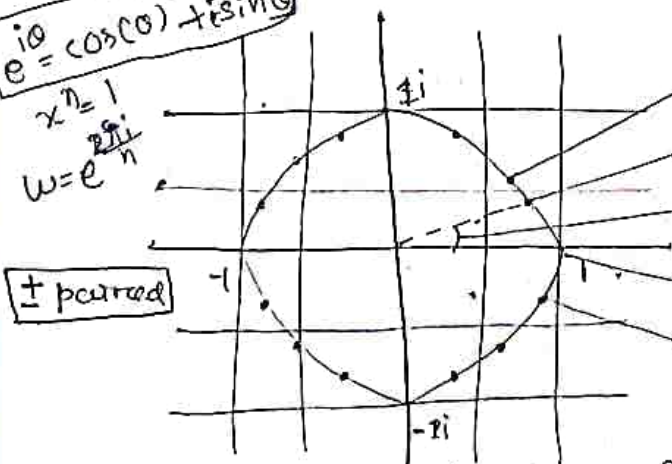
$\omega^2 = e^{\frac{2\pi(2)i}{n}}$

$\omega^1 = e^{\frac{2\pi i}{n}}$

angle $= \frac{2\pi}{n}$

$\omega^0 = 1$

$\omega^{n-1} = e^{\frac{2\pi(n-1)i}{n}}$

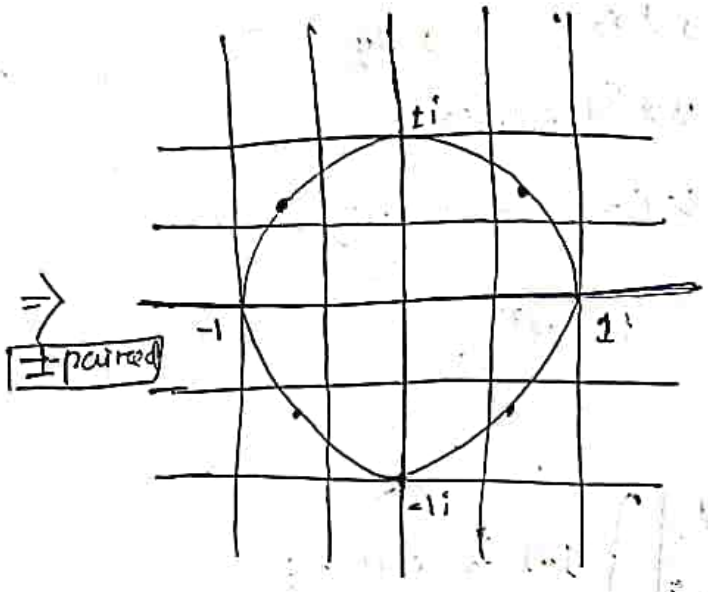


± paired

Evaluate $p(x)$ at $[\omega^0, \omega^1, \omega^2, \ldots, \omega^{n-1}]$

nth root of unity

$\omega^{i+\frac{n}{2}} = -\omega^i \Rightarrow (\omega^i, \omega^{i+\frac{n}{2}})$ are $\pm$ paired.



Evaluate $P_e(x^2)$ and $P_o(x^2)$ at $[\omega^0, \omega^2, \omega^4, \ldots, \omega^{4(\frac{n}{2}-1)}]$ $(\frac{n}{2})^{th}$ roots of unity.

Interpolation

$$P(x) = p_0 x^0 + p_1 x^1 + p_2 x^2 + \cdots + p_{n-1} x^{n-1}$$

$$P(x_0) = p_0 x_0^0 + p_1 x_0^1 + p_2 x_0^2 + \cdots + p_{n-1} x_0^{n-1}$$

$$P(x_1) = p_0 x_1^0 + p_1 x_1^1 + p_0 x_1^2 + \cdots + p_{n-1} x_1^{n-1}$$

$$\vdots$$

$$P(x_{n-1}) = p_0 x_{n-1}^0 + p_1 x_{n-1}^1 + p_2 x_{n-2}^2 + \cdots + p_{n-1} x_{n-1}^{n-1}$$

$$\begin{bmatrix} P(x_0) \\ P(x_1) \\ \vdots \\ P(x_{n-1}) \end{bmatrix} = \begin{bmatrix} x_0^0 & x_0^1 & x_0^2 & \cdots & x_0^{n-1} \\ x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & & & & \\ x_{n-1}^0 & x_{n-1}^1 & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{n-1} \end{bmatrix} \rightarrow \text{coefficients}$$

FFT

$x_k = \omega^k$, where $\omega = e^{\frac{2\pi i}{n}}$ → n = no. of pts.

$$\begin{bmatrix} P(\omega^0) \\ P(\omega^1) \\ P(\omega^2) \\ \vdots \\ P(\omega^{n-1}) \end{bmatrix} = \underbrace{\begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \cdots & \omega^0 \\ \omega^1 & \omega^1 & \omega^1 & \cdots & \omega^1 \\ \omega^2 & \omega^2 & \omega^2 & \cdots & \omega^2 \\ \vdots & & & & \\ \omega^0 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix}}_{\text{Discrete Fourier Transform (DFT) matrix}} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{n-1} \end{bmatrix}$$

$x_k = \omega^k$

↳ [redacted]

FFT (co-efficient Representation),

where $\omega = e^{\frac{2\pi i}{n}}$.

n → no. of points.

$$\begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ \vdots \\ P_{n-1} \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \cdots & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \cdots & \omega^{n-1} \\ \omega^0 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & & & & \\ \omega^0 & \omega^{(n-1)} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix}$$

$FFT([P_0, P_1, P_2 \cdots P_{n-1}]) \rightarrow [P(\omega^0), P(\omega^1), P(\omega^2) \cdots P(\omega^{n-1})]$

IFFT ([redacted]) → [redacted]

↗ inverse FFT

$IFFT([P(\omega^0), P(\omega^1) P(\omega^2), \cdots, P(\omega^{n-1})]) \rightarrow [P_0, P_1, P_2 \cdots P_n]$

IFFT (point value representation) $\Leftrightarrow$ FFT (point-value Representation)

↓ inverse FFT

where $\omega = \frac{1}{n} e^{\frac{2\pi i}{n}}$.

n → no. of points.

$A(x) = x^2 + 1$

$B(x) = x + x$

multiply = $x^2(x+x) + 1(x+1) \rightarrow O(n^2)$

↳ reduce [redacted] using divide and conquer [redacted]

point value → $O(n)$

FFT → coefficient → (→ $O(n)$) point value — point value

IFFT → point value → coefficient

Algorithm: Fast Fourier Transform (FFT)

```
1  FFT(P) // P = [P0, P1, P2, ... Pn-1] (co-efficient representation)
2    n = len(P) // n is a power of 2
3    if (n == 1)
4        return P
5    ω = e^(2πi/n)
6    Pe, Po = [P0, P2, P4 ... Pn-2], [P1, P3, P5 ... Pn-1]
```

7 $y_e, y_o = FFT(P_e), FFT(P_o)$.

8 $y = [0] * n$

9 for j in range $(\frac{n}{2})$:

10 $y[j] = y_e[j] + \omega^j y_o[j]$.

11 $y[j + \frac{n}{2}] = y_e[j] - \omega^j y_o[j]$

12 return y.

FFT: $P(x) = P_0 x^0 + P_1 x^1 + P_2 x^2 + \cdots + P_{n-1} x^{n-1}$

$P(x) = [P_0, P_1, P_2, \ldots, P_{n-1}]$

$\omega = e^{\frac{2\pi}{n}} : [\omega^0, \omega^1, \omega^2, \ldots, \omega^{(n-1)}]$

$n = 1 \Rightarrow P(1)$ ← Base condition ← line 3, 4 $\omega^0 = 1$, $e^{\frac{2\pi \cdot 0 \cdot i}{n}} = e^0$

FFT: $P_e(x^2) = [P_0, P_2, P_4, \ldots, P_{n-2}]$ // even degree

$[\omega^0, \omega^2, \ldots \omega^{n-2}]$

line 6, 7

$y_e = [P(\omega^0), P(\omega^2), \ldots P_e(\omega^{n-2})]$

FFT $= P_o(x^2) = [P_1, P_3, P_5, \ldots P_{n-1}]$ // odd degree

$[\omega^0, \omega^2, \ldots, \omega^{n-2}]$

$y_o = [P(\omega^0), P(\omega^2), \ldots, P(\omega^{n-2})]$.

line 9, 10, 11

$P(\omega^j) = y_e[j] + \omega^j y_o[j]$.

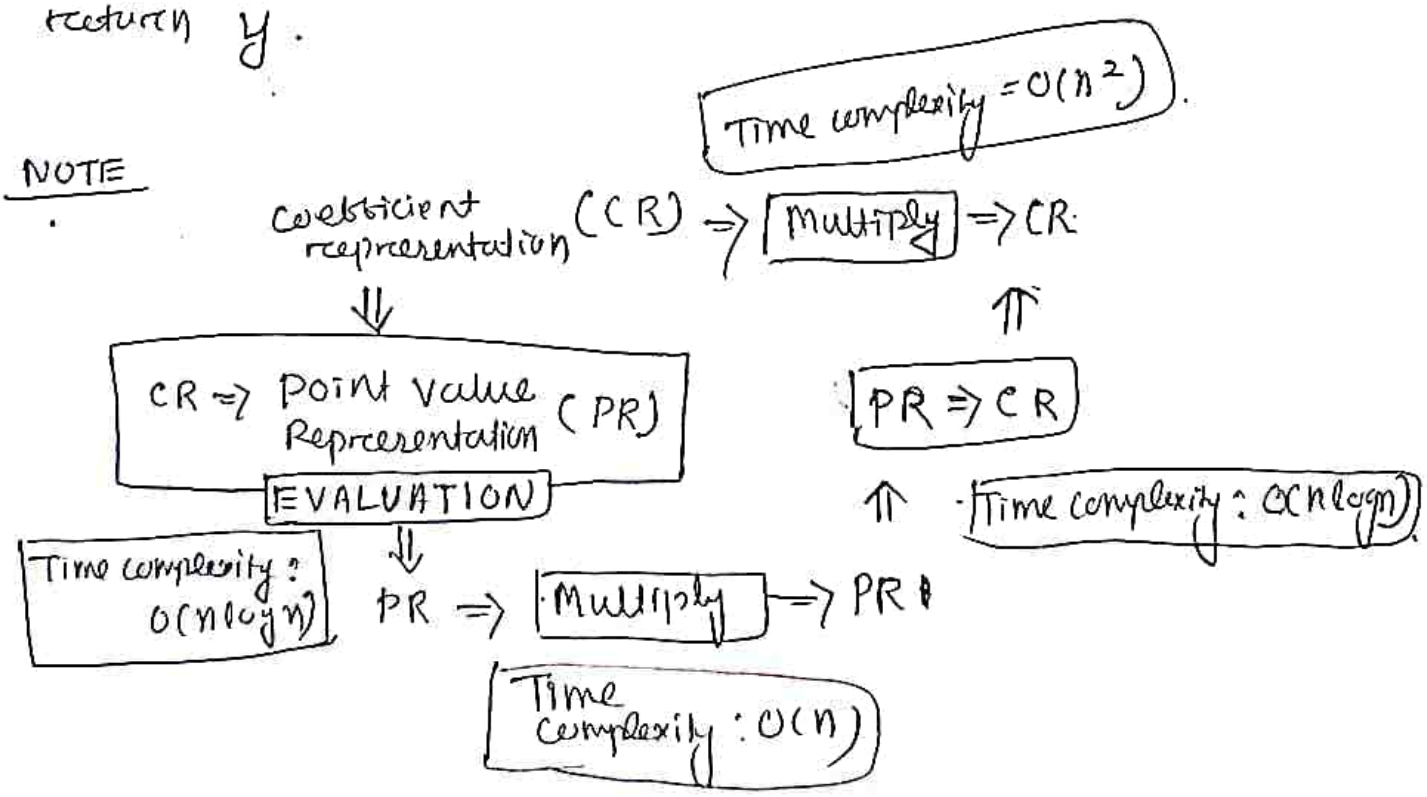
$P(\omega^{j + \frac{n}{2}}) = y_e[j] - \omega^j y_o[j]$ where $j \in \{0, 1, \ldots, (\frac{n}{2} - 1)\}$

line 12 $y = [P(\omega^0), P(\omega^1), \ldots P(\omega^{n-1})]$

Algorithm : Inverse Fourier transform (IFFT).

```
IFFT(P) // P = [P0, P1, P2, ---- Pn-1] (Coefficient representation)
n = length(P) // n is a power of 2.
if (n == 1),
  return P.
w = 1/n x e^(-2πi/n).
Pe, Po = [P0, P2, P4, ... Pn-2] ; [P1, P3, P5, --- Pn-1].
ye, yo = IFFT(Pe), IFFT(Po)
y = [0] x n
For j in range(n/2).
    y[j] = ye[j] + w^j yo[j].
    y[j + n/2] = ye[j] - w^j yo[j].
return y.
```

NOTE



The time complexity for multiply 2 polynomials using FFT and IFFT is.

$= O(2n \log n) + O(n)$

$= \boxed{O(n \log n)}$ which is less than $O(n^2)$.