Evaluating Algorithms

↓

performance evaluation

priori estimates (performance analysis) — posteriori Testing (performance measurement)

priori estimates (performance analysis):
- space complexity S(P)
- time complexity

if P is an algorithm

$S(P) = C + S_P$ (depends up on the input given at runtime)

(fixed part) (variable part)

- space for code
- space for variables
- space for constants.

→ C is independent of input and output characters.

→

Ex:-

```
Alg  Add (a,b,c)
{
     c ← a+b;
     return c;
}
```

a - 1word
b - 1word
c - 1word

$S(P) = C + S_P$ (no bops)

$= 3 + 0$

$= 3$ words.

Ex-2:-

```
Alg { sum(a,n)
{
    s ← 0;
    for{i ← 1 to n  do
        s ← s+a[i];
    return s;
}
```

s = 1 word
i = 1 word
n = 1 word

$S(P) = C + S_P$

$S(P) = 3 + n$

$S(P) \geq n+3$ words

$S(P) \geq O(n)$

Time complexity:-

* Amount of execution time required to execute an algorithm. $T(P)$

$$*T(P) = \underset{\text{compile time}}{C} + \underset{\text{Runtime.}}{R}$$

* It can be calculated by using frequency count method. * s/e → (steps for execution) How many times this step is executed.

* Frequency. (How many no. of times a statement or instruction executes)

if s/e = 0, it is not executed ({, name of algorithm)

s/e = 1, it is an executable instruction.

→ whether it is a executable statement or not is defined by s/e.

→ How many no. of times this instruction is executed is defined by frequency.

* Total steps = s/e * frequency.

	Algorithm	sle	frequency	Total steps = sle * frequency.
		0	0	0
1	Alg sum(a, n)	0	0	0
2	{	0	1	1
3	s ← 0;	1	1	1
	for i ← 1 to n do	1	n+1	n+1
4	s ← s + a[i];	1	n	n
5	return s;	1	1	1
6		0	0	0
7	}			2n+3 steps

Time complexity = O(n)

Matrix addition:-

```
Alg madd (a,b,c,m,n)
{
   a,b,c,m,n
   for i ← 1 to m do
      for j ← 1 to n do
         c[i][j] = a[i][j] + b[i][j];
   return c[i][j];
}
```

sle	frequency	Total steps = sle * frequency.
0	0	0
0	0	0
1	m+1	m+1
1	m(n+1)	mn+m
1	mn	mn
0	0	0
		2mn+2m+2

$2mn + 2m + 1$

if $m = n$

Time complexity = $O(n^2)$

else

Time complexity = $O(n^3)$

Matrix Multiplication :-

```
Alg mmul(a,b,c,n)
{
    for(i=1; i<=n; i++)
    {
        for(j=1; j<=n; j++)
        {
            c[i][j]:=0;
            for(k=1; k<=n; k++)
            {
                c[i][j] = c[i][j]+ A[i][k]+B[k][j];
            }
        }
    }
}
```

s/e	frequency	Total no of steps = s/e * frequency
0	0	0
0	0	0
1	$n+1$	$n+1$
0	$n(n+1)$ 0	0
1	$n(n+1)$	$n(n+1)$
0	0	0
1	$n*n$	n^2
1	$n^2(n+1)$	$n^2(n+1)$
0	0	0
1	n^3	n^3

$2n^3 + 3n^2 + 2n + 1$

$O(n^3)$

Find out the time complexity of sum of n natural numbers.

```
Alg   Add ( n)
{   sum := 0;
    for (i=1 ; i<=n ; i++)
      {
        sum = sum + i;
      }
    return sum;
}
```

Sle	Frequency	Total number of steps
0	0	0
0	0	0
1	n+1	n+1
1	n(n+1)	$n(n+1) = n^2 + n$
0	0	0
0	0	0
0	0	0

$n^2 + 2n + 1$

$O(n^2)$

sle	frequency	Total
Alg ~~Sum~~	0	0
0	0	0
0	1	1
1	n+1	n+1
1	0	0
0	n	n
1	0	0
0	1	1
1	0	0
0		
		2n+3
		O(n)

Find out the time complexity of factorial of a given number.

```
Alg  fact(n)
{
    fact = 1;
    for(i=1; i<=n; i++)
     {
        fact = fact*i;
     }
    return fact;
}
```

sle	Freq	Total.
0	0	0
0	0	0

1	1	1
1	n+1	n+1
0	0	0
1	n	n
0	0	0
1	1	1
0	0	0

$2n + 3$

$O(n)$

Find out the time complexity of sum of n odd natural numbers.

```
Alg  sum(n)
{  sum:= 0;
   for (i=1; i<=n ;i=i+2)
   {
        if(i%2!=0)
        {
           sum = sum+i;
        }
   }
   return sum;
}
```

S/e	Freq	Total.
0	0	0
0	0	0
1	1	1
1	n/2+1	n/2+1

0	0	0
1	n/2	n/2
0	0	0
0	0	0
1	1	1
0	0	0

$n+3$

$O(n)$.

Find out the (factorial) Time complexity of factorial of a given number using recursion.

Algorithm fact(n)

{

if (n ≤ 1)

return 1;

else

return n * fact(n-1);

}

Sle	Freq	Total.
0	0	0
0	0	0

Sle	Freq		Total.	
	n≤1	n>1	n≤1	n>1
0	0	0	0	0
0	0	0	0	0
1	1	1	1	1
1	1	0	1	0

0	0	0	0	0
1	0	1+(n-1)	0	1+n-1
0	0	0	0	0
			2	2+(n-1)

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

find out the time complexity of reverse of of a given number.

Algorithm rev(n, r)

```
{   rev = 0;
    r = 0;
    while(n>0):
      { r = n % 10;
        rev = rev * 10 + r;
        n = n / 10;
      }
      return rev;
}
```

* order of growth of functions :-

Running Time complexity depends on :-

input size

Nature of program

operations

→ order of growth of functions provides the behaviour of an algorithm

$O(1)$ → constant

$O(n)$ → linear

$O(\log n)$ → logarithmic

$O(n^2)$ → quadratic

$O(n^3)$ → cubic

$O(2^n)$ → exponential

$O(n!)$ = factorial

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$

* Asymptotic Notations (or) mathematical Notations

Big oh (O)

omega (Ω)

Theta (Θ)

little oh (o)

little omega (ω)

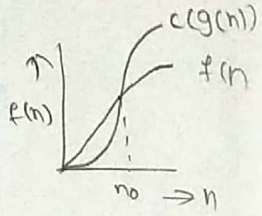
Big oh:

$f(n) = O(g(n))$

$f(n) \leq c\, g(n) \quad \forall n,\ n \geq n_0$, c constant and n_0 is given input.

upper bound

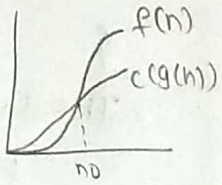
largest amount of time taken.



omega

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

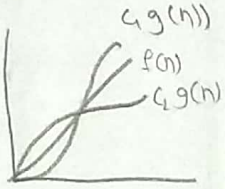
$f(n) \geq c(g(n))$



Theta:

$f(n) = \Theta(g(n))$ iff

$c_1 g(n) \leq f(n) \leq c_2 \cdot g(n)$



little oh:

$f(n) = o(g(n))$ iff.

$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$

little omega

$f(n) = \omega(g(n))$ iff.

$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$

Find out the space complexity of sum of n elements in array using Recursion :-

```
Algorithm Rsum (a, n)
{
    if (n <= 0) then
        return 0;
    else
        return Rsum (a[n-1] + a[n]);
}
```

The above recursive algorithm occupies
1 word - pointing to an array name
1 word - storing the return address
1 word - for variables

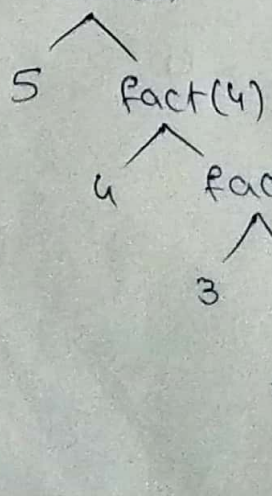
$$Sp \geq 3*(n+1)$$

(The above recursive algorithm occupies)

whenever a recursive call is executed, how many times a recursive call is executed

n+1 times a recursive call is executed

n=5 fact(5) . 6 times.



∴ The space complexity of the above algorithm is $S_p > 3*(n+1)$

n+1 is called depth of recursion.

In every recursive call it occupies 3 words that are mentioned above.

Ch-2

Divide & conquer

control abstraction of divide & conquer.

Algorithm:-

```
Algorithm DAC(P)
{
    if (P is too small) then
        return solution(P);
    else
    {
        Divide P into P1, P2, P3 .... Pk where
                                    Pk ≥ 1 subProblems;
        Apply DAC to each and every subProblem;
        return combine( DAC(P1), DAC(P2), ....
                                    .... DAC(Pk) );
    }
}
```

→ The recurrence relation for divide and conquer is

$$T(n) = \begin{cases} g(n) & \text{if } p \text{ is small} \\ T(n_1)+T(n_2)+\dots & \text{if } p \text{ is complex} \\ \quad +T(n_k)+f(n_k) & \end{cases}$$

where $T(n)$ is the time taken for performing the divide and conquer if the input size is n.

$g(n)$ is the time taken for solving problem p if p is a small problem.

$T(n_1), T(n_2) \dots T(n_k)$ are the time taken for solving each subproblem. if problem p is divided into $n_1, n_2, n_3 \dots n_k$ subproblems

$f(n)$ is the time taken for dividing problem p into $n_1, n_2, n_3 \dots n_k$ subproblem and time taken for combining the solution of all sub problems.

The above recurrence relation can be simply write as

$$T(n) = aT(n/b) + f(n)$$

wher a and b are constants.

The above formula is called as general divide & conquer recurrence relation.

Advantages:-

1) solving difficult problems by breaking the problem into subproblems.
2) extracting parallism by solving sub-problems on different processors
3) Efficient use of memory, all subProblems can be solved with in the cache and no need to access the main memory.
4) More accurate results can be obtained in the case of computations involving rounded arithmatic.

Applications of divide & conquer:-

Binary search
finding the max & min
merge sort
quick sort
strassen's matrix multiplicatio.

Binary search:-

Binary search is an efficient searching method, while searching the element using this method, the most essential thing is that the elements in the array should be in sorted order (ascending order)

An element which is to be searched from the list of elements stored in an array is called key element.

A[mid] be the mid element of an array A, then there are 3 conditions

that needs to be tested while searching the array using this method

1. If key = = A[mid] then the key element is present in the list.

2. If key < A[mid], then search the left sublist

3. If key > A[mid], then search the right sublist.

Algorithm:-

```
Algorithm Bsearch (A, key, low, high)
{
    [low := A[0];
     high := A[n-1];]
    if (low = = high)
    {
        if (key = = A[low])
        {
            return low;
        }
        else
        {
            return 0;
        }
    }
    else
    {
        mid := [(low+high)/2];
        if (key = = A[mid])
            return mid;
```

```
else if ( key < A[mid])
        return Bsearch (A, key, low, mid-1);
else
        return Bsearch (A, key, mid+1, high);
```

Ex:-

Apply the recursive binary search on the following elements 10, 20, 30, 40, 50, 60, 70

key element = 60

```
low                                  high
10   20   30   40   50   60   70
0    1    2    3    4    5    6
                          key element = 60
```

$$mid = \left\lfloor \frac{0+6}{2} \right\rfloor = 3, \quad a[3] = 40$$

```
low            mid            high
10   20   30   40   50   60   70
```

key element 60 is compared with a[3] = 40

key > a[mid]

60 > 40 in the right sublist

Now we have to search the key element

```
                low                  high
10   20   30   40 [ 50   60   70 ]
                    4              6
```

The right sublist contains 3 elements

$$mid = \frac{4+6}{2} = 5 \qquad a[5] = 60$$

```
                low   mid    high
10   20   30   40   50   60   70
```

The key element 60 is compared with a[mid] = 60, our key element 60 is found at 5th position in an array. Here the search is successful.

key == a[mid]

60 == 60

mid = 5

Advantages:-

Binary search is an optimal search algorithm. i.e; search the desired elemen very efficiently.

Disadvantage:-

This algorithm requires the list to be in sorted order.

Applications:-

* It is used to search the desired record from the data base efficiently.
* for solving Non-linear equations with one known, this method is used.

Analysis or time complexity of Binary search:-

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \text{ (list contains a single elemen} \\ T(n/2)+1 & \text{if } n > 1 \text{ (list contains mo than one elemen} \end{cases}$$

This is the recurrence relation for binary search with input size n

where $T(n)$ = Time required for performing for binary search with input size n

$T(n/2)$ = Time required to search the key element either in the left sublist or right sublist.

$T(n/2) + 1$

↳ only one comparision is required for comparing the key ele and mid ele.

$$* \quad T(n) = T(n/2) + 1 \rightarrow (1)$$

(Analysis of Time complexity of Binary search:-)

sub $n/2$ in place of n in equ (1)

$$T(n/2) = T(n/4) + 1 \rightarrow (2)$$

sub $n/4$ in place of n in equ (1)

$$T(n/4) = T(n/8) + 1 \rightarrow (3)$$

sub equ (2) in equ (1)

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

$$T(n) = T(n/4) + 1 + 1 \rightarrow (4)$$

sub equ (3) in equ (4)

$$T(n) = T(n/8) + 1 + 1 + 1$$

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

$$\vdots$$

$$* \quad T(n) = T(n/2^k) + k \rightarrow (5)$$

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

$$n = 2^k$$

Apply log on both sides

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

$$\log n = k \log_2 2$$

$$\log n = k$$

sub k value in eqn (5)

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

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

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

$$* \quad T(n) = O(\log n)$$

The time complexity of binary search is

best case : O(1) (it occurs if the list contains a single element)

Average case : O(log n)

worst case : O(log n)

Apply the recursive binary search algorithm for the following elements to search the key element 151, -14, 7

List :- -15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151

Given list

low													h
0	1	2	3	4	5	6	7	8	9	10	11	12	13
-15	-6	0	7	9	23	54	82	101	112	125	131	142	151

(i) key element = 151

$$Mid = \frac{0+13}{2} = 6 \quad a[6] = 54$$

key == 54 False

key > 54 True

151 > 54 we have to search right sublist, ignore left sublist

low = mid+1 = 7

7	8	9	10	11	12	13
82	101	112	125	131	142	151

$$mid = \frac{7+13}{2} = 10 \quad a[10] = 125$$

key == 125 False

key > 125 True

low = mid+1 = 11

131 142 151

mid = $\frac{11+13}{2}$ = 12

a[mid] = 142

key == a[mid] = false

key > a[mid] True.

low = mid + 1 = 13

13

151

key == a[mid] = True.

The key element 151 is compared with a[mid] = 151 our key element is found at 13th position. Hence, search is successful.

(ii)

Given list.

0 1 2 3 4 5 6 7 8 9 10 11 12 13

-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151

mid = $\frac{\text{low} + \text{high}}{2}$ = $\frac{0+13}{2}$ = 6 a[6] = 54

keyelement = -14

key == a[6] False

key > a[6] false

key < a[6] true.

high = mid - 1 = 5

0 1 2 3 4 5

-15, -6, 0, 7, 9, 23

mid = $\frac{0+5}{2}$ = 2 a[2] = 0

key == a[2] false

key > a[2] false

key < a[2] true

high = mid - 1 = 2 - 1 = 1

0 1
-15 -6

mid = $\frac{0+1}{2}$ = 0 a[0] = -15

key == a[mid] false

key < a[mid] false

key > a[mid] true.

high = mid - 1

Element not found.

(iii) Given list

0 1 2 3 4 5 6 7 8 9 10 11 12
-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142

keyelement = 7.

mid = $\left\lfloor \frac{low+high}{2} \right\rfloor$ = $\frac{0+13}{2}$ = 6

A[mid] = A[6] = 54.

keyelement = 7 is compared with A[m

7 < 54 we have to search the left sublist ighore the rightsubli

-15, -6, 0, 7, 9, 23 | 54 | 82 101 112 125 131,

mid

left sublist — right sublist.

consider left sublist — Ignore rightsub

0 1 2 3 4 5
-15 -6 0 7 9 23

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

A[mid] = A[2] = 0

key > A[mid] i.e; 7 > 0

-15 -6 | 0 | 7 9 23

left mid right

key ele 7>0 so key element is there in right sublist, Ignore left sublist

3 4 5

7 9 23

$mid = \left[\frac{3+5}{2}\right] = 4$ A[4] = 9

key < A[mid] i.e; 7 < 9

3 4 5

7 | 9 | 23

left Right

- high = mid - 1

3

7 mid = 3

key = = A[mid]

7 = = A[3]

7 = = 7

key element found at A[mid] = A[3]

i.e; 3rd position

search is successful.

* Merge sort:-

Merge sort is an example of divide and conquer technique.

It sorts a given array $A[0, \ldots n-1]$ by dividing into two sub-parts

$$A[0 \ldots n-1]$$

$$A[0 \ldots \lfloor n/2 \rfloor - 1] \qquad A[\lfloor \tfrac{n}{2} + 1 \rfloor \ldots n-1]$$

→ sorting each of the sub part recursively and then merging the two smaller sorted arrays into a single sorted sequence of n elements.

→ To sort a sequence of n elements the following steps are followed:-

1. Divide the sequence into two sequences of length $\frac{n}{2}$ and $\frac{n}{2}$
2. Recursively sort each of the two subsequence
3. Merge the sorted subsequences to obtain the final sorted sequence.

Algorithm:-

```
Algorithm mergesort(low, high)
// a[low:high] is an array to be sorted
{
    if (low < high)
    { mid := ⌊(low+high)/2⌋;
```

```
        Mergesort (low, mid);
        Mergesort (mid+1, high);
        merge (low, mid, high);
    }
}
Algorithm merge (low, mid, high)
{
    h := low;
    i := low;
    j := mid+1;
    while((h ≤ mid) && (j ≤ high))
    {
        if (a[h] < a[j])
        {
            b[i] = a[h];
            h++;
        }
        else
        { b[i] = a[j];
            j++;
        }
        i++;
    }
    if (h > mid)
    { for (k = j to high) do
        {
            b[i] = a[k];
            i++;
```

```
    }
}
else
{
    for (k = low to mid) do
    {
        b[i] = a[k];
        i++;
    }
}
for (k = low to high) do
{
    a[k] = b[k];
}
}
```

Apply the recursive mergesort Algorithm and sort the following elements.

~~8 10 2 85~~

310 285 179 652 351 423 861 254 450 520

low = 0, high = 9

310	285	179	652	351	423	861	254	450	520
0	1	2	3	4	5	6	7	8	9

$$mid = \left\lfloor \frac{0+9}{2} \right\rfloor = 4$$

310	285	179	652	351	423	861	254	450	520
0 low	1	2	3	4 high	5 low	6	7	8	9 high

$$mid = \left\lfloor \frac{0+4}{2} \right\rfloor = 2$$

$$mid = \left\lfloor \frac{5+9}{2} \right\rfloor = 7$$

310	285	179	652	351	423	861	254	450	520
0 low	1	2 high	3 low	4 high	5 low	6	7 high	8 low	9 high

mid = ⌊(0+2)/2⌋ = 1 | mid = ⌊(3+4)/2⌋ = 3 | mid = ⌊(5+7)/2⌋ = 6 | mid = ⌊(8+9)/2⌋ = 8

310 285 179 | 652 351 | 423 861 254 | 450 520

mid = ⌊(0+1)/2⌋ = 0 | mid = ⌊(5+6)/2⌋ = 5

310 285 179 | 652 351 | 423 861 254 | 450 520

285 310 179 | 423 861 254

179 285 310 351 652 | 254 423 861 450 520

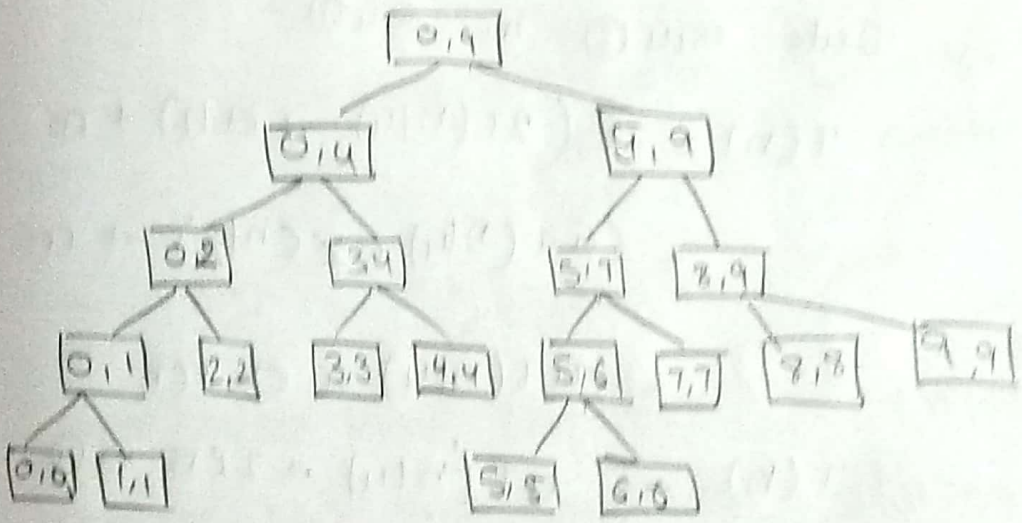
low mid mid+1 high | mid+1 high

b[i] = [179, 285, 310, 351, 652] | b[i] = [254, 423, 450, 520]

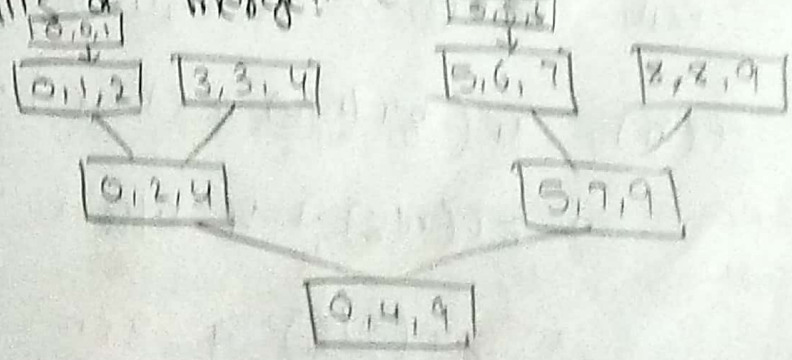
low mid | mid+1 high

[179 254 285 310 351 423 450 520 652 861]

Tree calls of merge sort: (low, mid)



Tree calls of merge: (low, mid, high)



Analysis or time complexity of merge sort

$$T(n) = \begin{cases} a & \text{if } n=1 \text{ (list contains only element)} \\ 2T(n/2) + cn & \text{if } n>1 \text{ (list contains more than one element)} \end{cases}$$

$2T(n/2)$ ← Time required for sorting two sublists.

cn → Time required for merging two sorted sublists.

$$T(n) = 2T(n/2) + cn \rightarrow (1)$$

Substituting $n/2$ in place of n in (1)

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

$$T(n/2) = 2T(n/4) + cn/2 \rightarrow (2)$$

Substituting $n/4$ in place of n in (1)

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

$$T(n/4) = 2T(n/8) + cn/4 \rightarrow (3)$$

Sub eqn (2) in eqn (1)

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

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

$$= 4T(n/4) + cn + cn$$

$$T(n) = 4T(n/4) + 2cn \rightarrow (4)$$

Sub eqn (3) in eqn (4)

$$T(n) = 4(2T(n/8) + cn/4) + 2cn$$

$$= 8T(n/8) + 4cn/4 + 2cn$$

$$= 8T(n/8) + 3cn$$

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

$\vdots$

$T(n) = 2^k T(n/2^k) + kcn \rightarrow (5)$

Let $n/2^k = 1$

$n = 2^k$

Apply log on both sides

$\log n = \log_2 2^k$

$= k \log_2 2 = k * 1$

$\log n = k$

Substituting k value in eqn (5)

$= n T(2^k/2^k) + \log n\, cn$

$= nT(1) + cn \log n$

$= na + cn \log n$ ($\because$ a & c are constants)

$= n + n \log n$

* $T(n) = O(n \log n)$

Best case time complexity : $O(n \log n)$

Average case time complexity : $O(n \log n)$

Worst case time complexity : $O(n \log n)$

Finding the Maximum & Minimum :-

In divide and conquer Technique, the list of elements is divided at the mid point in order to obtain two sublists.

From the both subtists, Maximum &

minimum elements

Two maximum & minimum elements are compared and from them real max and minimum elements are determined. This process is carried out for the entire list in a recursive manner.

Algorithm:-

```
Algorithm Maxmin(low, high, max, min)
// problem description : finding maximum
   and minimum elements recursively
// input : low, high are used as index to
           array A.
// output : The max and min will contain
           the maximum and minimum elements
           from the given array A.
{
  if(low == high)        (array contains single element)
   {
     min
     max = A[low];
     max = A[high];
   }
  else if ( low == high-1)
   {
     if ( A[low] < A[high] )
      {  max = A[high];
         min = A[low];
      }
```

```
else
{
    max = A[low];
    min = A[high];
}
}
else
{
    mid = ⌊(low + high)/2⌋ ;
    MaxMin(low, mid, max, min);
    MaxMin(mid+1, high, max1, min1);
    if (max < max1)
    {
        max = max1;
    }
    else
    {
        max = max;
    }
    if (min > min1)
    {
        min = min1;
    }
    else
    {
        min = min;
    }
}
```

Ex:-

Consider a list of some elements from which find out maximum & minimum elements.

low							high
50	40	-5	-9	45	90	65	25	75
0	1	2	3	4	5	6	7	8

$$mid = \left\lfloor \frac{(low + high)}{2} \right\rfloor = \left\lfloor \frac{0+8}{2} \right\rfloor = 4$$

low				mid	mid+1			high
50	40	-5	-9	45	90	65	25	75
					5	6	7	8

$$mid = \left\lfloor \frac{0+4}{2} \right\rfloor = 2 \qquad mid = \left\lfloor \frac{5+8}{2} \right\rfloor = 6$$

0	1	2	3	4				
50	40	-5	-9	45	90	65	25	75
low		mid	mid+1	high.	5	6	7	8

$$mid = \left\lfloor \frac{0+2}{2} \right\rfloor = 1 \qquad mid = \left\lfloor \frac{5+6}{2} \right\rfloor =$$

0	1	2	3	4				
50	40	-5	-9	45				
low	mid	mid+1 high						

$max = 50$ $\quad max_1 = -5$ $\quad max_1 = 45$ $\quad max_1 = 90$ $\quad max_2 = 75$

$min = 40$ $\quad min_1 = -5$ $\quad min_1 = -9$ $\quad min_1 = 65$ $\quad min_2 = 25$

$max = 50$

$min = -5$

$max_1 = 90$

$min_1 = 25$

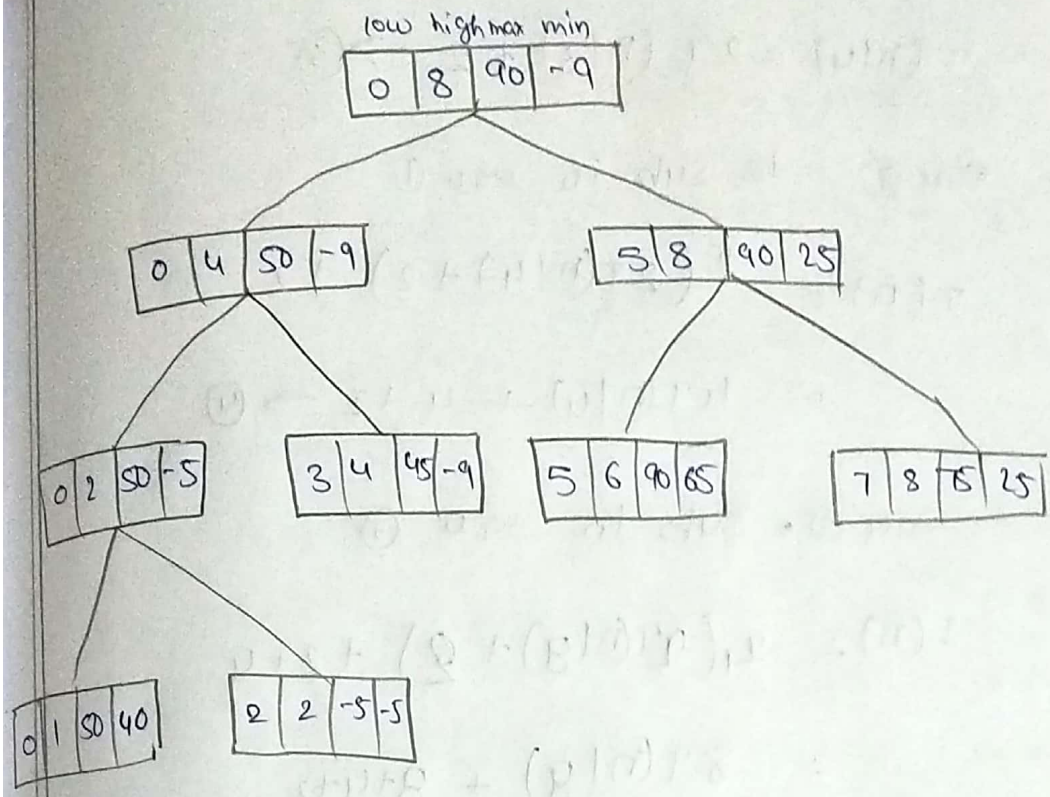
$max = 50$

$min = -9$

$max = 90$

$min = -9$

Tree calls for finding maximum & minimum:-



Analysis or Time complexity of maximum & minimum Algorithm:-

The amount of time required to find maximum & minimum is defined by the following recurrence relation

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

$T(n) = 2T(n/2) + 2 \rightarrow ①$

Sub $n/2$ in place of n in ①

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

$T(n/2) = 2T(n/4) + 2 \rightarrow ②$

Sub $n/4$ in place of n in ①

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

$T(n/4) = 2T(n/8) + 2 \rightarrow ③$

eqn ② is sub in eqn ①

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

$= 4T(n/4) + 4 + 2 \rightarrow ④$

eqn ③ sub in eqn ④

$T(n) = 4(2T(n/8) + 2) + 2 + 4$

$= 8T(n/8) + 8 + 4 + 2$

$= 8T(n/8) + 2^3 + 2^2 + 1$

$= 8T(n/8) + 2^1 + 2^2 + 2^3$

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

$T(n) = 2^k T(n/2^k) + 2^1 + 2^2 + 2^3 + \ldots + 2^k \rightarrow$

Let $n/2^k = 2$

$n = 2 \cdot 2^k$

$n = 2^{k+1}$

apply log on both sides

$\log n = \log_2 2^{k+1}$

$= (k+1) \log_2 2$

$\log n = k+1 \quad (1)$

$\log n = k+1$

$\Rightarrow k = \log n - 1$

k value is substituted in eq (5)

$$T(n) = 2^{\log n - 1} T(2) + 2^1 + 2^2 + 2^3 + \cdots + 2^{\log n - 1}$$

$$= \frac{2^{\log_2 n}}{2} \cdot 1 + 2^1 + 2^2 + 2^3 + \cdots + 2^{\log n - 1}$$

$$= \frac{2^{\log_2 n}}{2} + \left[2^1 + 2^2 + 2^3 + \cdots + 2^{\log n - 1}\right]$$

$\because a^{\log_c b} = b^{\log_c a}$

$= n^{\log_2 2}$

$= n \times 1$

$= n$

$$= \frac{n}{2} + \left[2^1 + 2^2 + \cdots + 2^{\log n - 1}\right]$$

GP

$a = 2$

$r = \frac{2^2}{2} = 2$

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

$$= \frac{n}{2} + \left[\frac{2\left(2^{\log_2 n - 1} - 1\right)}{2 - 1}\right]$$

$$= \frac{n}{2} + \left[\frac{2 \cdot 2^{\log_2 n - 1} - 2}{1}\right]$$

$$= \frac{n}{2} + \left[\frac{2 \cdot 2^{\log_2 n}}{2} - 2\right]$$

$$= \frac{n}{2} + \left[2^{\log_2 n} - 2\right]$$

$$= \frac{n}{2} + [n - 2]$$

$$= \frac{3n}{2} - 2$$

Time complexity $= O(n)$

* Quick sort:-

Quick sort is a sorting algorithm that uses divide and conquer strategy.

The three steps of the quick sort are as follows:-

* Divide:-

The given array splitted into two sub arrays based on the pivot element. All the elements that are less than pivot element should be in left sub array and all the elements that are greater than the pivot element should be in right sub array.

* Recursively sort the two sub arrays.

* conquer:-

→ combine the two sorted sub arrays into a single sorted array.

→ Quick sort is the fastest known sorting algorithm among all sorting algorithms

→ In Quick sort the worst case occurs, when all the elements are already sorted.

→ In quick sort the worst case also occurs when the pivot element is a minimum or maximum of all the elements in the array.

Algorithm:-

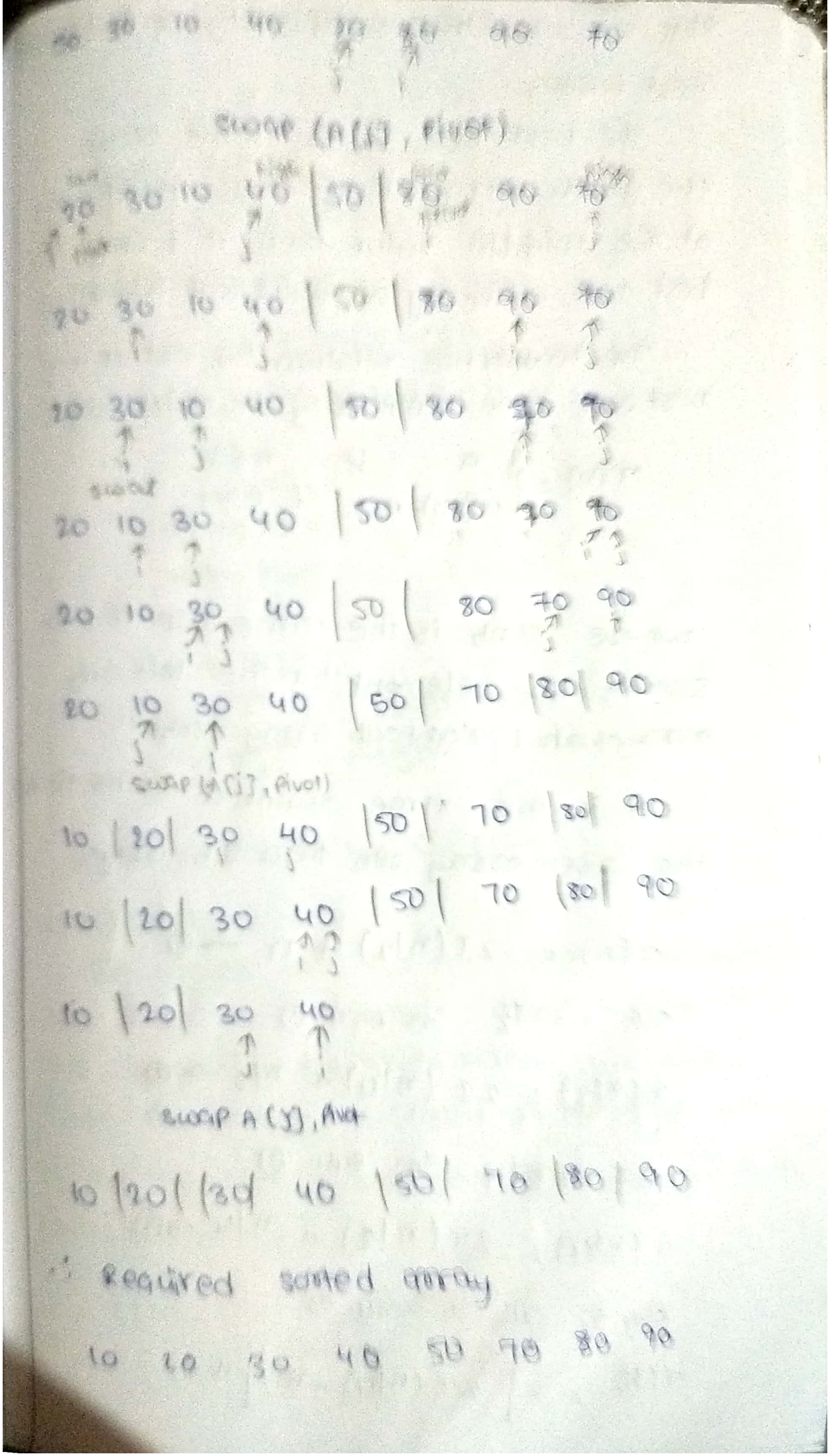
```
Algorithm Quicksort (A, low, high)
// problem description:- we have to sort given elements
in the array using quicksort.
// input:- The given array A contains two indices
low and high.
// output:- The finalresult is the sorted array of
given elements.
{
   if (low < high)
   {
      m ← Partition (A, low, high);
      Quicksort (A, low, m-1);
      Quicksort (A, m+1, high);
   }
}
Algorithm Partition(A, low, high)
{
   pivot = A[low];
   i := low;
   j := high;
   while (i <= j) do
   {
      while (A[i] ≤ pivot ) do
            i++;
      while (A[j] ≥ pivot) do
            j--;
      if (i <= j) then
            swap (A[i], A[j]);
   }
```

```
if (i>j)
{
    swap(A[j], pivot);
    return j;
}
}
```

Apply Quicksort for the following elements

50	30	10	90	80	20	40	70
i 0	1	2	3	4	5	6	7 j
50	30	10	90	80	20	40	70
low pivot							high

50	30	10	90	80	20	40	70
	i						j
50	30	10	90	80	20	40	70
		j					j
50	30	10	90	80	20	40	70
			i				j
50	30	10	90	80	20	40	70
			i			j	
50	30	10	40	80	20	90	70
			i			j	
50	30	10	40	80	20	90	70
				i		j	
50	30	10	40	80	20	90	70
				i	j		
50	30	10	40	20	80	90	70
				i	j		
50	30	10	40	20	80	90	70
					i	j	



Analysis or Time complexity of Quick sort

Best case:-

The best case in Quicksort occurs, if the given array is always partitioned at the middle value then it brings the best case efficiency of quicksort algorithm.

The recurrence relation for obtaining best case time complexity in quick sort is

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

where $2T\,n/2$ is the time required for sorting the elements in the left sub array $T(n/2)$ and right sub array $T(n/2)$

n is the time required for partitioning the given array into two sub arrays.

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

sub $n = n/2$ in equ (1)

$$T(n/2) = 2T(n/4) + n/2 \rightarrow (2)$$

sub $n = n/4$ in equ (1)

$$T(n/4) = 2T(n/8) + n/4 \rightarrow (3)$$

equ (2) sub in equ (1)

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

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

$T(n) = 4T(n/4) + 2n \rightarrow ④$

sub eqn ③ in ④

$T(n) = 4[2T(n/8) + n/4] + 2n$

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

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

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

$n/2^k = 1$

$n = 2^k$

apply log on b.s

$\log n = k \log_2 2$

$\log n = k$

$\Rightarrow k = \log n$

$T(n) = 2^{\log n} T(1) + (\log n)\, n$

$T(n) = an + n \log n$

$\Rightarrow$ Time complexity $= O(n \log n)$

worst case:-

The worst case for quicksort occurs when the pivot is minimum or maximum of all the elements in the given array. This happens when the array is sorted or reverse sorted.

The recurrence relation for quick sort in worst case is

$$T(n) = \begin{cases} a & \text{if } n = 1 \\ T(n-1) + cn & \text{if } n > 1 \end{cases}$$

$T(n) = T(n-1) + cn \rightarrow ①$

[Put $n = n/2$ in ①

$T(n/2) = T(\frac{n}{2} - 1) + c\, n/2$]

$n = n-1$ sub in eqn ①

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

$T(n-1) = T(n-2) + c(n-1) \rightarrow ②$

Sub $n = n-2$ in eqn ①

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

$T(n-2) = T(n-3) + c(n-2) \rightarrow ③$

Sub eqn ③ in ②

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

Sub ④ in ①

$T(n) = T(n-3) + c(n-2) + c(n-1) + cn$

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

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

$= T(n-3) + 3c(n-1)$

$T(k) = T(n-k) + kcn - kc \rightarrow ⑤$ $(1+3+6+ \ldots (k-1)$

$= T(n-k) + kcn - \left(\frac{k(k-1)}{2}\right)c$

$n-k = 1$

$k = n-1$

k value substituted in ⑤

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

$$= T(n-n+1) + (n^2-n)c - \frac{c(n-1)(n-2)}{2}$$

$$= T(1) + cn^2 - cn - \frac{c(n^2-3n+2)}{2}$$

$$= a + cn^2 - cn + \frac{c(n^2-3n+2)}{2}$$

Time complexity = $O(n^2)$

Average case :-

Average case occurs in quicksort when we taken a random array.

This is almost equal to best case.

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

∴ Best case = $O(n \log n)$

Average case = $O(n \log n)$

worst case = $O(n^2)$

* Master Theorem :-

Master Theorem provides a solution to the recurrence relation of the form

$$T(n) = aT(n/b) + f(n)$$

where a, b are constants and $a \geq 1, b \geq 1$

→ If $f(n) \in O(n^d)$ then

case i) :-

$$T(n) = O(n^d) \quad \text{if } a < b^d$$

case (ii) :-

$$T(n) = O(n^d \log n) \quad \text{if } a = b^d$$

case (iii)

$$T(n) = O(n^{\log_b a}) \quad \text{if } a > b^d$$

Ex:-

1) Find out the time complexity of the follo recurrence relations

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

$T(n) = a \cdot T(n/b) + f(n) \rightarrow ①$

the above recurrence relation is compared with ①

$a = 2 \quad b = 2 \quad f(n) = n$

$f(n) \in O(n^d) = O(n)$

$\Rightarrow d = 1$

$a = b^d$ (case ii)

$2 = 2^1$

$2 = 2$

∴ Time complexity $= O(n^d \log n)$

$* \; T(n) = O(n^1 \log n)$

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

$a = 4 \quad b = 2 \quad f(n) = n^2$

$f(n) \in O(n^d)$

$n^2 = n^d$

$d = 2$

$a = b^d$

$4 = 2^2$

$4 = 4$ (case ii)

Time complexity $= O(n^d \log n) = O(n^2 \log n)$

3) $T(n) = 8T(n/8) + n^3$

$a = 8 \quad b = 8 \quad f(n) = n^3$

$f(n) \in O(n^d)$

$n^3 = n^d$

$d = 3$

$a \quad b^d$

$8 < 8^3$

$8 < 512$ (case i) $a < b^d$

Time complexity $T(n) = O(n^d) = O(n^3)$

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

$a = 7 \quad b = 2 \quad f(n) = n$

$f(n) \in O(n^d)$

$n \in n^d$

$d = 1$

$7 > 2^1 \quad a > b^d$ case (iii)

Time complexity $= O(n^{\log_b a})$

$= O(n^{\log_2 7})$

UNIT-II

GREEDY METHOD

General method:

→ The Greedy method is a straight forward method. This method is popular for obtaining optimal solutions.

→ Greedy method works in stages by considering only one input at a time. In each stage, a decision is made regarding whether a particular input is a feasible solution or not.

→ If the inclusion of the next input into the particular constructed optimal solution will result in an infeasible solution. Then this input is not added to the particular solution, otherwise it is added.

→ These problems have 'n' inputs and require us to obtain a subset that satisfies some constraints.

→ Any subset that satisfies these constraints is called a "feasible solution."

→ A feasible solution either maximises or minimises the given objective function is called a "optimal solution".

→ there are two types of paradigms.

* subset paradigm
* ordering paradigm

→ In subset paradigm no grantee that all inputs are included.

→ In ordering paradigm all inputs should be present but they can be ordered in its own way.

Examples of subset paradigm are

* knapsack problem
* Job sequencing with deadlines problem
* Minimum cost spanning trees.

Examples of ordering paradigm are

* optimal storage on tapes.
* optimal merge pattern
* single source shortest path problem

→ control abstraction for greedy method

```
Algorithm Greedy(a,n)
// a[1:n] contains n number of inputs
{
    solution:= 0 ;
    for i = 1 to n do
    {
        x = select(a);
        if(feasible(solution, x)) then
        {
            solution= union(solution, x);
        }
    }
    return solution;
}
```

→ In greedy method, the following functions are used.

1) The function 'select' selects an input from the given array and removes that input from that array.

2) The selected input is assigned to the variable x.

3) 'feasible' is a boolean valued function that determines whether x can be included into the solution vector or not.

4) If it is included, the function 'union' combines input x, with the solution and update the objective function.

5) The feasible solution with maximum value or minimum value can be treated as optimal solution.

Differences between divide & conquer and greedy method:-

Divide & conquer	Greedy method.
1) Divide & conquer is used to obtain a solution to the given problem.	1) Greedy method is used to obtain a optimal solution to the given problem.
2) In this technique the given problem is divided into small sub problems. These sub problems are solved independently. finally, all the solutions of subproblems	2) In greedy method, a set of feasible solutions are generated and from them we pick up the optimal solution.

are collected together to get the solution to the given problem.	
3) In this method duplications in sub-problems may occur.	3) In greedy method, the optimal solution is generated without revising previously generated solutions.
4) Divide and conquer is less efficient because of rework on solutions.	4) Greedy method is comparatively efficient than divide & conquer.
Ex:- Merge sort, binarysearch, finding min and max, quick sort.	Ex:- Knapsack problem, Job sequencing with deadlines problem, MST, optimal storage on tapes, optimal merge pattern, single source shortest path problem.

* Knapsack or fractional knapsack problem.

→ The knapsack problem can be stated as follows. Suppose there are 'n' objects from $i = 1, 2, 3 \ldots n$. Each object i has some weight w_i & some profit value is associated with each object i's denoted as P_i and consider one knapsack (or) a bag with maximum capacity m. our main objective is to fill the knapsack by placing objects with maximum profit.

→ If a fraction x_i, $0 \leq x_i \leq 1$ of object i is placed into the knapsack, then the profit of $P_i \cdot x_i$ is earned.

1. If the object is completely placed into the knapsack then $x_i = 1$.

2. If the object is not placed into the knapsack then $x_i = 0$

3. If a part of the object is placed into the knapsack then $x_i = \frac{\text{remaining capacity of the knapsack}}{\text{weight of the object}}$

* since the capacity of the knapsack is M we require the total weight of all choosen objects to be atmost 'M'

$\Rightarrow$ Maximize $\sum_{i=1}^{n} p_i x_i$

subjected to $\sum_{i=1}^{n} w_i x_i \leq M$

and $0 \leq x_i \leq 1$ and $1 \leq i \leq n$

* By placing i^{th} object into the knapsack with weight w_i, we get the profit p_i.

* since the capacity of the knapsack is M

(i) $w_1 + w_2 + w_3 + \cdots + w_n \leq M$, then no decision is required because weare placing all the objects into the knapsack.

(ii) $w_1 + w_2 + w_3 + \cdots + w_n > M$, then we have to select some objects which maximize the profit.

consider the following instance of knapsac problem $n = 3$, $M = 20$

$(P_1, P_2, P_3) = (25, 24, 15)$, $(w_1, w_2, w_3) = (18, 15, 10)$

Find out the optimal solution for the given problem.

To findout the optimal solution for a given problem, first we have to

findout all feasible solutions then we pick up the feasible solution with maximum profit taken as a optimal solution.

Different feasible solutions for the given problem are

1. Maximum profit
2. minimum weight
3. Profit per unit weight (P/w ratio)

1. Maximum profit:-

$$\sum_{i=1}^{n} w_i x_i \leq M = w_1x_1 + w_2x_2 + w_3x_3 \leq M$$

$$= (18*1 + 15*\frac{2}{15} + 10*0) \leq 20$$

$$= (18+2) \leq 20$$

$$= 20 \leq 20$$

$$\sum_{i=1}^{n} P_i x_i = P_1x_1 + P_2x_2 + P_3x_3$$

$$= 25*1 + 24*\frac{2}{15} + 15*0$$

$$= 25 + \frac{16}{5} = 28.2$$

2. Minimum weight:-

$$\sum_{i=1}^{3} w_i x_i = w_1x_1 + w_2x_2 + w_3x_3 \leq 20$$

$$= 18*0 + 15*\frac{10}{15} + 10*1$$

$$= 10+10$$

$$= 20 \leq 20$$

$$\sum_{i=1}^{3} P_i x_i = P_1x_1 + P_2x_2 + P_3x_3$$

$$= 25*0 + 24*\frac{10}{15} + 15*1$$

$$= 0 + 16 + 15$$

$$= 31$$

3) Plw ratio

$$\left(\frac{P_1}{\omega_1}, \frac{P_2}{\omega_2}, \frac{P_3}{\omega_3}\right) = \left(\frac{25}{18}, \frac{24}{15}, \frac{15}{10}\right)$$

$$= (1.3, 1.6, 1.5)$$

$$\sum_{i=1}^{3} \omega_i x_i = \omega_1 x_1 + \omega_2 x_2 + \omega_3 x_3$$

$$= 18 * 0 + 15 * 1 + 10 * \frac{5}{10}$$

$$= 15 + 5$$

$$= 20 \leq 20$$

$$\sum_{i=1}^{3} P_i x_i = P_1 x_1 + P_2 x_2 + P_3 x_3$$

$$= 25 * 0 + 24 * 1 + 15 * \frac{5}{10}$$

$$= 0 + 24 + \frac{15}{2}$$

$$= 24 + 7.5$$

$$= 31.5$$

→ Among three feasible solutions, which feasible solution has maximum profit value that feasible solution can be taken as optimal solution.

→ By observing the above three feasible solutions, solution-3 has maximum profit value 31.5 hence solution-3 can be taken as optimal solution

→ solution vector $x = (x_1, x_2, x_3)$

$$= (0, 1, \frac{5}{10})$$

$$= (0, 1, \frac{1}{2})$$

→ Maximum profit = 31.5

Find out the optimal solution for the given knapsack instance (i) $n=3$, $m=20$, $(P_1, P_2, P_3) = (15, 12, 19)$ $(w_1, w_2, w_3) = (20, 15, 10)$

(ii) $n=3$, $m=20$, $(P_1, P_2, P_3) = (30, 21, 18)$, $(w_1, w_2, w_3) = (18, 15, 10)$

(iii) $n=7$, $m=15$ $(P_1, P_2, P_3, P_4, P_5, P_6, P_7) = (10, 5, 15, 7, 6, 18, 3)$, $(w_1, w_2, w_3, w_4, w_5, w_6, w_7) = (2, 3, 5, 7, 1, 4, 1)$

(i) $n=3$, $m=20$ $(P_1, P_2, P_3) = (15, 12, 19)$ $(w_1, w_2, w_3) = (20, 15, 10)$

$$\left(\frac{P}{w}\right) = \left(\frac{P_1}{w_1}, \frac{P_2}{w_2}, \frac{P_3}{w_3}\right) = \left(\frac{15}{20}, \frac{12}{15}, \frac{19}{10}\right)$$

$$= (0.75, 0.8, 1.9)$$

$$\sum_{i=1}^{3} w_i x_i = w_1 x_1 + w_2 x_2 + w_3 x_3$$

$$= 20 * 0 + 15 * \frac{10}{15} + 10 * 1$$

$$= 20$$

Maximum profit $= P_1 x_1 + P_2 x_2 + P_3 x_3$

$$= 15 * 0 + 12 * \frac{10}{15} + \frac{19}{10} \times 1$$

$$= 0 + 8 + 19$$

$$= 27$$

Solution vector $x = (x_1, x_2, x_3)$

$$= \left(0, \frac{10}{15}, 1\right)$$

(ii) $n=3$, $m=20$, $(P_1, P_2, P_3) = (30, 21, 18)$ (w_1, w_2, w_3)

$= (18, 15, 10)$

$$\frac{P}{w} = \left(\frac{30}{18}, \frac{21}{15}, \frac{18}{10}\right) = (0.72, 1.4, 1.8)$$

$$\sum_{i=1}^{3} = w_i x_i = w_1 x_1 + w_2 x_2 + w_3 x_3$$

$$= 18 * 0 + 15 * \frac{10}{15} + 10 * 1$$

$$m = 10 + 10 = 20$$

$= 18 \times \frac{60}{18} + 15 \times \frac{10}{15} + 10 \times 1$

$= 10 + 10$

$m = 20$

Maximum profit $= P_1x_1 + P_2x_2 + P_3x_3$

$= 130 \times \frac{10}{18} + 21 \times \frac{10}{15} + 18 \times 1$

$= 34.67 \cdot 32$

solution vector $= (\frac{10}{18}, 1, 1)$ $(0, \frac{2}{3}, 1)$

(iii) $n = 7 \quad m = 15$

$(P_1, P_2, P_3, P_4, P_5, P_6, P_7) = (10, 5, 15, 7, 6, 18, 3)$

$(\omega_1, \omega_2, \omega_3, \omega_4, \omega_5, \omega_6, \omega_7) = (2, 3, 5, 7, 1, 4, 1)$

$(\frac{P_i}{\omega}) = (\frac{10}{2}, \frac{5}{3}, \frac{15}{5}, \frac{7}{7}, \frac{6}{1}, \frac{18}{4}, \frac{3}{1})$

$= (5, 1.67, 3, 1, 6, 4.5, 3)$

Maximum weight

$= \omega_1x_1 + \omega_2x_2 + \omega_3x_3 + \omega_4x_4 + \omega_5x_5 + \omega_6x_6 +$

$= 2 \times \frac{14}{2} + 3 \times 0 + 5 \times 0 + 7 \times 0 + 1 \times 1 + 4 \times 0 +$

$= 14 + 1$

$= 15$

Maximum profit $= P_1x_1 + P_2x_2 + P_3x_3 + P_4x_4 + P_5x_5 + P_6x_6$

$= 10 \times \frac{14}{2} + 0 + 0 + 0 + 0 + 6 \times 1 +$

$= 70 + 6$

$= 76$

$$\frac{P}{\omega} = \left(\frac{10}{2}, \frac{5}{3}, \frac{15}{5}, \frac{7}{7}, \frac{6}{1}, \frac{12}{4}, \frac{2}{1}\right)$$

$$= (5, 1.67, 3, 1, 6, 4.5, 3)$$

≤ 15

Maximum profit

$$= P_1x_1 + P_2x_2 + P_3x_3 + P_4x_4 + P_5x_5 + P_6x_6 + P_7x_7$$

$$= 10 * \frac{3}{10} + 5 * 0 + 15 * 0 + 7 * 0 + 6 * 2 + 18 * 0 + 3 * 0$$

$$=$$

$$\sum_{i=1}^{7} \omega_i x_i = \omega_1 x_1 + \omega_2 x_2 + \omega_3 x_3 + \omega_4 x_4 + \omega_5 x_5 + \omega_6 x_6 + \omega_7 x_7$$

$$= 2* + 3* + 5* + 7* + 1*2 + 4* + 1*$$

$$= 3*1*1 + 2*1 + 4*1 + 5*1 + 1*1 + 3*\frac{2}{3}$$

$$= 15$$

$$\sum_{i=1}^{7} P_i x_i = 6*1 + 10*1 + 18*1 + 15*1 + 3*1 + 5*\frac{2}{3}$$

$$= 6 + 10 + 18 + 15 + 3 + 3.3$$

$$= 55.3$$

Greedy knapsack.

Algorithm Greedyknapsack(n, P, w, M, x)

// P[1:n] profits of n objects.

// w[1:n] weights of n objects.

// M is the capacity of the knapsack.

// x[1:n] is the solution vector

// ordered such that $\frac{P[i]}{w[i]} \geq \frac{P[i+1]}{w[i+1]}$

```
{
    for i:=1 to n do
    { if (w[i]>M) then break;
      else
        {
            x[i]:=1;
            M:=M-w[i];
        }
    }
    if (i<=n) then
    {
        x[i]:= M/w[i] ;
    }
}
```

Time complexity :-

The time complexity of greedy knapsack problem can be calculated by using the following steps.

step-1:-
sorting of all objects in decreasing order of p/w can take nlogn time.

step-2:-
we need one scan to find out maximum p/w ratio for all objects can take n time.

∴ Time complexity = $O(n\log n)^{1st} + O(n)^{2nd} = O(n\log n)^{MAX}$

Job sequencing with deadlines problem:-
The arrangement of Jobs on a single processor with deadline constraint is called as Job sequencing with deadlines problem.

we arrange n Jobs on a single processor in a sequence to obtain maximum profit, subjected to dead lines

The Job sequencing problem is stated as follows:

consider n Jobs, each Job is associated with a deadline d_i ($d_i > 0$) and a profit P_i ($P_i > 0$)

for any Job i, the profit P_i is earned, if the Job is completed by its deadline.

for executing a Job, it is processed on a processor for one unit of time. only one machine or processor is available for processing all Jobs.

At a time, only one Job is scheduled.

A feasible solution for this problem is a subset of jobs such that each job in the subset can be completed by its deadline

→ An optimal solution is a feasible solution with maximum profit

EX1-

Solve the following job sequencing with deadlines problem using greedy Algorithm

$n = 4$, profits : $(P_1 - P_4) = (100, 10, 15, 27)$

deadlines : $(D_1 - D_4) = (2, 1, 2, 1)$.

SNO	feasible solution	processing sequence	Maximum profit.
1	(1)	1	100
2	(2)	2	10
3	(3)	3	15
4	(4)	4	27
5	(1,2)	2,1	100+10=110
6	(1,3)	1,3 or 3,1	100+15=115
7	(1,4)	4,1	100+27=127
8	(2,3)	2,3	10+15=25
9	(2,4)	2,4 or 4,2	10+27=37
10	(3,4)	4,3	15+27=42

Solution-7 is optimal. In this solution only Jobs 1 and 4 is processed and the profit is 100+27=127. These two Jobs must be

processed in the order: Job 4 is followed by Job 1 and the processing of Job 4 begins at time 0 and that of Job 1 is completed at time 2.

(OR)

Arrange the Jobs in decreasing order based on the profit

J_1 J_2 J_3 J_4

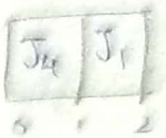
$(P_1 - P_4) = (100, 10, 15, 27)$

$(D_1 - D_4) = (2, 1, 2, 1)$

$= (100, 27, 15, 10)$

$= (2, 1, 2, 1)$

Gaant



J	Assigned slot	selected Job	Action	Max profit
ϕ	-	J_1	Assigned to slot [1,2]	100
$\{3\}$	[1,2]	J_4	Assigned to slot [0,1]	100+27=127
$\{3,4\}$	[0,1][1,2]	J_3	Rejected	127.
$\{3,4\}$	[0,1][1,2]	J_2	Rejected	127.

$J = \{J_1, J_4\}$ with processing sequencing

$4 \rightarrow 1$ with maximum profit = 127

$\therefore$ optimal solution $J = \{J_1, J_4\}$ with

max profit 127.

2. Find out the optimal solution for the following Jobsequencing with deadline

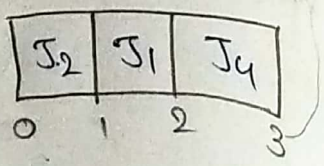
$n=5 \quad (P_1 - P_5) = \{20, 15, 10, 5, 1\}$

$(D_1 - D_5) = (2, 2, 1, 3, 3)$

Arrange the Jobs in decreasing order

J_1	J_2	J_3	J_4	J_5
20	15	10	5	1
2	2	1	3	3

maximum deadline number



J	Assigned slot	selected Job	Action	max profit
ϕ	–	J_1	Assigned to slot [1, 2]	20
$\{J_1\}$	[1,2]	J_2	Assigned to slot [0, 1]	20+15 =35
$\{J_1, J_2\}$	[0,1] [1,2]	J_3	Rejected	20+15=35
$\{J_1, J_2\}$	[0,1] [1,2]	J_4	Assigned to slot [2,3]	20+15+5 =40
$\{J_1, J_2, J_4\}$	[0,1][1,2][2,3]	J_5	Rejected	40

$J = \{J_1, J_2, J_4\}$ with processing sequence $2 \rightarrow 1 \rightarrow 4$

with max. profit = 40

find out the optimal solution for the Job Sequencing with dead lines problem

$n=7 \quad (P_1 - P_7) = (3, 5, 20, 18, 1, 6, 30) \quad (D_1 - D_7) = (1, 3, 4, 2, 1, 2$

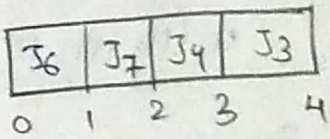
$(P_1 - P_7)$ = (3, 5, 20, 18, 6, 30)

$(P_1 - D_7)$ = (1, 3, 4, 3, 2, 1, 2)

Arrange the profits in decreasing order

(30, 20, 18, 6, 5, 3, 1)

(2, 4, 3, 2, 1, 1, 3)



J	Assigned slot	selected Job	Action	max profit.
ϕ	-	J_7	Assigned to slot [1,2]	30
$\{J_7\}$	[1,2]	J_3	Assigned to slot [3,4]	30+20=50
$\{J_7, J_3\}$	[1,2] [3,4]	J_4	Assigned to Slot [2,3]	50+18 = 68
$\{J_7, J_3, J_4\}$	[1,2] [2,3] [3,4]	J_6	Assigned to slot [0,1]	68+6 = 74
$\{J_7, J_3, J_4, J_6\}$	[0,1] [1,2] [2,3] [3,4]	J_2	Rejected	74
$\{J_7, J_3, J_4, J_6\}$	[0,1] [1,2] [2,3] [3,4]	J_1	Rejected	74
$\{J_7, J_3, J_4, J_6\}$	[0,1] [1,2] [2,3] [3,4]	J_5	Rejected	74

~~Arrange the~~

$J = \{J_7, J_3, J_4, J_6\}$ with processing sequence $6 \to 7 \to 4 \to 3$ with maxprofit = 74

```
Algorithm GreedyJobsequencing (d, J, n)
// d is the deadlines of n jobs
// n is the number of jobs.
// J is the set of jobs that are completed
   within their deadline.

{
      J = {1};
      for { i := 2 to n do
      {
          if (all jobs that are completed with
              in their deadline) then
                  J = J ∪ {i};
      }
}
```

Time complexity of Jobsequencing with deadlines problem can be calculated by using the following steps:-

(i) sort the given Jobs according to maximum profit that can take nlogn time.

(ii) for each Job, find out the slot in array of size n can take n^2 time (gaant chart)

∴ Totaltime = $O(n\log n) + O(n^2)$

$= O(n^2)$

Find out the optimal solution for the following sobsequencing with deadline problem

$n=6$ $(P_1-P_6) = (200, 180, 190, 300, 120, 100)$

$(D_1-P_6) = (5, 3, 3, 2, 4, 2)$

(i) write the optimal schedule that gives maximum profit (3values)?

(ii) Are all the Jobs completed in the optimal schedule?

(iii) what is the maximum earned profit?

$J_1\ J_2\ J_3\ J_4\ J_5\ J_6$

$n=6$ $(P_1-P_6) = (200, 180, 190, 300, 120, 100)$

$(D_1-P_6) = (3, 3, 3, 2, 4, 2)$

Arrange the Jobs in decreasing order.

$= (300, 200, 190, 180, 120, 100)$

$= (2, 5, 3, 3, 4, 2)$

J_2	J_4	J_3	J_5	J_1
	2		3	4 5

J	Assigned slot.	selected Job	Action	Max profit.
ϕ	-	J_4	Assigned to slot [1,2]	300
J_4	[1,2]	J_1	Assigned to slot [4,5]	300+200 = 500
J_4, J_1	[1,2][4,5]	J_3	Assigned to slot [2,3]	300+200+190 = 690
J_4, J_1, J_3	[1,2][2,3][4,5]	J_2	Assigned to slot [0,1]	690+180 = 870
J_4, J_1, J_3, J_2	[0,1][1,2][2,3][4,5]	J_5	Assigned to slot [3,4]	870+120 = 990

$\{J_4, J_1, J_3, J_2, J_5\}$ [0,1] [1,2] [2,3] J_6 Rejected 990

[3,4] [4,5]

$J = \{J_4, J_1, J_3, J_2, J_5\}$ with processing sequence $2 \rightarrow 4 \rightarrow 3 \rightarrow 5 \rightarrow 1$

we are given 9 Jobs J_1 to J_9. The execution of each Job requires 1 unit of time. we can execute one Job at a time with 1 unit duration. Profits and deadlines are listed below.

$P_1 - P_9 = \{15, 20, 30, 18, 18, 10, 23, 16, 25\}$

$d_1 - d_9 = \{7, 2, 5, 3, 4, 5, 2, 7, 3\}$

(i) what is the maximum profit earned.

a) 147 b) 165 c) 167 d) 175

(ii) Are all Jobs completed in the schedule that gives maximum profit.

a) all Jobs are completed b) J_1 and J_6 are left out

c) J_1 and J_4 are left out d) J_4 and J_6 are left out

$n = 9$ $(P_1 - P_9) = \{\overset{J_1}{15}, \overset{J_2}{20}, \overset{J_3}{30}, \overset{J_4}{18}, \overset{J_5}{18}, \overset{J_6}{10}, \overset{J_7}{23}, \overset{J_8}{16}, \overset{J_9}{25}\}$

$(d_1 - d_9) = \{7, 2, 5, 3, 4, 5, 2, 7, 3\}$

Arrange the Jobs in descending order.

$= \{30, 25, 23, 20, 18, 18, 16, 15, 10\}$

$= \{5, 3, 2, 2, 4, 3, 7, 7, 5\}$

J_2	J_7	J_9	J_5	J_3	J_1	J_8

0 1 2 3 4 5 6 7

J	Assigned slot	Selected Job	Action	Max Profit
ϕ	–	J_3	Assigned to slot [4,5]	30
J_3	[4,5]	J_9	Assigned to slot [2,3]	30+25
J_3, J_9	[2,3] [4,5]	J_7	Assigned to slot [1,2]	55 23 78 ~~35~~ 23
J_3, J_9, J_7	[1,2] [2,3] [4,5]	J_2	Assigned to slot [0,1]	78+20
J_3, J_9, J_7, J_2	[0,1] [1,2] [2,3] [4,5]	J_5	Rejected [3,4]	98+18.
J_3, J_9, J_7, J_2, J_5	[0,1] [1,2] [2,3] [4,5] [3,4]	J_4	Rejected	~~98+18~~ 98+18
J_3, J_9, J_7, J_2, J_5	[0,1] [1,2] [2,3] [3,4] [4,5]	J_8	Assigned to slot [6,7]	98+18+16
$J_3, J_9, J_7, J_2, J_5, J_8$	[0,1] [1,2] [2,3] [3,4] [4,5] [6,7]	J_1	Assigned to slot [5,6]	98+18+16+15
$J_3, J_9, J_7, J_2, J_5, J_8, J_1$	[0,1] [1,2] [2,3] [3,4] [4,5] [5,6] [6,7]	J_6	Rejected	147

(i) 147

(ii) d

Optimal storage on Tapes:-

→ Computer programs are earlier stored on magnetic tape when the user wants any program then that program can be retrieved from the tape.

→ Consider a set of programs $P_1, P_2, P_3, \ldots P_n$ are n programs with lengths $l_1, l_2, l_3 \ldots l_n$

Our main objective is to store all the programs on a single tape and the average amount of

time required to access a ... from the tape is to be minimized.

→ At initial stage, A tape is positioned at front. After storing all the programs, A tape is positioned at after the last program.

for accessing a random program we are always come back from the beginning.

→ The time required to access a random program P_J is t_J, it can be calculated by using the following formula.

$$* \; t_J = \sum_{k=1}^{J} l_k$$

$$= l_1 + l_2 + l_3 + \ldots + l_J$$

Ex:- $$t_5 = \sum_{k=1}^{5} l_5$$

$$= l_1 + l_2 + l_3 + l_4 + l_5$$

→ The average amount of time required to access a particular program from the tape is called Mean Retrieval time. (MRT)

$$* MRT = \frac{1}{n} \Sigma (t_J)$$

Example:-

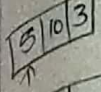
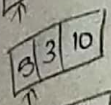
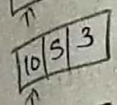
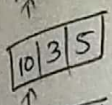
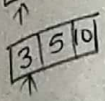
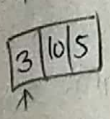
1) Find out the optimal retrieval order and optimal mean retrieval time for the following instance:

$n = 3, (l_1, l_2, l_3) = (5, 10, 3)$ n is the no. of programs

~~Order~~ — ~~Total retrieval time~~ — ~~MRT~~

3 programs can be arranged in 3! ways

= 6 ways

order		Total retrieval time	MRT
1, 2, 3		5 + (5+10) + (5+10+3)	38/3
1, 3, 2		5 + (5+3) + (5+3+10)	31/3
2, 1, 3		10 + (10+5) + (10+5+3)	43/3
2, 3, 1		10 + (10+3) + (10+3+5)	41/3
3, 1, 2		3 + (3+5) + (3+5+10)	29/3
3, 2, 1		3 + (3+10) + (3+10+5)	34/3

(29/3 min)

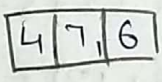
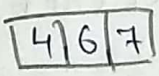
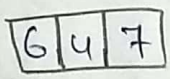
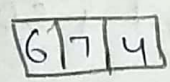
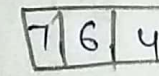
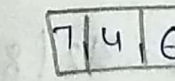
* optimal retrieval order = (3, 1, 2)

* M.R.T = 29/3, Total retrieval time = 29

2) Find out all feasible solutions for the following 3-optimal storage on tapes instance n = 3
$l_1, l_2, l_3 = (4, 7, 6)$. Find M.R.T?

n = 3

3 programs can be arranged in 3! ways = 6

order		Total retrieval time	MRT
1, 2, 3		4 + (4+7) + (4+7+6)	32/3
1, 3, 2		4 + (4+6) + (4+6+7)	31/3
3, 1, 2		6 + (6+4) + (6+4+7)	33/3
3, 2, 1		6 + (6+7) + (6+4+7)	36/3
2, 3, 1		7 + (7+6) + (7+6+4)	37/3
2, 1, 3		7 + (7+4) + (7+6+4)	35/3

optimal retrieval order (1, 3, 2)

M.RT = 31/3

Total retrieval time = 31

NOTE :-

If the no. of programs are more then we are unable to find all the orders. Hence we can directly find the Total retrieval time by arranging all the lengths of programs in ascending order.

Ex:-

Find out the total retrieval time and optimal mean retrieval time for the programs with lengths 12, 5, 8, 32, 7, 5, 18, 13

arrange all the programs in ascending order of their lengths

(do = 32, 18, 13, 12, 8, 7, 5, 5)

d.o = 5, 5, 7, 8, 12, 13, 18, 32

Total retrieval time

$= 5 + (5+5) + (5+5+7) + (5+5+7+8) +$
$(5+5+7+8+12) + (5+5+7+8+12+13)$
$+ (5+5+7+8+12+13+18) + (5+5+7+8+$
$12+13+18+32)$

$= 5+10+17+25+37+50+68+100$

$= 312$

Mean retrieval time $= 312/8 = 39$

NOTE :-

If more than one tape is available, then arrange the programs tape by tape

Algorithm:-

Algorithm GreedyOptimalStorage (n, k)

// n is the no. of programs

// k is the no. of Tapes

// Assign programs to tapes, Assume all the

Findout the total retrieval time for the following programs with lengths 12, 5, 8, 32, 7, 5, 18, 13 with 3 Tapes.

Arrange all the programs in ascending order of their lengths.

~~32, 18, 13, 12,~~

a.o = 5, 5, 7, 8, 12, 13, 18, 32

T_0 : 5 8 18

T_1 : 5 12 32

T_2 : 7 13

Total retrieval time for T_0 tape

= 5 + (5+8) + (5+8+18) = 49

MRT = 49/3

Total retrieval time for T_1 tape

= 5 + (5+12) + (5+12+32) = 71

MRT = 71/5

Total retrieval time for T_2 tape

= 7 + (7+13) = 27

MRT = 27/2

Total retrieval time for 3 Tapes = 49+71+27 = 147

EX-2:-

Find an optimal placement for 13 programs on 3 tapes T_0, T_1, T_2 where the programs are of length 12, 5, 8, 32, 7, 5, 18, 26, 4, 3, 11, 10, 6

Arrange all the programs in A.O

3, 4, 5, 5, 6, 7, 8, 10, 11, 12, 18, 26, 32

T_0: 3 5 8 12 32

T_1: 4 6 10 18

T_2: 5 7 11 26

32
28
60

Total time required for tape T_0

$= 3 + (3+5) + (3+5+8) + (3+5+8+12) + (3+5+8+12+32)$

$= 3 + 8 + 16 + 28 + 60$

$= 115$

M.R.T = 115/5

Total time required for tape T_1

$= 4 + (4+6) + (4+6+10) + (4+6+10+18)$

$= 4 + 10 + 20 + 38 = 72$

26
23
49

MRT = 72/4

M.R.T = 72/4

Total time required for Tape T_2

$= 5 + (5+7) + (5+7+11) + (5+7+11+26)$

$= 5 + 12 + 23 + 49 = 89$

M.R.T = 89/4

Algorithm optimalpattern (n, t)
// n is the number of programs
// t is the no. of tapes
// Assign programs to tapes, assume all the programs are arranged in ascending order of their length

{
    j := 0; // tape number
    for i := 1 to n do
    {
        print ("program", i, "was put on tape", j);
        j := (j+1) mod t;
    }
}

* optimal merge pattern:
Let A and B are two sorted lists with m and n, for combining list A and list B into a single sorted list C, we must require m+n comparisons.

Ex:

List A	List B	List C
→ 3	→ 5	3
7	6	5
9	8	6
12	11	7
m=4	n=4	8
		9
		11
		12

O(m+n) = O(max(m,n))

No. of comparisions are required

$= m+n = 4+4 = 8$

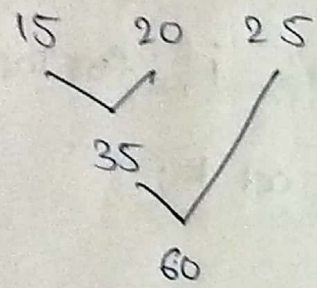
If we consider three lists with different sizes, we require many orderings are possible to combine them as a single list

Ex:-

Lists:	A	B	C
Records:-	15	20	25

→ How many no. of orderings are required $= nC_2 = 3C_2 = 3$

order 1

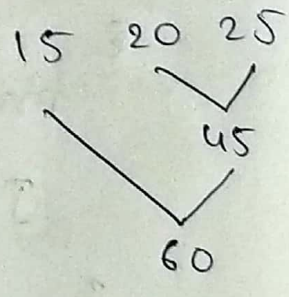


cost (or) no. of merges $= 35+60$

$= 95$

order ((A, B), C)

order 2



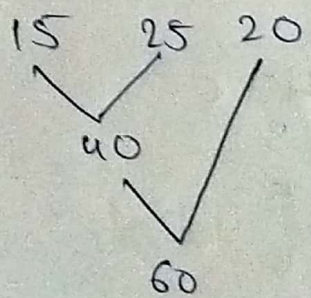
cost (or) no. of merges

$= 45+60$

$= 105$

order = (A, (B, C))

order 3



Cost $= 40+60 = 100$

order = ((A, C), B)

By observing the above three orders, which order has minimum number of merges or cost that order can be treated as optimal merge order.

→ optimal merge order = ((A, B), C) with total no. of merges = 95

If the number of lists or files are more, it is difficult to find orderings, hence, we are using a greedy method to find the optimal merge pattern tree.

Greedy procedure for constructing the optimal merge pattern tree:-

construct the binary tree, in that binary tree, all the leaf nodes are represented as squares (inputs) and all the internal nodes represented as circles (merges)

step-1:-

Arrange all the files in ascending order based on the number of records.

step-2:-

Identify the two smallest sized files and merge them. Insert a merged file in the sorted list at its correct position.

step-3:-

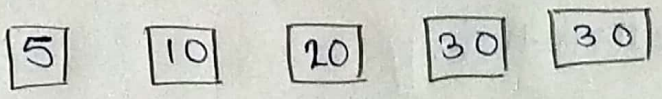
Repeat step-2 until a binary tree is formed with minimum number of comparisions.

Ex :-

Let the files are x_1, x_2, x_3, x_4, x_5 are to be merged with 20, 30, 10, 5, 30 records respectively Apply Greedy method to find the optimal merge pattern.

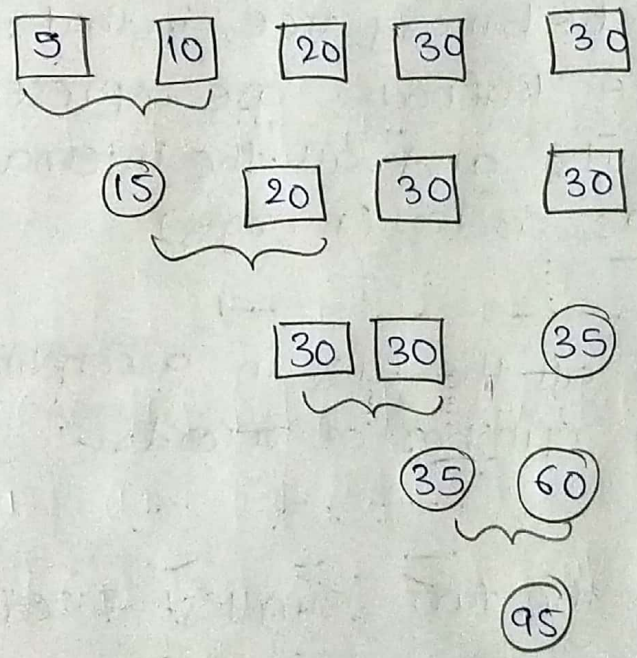
Files:	x_1	x_2	x_3	x_4	x_5
Records:	20	30	10	5	30

step-1:- Arrange the records in Ascending order.

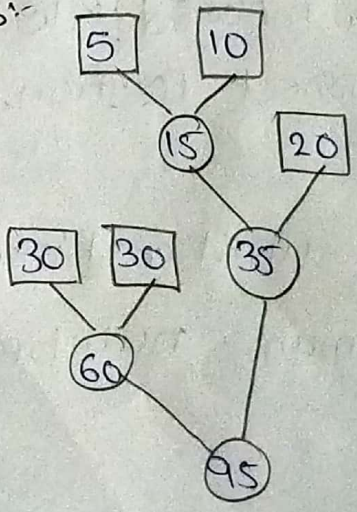


inputs.

step-2:-



step-3:-



$\sum_{i=1}^{n} d_i x_i$

d_i = distance from root to input node

x_i = input

$= 5\times3 + 10\times3 + 20\times2 + 30\times2 + 30\times2$

$= 15 + 30 + 40 + 60 + 60$

$= 205$

(or)

circle files

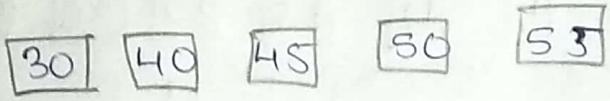
Total no. of merges : $15 + 35 + 60 + 95 = 205$

Ex2:-

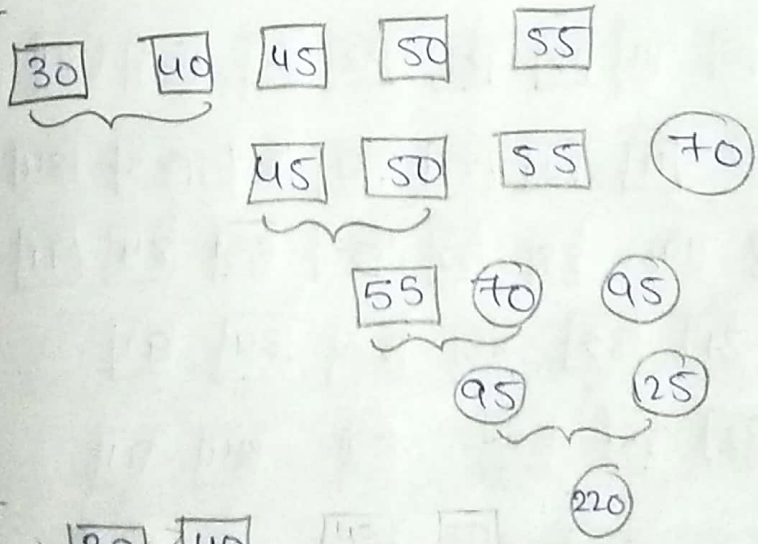
2) Files x_1 x_2 x_3 x_4 x_5

Records:- 40 30 50 45 55

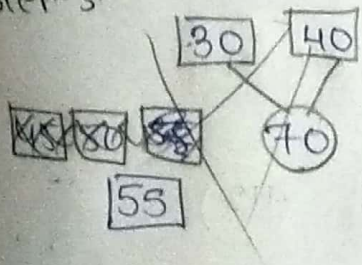
step-1:- Arrange the records in ascending order.



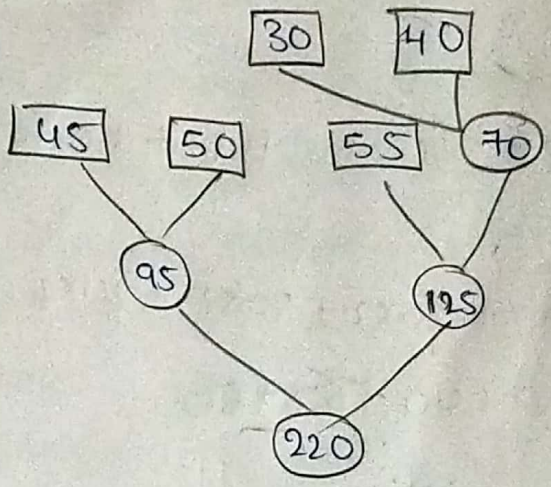
step-2:-



step-3:-



125



Total no. of merges: 70 + 95 + 125 + 220

= 510

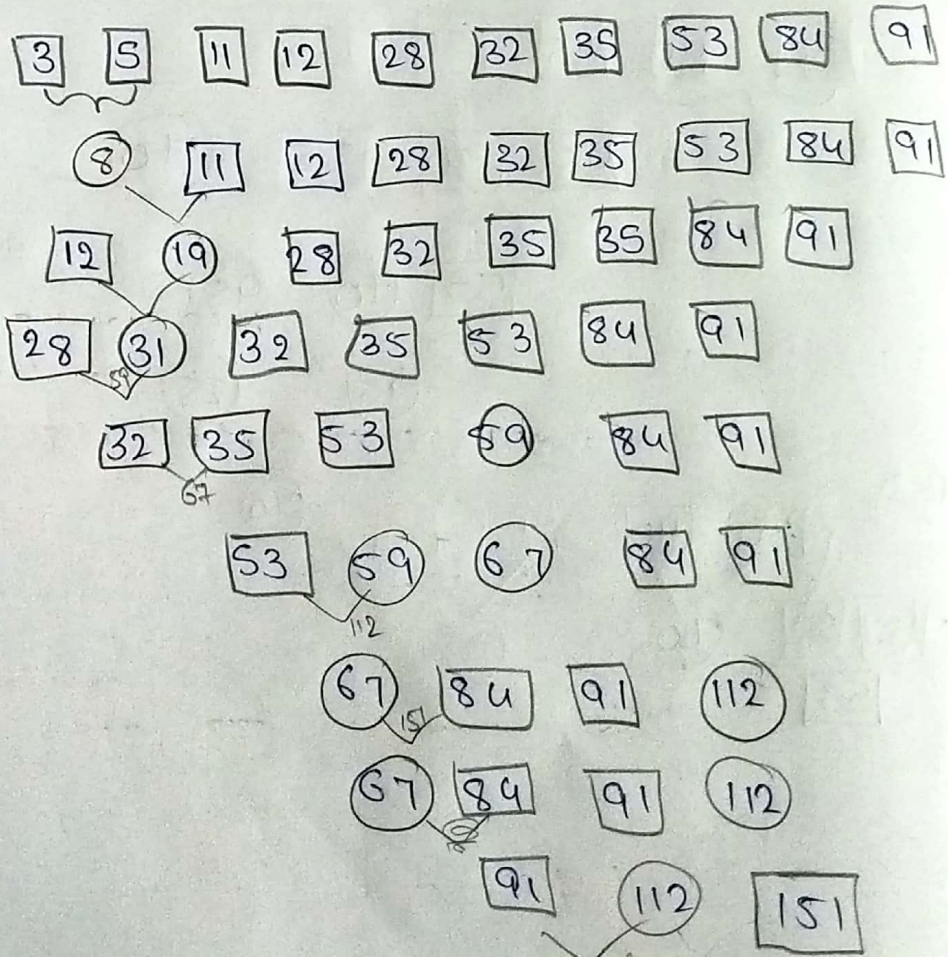
3) Files:-

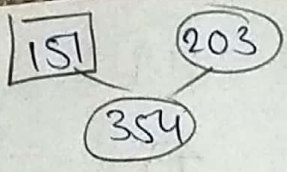
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
28	32	12	5	84	53	91	35	3	11
2	5	7	26	11	16	21	17	35	42

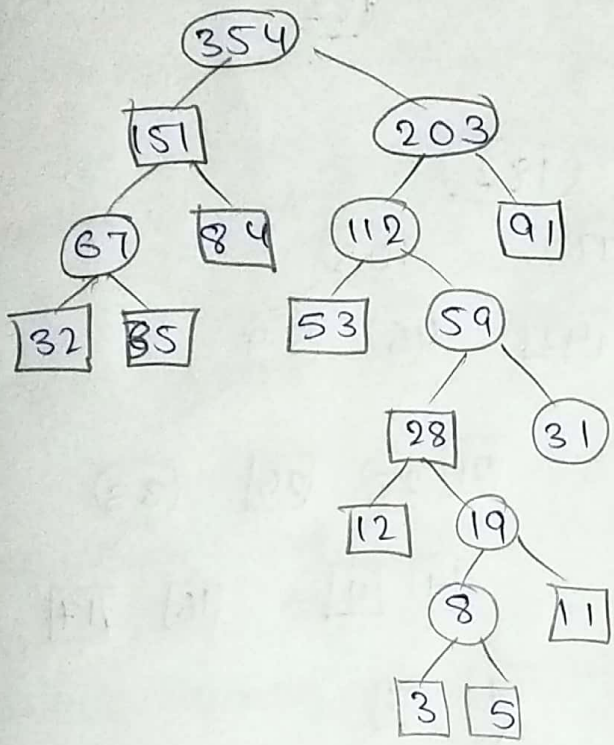
Arrange these in Ascending order

3 5 11 12 28 32 35 53 84 91

step-1:-





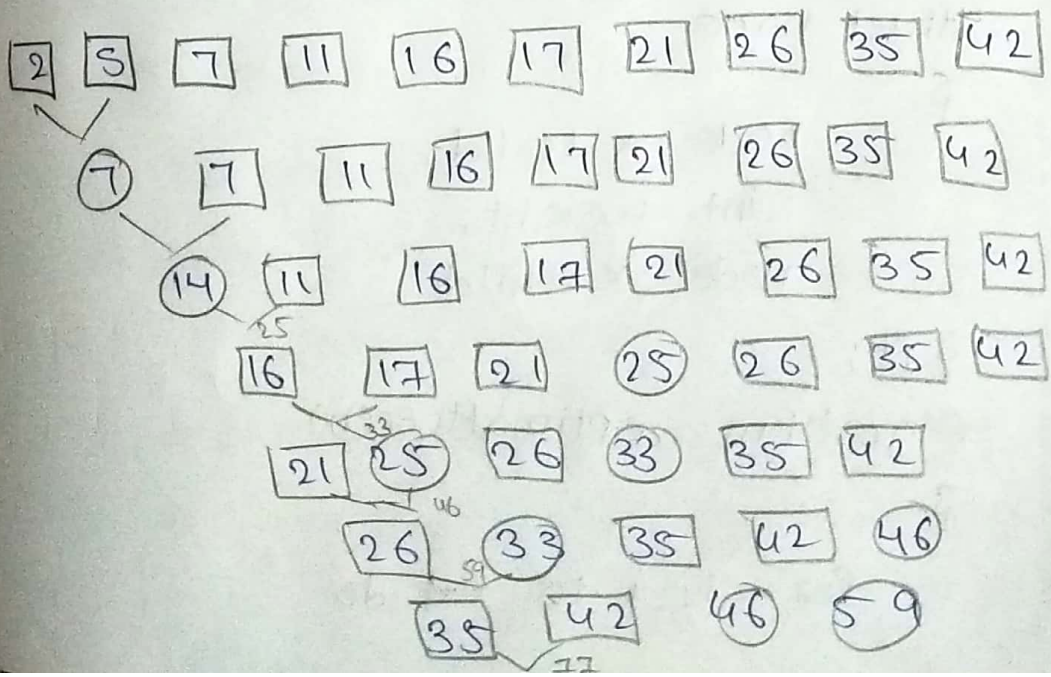


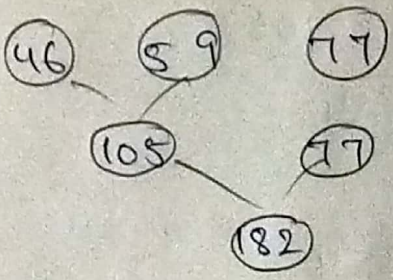
no. of merges = 354 + 151 + 203 + 67 + 112 + 59 + 31 + 19 + 8

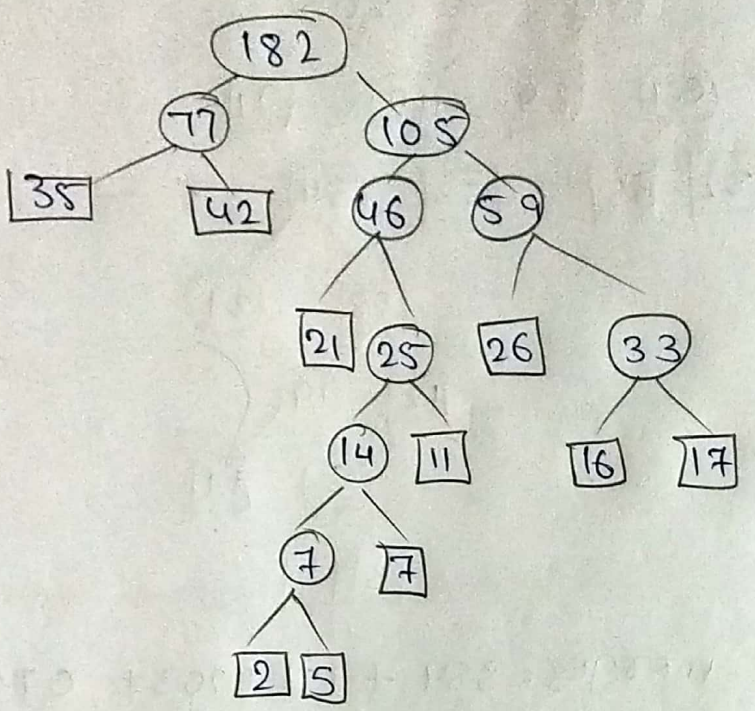
= 1004

iii)

Files:	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
Records:	2	5	7	26	11	16	21	17	35	42







No. of merges = 182 + 77 + 105 + 46 + 59

+ 25 + 33 + 14 + 7

= 548

Algorithm for optimal merge pattern:-

```
struct node
{
        node  *lchild;
        int   weight;
        node  *rchild;
};pt;
Algorithm  optimaltree(n)
{
    for  i:=1 to n-1 do
```

```
{ pt := newnode;        //create a new node
  pt -> lchild := least(list);
  pt -> rchild := least (list);
  pt -> weight := (pt -> lchild) -> weight +
                  (pt -> rchild) -> weight;
  Insert (list, pt);

}
return least (list);

}
```

→ In the above algorithm, the least function is used to find the minimum element in the list, after finding the minimum ele it is deleted from the list.

Insert function is used for inserting an element to the list at its correct position in the list.

Analysis or Time complexity:-

The time complexity of optimal merge pattern can be calculated by using the following steps -

- for arranging the given files or programs in ascending order we require O(nlogn) time.
- for constructing the binary tree by taking n number of files take O(n) time.

∴ Time complexity = $O(n \log n) + O(n)$

$= O(n \log n)$

* Huffman (code tree) coding:-

David Huffman discovered an optimal algorithm for encoding documents.

Huffman codes are used in a wide variety of practical applications including fax machines, modems, computer networks and height definition Televisions.

A Huffman code has one leaf node for each different character in the document. A code for each character is then determined by the root to leaf path through that character.

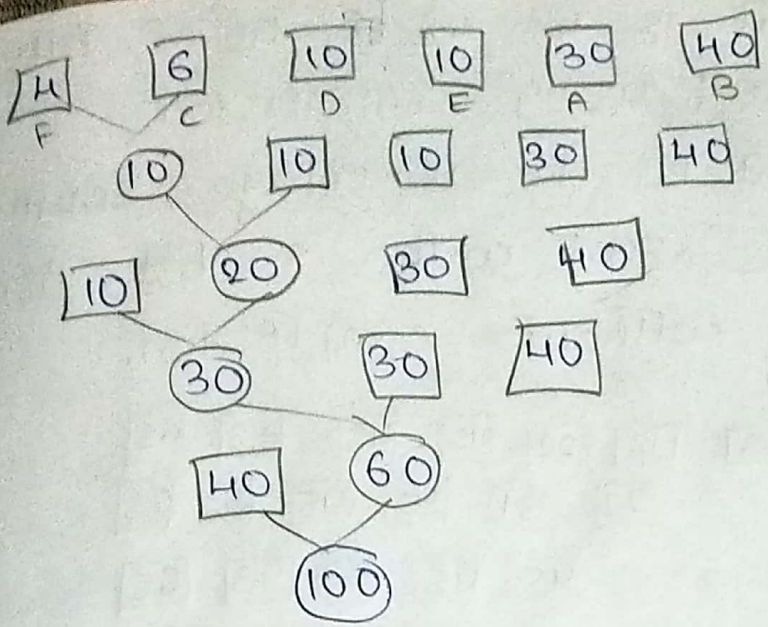
The left branches labelled '0' and right branches are labelled '1'.

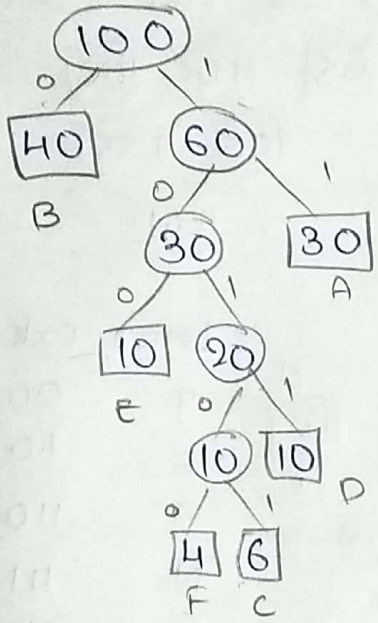
For example, a root to leaf path that goes right - left - right gives the code 101

Ex:-

Symbols A B C D E F are being produced by the information source with probabilities in percentages 30, 40, 6, 10, 10, 4 respectively. construct the Huffman binary tree and find out the code-word for each symbol

Symbols :	A	B	C	D	E	F
probabilities:	30	40	6	10	10	4





symbol	codeword
A	11 (2×30)
B	0 (1×40)
C	1011 (5×6)
D	1011 (4×10)
E	100 (3×13)
F	10100 (5×4)
	220

Information to be transmitted over the internet contains the following characters with their associated frequencies as shown below

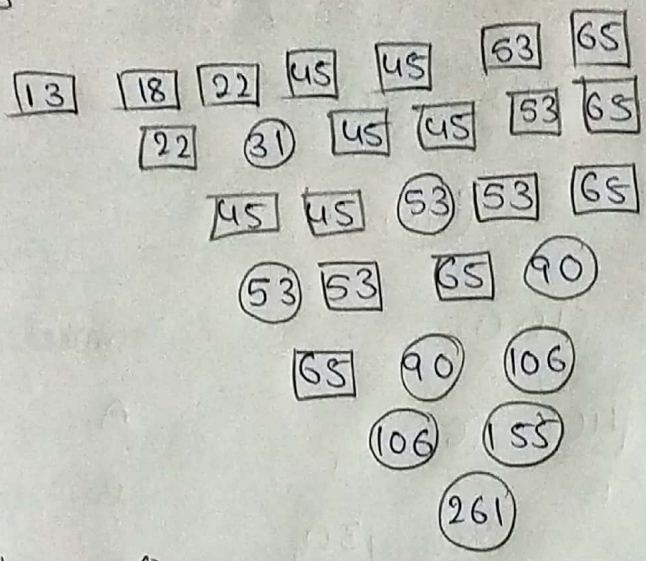
characters	a	e	l	n	o	s	t
Freq	45	65	13	45	18	22	53

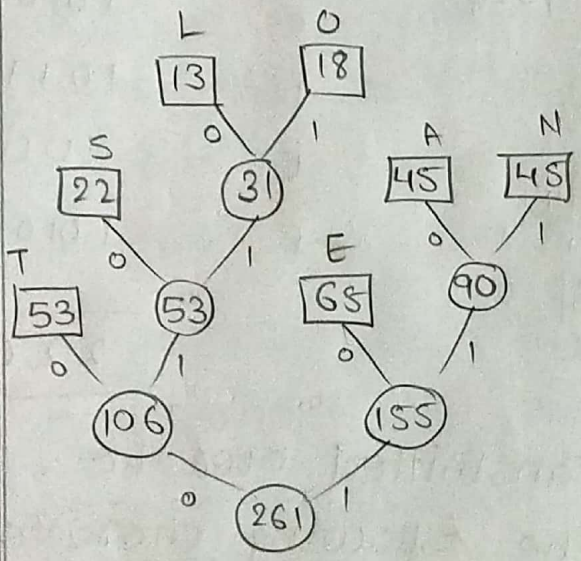
Use the Huffman Technique to answer the following questions.

(i) Build the Huffman code tree for the message and find the code word

(ii) what is the total no. of bits to be transmitted using compression.

(iii) what is the percentage saving, if the data is sent with 8 bit ASCII values without compression.





Letter	codeword	
T	00	2*53 = 106
E	10	2*65 = 130
A	110	3*45 = 135
N	111	3*45 = 135
S	010	3*22 = 66
L	0110	4*13 = 52
O	0111	4*18 = 72
		696

(ii) If we assume that the message contain consist of only characters A, E, L, N, O, S, T then the number of bits transmitted will be:

$(3*45) + (2*65) + (4*13) + (3*45) + (4*18) + (3*22) + (53*2) = 696$

(iii) If the message is sent uncompressed with 8-bit ASCII representation for the characters we have 261, character, to represent we require 8-bits. total no. of bits. $261*8$

No. of bits save = no. of bits required without compression - no. of bits required without compression.

= 2088 - 696

= 1392.

Minimum cost spanning tree:-

→ Let G(E,V) is an undirected graph, then a sub-graph T = (V, E') of G, is a spanning tree of G if and only if T is a tree.

→ A spanning tree is a minimal subgraph G' of G such that V(G') = V(G) and G' is connected. A minimal sub graph is one with the fewest no. of edges.

→ Any connected graph with n vertices must have atleast n-1 edges and all the connected graphs with n-1 edges are trees. Each edge in the graph is associated with a cost (or) weight.

→ The cost of minimum spanning tree is the sum of costs of edges in that spanning tree.

→ there are two popular algorithms to construct minimum cost spanning tree.

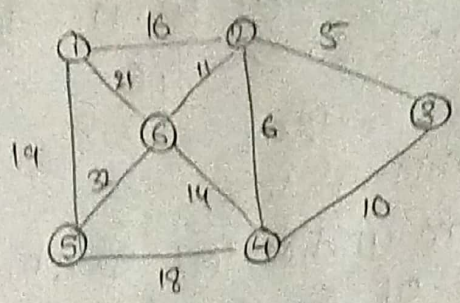
1. Prim's algorithm
2. Kruskals algorithm.

→ to obtain the spanning tree for a given graph the no. of edges that are to be removed is called as circuit rank.
∴ circuit rank = E - V + 1
V = no. of vertices E = no. of edges.

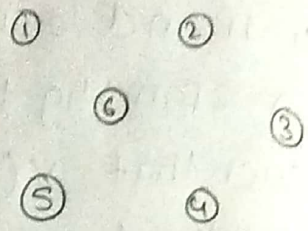
Prims algorithm:-

1. Initially the spanning tree is empty.
2. Consider any vertex from u and add it to the spanning tree.
3. Find all the neighbouring adjacent vertices for that node, consider all edges of that vertex and add the min cost edge to spanning tree.
4. Repeat the entire process until all the vertices are visited, without forming a cycle.

1. Find out the minimum cost for the following graph using prim's algorims.

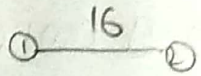


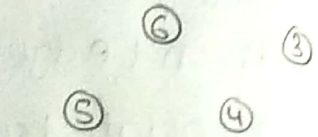
Step-1:-
We will consider all the vertices first.



Minimum cost = 0

Step-2:-
consider vertex 1, find out all the neighbouring vertices with cost, select the minimum cost edge and add it to the spanning tree without forming a cycle.

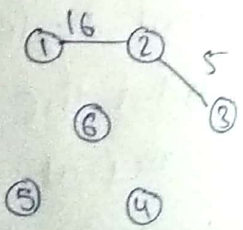




neighbouring edges	selected edge	mincost = 16
1-2 = 16 ✓	(1,2)	
1-6 = 21		
1-5 = 19		

Step-3:-
consider vertex 2, find out the neighbouring vertices of 1 & 2. among them we have to select the vertex with min cost & add it to the spanning tree without forming cycle.

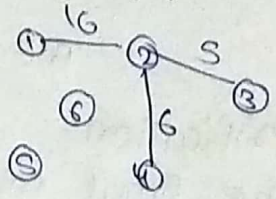
neighbouring edges	selected edge	graph
2-3 = 5 ✓	(2-3)	
2-4 = 6		
2-6 = 11	mincost = 21	
1-6 = 21		
1-5 = 19		



step-3 :- neighbouring vertices of 1, 2 & 3 among them we have to select the vertex with min cost & add it to the spanning tree without forming a cycle.

neighbouring edges	selected edge
2-4 = 6 ✓	(2,4)
2-6 = 11	
1-6 = 21	
1-5 = 19	
3-4 = 10	

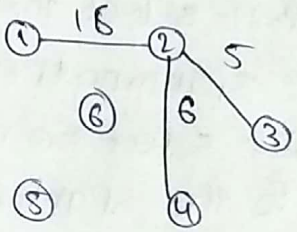
graph



Min cost = 27

step-5 :-
Find out the neighbouring edges of 1, 2, 3 & 4. among them we need to select the vertex with min cost & add it to the spanning tree without forming cycle.

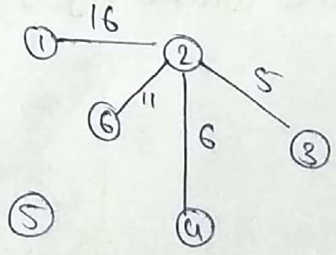
2-6 = 11	(3,4)
1-6 = 21	
1-5 = 19	
3-4 = 10 ✓	
4-5 = 18	
4-6 = 14	



since adding (3,4) to spanning tree it forms a cycle

step-6 :- Find out the neighbouring edges of 1,2,3,4 select min cost edge and add it to the spanning tree without forming a cycle.

2-6 = 11 ✓	(2,6)
1-6 = 21	
1-5 = 19	
4-5 = 18	
4-6 = 14	



Min cost = 38

step-7 :-
Find out the neighbouring edges of 1, 2, 3, 4, 5 select min cost edge & add to the spanning tree.

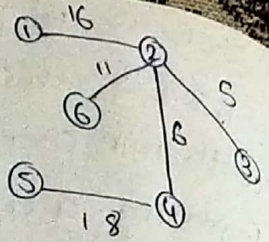
1-6 = 21

1-5 = 19

4-5 = 18 ✓ (4, 5)

4-6 = 14 (forming cycle)

6-5 = 32



Minimum cost of

∴ The minimum cost of spanning tree = 56

Verification:-

→ The final minimum cost spanning tree contains the following steps. It has (n-1) edges.

→ The no. of edges that are removed from the graph

= E - V + 1

* Krushkal's Algorithm:-

Step-1:- Arrange all the edge costs in ascending order of their weights or costs.

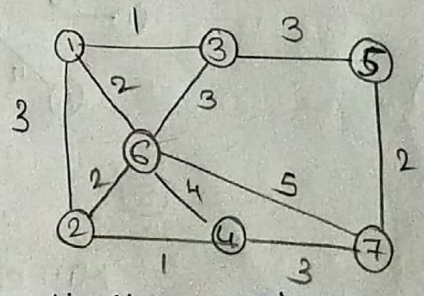
Step-2:- select the smallest edge weight and add it to the spanning tree.

Step-3:- select the next smallest edge weight and add it to the spanning tree without forming a cycle.

Step-4:- Repeat step-3 for all the edges until (n-1) edges and n vertices are present in the spanning tree.

Step-5:- when all the vertices gets visited, then stop the procedure.

1) Find out the minimum cost spanning tree for the following graph using krushkals algorithm.

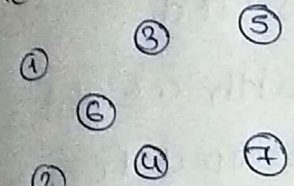


Arrange all the weights of edges in ascending order

edge	cost	Accepted / rejected
1, 3	1	accepted
2, 4	1	accepted
1, 6	2	accepted
2, 6	2	accepted

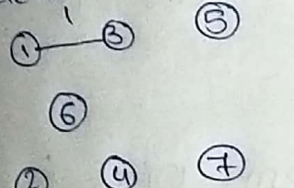
5,7	2	accepted
1,2	3	rejected
3,6	3	rejected
3,5	3	accepted
4,7	3	rejected
4,6	4	rejected
6,7	5	rejected.

step-1:- Initially the spanning tree is empty.



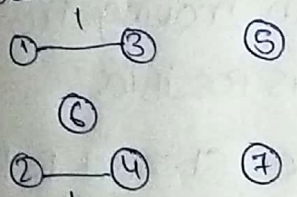
minimum cost = 0

step-2:- First select the smallest edge weight then add it to the spanning tree



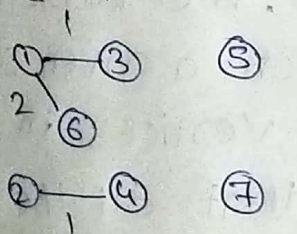
minimum cost = 1

step-3:- Next select the edge (2,4) with cost 1 add it to the spanning tree without forming a cycle.



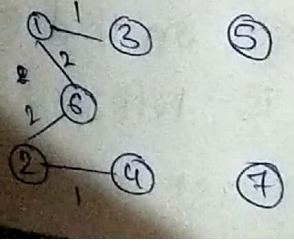
minimum cost = 2

step-4:- Next select the edge (1,6) with weight 2 add it to the spanning tree without forming a cycle



minimum cost = 4

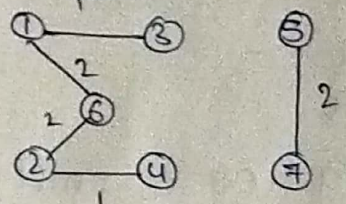
Step-5:- Next select the edge (2,6) with weight 2 add it to spanning tree without forming a cycle.



Minimum cost = 6

Step-6:-

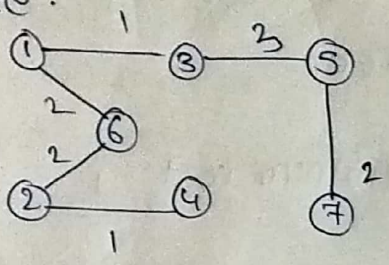
Next select the edge (5,7) with weight 2 add it to the spanning tree without forming a cycle.



Minimum cost = 6+2 = 8

Step-7:- (1,2) rejected, (3,6) rejected.

Next select the edge (3,5) with weight 3 add it to the spanning tree without forming a cycle.



Minimum cost = 11

→ Single source shortest path problem - Dijkstra's Algorithm:-

A graph is used to represent the distance between two cities. Everybody interested in moving from one city to other city as quickly as possible.

For that purpose, single source shortest path is used.

Let G = (V, E) be a graph, The problem is to find the shortest path from a given source vertex V to remaining all the vertices in the given graph. It is assumed that all the distances are positive.

The length of the path is defined as the sum of the weights of the edges on that path. The starting vertex of the path is referred to as the source vertex. and the last vertex

of the path is referred to as destination vertex.

The Greedy strategy is to generate the shortest path from source vertex V to remaining all the vertices in increasing order of length by using the following steps:

step 1:- Make sure that there is no negative edge weights in the given graph. set distance to source vertex as 0 and set all other distances to infinity.

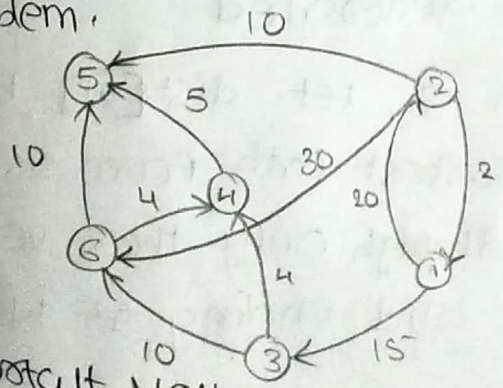
step-2:- Relax all vertices adjacent to the current vertex.

step-3:- choose the closest vertex as the next current vertex.

step-4:- Repeat step 2 and step-3 until reach the destination.

Ex:-

find out the shortest path from source vertex to remaining all the vertices in the given graph using single source shortest path problem.



Shortcut Method:-

	2	3	4	5	6
1	20	15	∝	∝	∝
1,3	20	15	19 (15+4)	∝	25 (15+10)
1,3,4	20	15	19	24 (15+4+5)	25
1,3,4,2	20	15	19	24	25

1,3,4,2,5,6	20	15	19	24	25
1,3,4,2,5,6	20	15	19	24	25

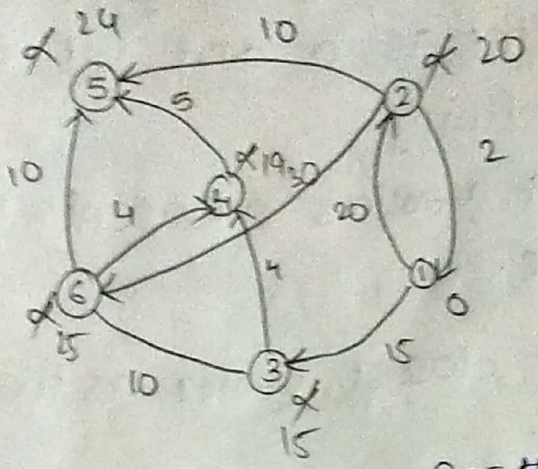
1-2 = 20

1-3 = 15

1-4 = 1-3-4 = 15+4 = 19

1-5 = 1-3-4-5 = 15+4+5 = 24

1-6 = 1-3-6 = 15+10 = 25



→ The algorithm used for the single source shortest path problem using greedy metho leads to a simple algorithm is called Dijkstra's Algorithm.

→ Let 's' denotes the set of vertices (including v) to which the shortest path have already been generated

→ For 'w' not in s, let dist[w], be the length of the shortest path from source vertex v going through only those vertices that are in 's' and ending at w

Solution:-

step 1:- Initially 's' contains empty
We will start from source vertex 1.
Hence s = {1}

dist [2] = 1-2 = 20

dist [3] = 1-3 = 15

dist [4] = ∝

dist [5] = ∝

dist [6] = ∝

Among all the above dist[w's], dist[3] is minimum. Hence, we select vertex 3. This vertex can be added to S.

∴ S = {1, 3}

step-2:-

Now, we have to find out the adjacent vertices of vertex 3. They are 4 and 6

dist[2] = 1-2 = 20

dist[4] = $1 \xrightarrow{3} 4$ u=3, w=4

= min{dist[w], dist[u] + cost[u, w]}

= min{∝, 15 + cost[3,4]}

= min{∝, 15+4}

= min{∝, 19}

dist[4] = 19

dist[6] = $1 \xrightarrow{3} 6$ u=3 w=6

= min{dist[w], dist[u] + cost[u,w]}

= min{∝, 15 + cost[3,6]}

= min{∝, 15+10}

= min{∝, 25}

= 25

dist[5] = ∞

among dist[2], dist[4], dist[6], dist[5]
dist[4] is minimum. Hence we select vertex 4. It can be added to S

$S = \{1, 3, 4\}$

step-3:-

Now we have to select the adjacent vertices of vertex 4. only vertex 5 is adjacent to vertex 4.

$dist[5] = 1 \xrightarrow{4} 5$ $\quad u=4 \quad w=5$

$= min\{dist[w], dist[u] + cost[u,w]\}$

$= min\{\infty, 19 + cost(4,5)\}$

$= min\{\infty, 19+5\}$

$= min\{\infty, 24\}$

$dist[5] = 24$

$dist[2] = 1-2 = 20$

$dist[6] = 25$

step-4:- $S = \{1, 3, 4, 2\}$

Now, we have to select the adjacent vertices of vertex 2. ~~(only vertex 5 is adjacent to)~~ vertex 5, vertex 6 that are vertex 5 is adjacent to vertex 2

$dist[5] = 1 \xrightarrow{2} 5$ $\quad u=2 \quad w=5$

$= min\{dist[w], dist[u] + cost[u,w]\}$

$= min\{24, 20+10\}$

$= min\{24, 30\}$

$dist[5] = 24$

$1 \xrightarrow{2} 6 = 25$.

dist[6] = 25

Among dist[5] and dist[6], dist[5] is minimum select vertex 5 that can be added to S

$S = \{1, 3, 4, 2, 5\}$

Step-5:-

Now we have to select the adjacent vertices of S. There are no adjacent vertices for vertex 5

$\therefore 1 \to 6 = 25$

$\therefore$ vertex 6 can be added to S.

$\therefore S = \{1, 3, 4, 2, 5, 6\}$

$\therefore$ the shortest path from source vertex to remaining all the vertices are.

$1 \to 2 = 20$ $(1 \to 2)$

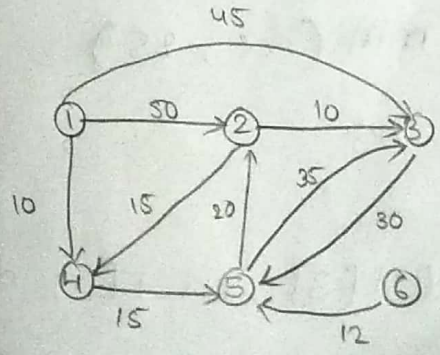
$1 \to 3 = 15$ $(1 \to 3)$

$1 \to 4 = 19$ $(1 \to 3 \to 4 = 15+4 = 19)$

$1 \to 5 = 24$ $(1 \to 3 \to 4 \to 5 = 15+4+5 = 24)$

$1 \to 6 = 25$ $(1 \to 3 \to 6 = 15+10 = 25)$

*** Apply Dijkstra's Algorithm for the following graph and findout the shortest distance from source vertex to remaining all the vertices in the given graph.



step-1:-

Initially 's' contains empty. we will start from source vertex 1.

Hence $S = \{1\}$

step-2:-

$dist[2] = 1-2 = 50$

$dist[3] = 1-3 = 45$

$dist[4] = 1-4 = 10$

$dist[5] = \infty$

$dist[6] = \infty$

Among all the above dist[W's] dist[4] is minimum. Hence we select vertex 4. This vertex can be added to S

$S = \{1, 4\}$

step-2:-

Now, we have to choose the adjacent vertex of vertex 4. They are only one vertex is adjacent to 4 i.e. vertex 5

$dist[2] = 1-2 = 50$

$dist[3] = 1-3 = 45$

$dist[5] = 1-4-5$

$= 1 \xrightarrow{4} 5$

$= \min(dist[W], dist[4] + \dots)$

$= \min(\infty, 10+15)$

$= \min(\infty, 25)$

$dist[5] = 25$

$dist[6] = \infty$

among dist[2] dist[3], dist[5] dist[6] dist[5] is minimum Hence we select

vertex 5 and add it to S

$S = \{1, 4, 5\}$

Step-3:-

Now we have to select the adjacent vertices of vertex 5. vertex 2 and 3 are adjacent to 5

$1 \xrightarrow{5} 2$ $\quad u = 5 \quad w = 2$

$= \min[dist[w], dist[u] + cost(u, w)]$

$= \min[50, 25 + 20]$

$dist[2] = 45$

$1 \xrightarrow{5} 3$ $\quad u = 5 \quad w = 3$

$= \min[dist[w], dist[u] + cost(u, w)]$

$= \min[45, 25 + 35]$

$= \min[45, 60]$

$dist[3] = 45$

$dist[6] = \infty$

Among dist[2], dist[3], dist[6], dist[2] is minimum. Hence we select vertex 2 and add it to S

$S = \{1, 4, 5, 2\}$

Step-4:-

Now we have to select adjacent vertices of vertex 2. vertex 3 is adjacent to 2

$1 \xrightarrow{2} 3$ $\quad u = 2 \quad w = 3$

$= \min[45, 50 + 10]$

$dist[3] = 45$

$dist[6] = \infty$

Among these dist[3] is minimum. so, add it to S

$S = \{1, 4, 5, 2, 3\}$

There is no incoming vertex to vertex 6 so, add it to S

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

The shortest path from source vertex to remaining vertices are

$1 \to 2 = 45$

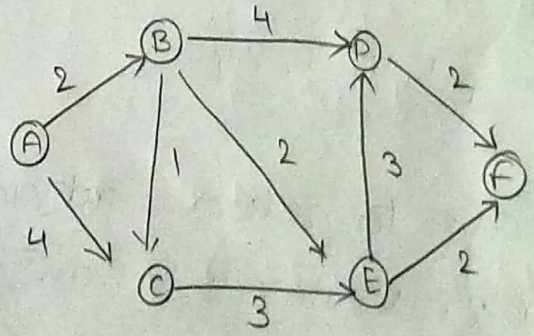
$1 \to 3 = 45$

$1 \to 4 = 10$

$1 \to 5 = 25$

$1 \to 6 = \infty$

Apply Dijkstra's Algorithm for the following graph and find out the shortest distance from source vertex to remaining vertices



Step-1:-

Initially S is empty. We will start from source vertex A

$S = \{A\}$

dist[B] = A − B = 2

dist[C] = A − C = 4

dist[D] = ∞

dist[E] = ∞

dist[F] = ∞

Among all the above dist[W's] dist[B] is minimum. Hence we select vertex B. This vertex can be added to S

$S = \{A, B\}$

step-2:

Now, we have to choose the adjacent vertices of vertex B. vertex C, vertex D, vertex E are adjacent to B

dist[C] = $A \xrightarrow{B} C$ u=B w=C

= min(dist[w], dist[u] + cost(u,w))

= min(4, 2+1)

→ dist[C] = 3 ✓

dist[D] = $A \xrightarrow{B} D$ u=B w=D

= min[dist[w], dist[u] + cost(u,w)]

= min[∞, 2+4]

→ dist[D] = 6 ✓

dist[E] = $A \xrightarrow{B} E$ u=B w=E

= min[dist[w], dist[u] + cost(u,w)]

= min[∞, 2+2]

→ dist[E] = 4

→ dist[F] = ∞

Among these distance of C is minimum so, add it to S

$$S = \{A, B, C\}$$

step-3:-

Now we have to choose the adjacent vertices of vertex C. E is the adjacent vertex of vertex C

$$dist[E] = A \xrightarrow{C} E \qquad u = C \quad \omega = E$$

$$= \min[dist[\omega], dist[u] + cost[u, \omega]]$$

$$= \min[4, 3 + 3]$$

$$dist[E] = 4$$

$$dist[D] = 6$$

$$dist[F] = \infty$$

Among these dist[E] = 4 is minimum. so add it to S

$$S = \{A, B, C, E\}$$

step-4:-

Now select the adjacent vertices of vertex E. D, F are adjacent to E

$$dist[D] = A \xrightarrow{E} D \qquad u = E \quad \omega = D$$

$$= \min[dist[\omega], dist[u] + cost[u, \omega]]$$

$$= \min(6, 4 + 3)$$

$$\rightarrow dist[D] = 6$$

$$dist[F] = A \xrightarrow{E} F \qquad u = E \quad \omega = F$$

$$= \min[dist[\omega], dist[u] + cost[u, \omega]]$$

$$= \min(\infty, 4 + 2)$$

→ dist[F] = 6

Among these dist[D] is minimum. So, add it to S

S = { A, B, C, E, D }

Step-5:-

Now select the adjacent vertices of D

F is the adjacent vertex of D

u = D w = F

dist[F] = A $\xrightarrow{D}$ F

= min[dis[w], dist[u] + cost(u,w)]

= min(6, 6+2)

dist[F] = 6

ADD vertex F to S

S = { A, B, C, E, D, F }

Step-6:-

The shortest path from source vertex to remaining vertices:-

A → B = A − B = 2

A → C = A − B − C = 3

A → D = A − B − D = 6

A → E = A − B − E = 4

A → F = A − B − E − F = 2+2+2 = 6

Algorithm:-

Algorithm shortestPath (v, cost, dist, n)

// dist[J], 1 ≤ J ≤ n, is set to the length of the shortest path from vertex v.

// to vertex J in a diagraph.

// n is the number of vertices.

// dist[v] is set to 0

// G is represented by cost adjacency matrix [1:n, 1:n]

```
{
    for i:=1 to n do
    {
        s[i] := false;        // Initialize s
        dist[i]:= cost[v, i];
    }
    s[v] := true;
    dist[v] := 0;             // put v in s
    for num:=2 to n do
    {
      //Determine n-1 paths from v
      choose u from among those
      vertices not in s, such that
      dist[u] is minimum.
        s[u]:= true;          // put u in s
        for (each w adjacent to u
              with s[w] = false) do
        {
            // update distances
            if (dist[w] > dist[u] +
                cost[u,w]) then
                dist[w]:= dist[u] + cost[u
        }
    }
}
```

Relaxation property

Time complexity:-

The time complexity of single source shortest path problem can be calculated by using the following steps:-

→ for finding the shortest path to all the vertices. It can take O(n) time.

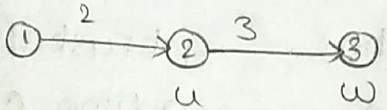
→ Each vertex is to be relaxed i.e; n number of vertices to be relaxed

∴ Time complexity = O(n). O(n)

= O(n²)

→ where n is the no of vertices in the given graph.

Relaxation property:-




```
if (dist[w] > dist[u] + cost[u,w]) then
{
    dist[w] = dist[u] + cost[u,w];
}
```

Differences b/w Prims & Krushkals Algorithm:-

1) Prims Algorithm builds a minimum cost spanning tree by adding one vertex at a time.	1) Kruskals Algorithm build a minimum cost spanning tree by adding one edge at a time
2) The next vertex to be added is always the one nearest to a current vertex already on the graph.	2) The next edge is always the minimum edge weight only if it doesn't form a cycle.

3) prims algorithm span from one node to another node in the given graph.	3) Kruskal's Algorithm select the edges in a way that the position of edges is not based on the last step.
4) In prims algorithm, ~~the spanning tree~~ graph must be a connected graph.	4) In Kruskals algorithm it can be function on disconnected graphs.
5) prims algorithm is used for obtaining M.S.T by selecting the adjacent vertices of already selected vertices	5) Krushkal Algorithm is used for obtaining M.S.T by selecting the minimum weighted edges.
6) prims algorithm initia - lizes with a node.	6) Kruskal's algorithm initializes with a edge.
7) prims algorithm has time complexity of $O(n^2)$ or ~~(E+V)~~ $O(V^2)$ n, V are no. of vertices.	7) Kruskal's algorithm has a time complexity of $O(E \log E)$ where E is the no. of edges