* Algorithm Insertion sort (a, n)
// sort the array a[1:n] into nondecreasing order, n ≥ 1

```
{
    for j := 2 to n do
    {
        // a[1:j-1] is already sorted.
        item := a[j]; i := j-1;
        while ((i ≥ 1) & (item < a[i])) do
        {
            a[i+1] := a[i]; i := i-1;
        }
        a[i+1] := item;
    }
}
```

ex-

For (j = 2)

j	i	item
2	1	a[2] = 3
	0	
3	2	a[3] = 1
	1	
	0	

item < a[i]

{ a[i+1] = a[i]
  a[2] = a[1]

* Time Complexity of insertion sort
If we take a closer look at the insertion sort code, we can notice that every iteration of while loop reduces one inversion. The while loop executes only if i > j
arr[i] < arr[j]

∴ total no of while loop iteration [for all value of i] is same as no of inversions. Therefore overall time complexity of insert sort is $O(n+f(n))$ where $f(n)$ is inversion count. if the inversion count is $O(n)$, then tym complexity of insertion sort is $O(n)$. they can be $n*(n-1)/2$ inversions. worst case time complexity in ~~insertn sort~~ is $O(n^2)$.

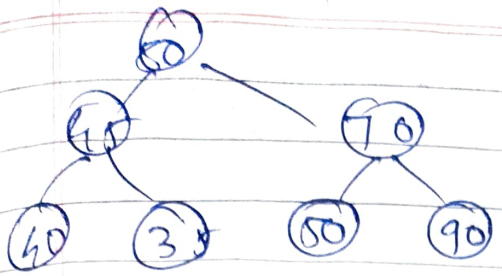
* Heap sort -

A max(min) heap is a complete binary tree with the property that the value at each node is at least as large as (as small as) the values at its children (if they exist).

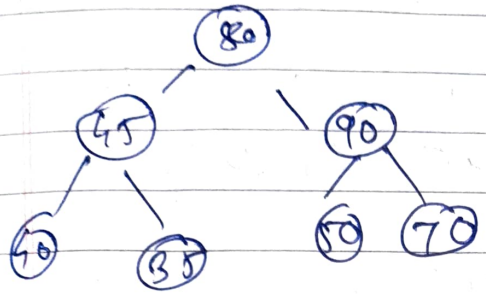
- The defn of max heap implies that one of the largest element is at the root of the heap.
- To insert an element into the heap, one adds it "at the bottom" of the heap & then compares it with its parent, grandparent, greatgrandpar

* ex -

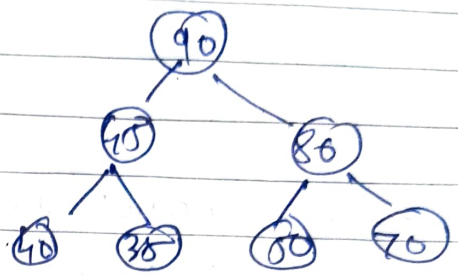
```
Algorithm Insert (a,n)
{
// insert a[n] into heap which is sorted in
                          a[1:n-1]
 i := n; item := a[n];
while ((i > 1) and (a[i/2] < item)) do
     {
       a[i] := a[[i/2]]; i := [i/2];
     }
  a[i] := item; return true;
}
```


60
15
70
40
35
50
90

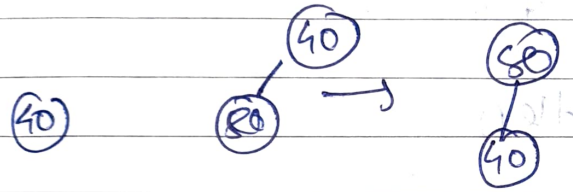
1


80
45
90
40
35
50
70

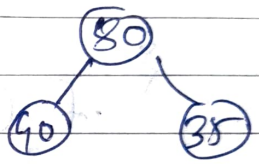
2


90
45
80
40
35
50
70

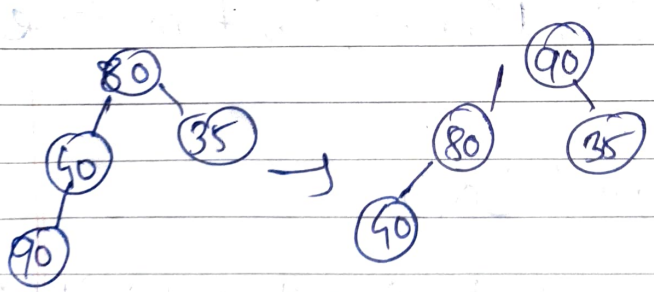
3


40
40
80
50
40

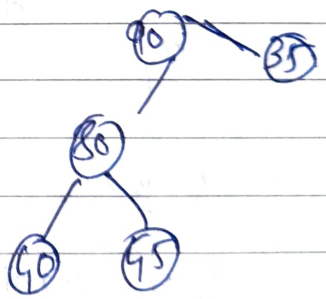
(a) (b)


80
40
35

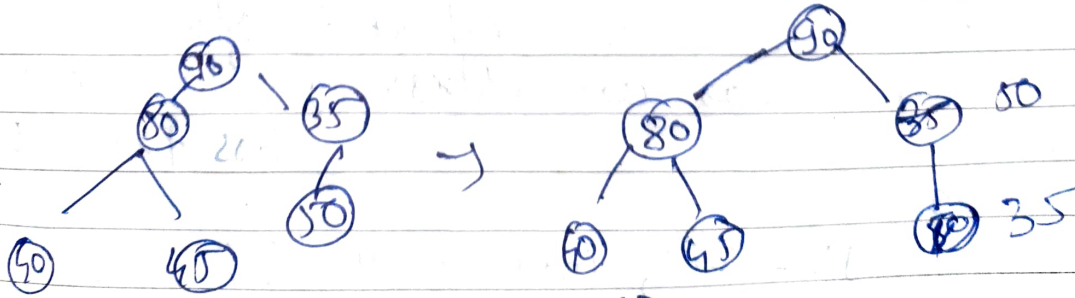
(C)


80
35
40
90
90
80
35
40

(d)

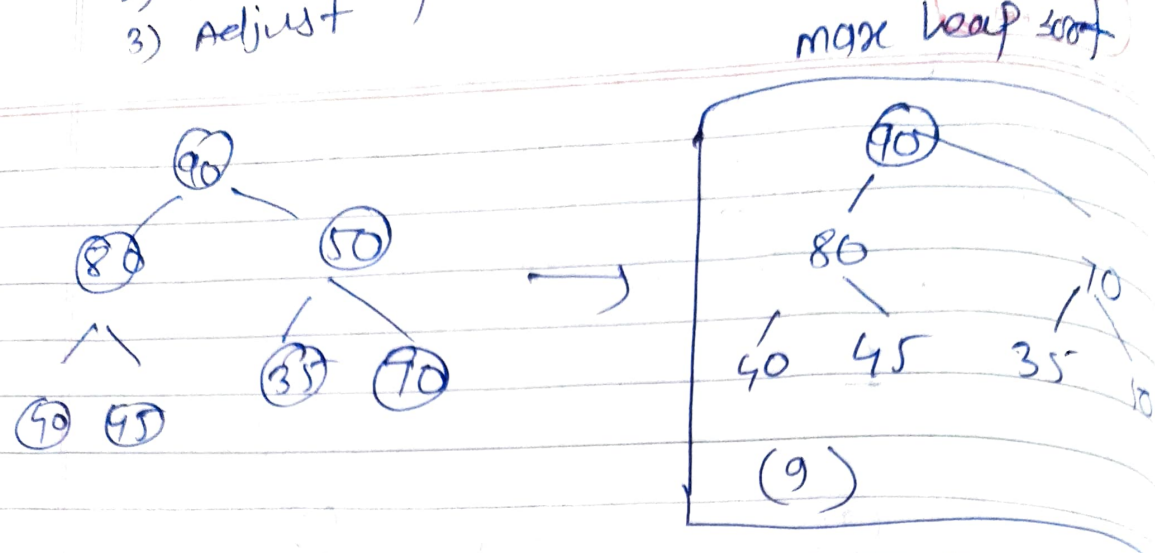

90
35
80
40
45

(e)


90
80
35
40
45
50
90
80
35
40
45
50
35

(f)

1) insert
2) Delete } Algorithem use
3) Adjust



ex: Forming a heap from the set
{ 40, 80, 35, 90, 45, 50, 70 }

* Algorithem DelMax (a, n, x)
// Delete the maximum from heap a[1:n] & store it in x.

```
{
  if (n=0) then
  {
    write ("heap is empty"); return false
  }
  x := a[1]; a[1] := a[n];
  Adjust (a, 1, n-1); return true;
}
```

* Algorithem Adjust (a, i, n)

// The complete binary tree with roots 2i & 2i+1
// combied with nodes i to form a heap
rooted at i. No
// node has an address greater than n or
less than 1.

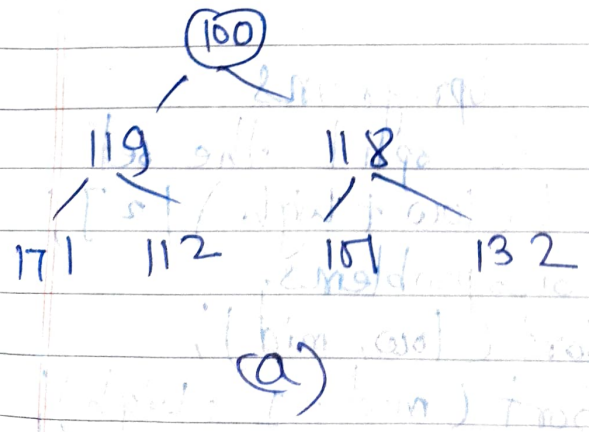
```
{
  j := 2i, item := a[i];
```

```
while (j<n) do
{
  if ((j<n) & (a[i] < a[j+1])) then j := j+1;
  // compare left & right child.
  // and let j be larger child.
  if (item >= a[i]) then break;
        // A position for item is found.
  a[[j/2]] := a[i]; j := 2j;
}
  a[[j/2]] := item;
}
```

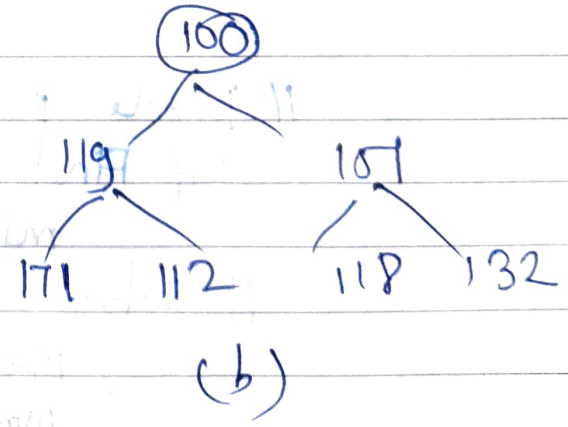
* Algorithm Heapify (a, n)

① // Readjust the element in a[1:n] to form a heap.

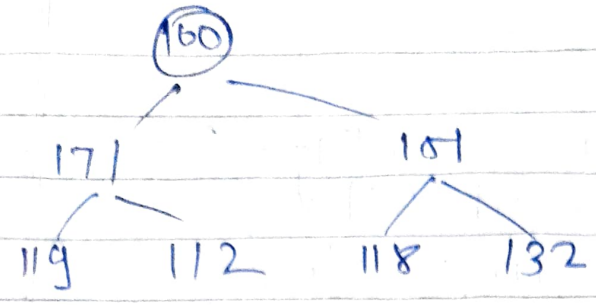
```
{
  for i := [n/2] to 1 step -1 do
        Adjust (a, i, n);
}
```



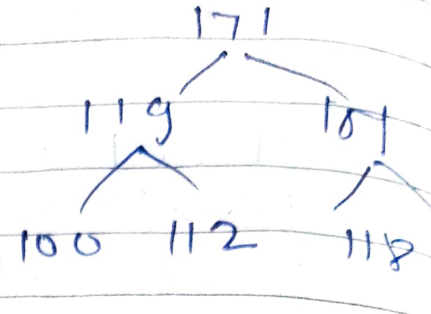
(a)



(b)



(c)



(d)

* Heap sort tym Complexity :

$O(n \log_n)$ - Reverse tym complexity

$O(n)$ - Adjust

$O_n \log_n$ - worst tym complexity.

Merge sort what is divide & ~~conquer~~ conquer statrgy ?

* Algorithm Merge Sort (low, high)

① // allow : high is a global array to be sorted.
// small (p) is true if there is only one
// to sort. list is already sorted. element.

{

if (low < high) then // If there are more than one element

{

// Divide p into subprograms

// Find where to split the set.

mid := [(low + high) / 2];

// Solve the subproblems.

Merge Sort (low, mid);

Merge Sort (mid + 1 , high);

// Combine the solutions

Merge (low, mid, high);

}

* Divide & Conquer

(4)

```
Algorithm DAndC (P)
{
    if Small (P) then return S (P);
    else
    {
        divide P into smaller instances
                                P1, P2, ... Pk, k >= 1;
        Apply DAndC to each of these
                                    subproblems;
        return Combine (DAndC (P1), DAndC (P2),
                                    DAndC (Pk));
    }
}
```

* Counting Sort:

(5) Let a[1...n] be an array of numbers to be sorted. let 1 <= A[i] <= k, for i = 1, 2, ... n.

- for each element in a[i], the number of elements of A < A[i] is determined. If these are 5 elements < a[3], then a[3] goes to 6th place in the sorted array.
- B[1..n] stores the sorted array & C[1..k] provides temporary working space. Array C is neccessary to handle situation in which several element have same value.

Algorithm : Counting Sort

```
Algorithm Counting sort (A, B, k)
{
  for (i := 1 to k) //      loop 1
    c[i] := 0;
  for (j := 1 to length of [A]) // loop 2
    C[A[j]] := c[A[j]] + 1;
                    // c[i] = the no of elem
  equal to i
    for (i := 2 to k) // loop 3
      c[i] := c[i] + c[i-1];
                    // c[i] = the no of elem
  less than or equal to i
    for (j := length [A] down to 1) // loop 4
    {
      B[c[A[j]]] := A[j];
      c[A[j]] := c[A[j]] - 1;
    }
}
```

* Binary Search

```
Algorithm BinSearch (a, n, x)
// Given an array a[1:n] of elements in
     nondecreasing
// order, n >= 0, determine whether x is
     present
// if so, return j such that x = a[j];
                    else return 0.
{
```

```
    low := 1; high := n;
    while (low ≤ high) do
      {
        mid := [(low + high)/2];
        if (x < a[mid]) then high :=
                              mid - 1;
        else if (x > a[mid]) then low := 
                              mid + 1;
             else return mid;
      }
      return 0;
}
```

* Recursive Binary Search

(4)

```
Algorithm BinSrch(a, i, l, x)
// Given an array a[i:l] of elements in
                    nondecreasing
// order, 1 ≤ i ≤ l, determine whether x
                    is present
// if so, return j such that x = a[j];
                    else return 0.
{
  if (l = i) then // If Small(P)
    {
      if (x = a[i]) then return i;
      else return 0;
    }
  else
  {
    // Reduce P into a smaller subproblm
```

```
    mid := [(i+l)/2];
    if (x = a[mid]) then return mid;
    else if (x < a[mid]) then
        return BinSrch (a, i, mid-1, x);
    else return BinSrch (a, mid+1, l, x);
}
```

Ex:- let select the 14 entries

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

place in a[1:14], & simulate the steps that BinSearch goes through as it searches for different value of x. only variables low, high & mid need to be traced. we try following value for x: 151, -14 & 9 two successful search & one unsuccessful.

→

x: 151	low	high	mid
	1	14	7
	8	14	11
	12	14	13
	14	14	14 found

x: -14	low	high	mid
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	not found

Q:]

low	high	mid
1	14	7
1	6	3
4	6	5

round

* Algorithm Merge (low, mid, high)
// a[low: high] is global array containing two sorted.
// subsets in a[low: mid] & in a[mid+1: high]
// the goal is to merge these two sets into a single set residing
// in a[low: high]. b[] is an auxiliary global array.

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

```
else
    for k := h to mid do
    { b[i] := a[k]; i := i+1; }
    for k := low to high do a[k] := b[k]
}
```

Quick Sort

In quicksort, the division into two subarray is made so that the sorted subarray do not need to be merged later. This is accomplished by rearranging the elements in a[1:n] such that a[i] ≤ a[j] for all i between 1 and m and all j between m+1 and n for some m, 1 ≤ m ≤ n. Thus the elements in a[1:m] and a[m+1:n] can be independently sorted. No merge is needed. The rearrangement of element accomplished by picking some element of a[], say t = a[s], and rearranging the other elements so that all element appearing before t in a[1:n] are less than or equal to t and all element appearing after t are greater than or equal to t. This is the partitioning.

* Algorithm QuickSort(p,q)
// sorts the elements a[p], ... a[q] which reside in the global
// array a[1:n] into assending order;
a[n+1] is considered to
// be defined & must be ≥ all element in
a[1:n]

{
if (p<q) then // if there are ~~one~~ more than one element
{
// divide P into two subproblems
j := partition(a, p, q+1);
// j is the position of partitioning elements.
// solve the subproblems
QuickSort(p, j-1);
QuickSort(j+1, q);
// There is no need for combining sol^n
}
}

* Algorithm Partition(a, m, p)
// within a[m], a[m+1], ... a[p-1] the elements are
// rearranged in such a manner that if initially t = a[m],
// then after completion a[q] = t for some q between m
// & p-1, a[k] ≤ t for m ≤ k < q, & a[k] ≥ t
// for q < k < p. q is returned. set a[p] = ∞.

* Algorithm interchange (a, i, j)
// Exchange a[i] with a[j].
{
p := a[i];
a[i] := a[j]; a[j] := p;
}

* Algorithm Partition (a, m, p)
{
v := a[m]; i := m; j := p;
repeat
{
repeat
i := i + 1;
until (a[i] ≥ v);
repeat
j := j - 1;
until (a[j] ≤ v);
if (i < j) then interchange (a, i, j);
} until (i ≥ j);
a[m] := a[j]; a[j] := v;
return j;
}

~~Base case~~

- Best case behavior of quicksort occur two regions of size $n/2$

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

- The average case running time of quick sort is much closer to the best case than worst case
- worst case is rare occurence it occure array is already sorted.
- Average case sorting time of quick sort is $O(n \log n)$. It is high space complexity being recursive
- It is an place algorithm, it is not stable
- quick sort, Merge sort both use divide & conquer strategy
- quick sort spends more time in dividing while merge sort spends more time in combining.

i	pos	①	②	③	④	⑤	⑥	⑦	⑧	⑨	⑩
2	9	65	70	75	80	85	60	55	50	45	too
3	8	65	45	75	80	85	60	55	50	70	too
4	7	65	45	50	80	85	60	55	75	70	too
5	6	65	45	50	55	85	60	80	75	70	too
6	5	65	45	50	55	60	85	80	75	70	too
		60	45	50	55	65	85	80	75	70	too

Notice that remaining elements are unsorted but partitioned about $a[5] = 65$.

* Strassen's Matrix multiplication

Let A & B be two n x n matrices. The product matrix C = AB is also an n x n matrix whose i, jth element is formed by making the elements in the i^{th} row of A & the j^{th} column of B & multiplying them

$$C(i,j) = \sum_{1 \le k \le n} A(i,k) B(k,j)$$

for all i & j between 1 & n. To compute C(i,j) using formula, we need n multiplictn As the matrix C has n^2 element, the time for the ~~not~~ resulting matrix multiplicatn algorithm which we refer to as the conventional methods is $O(n^3)$.

The divide & conquer strategy suggests the another way to compute the product of two n x n matrices. For simplicity n is the power of 2, that exists a nonnegative integer k such that $n = 2^k$.

- Since matrix multiplication is more expensive than matrix addition we can attempt to reformulate eqn for Cij so as to have fewer multiplication & possibly more additions.

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$Q = (A_{21} + A_{22}) B_{11}$$
$$R = A_{11}(B_{12} - B_{22})$$
$$S = A_{22}(B_{21} - B_{11})$$

$T = (A_{11} + A_{12})\, B_{22}$

$U = (A_{21} - A_{11})(B_{11} + B_{12})$

$V = (A_{12} - A_{22})(B_{21} + B_{22})$

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U$

* Matrix multiplictn

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$C_{11} = A_{11}B_{11} + A_{12}B_{21}$

$C_{12} = A_{11}B_{12} + A_{12}B_{22}$

$C_{21} = A_{21}B_{11} + A_{22}B_{21}$

$C_{22} = A_{21}B_{12} + A_{22}B_{22}$

* Intro

- Consider a problem p with n inputs say $x_1, x_2 \ldots x_n$.
- we want to find values of $x_1, x_2 \ldots x_n$ satisfying given constraints called feasible soln, p is called objective function.
- A feasible soln that maximizes or minimize objective functn called optimal soln.
- Algorithm for most of the optimizatn prblm can be obtained using greedy method.

* Elements of greedy strategy

A greedy algorithm obtains an optimal sol^n to prblm by making a sequence of choic. This strategy does not always produce an optimal sol^n.

It has two property:-

① Greedy choice property:-

- a globally optimal sol^n can be obtained by making locally optimal (Greedy) choice.
- We make whatever choice seems best at the moment & solve the subproblems arising after choice is made.
- A choice made by greedy algorithm may depend on choices so far, it is independent on any future choice.
- Greedy strategy usually progress on a top down fashion, iteratively reducing each problem instance to smaller one.

② Optimal substructure:-

A problem is said to have optimal substructure property if an optimal sol^n to the prblm contains optimal sol^n to its subproblems.

- control abstraction:

let a[1:n] be set of n inputs we want to find optimal sol^n for this input. let select the funct^n which select an i/p input from a. Assign this selected input to variable say x. feasible is a funct^n which return true if x can include in sol^n vector. feasible returns false otherwise.

function union update soln vector
- this method may not always give optimal soln.

* Greedy Algorithm

Algorithm Greedy (a,n)
{
  soln := ϕ ; // initialize the soln
  for (i:= 1 to n) do
    {
      x := select (a);
      if (Feasible (solution, x)) then
          solution := union (solution, x);
    }
    return solution ;
}

* knapsack problem
- suppose a student is going on a trip He has a single knapsack that can carry items of weight at most 'm'
- he is allowed to break items into fraction arbitrarily (like 0.25 kg - clips) each item i has some utility value say pi He wants to fill his knapsack with items most useful to him with total weight at most m.
- this problem can be formally put into form as:

- Given: A knapsack of capacity m & n object each object i has weight w_i $i = 1, 2, \ldots n$
- if fraction x_i of object is is included profit $p_i * x_i$ is earned.

- objective: To find soln vector $(x_1, x_2 \ldots x_n)$ such that the total profit earned is

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le n$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n$$

i) it is easy to observe that: if

$$\sum_{1 \le i \le n} w_i : \le m$$

then $x_i = 1$ for all i is optimal soln.

ii) All optimal solutn these are 3 approaches

① Greedy by profit: we try to fill the knapsack by including next the object with largest profit. if that object does not fit in the knapsack, then the fraction of it is included to fill the knapsack.

A thief enters a store & see following items:

A	B	C
[$100]	[$10]	[$120]
2pd	2pd	3pd

His knapsack holds 4 pounds. what should he steal to maximize profit?

→ soln:

① Fractional knapsack problem: Thief can take a fraction of an item

→ solution = + 2 pounds of item A
2 pounds of item C

2 pds A $100	2 pds C $80

② 0-1 knapsack problem: Thief can only take or leave item. He can't take a fraction.

→ solution = 3 pounds of item C

3 pds C $120	

Algorithm Greedy knapsack (m, n)

// P[1:n] & w[1:n] contain the profits & weights resp of n object ordered such that
- P[i]/w[i] >= p[i+1]/w[i+1]
// m is the size of knapsack & x[1:n] is the soln vector

```
{
  for i := 1 to n
    { do x[i] := 0.0; }  // initialize x
  U := m;
    for i := 1 to n do
      {
        if (w[i] > U) then break;
          x[i] := 1.0;
          U := U - w[i];
      } // end of for
  if (i <= n) then
  x[i] := U / w[i];
}
```

Example ①

- Find a soln for knapsack prblm
  n = 3 p = (25, 24, 15)
  m = 20 w = (18, 15, 10)

→ method 1 : Arrange the object in non-increasing i.e decreasing order of profit.

p = (25 (P1), 24 (P2), 15 (P3))
w = (18, 15, 10)
     1   2   3

select object initially	solution	Remaining capacity m = 20
1	x1 = 1	(m - w) = (20 - 18) = 2
2	x2 = 2/15	(2 - 15) = 0
3	x3 = 0	0

- Now $X = (x_1, x_2, x_3) = (1, 2/15, 0)$
- To find profit

$\Sigma p_i * x_i = (x_1)(P_1) + (x_2)(P_2) + (x_3)(P_3)$

$= (1)(25) + (2/15)(24) + (0)(15)$

$= 25 + 3.2$

$\Sigma p_i * x_i = 28.2$

<u>Method 2</u> : Arrange the object in increasing order of weight :

$P = (15, 24, 25)$ $\quad w = (10, 15, 18)$

$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad 3 \quad 2 \quad 1$

select object	solution	Remaining capacity
initially		$m = 20$
3	$x_3 = 1$	10
2	$x_2 = 10/15 = 2/3$	$[10 - 15] = 0$
1	$x_1 = 0$	0

Now $x = (x_1, x_2, x_3) = (0, 2/3, 1)$

To find profit

$\Sigma p_i * x_i = (x_1)(P_1) + (x_2)(P_2) + (x_3)(P_3)$

$= (0)(25) + (2/3)(24) + (1)(15)$

$= 16 + 15$

$\Sigma p_i * x_i = 31$

<u>method 3</u> : Arrange the p & w in non-increasing order of p/w ratio i.e decreasing order.

$P = (25, 24, 15)$

$w = (18, 15, 10)$ $\quad P/w = (25/18, 24/15, 15/10)$

$= (1.37, 1.6, 1.5)$ 1 2 3

Now arrange p/w ration in decreasing order
p/w = (1.6, 1.5, 1.37)
2 3 1
Arrange weight in decreasing order
w = (15, 10, 18)

select object initially	solution	Remaining capacity m = 20
2	$x_2 = 1$	5
3	$x_3 = 5/10 = 1/2$	[5-10] = 0
1	$x_4 = 0$	0

Now $x = (x_1, x_2, x_3) = (0, 1, 1/2)$
To find profit
$\sum p_i x_i = (x_1)(p_1) + (x_2)(p_2) + (x_3)(p_3)$
$= (0)(25) + (1)(24) + (1/2)(15)$
$= 24 + 7.5$
$\sum p_i x_i = 31.5$

② Find the soln of Knapsack problem
n = 7 p = (10, 5, 15, 7, 6, 18, 3)
m = 15 w = (2, 3, 5, 7, 1, 4, 1)

→

metho 3: Arrang p & w in non increasing order of p/w ratio.

p = (10, 5, 15, 7, 6, 18, 3) , w = (2, 3, 5, 7, 1, 4, 1)
p/w = (5, 1.6, 3, 1, 6, 4.5, 3)
1 2 3 4 5 6 7
Now arrange p/w ration in decreasing order

P/W = (6, 5, 4·5, 3, 3, 1·6, 1)
5 1 6 3 7 2 4

Arrange weight in decreasing order
w = (1, 2, 4, 5, 1, 3, 7)

select object initially	solution	Remaining capacity m = 15
5	$x_5 = 1$	14
1	$x_1 = 1$	12
6	$x_6 = 1$	8
3	$x_3 = 1$	3
7	$x_7 = 1$	2
2	$x_2 = 2/3$	0
4	$x_4 = 0$	0

Now $X = (x_1, x_2, x_3, x_4, x_5, x_6, x_7)$
$= (1, 2/3, 1, 0, 1, 1, 1)$

To find profit $\sum p_i x_i = (x_4)(p_1) + (x_2)(p_2) + (x_3)(p_3)$
$+ (x_4)(p_4) + (x_5)(p_5) + (x_6)(p_6) + (x_7)(p_7)$
$= (1)(10) + (2/3)(5) + (1)(15) + (0)(7) +$
$(1)(6) + (1)(18) + (1)(3)$
$= 10 + 3.3 + 15 + 6 + 18 + 3$
$\sum p_i x_i = 55.33$

In case sum of all weights is $\leq m$ then $x_i = 1$, $1 \leq i \leq n$ is an optimal soln.
If $P_1/w_1 \geq P_2/w_2 \geq \cdots P_n/w_n$, then Greedy knapsack generates an optimal soln to the given instance of knapsack problem.

Job sequencing with deadlines:

- we are given a set of n jobs associated with job i is an integer deadline di >=0 & profit pi>0.
- For any job i the profit pi is earned if & only if the job is completed by its deadline.
- To complete a job, one has to process the job on machine for one unit of time
- only one machine is available for processing jobs
- A feasible soln for this problem is subset J of of jobs such that each job in this subset can be completed by its deadline
- The value feasibe soln j is the sum of profits of jobs in J.
- optimal soln is feasible soln with max value

- problem: given jobs [J1,.. Jn) with deadlin (d1, ...dn) & profits (P1,.. Pn) to find subset of {J1,...Jn), J= {Ji1 .. Jik such that $\sum_{j=1}^{k} P_{ij}$ is max.

Strategies:

① we shall arrange the jobs in non increasing order of profit. A subset of jobs i1, i2 ... ik is feasible if the su with, d(i1) <= d(i2) <= ... <=d(ik) is fe

② Thus we shall select a sequence such that

$d(J_1) <= d(J_2) <= \dots <= d(J_k)$

③ for each selected job we must have $d(J_r) >= r$

- The place for ith job is obtained
  - a) $d(J_1) <= \dots <= d(J_r) <= d(i) < d(J_{r+1}) <= \dots$
  - b) we can push the jobs from J_{r+1} to J_k by one shot
  - c) Also, $d(i) >= (r+1)$
- In case we shall select job i otherwise reject.

Ex: using job sequencing find the profit for given data.

$n = 4$ $p = (100, 10, 15, 27)$ $d = (2, 1, 2, 1)$
1 2 3 4

→ Arrange the data in non increasing order of given profit

$n = 4$ $p = (100, 27, 15, 10)$ $d = (2, 1, 2, 1)$
1 4 3 2

job considered	job sequence		solution
1	1	[1, 2]	{1}
4	{4, 1}	[0, 1]	{1}
3	Reject 3		{1, 4}
2	Reject 2		{1, 4}

profit = job 1 + job 2
= 100 + 27 = 127

① since job 1 can be executed in 0-1 or 1-2 interval so it is executed.

④ job 4 can be executed in 0-1 as it has deadline 1 because if job1 is getting executed in interval 1-2 then we have 0-1 interval free, so job 4 can get executed in 0-1 interval

③ job 3 is rejected as it has deadline 2 i.e 0-1 & 1-2 interval in which job 4 & job 1 has already been executed, so there is no interval left for job 3 to execute.

② job 2 is rejected as it has deadline 1 i.e 0-1 in which job4 has already been executed so there is no interval left for job2 execute.

(12) * Algorithm JS(d, j, n)
// d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines. n ≥ 1.
// are ordered such that p[1] ≥ p[2] ≥ ··· ≥ p[n].
// i is the ith job in the optimal soln. 1 ≤ i ≤ k.
// Also, at termination d[J[i]] ≤ d[J[i+1]]
1 ≤ i < k.

```
{
  d[0] := J[0] := 0;      // Initialize
  J[1] := 1;              // include job 1.
  k := 1;
  for i := 2 to n do
  {
    // consider jobs in nonincreasing order p[i].
    // position for i & check feasibility of insertion
      r := k;
  while ((d[J[r]] > d[i]) &
         (d[J[r]] ≠ r)) do r := r - 1;
  if ((d[J[r]] ≤ d[i]) & (d[i] > r)) then
```

```
    {
    // insert i into J[].
    for q := k to (r+1) step -1 do J[q+1] :=
                                        J[q];
      J[r+1] := i;  k := k+1;
    }
  }
  return k;
}
```

* Algorithm GreedyJob (d, J, N)
// J is a set of jobs that can be completed by their deadlines

```
{
    J := {1};
    for i := 2 to n do
    {
      if (all jobs in J ∪ {i} can be completed
          by their deadlines ) then J := J ∪ {i};
    }
}
```

ex Let $n = 4$, $(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$ & $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. The feasible Soln & values are:

	feasible Soln	processing sequence	value
1	(1, 2)	2, 1	110
2	(1, 3)	1, 3, or 3, 1	115
3	(1, 4)	4, 1	127
4	(2, 3)	2, 3	25

Job set	job considered	Assigned slot	Actn sequence
ϕ	2	—	Assign [0-1] —
{2}	4	[0-1] [1-2]	Assign {2}
{2,4}	1	[0-1] [1-2]	Assign {2,4}
{2,4}	3	[0-1] [1-2] [2-3]	Assign {2,4}
{2,3,4}	5	[0-1]	Reject jobs {2,3,4}

profit = job 2 + job 3 + job 4
= 15 + 7 + 12
= 34.

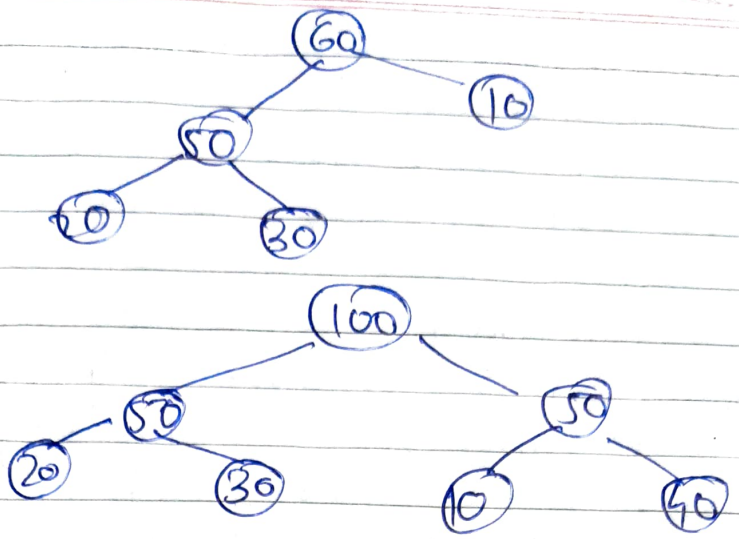
* Optimal Merge Pattern

- Two sorted files containing m & n records can be merged together to get a sorted file in O(m+n)
- If there are n files to be merged, there are several possible orders in which merging can take place with different associated costs

Ex n = 3 files of size 20, 30 & 10

1. [[1 2] 3] 20 + 30 = 50 50 + 10 = 60 T 110
2. [1 [2 3]] 30 + 10 = 40 20 + 40 = 60 T 100
3. [[1 3] 2] 20 + 10 = 30 30 + 30 = 60 T 90

The above merged patterns are two way merge patterns & can be represented form of binary merge trees.



- The optimal two way merge pattern corresponds to a binary merge tree with minimum weighted external path length
- The greedy approach is at every stage select the two smallest files & merge them. The new file obtained is a used in subsequent merges.
- Each file is a tree node with left child, right child & weight which is file length.
- The tree nodes are maintained as a list with two operatn.
- least (list) - returns the smallest element
- Insert (list, t) - insert the tree node in the list

* Dynamic Programming :-
- It is typically applied to optimizatn prblm optimizatn problm - the soln are plenty with each soln there is an associated value - it can be cost or profit. The soln that maximizes cost is known as optimal soln

Solving optimizt^n problem is to get optimal sol^n for ex: o/1 knapsack prblm.
- A brute force approach is generating all sol^n & then finding amongst them the one having lowest cost or highest profit. This will required a lots of efforts.
Effort can be reduced in two ways.
* In obtaining sol^n a strategy similer to divide & conquer strategy can be used divide prblm into subprblm, find the sol^n of main prblm. since process is carried out of get all possible sol^n, there is possibility of overlapping subprblm, same subprblm a part of two or more subprblm The solving of same subprblm repeatively can be avoided by storing its value once it is computed & using all over again
* Avoid solving those subprblm which are no way anywhere near optimal sol^n by attaching value with each subprblm.

* Matrix chain Multiplicat^n problem
Given a sequence (chain) $(A_1, A_2, \ldots A_n)$ of n matrices, the product $A_1 A_2 A_3 \ldots A_n$ is to be computed.
- There is already an algorithm to compute the product of two matrices which is to be repeatedly applied
- The matrix multiplicat^n algo for two matr. of order nxm by mxl involves nxmxl multiplict^n

The cost of multiplying n matrices will depend on how the chain is parenthesized

ex: N = 3 $(A_1 A_2 A_3)$ $A_1 = 10 \times 100$ ~~A_2~~
$A_2 = 100 \times 10$ ~~A_3~~
$A_3 = 10 \times 100$

* 1 $[[A_1, A_2] A_3]$
The product $[A_1 A_2]$ involves $10 \times 100 \times 10 = 10000$.
multiplictn giving a matrix which is 10×10.
The product of this 10×10 matrix with A_3 involves $10 \times 10 \times 100$ multiplictn.
Total multiplictn = 10000 + 10000 = 20000

* 2 $[A_1 [A_2, A_3]]$
The product $[A_2, A_3]$ involves $100 \times 10 \times 100 = 100000$. multiplicating giving a matrix which is 100×100.
The product of A_1 with this 100×100 matrix involves $10 \times 100 \times 1000$ multplictn.
Total multiplictn = 100000 + 100000 = 200000
- Thus the way the chain is parenthesized decides the cost of evaluating the product
- The optimal soln to matrix chain multiplicatn problm is that parenthesizatn that gives minimum cost.
- Given a set of n matrices there can be many ways of parenthesizing it.

1) $[[A_1 \ldots A_7][A_8 \ldots A_{16}]]$
$[[[A_1 \ldots A_4][A_5 \ldots A_7][A_8 \ldots A_{10}][A_{11} \ldots A_{16}]]$

2. $[A_1 \ldots A_2][A_3 \ldots A_{16}]]$
$[A_1 \ldots A_2][A_3 \ldots [[A_5 \ldots A_7][A_8 \ldots A_{10}]$
$[A_{11} \ldots A_{16}]]$

- Dynamic
those prblm which
substructure property.
- A prblm exhibits optimal substructure property if an optimal soln to the prblm contain within it optimal soln to sub prblm.
- dynamic programming algorithm can be broken into 4 steps.
① characterize the structure of an optimal soln.
② Recursively define the value of an optimal soln
③ compute the value of an optimal soln, typically in a bottom-up fashion
④ construct an optimal soln from the computed a informtn.

Travelling salesperson problem:
The travelling salesperson problem is to find a tour of minimum cost
- let $G = (V, E)$ be directed graph with $|V| = n > 0$ & edge costs c_{ij} where $c_{ij} > 0$

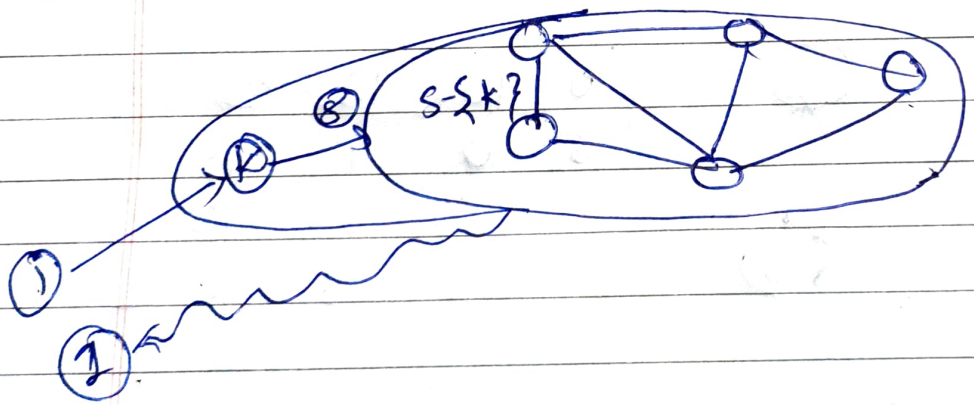
* Dynamic programming can be applied to those prblm which exhibit optimal substructure property.
- A prblm exhibits optimal substructure property if an optimal soln to the prblm contain within it optimal soln to sub prblm.
* dynamic programming algorithm can be broken into 4 steps.
① characterize the structure of an optimal soln.
② Recursively define the value of an optimal soln
③ compute the value of an optimal soln, typically in a bottom-up fashion
④ construct an optimal soln from the computed & inform tn.

* Travelling salesperson problem:
The travelling salesperson problem is to find a tour of minimum cost
- let $G = (V, E)$ be directed graph with $|V| = n > 0$ & edge costs c_{ij} where $c_{ij} > 0$ for $i \neq j$ & $c_{ij} = \infty$ if (i, j) not in E
- A tour of G is a directed simple cycle that includes every vertex in V.
The cost of tour is sum of cost edges on the tour without loss of generality assume tour starts & end at vertex 1.
- every tour consist of an edge $(1, k)$ for some k in $V - \{1\}$ & path from

vertex k to 1 going through each vertex in V - {1, k} exactly once.

① Let us denote by T(i, S) the problem of starting at vertex i & travelling through all vertices in set S & then reaching back to 1 as tour should always end at 1

② The optimal soln to T(i, S) will involve going from i to some vertex k & then travelling through all vertices in S - {k} in an optimal manner which is the optimal soln to the prblm T(k, S - {k})
Thus the problem stasfies substructure property.



② let g(i, S) denote the cost of the optimal tour starsting from i, visiting all vertices in S & ending at 1.
- The TSP problem is to get g(1, S)
- If S is empty g(i, ϕ) will be the cost of the tour starting at i visiting no vertices as S is empty & reaching 1

Thus the cost is of directly moving from i to 1 i.e C_{i1}

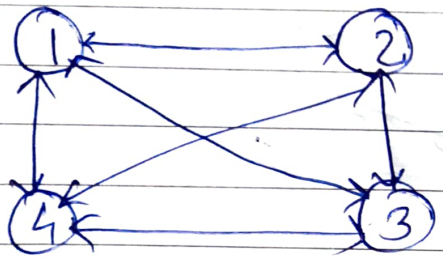
$g(i, \phi) = C_{i1} \quad 1 \le i \le n$

If S is not empty. The cost of moving from i to some k is C_{ik} & then $g(k, S-\{k\})$ is the optimal cost of reaching 1 after travelling through remaining vertices.

- we can close that k in S which gives the minmum cost $g(i, S) = \min_{k \in S} \{C_{ik} + g(k, S-\{k\})\}$

3. Compute the values into bottom up manner

* Consider a TSP instance given by following graph & cost matrix.



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

→ $g(i, S) = \min_{k \in S} \{C_{ik} + g(k, S-\{k\})\}$

$g(2, \phi) = C_{21} = 5$

$g(3, \phi) = C_{31} = 6$

$g(4, \phi) = C_{41} = 8$

Next Consider singleton set $\{2\}$ $\{3\}$ & $\{4\}$

$g(i, S) = \min_{k \in S} \{C_{ik} + g(k, S-\{k\})\}$

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

$= c_{23} + g(3, \phi) = 9 + 6 = 15$

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

$= c_{24} + g(4, \phi) = 10 + 8 = 18$

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

$= c_{32} + g(2, \phi) = 13 + 5 = 18$

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

$= c_{34} + g(4, \phi) = 12 + 8 = 20$

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

$= c_{42} + g(2, \phi) = 8 + 5 = 13$

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

$= c_{43} + g(3, \phi) = 9 + 6 = 15$

Next: Compute $g(i,s)$ with s containing 2 elements

$g(2, \{3,4\}) = \min(c_{23} + g(3,\{4\}),$
$c_{24} + g(4,\{3\}))$
$= \min(9+20, 10+15)$
$= \min(29, 25)$
$= 25$

$g(3,\{2,4\}) = \min(c_{32} + g(2,\{4\}),$
$c_{34} + g(4,\{2\}))$
$= \min(13+18, 12+13)$
$= \min(31, 25)$
$= 25$

$g(4,\{2,3\}) = \min(c_{42} + g(2,\{3\}),$
$c_{43} + g(3,\{2\}))$
$= \min(8+15, 9+18)$
$= \min(23, 27)$
$= 23$

Finally,

$g(1,\{2,3,4\})$
$= \min(c_{12} + g(2,\{3,4\}), c_{13} + g(3,\{2,4\}),$
$c_{14} + g(4,\{2,3\}))$
$= \min(35, 40, 43)$
$= 35$

* Computing the values in bottom up manner
Consider the knapsack instances $n=4, m=20$
Prblm $(w_1, w_2, w_3, w_4) = (16, 12, 8, 6)$ &
$(p_1, p_2, p_3, p_4) = (32, 20, 14, 9)$

H.W → $n=4, m=20$
$p = (15, 11, 13, 9)$
$w = (9, 6, 7, 4)$

(1) n=5, m=12, P = (10, 15, 6, 8, 4)
w = (4, 6, 3, 4, 2)

→ $f_5(12) = \max \begin{cases} 0 \\ 1 \end{cases} \begin{cases} f_4(12) \\ f_4(12-2)+4 \end{cases}$

$f_4(12) = \max \begin{cases} 0 \\ 1 \end{cases} \begin{cases} f_3(12) \\ f_3(12-4)+8 \end{cases}$

$f_4(10) = \max \begin{cases} 0 \\ 1 \end{cases} \begin{cases} f_3(10) \\ f_3(10-4)+8 \end{cases}$

$f_3(12) = \max \begin{cases} 0 \\ 1 \end{cases} \begin{cases} f_2(12) \\ f_2(12-3)+6 \end{cases}$

$f_3(8) = \max \begin{cases} 0 \\ 1 \end{cases} \begin{cases} f_2(8) \\ f_2(8-3)+6 \end{cases}$

$f_3(10) = \begin{cases} 0 \\ 1 \end{cases} \begin{cases} f_2(10) \\ f_2(10-3)+6 \end{cases}$

$f_3(6) = \begin{cases} 0 \\ 1 \end{cases} \begin{cases} f_2(6) \\ f_2(6-3)+6 \end{cases}$

$f_2(12) = \begin{cases} 0 \\ 1 \end{cases} \begin{cases} f_1(12) \\ f_1(12-6)+15 \end{cases}$ 10 / 10 : 25

$f_2(9) = \begin{cases} 0 \\ 1 \end{cases} \begin{cases} f(9) \\ f_1(9-6)+15 \end{cases}$ 10 / 15

$f_2(8) = \begin{cases} 0 \\ 1 \end{cases} \begin{cases} f_1(8) = 10 \\ f_1(8-6)+15 = 15 \end{cases}$

$f_2(5) = \begin{matrix} 0 \\ 1 \end{matrix}\begin{cases} f_1(15) & -\infty \\ f_1(5-6)+15 & \end{cases}$

$f_2(10) = \begin{matrix} 0 \\ 1 \end{matrix}\begin{cases} f_1(10) & 10 \\ f_1(10-6)+15 & \end{cases}$

10

$f_2(7) = \begin{matrix} 0 \\ 1 \end{matrix}\begin{cases} f_1(7) & \\ f_1(7-6)+15 & \end{cases}$

10

$f_2(6) = \begin{matrix} 0 \\ 1 \end{matrix}\begin{cases} f_1(6) & \\ f_1(6-6)+15 & \end{cases}$

0+15

$f_2(3) = \begin{matrix} 0 \\ 1 \end{matrix}\begin{cases} f_1(3) & 0 \\ f_1(3-6)+15 & \end{cases}$

$\equiv 0$

-8

$\therefore f_1(m) = P_1 \text{ if } w_1 \leq m$

$= 0 \quad w_1 > m$

$f_1(12) = 10 = f_1(7)$

$f_1(6) = 10$

$f_1(9) = 10 = f_1(5)$

$f_1(8) = 10 = f_1(6)$

$f_1(1) = 0$

$f_1(3) = 0 = f_1(2)$

$f_1(4) = 10$

$f_1(-3) = -\infty$

$f_1(0) = 0$

$= (1, 1, 0, 0, 1) = 4+6+2$

* String editing :-

- given two strings $X = x_1, x_2 \ldots x_n$ $Y = y_1, y_2 \ldots y_m$, where x_i's & y_i's are elements of finite set of symbol called alphabet.

X = sort

y = sport

- we want to transform x into y using a sequence of edit operⁿ on x.
- the edit operⁿ are insert, delete & change
- there is cost associated with each operⁿ
- the cost of sequence of operⁿ is sum of cost of individual operⁿ in the file sequence.
- the prblm of string editing is to find a minimum cost sequence of edit operⁿ that will transform x into y.
- let $D(x_i)$ = the cost of deleting x_i frm X
  $I(y_i)$ - the cost of inserting y_j into x
  $C(x_i, y_i)$ - the cost of changing x_i of x into y_j
- define $cost(i, j)$ = minimum cost of edit sequence for transforming $x_1, x_2 \ldots x_i$ to $y_1, y_2 \ldots y_j$ where $0 \leq i \leq n$ & $0 \leq j \leq m$
- $cost(i, j) = 0$ if $i = j = 0$
- to find $cost(m, n)$: minimum cost of transforming X into y.

- If $j = 0$, $i > 0$ transform x to Y by seqⁿ of delete operⁿ

$$cost(i, 0) = cost(i-1, 0) + D(x_i)$$

If $i=0$ $j>0$ transform x to y by sequen. of insert operatn

$$cost(0,j) = cost(0,j-1) + 1(y_j)$$

- If $i \neq 0$ $j \neq 0$ one of the three ways can be used

(i)- Transform $x_1, x_2, \ldots x_{i-1}$ into $y_1, y_2 \ldots y_j$ using a minimum cost edit sequence & then delete x_i. $cost(i,j) = cost(i-1,j) + D(x_i)$ →

(ii) transform $x_1, x_2, \ldots x_{i-1}$ into $y_1, y_2 \ldots y_{j-1}$ using a minimum cost edit sequence & then change x_i to y_j $cost(i,j) = cost(i-1,j-1) + c(x_i, y_j)$

(iii) Transform $x_1, x_2 \ldots x_i$ into $y_1, y_2 \ldots y_{j-1}$ using a minimum cost edit sequence & then insert y_j $cost(i,j) = cost(i,j-1) + 1(y_j)$

$$cost(i,j) = \begin{cases} 0 & \text{if } i=j=0 \\ cost(i-1,0) + D(x_i) & \text{if } j=0,\ i>0 \\ cost(0,j-1) + 1(y_j) & \text{if } i=0,\ j>0 \end{cases}$$

$$\text{if } x_i = y_j,\ cost(x_{i-1}, y_{j-1}) \text{ else } = \min\{cost(i-1,j-1) + c(x_i, y_j)\} \quad \text{if } i>0,\ j>0$$

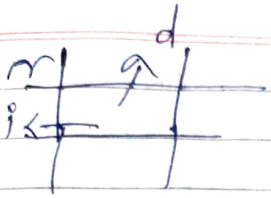
$$= cost(i-1,j-1) + c(x_i, y_j)$$

$$= cost(i,j-1) + 1(y_j)\ \}$$

method to find $cost(m,n)$

1. Compute 1st row of table with $i=0$
2. Compute 1st column of table with $j=0$
3. for ($i=1$ to n)

{ for ($j=1$ to m) compute $cost(i,j)$: }



$$\min \begin{cases} \text{diag} + 2 \\ \text{left} + 1 \\ \text{bp} + 1 \end{cases}$$

$r(x_3, y_4)$

- $x : a, a, b, a, b$ & $y : b, a, b, b$ x_4

		j		b	a	b	b
		i	y_0	y_1	y_2	y_3	y_4
	a	x_0	0	1+1	2	3	4
→	a	x_1	1	2 ij	1	2	3
	b	x_2	2	3+2	2+1	3	4
	a	x_3	3	2+2	3+2	2+2	3+1
	b	x_4	4	3+2	2+1	3+1	4
		x_5	5	4+1	3	2	(3)

$r(x_5, y_4) = 0$
$d(x_4) = 1 +$
$r(x_3, y_3) = 0 +$
$r(x_2, y_2) = 0 +$
$i(y_1) = 1 +$
$d(x_1) = 1 +$

} 3

$x : a\ a\ (b)\ a\ b$
$x_1\ x_2\ x_3\ x_4\ x_5$

$y : b\ a\ (b)\ b$
$y_1\ y_2\ y_3\ y_4$

(2) $x : a\ b\ c\ d\ (e)\ f$ Ans = 5
$y : a\ z\ c\ e\ (d)$

* Longest Common subsequence (LCS)

- given a sequence $X = \langle x_1, x_2 \dots x_m \rangle$, another sequence $Z = \langle z_1, z_2 \dots z_k \rangle$ is a subsequence of X if there exist a strictly increasing sequence $\langle i_1, i_2 \dots i_k \rangle$ of indices of X such that for all $j = 1, 2 \dots k$ we have $x_{i_j} = z_j$.

- Given two sequences X & Y we say that Z is common subsequence of X & Y if Z is a subsequence of both X & Y.
- If $X = \langle A, B, C, B, D, A, B \rangle$ AND $Y = \langle B, D, C, A, B, A \rangle$ then $Z = \langle B, C, A, B \rangle$ is common sequence of both X & Y.
- given two sequences X & Y, a maximum length common subsequence of X & Y is called longest common subsequence (LCS) of X & Y.

* Longest Common subsequence problm:

- given two sequences $X = \langle x_1, x_2 \dots x_m \rangle$ & $Y = \langle y_1, y_2 \dots y_n \rangle$ to find a maximum length common subsequence of X & Y
- let $c[i, j]$ = length of an LCS of x_i & y_j.
- let $c[0, 0] = 0$

Hence,

$$c[i, j] = \begin{cases} 0 & i = 0, j = 0 \\ c[i-1, j-1] + 1 & i, j > 0 \text{ & } x_i = y_j \\ \max\{c[i, j-1], c[i-1, j]\} & i, j > 0 \text{ & } x_i \neq y_j \end{cases}$$

i.e if $x_i = y_j$ then $c[i,j] = 1 +$ diagonal
else $c[i,j] = \max\{up, left\}$

$X = ABCBDAB \quad Y = BDCABA$

			B	D	C	A	B	A
	x_i	y_0	y_1	y_2	y_3	y_4	y_5	y_6
	x_0	0	0	0	0	0	0	0
A	x_1	0	0	0	0	1	1	1
B	x_2	0	1	1	1	1	2	2
C	x_3	0	1	1	2	2	2	2
B	x_4	0	1	1	2	2	3	3
D	x_5	0	1	2	2	2	3	3
A	x_6	0	1	2	2	3	3	4
B	x_7	0	1	2	2	3	4	4

* BFS

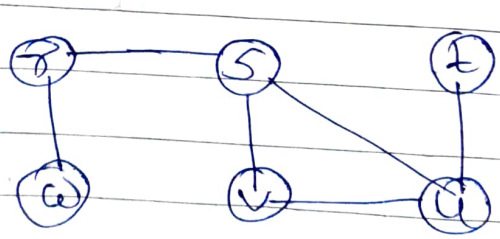
- The breadth-first-search procedure BFS shown on the next slide assume that the input graph $G = (V, E)$ is represented using adjacency lists.
- It maintains several additional data structures with each vertex in the graph
- The color of each vertex $u \in V$ is stored in the variable color[u] & predecessor of u is stored in the variable $\pi[u]$
- If u has no predecessor (e.g u is s or u or not been discovered) then $\pi[u] = NIL$
- The distance from the source s to vertex u computed by the algorithm is stored in $d[u]$.

The algorithm also uses a FIFO queue Q to manage the set of gray vertices.

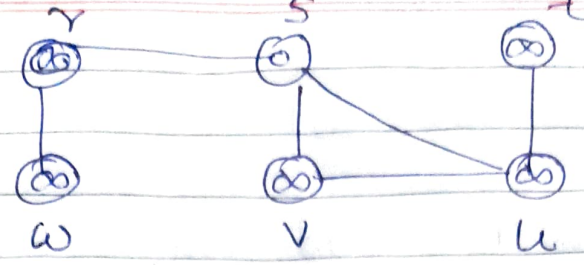
```
BFS (G, s)
for each vertex u ∈ V[G] - {s}
    do color[u] ← WHITE
       d[u] ← ∞
       π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← ϕ
ENQUEUE (Q, s)
while Q ≠ 0
    do u ← DQUEUE (Q)
       for each u ∈ Adj[u]
           do if color[v] = WHITE
              then color[v] ← GRAY
                   d[v] ← d[u] + 1
                   π[v] ← u
                   ENQUEUE (Q, v)
       color[u] ← BLACK
```

Ex: Draw BFS tree for G by starting at the vertex s.

→

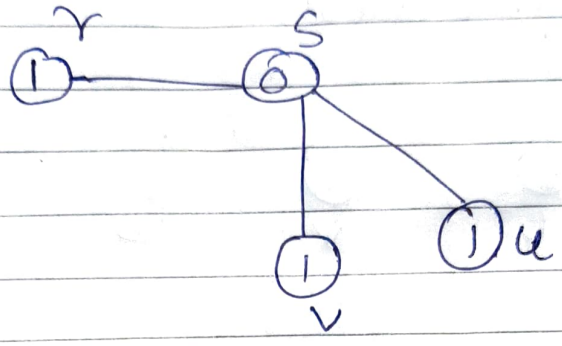


→ Let label with vertex v, x, denote $d[v]$. x^v



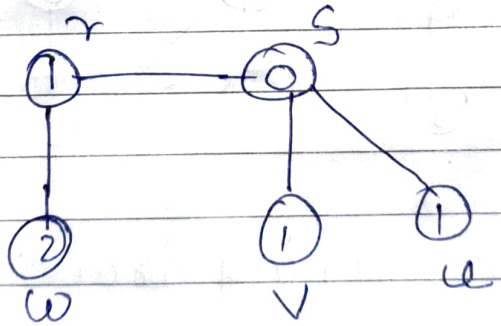
(I) visit vertices adjacent to s & add to Q.

(II)



color s black & delete it from Q.

(III) Explore r and add all the vertices adjacent to r to Q.

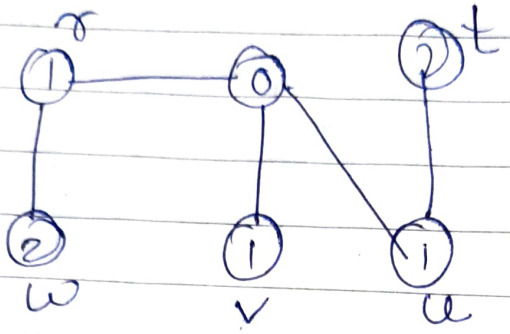


(IV) Color r black & remove it from Q.

(V) Explore v. Observe that all the vertices adjacent to v are already visited color v black delete from Q.

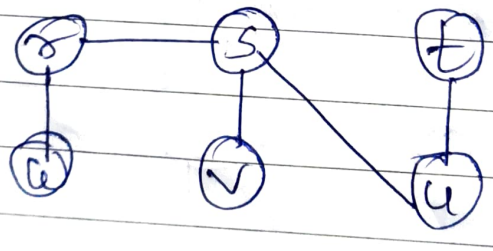
Q. [u]

⑥ explore u. add all the vertices adjacent to u to Q, color u black & delete u from Q.



Q [t]

⑦ Observe that t is explored. Color it black & remove from Q. Q is empty. Hence algorithm terminates.



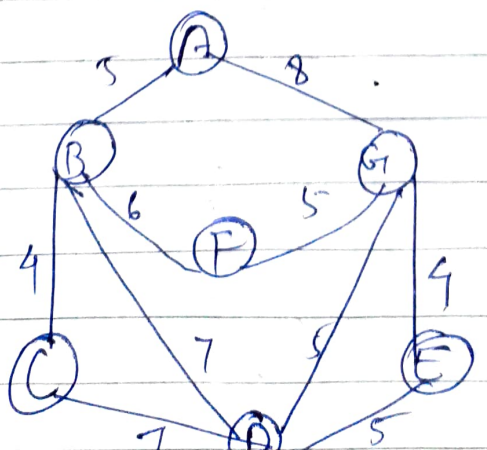
* Minimum spanning tree :-
  * spanning Tree :-
  - a tree (i.e, connected, acyclic graph) which contain all vertices of the graph.

* minimum spanning tree :-
  - the tree with the minimum sum of weights.

* spanning forest :-
If a graph is not connected, then there is a spanning tree for each connected component of the graph.

* kruskal's Algorithms :-
1) list all the edges of the graph G in the increasing order of weights.
2) select the smallest edge from the list & add it into the spanning tree T constructed so far, if the inclusion of this edge does not make a cycle.
3) if the selected edge with smallest weight forms a cycle, remove it from the list is empty.
4) Repeat steps 2 & 3 until tree contains n-1 edges or the list is empty.
5) if tree T contains less than n-1 edges & list is empty, no spanning tree is possible for graph, else return the MST T.

Q1) using kruskal algorithm find spanning tree of given graph G1.



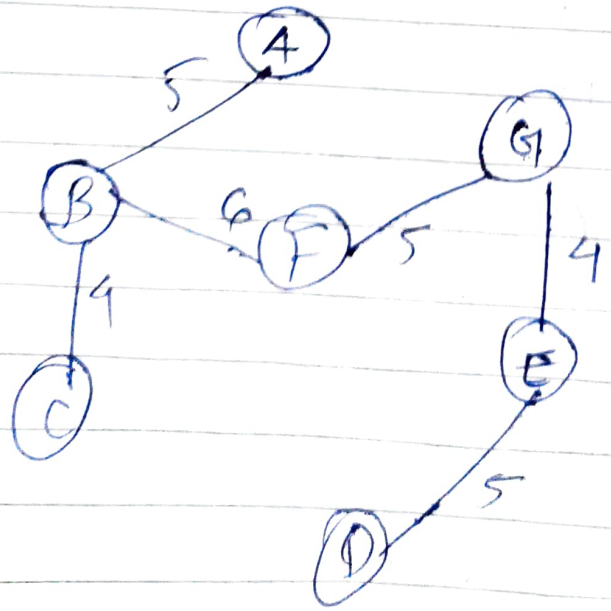
→ The edge in increasing order are

Edge	BC	EG	DE	DG	DF	AB	BF	CD	BD	AG
w(e)	4	4	5	5	5	5	6	7	7	8

Now we construct table which implemented kruskal's algorithm in which number of edges selected in spanning tree (u,v) is edge checked for inclusion in spanning tree & k are root of the trees to which u & v belong.

i	U	V	J	K	A	B	C	D	E	F	G	mincost
0					A	B	C	D	E	F	G	0
1	B	C	B	C	A	B	B	D	E	F	G	4
2	E	G	E	G	A	B	B	D	E	F	E	8
3	D	E	D	E	A	B	B	D	D	F	D	13
	D	G	D	D	—			Reject				
4	G	F	D	F		A	B	B	D	D	D	18
5	A	B	A	B		A	A	A	D	D	D	23
6	B	F	A	B		A	A	A	A	A	A	29

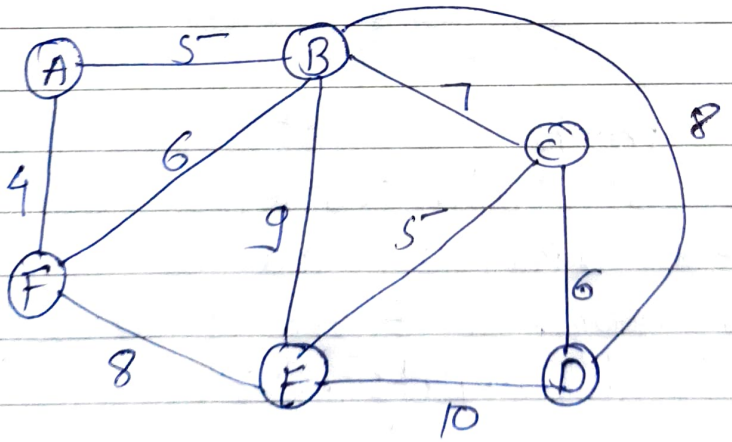
MST



* prim's Algorithm :-

- The idea as follows " at any stage in the algorithm, we can see that we have a set of vertices have already been included in the tree, the rest of vertices have not. The prim's algorithm then finds new vertex to add it to the tree by choosing the edge (v_i, v_j) the smallest among all edges, where v_i is in the tree v_j is yet to be included in the tree. The algorithm starts by selecting a vertex arbitrarily, & then in each stage, we add on edge to the tree.

1) use prim's algorithm, find MST of G



→ 1

in columns for vertices A to F give near of the respective vertex j & cost [j, near[j]]

i	K	l	j	A	B	C	D	E	F	mincost
1	A	F	—	0	A [5]	A [∞]	A [∞]	F [8]	0	4
2	—	—	B	0	0	B [7]	A [16]	B [8]	0	9
3	—	—	C	0	0	0	C [16]	C [5]	0	16
4	—	—	E	0	0	0	C [6]	0	0	21
5	6	—	D	0	0	0	0	0	0	27

* shortest path

- in a shortest path problems we are given a weighted, directed graph $G = (V, E)$ with weight function $w: E$ to R mapping edges to real valued weights. the weight of path $p = \langle v_0, v_1 \ldots v_k \rangle$ is the sum of the weight of its constituent edges.
- $w(p) = \sum w(v_{i-1}, v_i)$ i from 1 to k
- we define the shortest path weight from u to v by . . . .
- $\delta(u, v) = \min(w(p) : u \to v)$ if there is path from u to v
  ∞ otherwise

cost

The shortest path from vertex u to v is then defined as any path p with weight $w(p) = \delta(u, v)$

* Single source shortest paths
- Given a graph $G = (V, E)$, we want to find a shortest path from a given source vertex s from v to each vertex v in V
- initialize - single source (G, s)
  for each vertex $v \in V[G]$
    do $d[v] = \infty$
       $\Pi[v] = nil$
  $d[s] = 0$

* Algorithm relax
- Relax (u, v, w)
  if $d[v] > d[u] + w(u, v)$
  then $d[v] = d[u] + w(u, v)$
  $\Pi[v] = u$

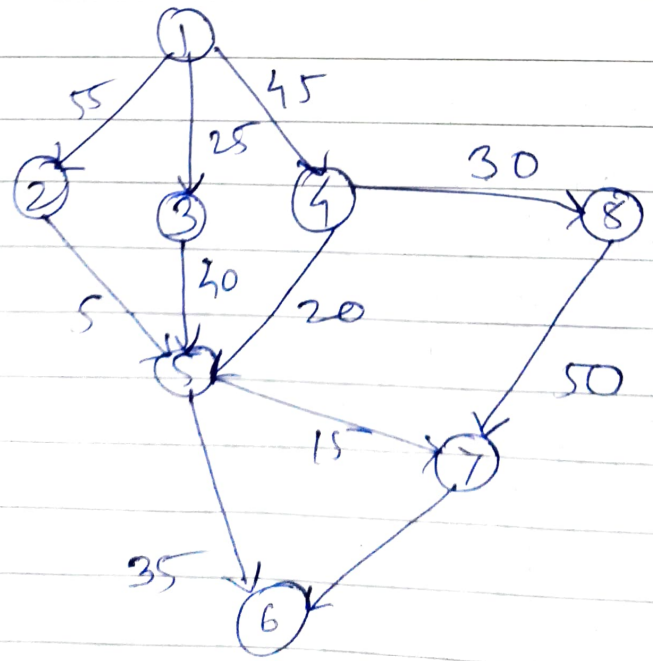
* Dijkstra's algorithm
- It solves the single source shortest path problem on weighted directed graph $G = (V, E)$ for which all edge weight are non negative
- Here we assume that $w(u, v) >= 0$ for each edge $(u, v) \in E$.
- It maintains set S of vertices whose final shortest path weight from source a have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with minimum shortest path.

estimate, add u to S & relaxes all edges leaving u.

- in following implementation, we use a min priority queue Q of vertices, keyed by their d values.

DIKSTRA (G, w, S)
INITIALIZE-SINGLE-SOURCE (G, S)
S ← ϕ
Q ← V(G)
while Q ≠ ϕ
    Do u ← Extract-Min (Q)
S ← SU{u}
for each vertex v ∈ Adj [u]
Do RELAX (u, v, w)

Apply Dijkstra's algorithm on following graph:

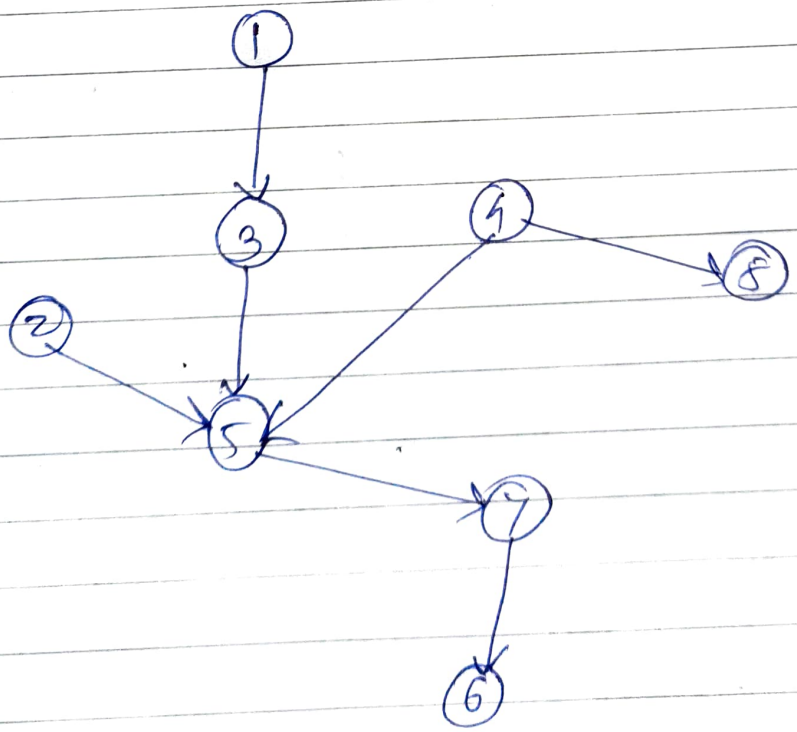


start from vertex 1.

initially.

vertex selected	1	2	3	4	5	6	7
–	0	55	25	45	∞	∞	∞
1	0	55	25	45	∞	∞	∞
3	0	55	25	45	65	∞	∞
4	0	55	25	45	65	∞	∞
2	0	55	25	45	60	∞	∞
5	0	55	25	45	60	95	75
7	0	55	25	45	60	85	75
6	0	55	25	45	60	85	75
8	0	55	25	45	60	85	125

get shortest path:



* Bellman Ford Algorithm:

```
BELLMAN-FORD (G, w, s)
  Initialize-Single-Source (G, s)
  For i = 1 to |V[G]| - 1
  do for each edge (u, v) ∈ E[G]
       do relax (u, v, w)
  For each edge (u, v) ∈ E[G]
  do if d[v] > d[u] + w(u, v)
       then return false
  return false
```

→ ex: Bellman ford algorithm find length of shortest paths from source z to all other vertices.