# UNIT - III GREEDY METHOD & DYNAMIC PROGRAMMING

SYLLABUS :-

Greedy method: General method, applications- Job sequencing with dead-lines, Knapsack problem, Minimum-cost spanning trees, Single source shortest path.

Dynamic Programming:- General method, applications - optimal binary search trees, 0/1 Knapsack, All pairs shortest path, The Travelling sales person problem.

## GREEDY METHOD

### GENERAL METHOD:-

→ The greedy method is the most possible straight forward design technique, used to determine a feasible solution that may or may not be optimal.

Feasible solution:-

→ Most problems have n inputs and its solution contains a subset of inputs that satisfies a given constraint (condition).

→ Any subset that satisfies the constraint is called feasible solution.

optimal solution:-

→ TO find a feasible solution that either maximizes or minimizes a given objective function. A feasible solution that does this

is called optimal solution.

→ The greedy method suggests that an algorithm works in stages, considering one input at a time.

→ At each stage, a decision is made regarding whether a particular input is in an optimal solution.

→ Greedy algorithms neither postpone nor revise the decisions (i.e, no backtracking).

Example:- Kruskal's algorithm. Minimal spanning tree that select an edge from a sorted list, check, decide and never visit it again.

Algorithm for Greedy Method:-

```
Algorithm Greedy(a,n)
  // a[1:n] contains the n inputs.
{
  solution:=0;
  for i=1 to n do
  {
   x:=select(a);
  if feasible(solution,x) then
  solution:=Union(solution,x);
  }
   Return solution;
  }
```

Selection:- Function that selects an input from a[] and removes it. The selected input's value is assigned to x.

Feasible:- Boolean-valued function that determines whether x can be included into the solution vector.

Union:- Function that combines x with solution and updates the objective function.

## APPLICATIONS:-

1. Job sequencing with deadline.
2. 0/1 Knapsack problem.
3. Minimum cost spanning trees.
4. Single source shortest path problem.

## JOB SEQUENCING WITH DEAD-LINES:-

→ There is set of n-jobs. For any job i, is a integer deadline $d_i \geq 0$ and profit $P_i > 0$, the profit $P_i$ is earned iff the job completed by its deadline.

→ To complete a job one had to process the job on a machine for one unit of time. only one machine is available for processing jobs.

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

→ The value of a feasible solution J is the sum of the profits of the jobs in J,

i.e, $\sum_{i \in j} P_i$

→ An optimal solution is a feasible solution with maximum value.

→ The problem involves identification of a subset of jobs which can be completed by its deadline.

→ Therefore the problem suites the subset methodology and can be solved by the greedy method.

Example:- obtain the optimal sequence for the following jobs.

$(P_1, P_2, P_3, P_4)$ = $j_1\ j_2\ j_3\ j_4$ $(100, 10, 15, 27)$

$(d_1, d_2, d_3, d_4)$ = $(2, 1, 2, 1)$

$n = 4$

| Feasible Solution | Processing Sequence | Value |
|---|---|---|
| $j_1 j_2$ $(1, 2)$ | $(2, 1)$ | $100+10=110$ |
| $(1,3)$ | $(1, 3)$ (or) $(3, 1)$ | $100+15=115$ |
| $(1, 4)$ | $(4, 1)$ | $27+100=127$ |
| $(2, 3)$ | $(2, 3)$ | $10+15=25$ |
| $(3, 4)$ | $(4, 3)$ | $15+27=42$ |

| Feasible solution | Processing Sequence | Value |
|---|---|---|
| (1) | (1) | 100 |
| (2) | (2) | 10 |
| (3) | (3) | 15 |
| (4) | (4) | 27 |

→ In the example, solution '3' is the optimal.
→ In this solution only jobs 1 and 4 are processed and the value is 127. These jobs must be processed in the order j4 followed by j1. The process of job 4 bagins at time 0 and ends at time 1.
→ And the processing of job 1 begins at time 1 and ends at time 2. Therefore both the jobs are completed within their deadlines.
→ The optimization measure for determining the next job to be selected in to the solution is according to the profit.
→ The next job to include is that which increases $\Sigma p_i$ the most, subject to the constraint that the resulting "j" is the feasible solution.
→ Therefore the greedy strategy is to consider the jobs in decreasing order of profits.
→ We must formulate an optimization measure to determine how the next job is chosen.

Algorithm:-

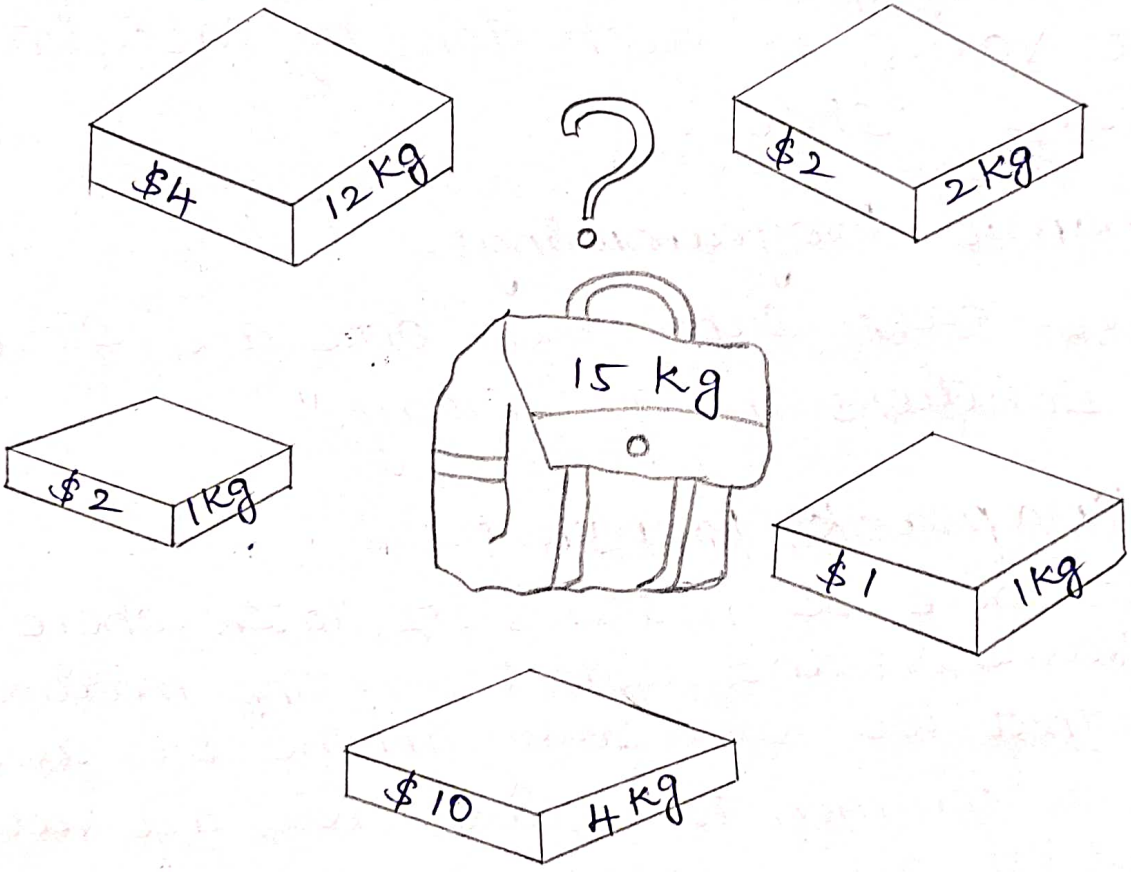
```
algorithm js(d, j, n)
// d → dead line, j → subset of jobs,
   n → total number of jobs.
// d[i] ≥ 1, 1 ≤ i ≤ n are the dead lines,
// the jobs are ordered such that p[1] ≥ p[2] ≥
                              ---- ≥ p[n].
// j[i] is the ith job in the optimal solution
   1 ≤ i ≤ k, k → subset range.
{
 d[0] = j[0] = 0;
  j[1] = 1;
  k = 1;
  for i = 2 to n do {
  r = k;
  while ((d[j[r]] > d[i]) and [d[j[r]] ≠ r))
                                    do
  r = r - 1;
  if((d[j[r]] ≤ d[i]) and (d[i] > r)) then
  {
  for q := k to (r+1) step -1 do j[q+1] = j[q];
   j[r+1] = i;
    k = k+1;
    }
   }
   return k;
  }
```

∴ Note:- The size of sub set j must be less than equal to maximum deadline in given list.

## KNAPSACK PROBLEM:-

→ The Knapsack problem or rucksack (bag) problem is a problem in combinatorial optimization.

→ Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to the given limit and the total value is as large as possible.



There are two versions of the problems.

1. 0/1 Knapsack problem.
2. Fractional Knapsack problem.
    a. Bounded Knapsack problem.
    b. Unbounded Knapsack problem.

Solutions to Knapsack problems:-

1. Brute-force approach:-
   → solve the problem with a straight forward algorithm.

2. Greedy Algorithm:-
   → Keep taking most valuable items until maximum weight is reached or taking the largest value of each item by calculating $V_i$ = value$_i$ / size$_i$.

3. Dynamic Programming:-
   → solve each subproblem once and store their solutions in an array.

0/1 Knapsack problem:-

→ Let there be n items, $Z_1$ to $Z_n$ where $Z_i$ has a value $V_i$ and weight $W_i$. The maximum weight that we can carry in the bag is W.

→ It is common to assume that all values and weights are nonnegative.

→ TO simplify the representation, we also assume that the items are listed in increasing

Order of weight.

Maximize $\sum_{i=1}^{n} v_i x_i$ subject to $\sum_{i=1}^{n} w_i x_i \leq W$,

$x_i \in \{0, 1\}$

→ Maximize the sum of the values of the items in the knapsack so that the sum of the weights must be less than the knapsack's capacity.

Greedy algorithm for knapsack:-

```
Algorithm GreedyKnapsack(m, n)
// P[i:n] and [1:n] contain the profits and
   weights respectively.
// if the n-objects ordered such that
   P[i]/w[i] >= P[i+1]/w[i+1], m → size
   of knapsack and x[1:n] → the solution
   vector.
{
 for i:=1 to n do x[i]:=0.0
  U:=m;
 for i:=1 to n do
 {
  if(w[i]>U) then break;
   x[i]:=1.0;
   U:= U-w[i];
  }
  if(i<=n) then x[i]:=U/w[i];
}
```

Example:-

consider 3 objects whose profits and weights are defined as $(P_1, P_2, P_3) = (25, 24, 15)$ $(W_1, W_2, W_3) = (18, 15, 10)$ $n=3 \rightarrow$ number of objects, $m=20 \rightarrow$ Bag capacity.

consider a Knapsack of capacity 20. Determine the optimum strategy for placing the objects in to the Knapsack. The problem can be solved by the greedy approach where in the inputs are arranged according to selection process (greedy ~~dev~~ strategy) and solve the problem in stages. The various greedy strategies for the problem could be as follows.

| $(X_1, X_2, X_3)$ | $\sum X_i W_i$ | $\sum X_i P_i$ |
|---|---|---|
| $(1, 2/15, 0)$ | $18 \times 1 + \frac{2}{15} \times 15 = 20$ | $25 \times 1 + \frac{2}{15} \times 24 = 28.2$ |
| $(0, 2/3, 1)$ | $\frac{2}{3} \times 15 + 10 \times 1 = 20$ | $\frac{2}{3} \times 24 + 15 \times 1 = 31$ |
| $(0, 1, 1/2)$ | $1 \times 15 + \frac{1}{2} \times 10 = 20$ | $1 \times 24 + \frac{1}{2} \times 15 = 31.5$ |
| $(1/2, 1/3, 1/4)$ | $1/2 \times 18 \times 1/3 \times 15 + 1/4 \times 10 = 16.5$ | $1/2 \times 25 + 1/3 \times 24 + 1/4 \times 15 = 12.5 + 8 + 3.75 = 24.25$ |

Analysis:- If we do not consider the time considered for sorting the inputs then all of the three greedy strategies complexity will be O(n).

## MINIMUM COST SPANNING TREE:-

SpanningTree:-

→ A sub graph 'n' of a graph 'G' is called as a spanning tree if

(i) It includes all the vertices of 'G'
(ii) It is a tree.

Minimum cost spanning tree:-

→ For a given graph 'G' there can be more than one spanning tree. If weights are assigned to the edges of 'G' then the spanning tree which has the minimum cost of edges is called as minimal spanning tree.

→ The greedy method suggests that a minimum cost spanning tree can be obtained by contacting the tree edge by edge.

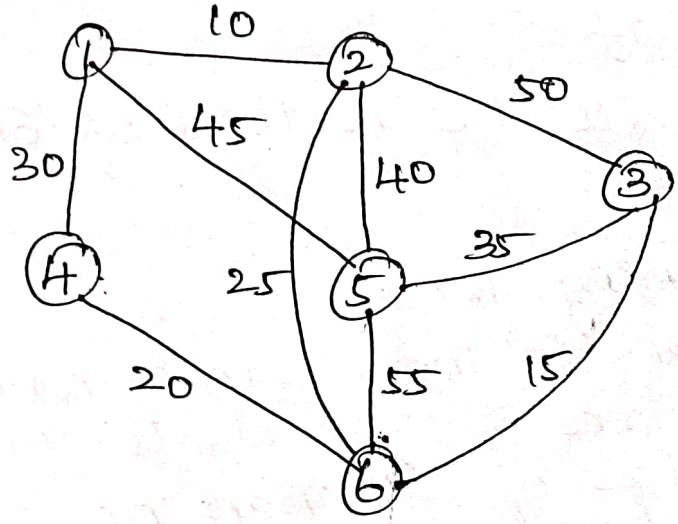
→ The next edge to be included in the tree is the edge that results in a minimum increase in the some of the costs of the edges included so far.

→ There are two basic algorithms for finding minimum cost spanning trees, and both are greedy algorithms.

→ Prim's algorithm
→ Kruskal's algorithm

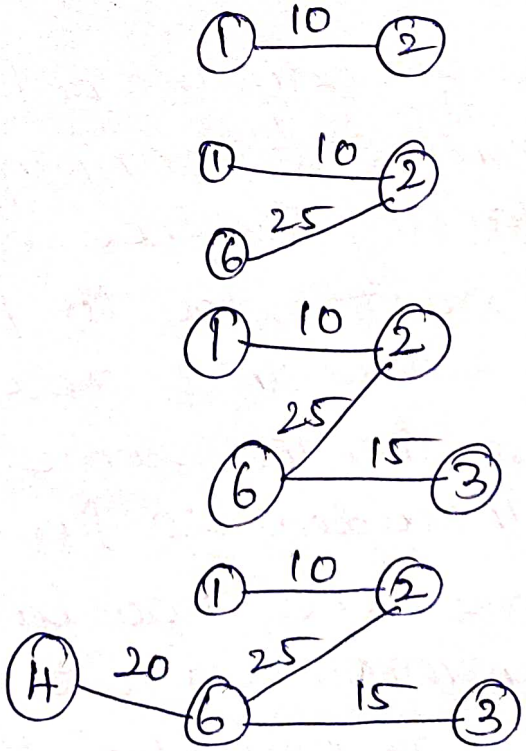
Prim's Algorithm :-

→ start with any one node in the spanning tree, and repeatedly add the cheapest edge, and the node it leads to, for which the node is not already in the spanning tree.

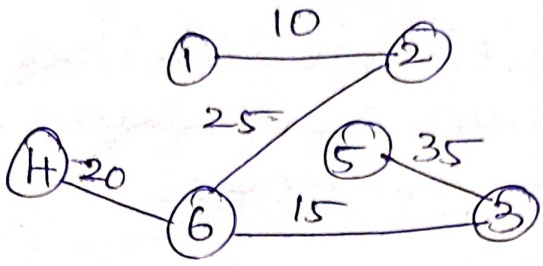



| Edge | cost |
|---|---|
| (1,2) | 10 |
| (2,6) | 25 |
| (3,6) | 15 |
| (6,4) | 20 |

spanning tree



| Edge | cost | spanning tree |
|---|---|---|
| | cycle forms | |
| (1,4) | reject | |
| (3,5) | 35 | |

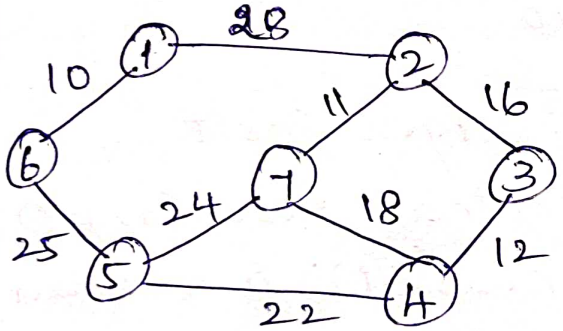


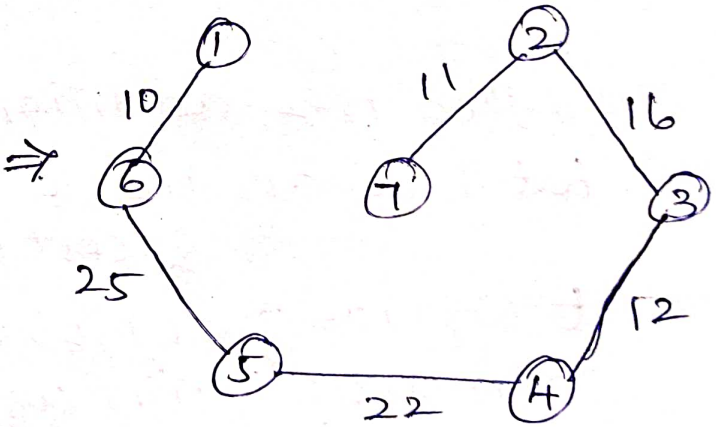

Algorithm:-

i) select an edge with minimum cost and include in to the spanning tree.

ii) Among all the edges which are adjacent with the selected edge, select the one with minimum cost.

iii) Repeat step 2 until 'n' vertices and (n-1) edges are been included. And the subgraph obtained does not contain any cycles.




Prim's minimum spanning tree algorithm:-

Algorithm Prim (E, cost, n, t)

// E is the set of edges in G. cost (1:n, 1:n) is the cost adjacency matrix of an n vertex graph such that cost (i, j) is either a positive

```
real number or ∞ if no edge (i,j) exists.
//A minimum spanning tree is computed and
  stored in the array T[1:n-1, 2).
//(t(i, 1), + t(i, 2)) is an edge in the
  minimum cost spanning tree. The final
  cost is returned
  {
    Let (K,1) be an edge with min cost in E
    Min cost := cost (x, 1);
    T(1, 1) := K; + (1,2) := 1;
    for i := 1 to n do //initialize near.
        if (cost (i,1) < cost (i, K) then n
                                east(i) : 1;
        else near (i) := K;
        near (K) : = near(1) : =0;
        for i := 2 to n-1 do
    {
    //find n-2 additional edges for t
    let j be an index such that near(i) ≠ 0
                & cost(j, near(i)) is minimum;
    t (i,1) := j + (i,2) := near(j);
    min cost := Min cost + cost(j, near(j));
    near (j) := 0;
    for k := 1 to n do //update near()
    if ((near (K) ≠ 0) and (cost {K, near(K))
                                > cost(K, j)))
    then near Z(K) := j; }
    return mincost; }
```

→ The algorithm takes four arguments

'E': set of edges.

'cost' is nxn adjacency matrix cost of (i,j) = +ve integer if an edge exists between i & j, otherwise infinity.

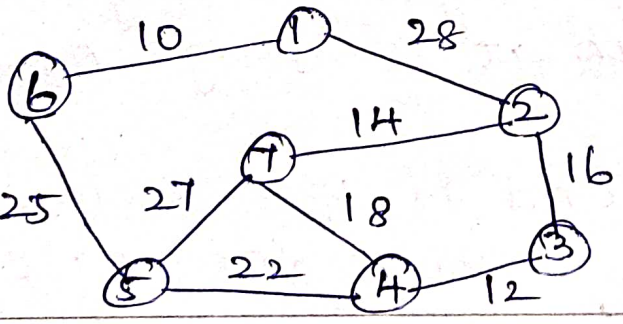
'n' is number of vertices.

't' is a (n-1) : 2 matrix which consists of the edges of spanning tree.

$E = \{(1,2),(1,6),(2,3),(3,4),(4,5),(4,7),(5,6),(5,7),(2,7)\}$

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

| Cost | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | ∞ | 28 | ∞ | ∞ | ∞ | 10 | ∞ |
| 2 | 28 | ∞ | 16 | ∞ | ∞ | ∞ | 14 |
| 3 | ∞ | 10 | ∞ | 12 | ∞ | ∞ | ∞ |
| 4 | ∞ | ∞ | 12 | ∞ | 22 | ∞ | 18 |
| 5 | ∞ | ∞ | ∞ | 22 | ∞ | 25 | 24 |
| 6 | 10 | ∞ | ∞ | ∞ | 25 | ∞ | ∞ |
| 7 | ∞ | 14 | ∞ | 18 | 24 | ∞ | ∞ |

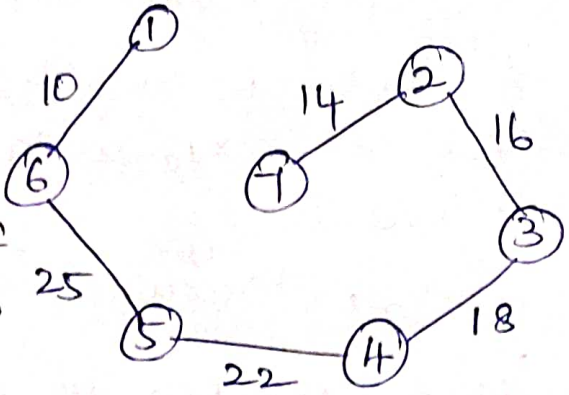


| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 1 | 6 | 5 | 4 | 3 | 2 |
| 2 | 6 | 5 | 4 | 3 | 2 | 7 |

→ Ending vertex

Starting Vertex

Edges of Spanning tree




Analysis:-

→ The time required by the prim's algorithm is directly proportional to the number of vertices. If a graph 'G' has 'n' vertices then the time required by prim's algorithm is $O(n^2)$.

2. Kruskal's Algorithm:-

→ In Kruskal's algorithm for determining the spanning tree we arrange the edges in the increasing order of cost.

i) All the edges are considered one by one in that order and deleted from the graph and are included in to the spanning tree.

ii) At every stage an edge is included, the sub-graph at a stage need not be a tree.

iii) At the end if we include 'n' vertices and n-1 edges without forming cycles then we

get a single connected component without any cycles. i.e a tree with minimum cost.

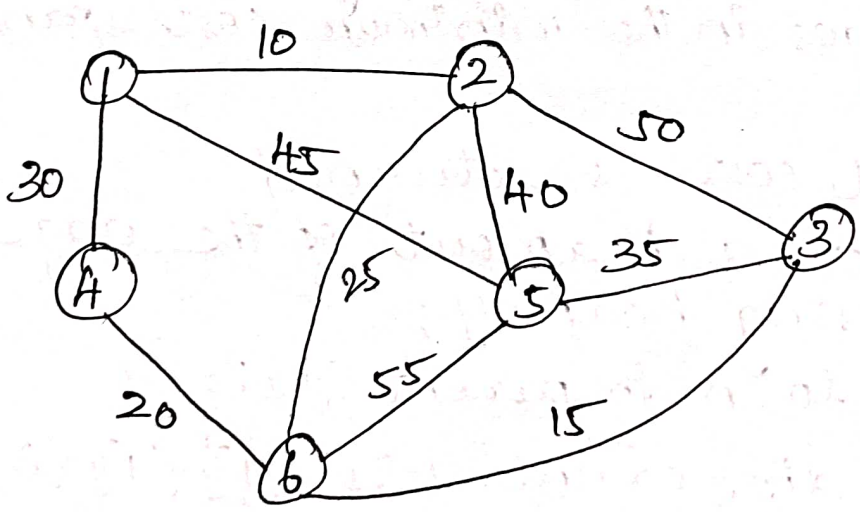
→ At every stage, as we include an edge in to the spanning tree, we get disconne-cted trees represented by various sets.
→ While including an edge in to the spanning tree we need to check it does not form cycle.
→ Inclusion of an edge (i, j) will form a cycle if i, j both are in same set.
→ otherwise, the edge can be included into the spanning tree.

Kruskal's minimum spanning tree algorithm:

```
Algorithm Kruskal(E, cost, n, t)
// E is the set of edges in G. 'G' has 'n'
   vertices.
// cost{u,v} is the cost of edge (u,v), t is the
   set of edges in the minimum cost spanning
   tree.
// The final cost is returned
   { construct a heap out of the edge
        costs using heapify;
     for i:=1 to n do parent(i):=-1
   //place in different sets {1}{1}{3}
   // each vertex is in different set
     i:=0; min cost:=0.0;
```

```
while (i<n-1) and (heap not empty) do
{
Delete a minimum cost edge (u,v) from
 the heaps; and reheapify using adjust;
 j:= find(u); k:= find(v);
  if(j≠k) then
  {
    i:=i+1;
    t(i,1)=u; t(i,2)=v;
    mincost:= mincost + cost(u,v);
    Union(j,k);
  }
}
if(i≠n-1) then write("No spanning tree");
   else return mincost;
}
```

Example:-

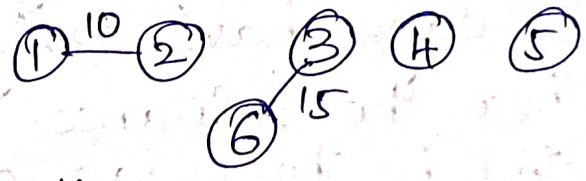
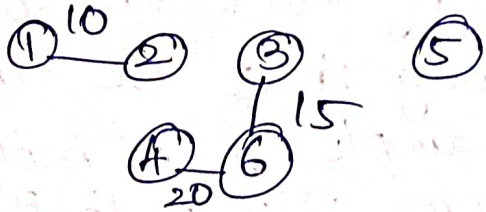
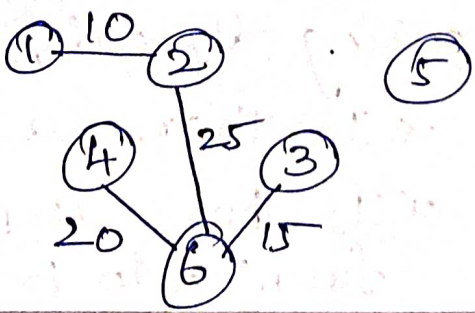


→ consider the above graph, using Kruskal's method the edges of this graph are considered for inclusion in the minimum cost spanning tree in the order (1,2), (3,6), (4,6), (2,6), (1,4), (3,5), (2,5), (1,5), (2,3) and (5,6).

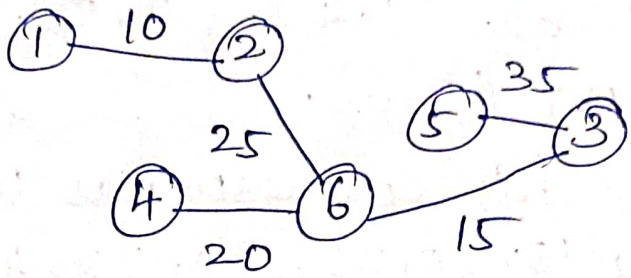
→ This corresponds to the cost sequence 10, 15, 20, 25, 30, 35, 40, 45, 50, 55.

→ The first four edges are included in T. The next edge to be considered is (1,4). This edge connects two vertices already connected in T and so it is rejected.

→ Next the edge (3,5) is selected and that completes the spanning tree.

| Edge | Cost | Spanning Forest |
|---|---|---|
| (1,2) | 10 |  |
| (3,6) | 15 |  |
| (4,6) | 20 |  |
| (2,6) | 25 |  |

| Edge | cost | spanning Forest |
|---|---|---|
| (1,4) | 30 | reject (it forms cycle). |
| (3,5) | 35 | |




Analysis:-

→ If the number of edges in the graph is given by |E| then the time taken for Kruskals algorithm is given by O(|E| log |E|).

## SINGLE SOURCE SHORTEST PATH :-

→ Graphs can be used to represent the high way structure of a state or country with vertices representing cities and edges representing sections of highway.

→ The edges have assigned weights which may be either the distance between the 2 cities connected by the edge or the average time to drive along that section of highway.

→ For example if A motorist wishing to drive from city A to B then we must answer the following questions

* Is there a path from A to B
* If there is more than one path from A to B which is the shortest path.

→ The length of a path is defined to be the sum of the weights of the edges on that path.

→ Given a directed graph G(V, E) with weight edge w(u, v), We have to find a shortest path from source vertex $S \in V$ to every other vertex $V \in V-S$.

→ To find single source shortest path for directed graphs G(V, E) there are two different algorithms.

1. Bellman - Ford algorithm.
2. Dijkstra's algorithm.

1. Bellman - Ford algorithm:-

→ Allow negative weight edges in input graph. This algorithm either finds a shortest path from source vertex $S \in V$ to other vertex $V \in V$ or detect a negative weight cycles in G, hence no solution.

→ If there is no negative weight cycles are reachable from source vertex $S \in V$ to every other vertex $V \in V$.

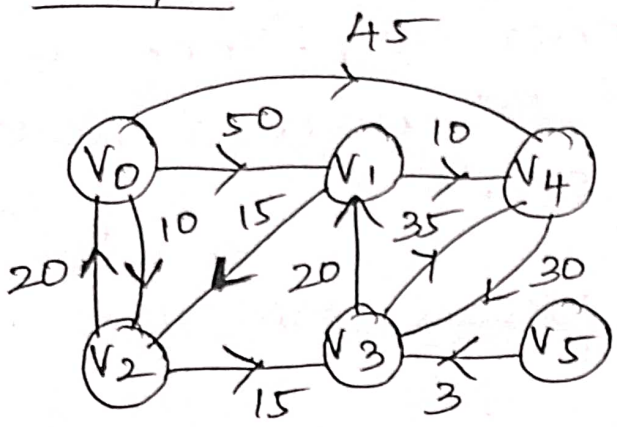
2. Dijkstra's algorithm:-

→ Allows only positive weight edges in the input graph and finds a shortest path from source vertex $S \in V$ to every other vertex $V \in V$.

Algorithm for finding shortest path :-

```
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
  graph g with n-vertices.
//dist[v] is zero
 {
 for i=1 to n do {
 s[i]= false;
 dist[i]=cost[v,i];
 }
 s[v]=true;
 dist[v]:=0.0; //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
  if (dist[w] > (dist[u] + cost[u,w])) then
   dist[w] = dist[u] + cost[u, w];
   }
}
```

Example:-




| Path | Length |
|---|---|
| 1) V0 V2 | 10 |
| 2) V0 V2 V3 | 25 |
| 3) V0 V2 V3 V1 | 45 |
| 4) V0 V4 | 45 |

Graph and shortest paths from V0 to all destinations

→ consider the above directed graph, if node 1 is the source vertex, then shortest path from 1 to 2 is 1, 4, 5, 2. The length is 10+15+20 =45.

→ To formulate a greedy based algorithm to generate the shortest paths, we must conceive of a multistage solution to the problem and also of an optimization measure.

→ This is possible by building the shortest paths one by one.

→ As an optimization measure we can use the sum of the lengths of all paths so far generated.

→ If we have already constructed 'i' shortest paths, then using this optimization measure, the next path to be constructed should be the next shortest minimum length path.

→ The greedy way to generate the shortest paths from V0 to the remaining vertices is to generate these paths in non-decreasing order of path length.

→ For this 1st shortest path of the nearest vertex is generated. Then a shortest path to the 2nd nearest vertex is generated and so on.

# DYNAMIC PROGRAMMING

## GENERAL METHOD:-

→ It is a strategy for designing algorithm.

→ Dynamic programming is also used in optimization problems.

→ Like divide and conquer method, Dynamic programming solves problems by combining the solutions of subproblems.

→ Moreover, Dynamic programming algorithm solves each sub problem just once & then saves its answer in a table, there by avoiding the work of recomputing the answer.

Two main properties:-

→ These two main properties suggest that the given problem can be solved using dynamic programming.

1. Overlapping subproblems.
2. Optimal substructure.

1. Overlapping subproblems:-

→ Similar to divide and conquer approach, Dynamic programming also combines solutions to sub problems.

→ It is mainly used where the solution of one subproblem is needed repeatedly.
→ The computed solutions are stored in a table, so that these don't have to be re-computed.
→ Hence, this technique is needed where overlapping subproblems exists.

For example:-
→ Binary search does not have overlapping subproblems, whereas recursive program of fibonacci numbers have many overlapping sub problems.

2. optimal sub structure (principle of optimality)
→ A given problem has optimal substructure property, if the optimal solution of the given problem can be obtained using optimal solutions of its sub problems.

For example:-
→ The shortest path problem has the following optimal substructure property.

Steps for Dynamic programming:-
→ Dynamic programming design involves four major steps.

1. Characterize the structure of an optimal solution.

2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution typically in a bottom-up fashion.
4. Construct an optimal solution from the computed information.

APPLICATIONS:-

→ Various problems that can be solved using dynamic programming are:-

1. All pairs shortest path
2. optimal binary search trees.
3. 0/1 knapsack problem.
4. Travelling salesperson problem.
5. Multi stage Graphs.

Example for dynamic programming - how it will applied.

1) Greedy method → It will always take only one decision.
2) Dynamic programming → It will took all possible sequences and picks best optimal solution.
   → It follows principle of optimality (sequence of solutions).

→ Every stage we take decisions.

→ Dynamic programming adopts tabulization method (or) memorization method.

Recursion definition of fibonacci term:-

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

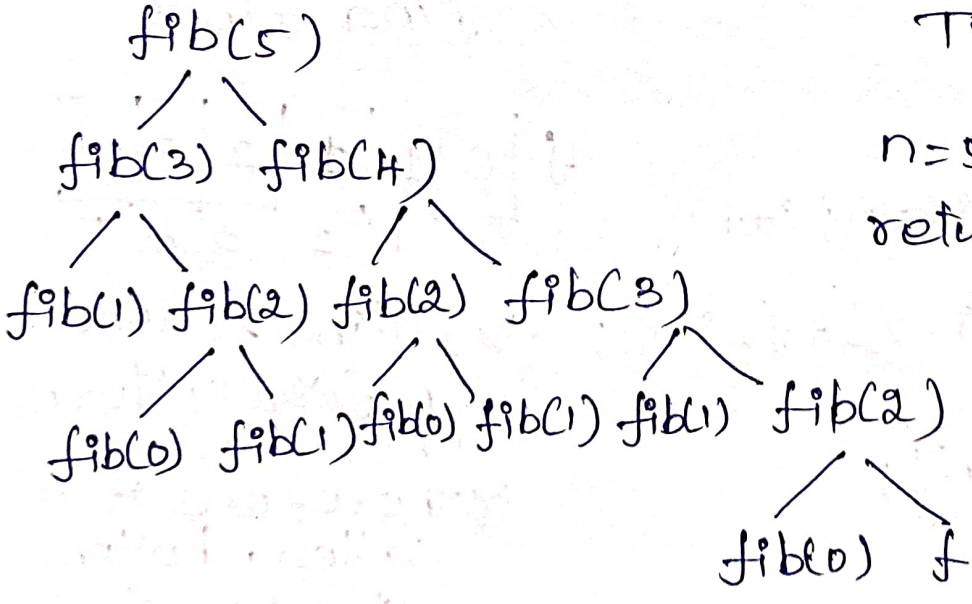
```
int fib(int n)
{
  if (n<=1)
     return n;
     return fib(n-2)+ fib(n-1);
}
```

* fibonacci series is

  0, 1, 1, 2, 3, 5, 8, 13....

  0th term, 1st term, 2, 3, 4, 5, 6, 7.....

* If I want to find 5th term in fibonacci (how it call recursively).




Trace the program

n=5, 5<=1 (false)

return fib(5-2)+ fib(5-1).

fib(3)+fib(4)

Recurrence relation:-

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

Time complexity is $O(2^n)$.

Disadvantage:-

→ If I call this fibonacci function, it will take too much time to compute.

Is there any way to reduce the function time.

→ When you observe the tracing tree, fib(1) is calling so many times.

→ why we are calling these function so many times & why can't we reduce this one.

To reduce the repetition (overlapping), we are moving to tabular method:-

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

Dynamic programming in Tabular method (It uses iterative approach)

find fib(5)

| f | 0 | 1 | 1 | 2 | 3 | 5 |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |

→ filling is done on from '0' term onwards.

→ Therefore it is called, bottom-up approach.

[Note:-
→ Memorization is done in the form of top down approach.
→ Time complexity for memorization is O(n)]

## OPTIMAL BINARY SEARCH TREE:-

Problem description:-
→ Let $\{a_1, a_2, \ldots a_n\}$ a set of identifiers such that $a_1 < a_2 < a_3$.
→ Let P(i) be the probability with which we can search for $a_i$ & q(i) be the probability of successful searching element x such that $a_i < x < a_{i+1}$ & $0 \le i \le n$.
→ In other words.
P(i) → probability of successful search.
q(i) → probability of unsuccessful search.
→ Also $\sum_{1 \le i \le n} P(i) + \sum_{1 \le i \le n} q(i)$ Then obtain a tree with minimum cost.
→ Such a tree with optimum cost is called optimal binary search tree.
→ TO solve this problem using dynamic programming method we will perform following steps.

Steps for optimal Binary Search Tree:-
Step 1:- Notations used Let,

$$T_{ij} = OBST(a_{i+1}, \ldots, a_j)$$

$C_{ij}$ denotes the cost $C(T_{ij})$

$W_{ij}$ is the weight of each $T_{ij}$.

$T_{on}$ is the final tree obtained.

$T_{oo}$ is empty.

$T_{i,i+1}$ is a single-node tree that has element $a_{i+1}$.

During the computations the root values are computed & $r_{ij}$ stores the root value of $T_{ij}$.

Step 2:-

→ The OBST can be build using the principle of optimality. consider the process of creating OBST.

→ Let $T_{on}$ be an OBST for the elements $a_1 < a_2 < \dots < a_n$ & Let L & R be its left & right subtree. Suppose that the root of $T_{on}$ is $a_k$, for some k.

→ Then the elements in the left subtree 'L' are $a_1, a_2, \dots, a_{k-1}$ & the elements in the right subtree 'R' are $a_{k+1}, a_{k+2}, \dots, a_n$.

→ The cost of computing the $T_{on}$ can be given as

$$C(T_{on}) = C(L) + C(R) + P_1 + P_2 + P_3 + \dots + P_n + q_0 + q_1 + q_2 + \dots + q_n$$

i.e

$$C(T_{on}) = C(L) + C(R) + W$$

where

$$W = P_1 + P_2 + \cdots + P_n + q_0 + q_1 + q_2 + \cdots + q_n.$$

→ If L is not an optimal BST for its elements, then we can find another tree 'L' for the same elements, with the property $C(L') < C(L)$ (i.e. optimal cost).

∴ Let T' be the tree with root $a_k$, left subtree L' & right subtree R.

Then,

$$C(T') = C(L') + C(R) + W$$

i.e $C(T') < C(L) + C(R) + W$

i.e $< C(T_{on})$

→ That means, T' is optimal than $T_{on}$. This contradicts the fact that $T_{on}$ is an optimal BST. Therefore L must be an optimal for its elements.

→ In the same manner we can obtain optimal binary search tree by building the optimal subtrees. This ultimately shows that optimal binary search tree follows the principle of optimality.

Step 3:-

→ We will apply following formula for computing each sequence.

$C(i,j) = \min_{i<k\le j} \{C(i,k-1) + C(k,j)\} + W(i,j)\}$

$W(i,j) = W[i,j-1] + P[j] + q[j];$

$r[i,j] = k$

Analysis:-

→ The computation of each c and r can be done using three nested for loops. Here the time complexity turns out to be $O(n^3)$.

Example:-

consider four elements $a_1, a_2, a_3$ and $a_4$ with $Q0 = 1/8$, $Q1 = 3/16$, $Q2 = Q3 = Q4 = 1/16$ and $P1 = 1/4$, $P2 = 1/8$, $P3 = 1/16$. construct an optimal binary search tree. Solving for $C(0,n)$:

step 1:-

computing all $C(i,j)$ such that $j-i=1$; $j=i+1$ and as $0 \le i < 4$; $i=0,1,2$ and 3; $i<k\le j$ start with $i=0$; so $j=1$; as $i<k\le j$, so the possible value for $k=1$

$W(0,1) = P(1) + Q(1) + W(0,0) = 4+3+2 = 9$

$C(0,1) = W(0,1) + \min\{C(0,0) + C(1,1)\}$

$= 9 + [0+0] = 9$ ft $(0,1) = 1$.

Next with $i=1$; so $j=2$; as $i<k\le j$; so the possible value for $k=2$

$W(1,2) = P(2) + Q(2) + W(1,1) = 2+1+3 = 6$

$C(1,2) = W(1,2) + \min\{C(1,1) + C(2,2)\}$

$= 6 + [(0+0)] = 6$ ft $(1,2) = 2$

Next with $i=2$; so $j=3$; as $i<k\leq j$; so the possible value for $k=3$

$W(2,3) = P(3) + Q(3) + W(2,2) = 1+1+1 = 3$.

$C(2,3) = W(2,3) + \min\{C(2,2) + C(3,3)\}$.

$= 3 + [(0+0)] = 3$ ft $(2,3) = 3$

Next with $i=3$; so $j=4$; as $i<k\leq j$; so the possible value for $k=4$

$W(3,4) = P(4) + Q(4) + W(3,3) = 1+1+1 = 3$

$C(3,4) = W(3,4) + \min\{[C(3,3) + C(4,4)]\}$

$= 3 + [(0+0)] = 3$ ft $(3,4) = 4$

Step 2:-

computing all $C(i,j)$ such that $j-i=2$; $j = i+2$ and as $0\leq i<3$; $i=0,1,2$;

$i<k\leq j$.

Start with $i=0$; so $j=2$; as $i<k\leq j$, so the possible values for $k=1$ and 2.

$W(0,2) = P(2) + Q(2) + W(0,1) = 2+1+9 = 12$

$C(0,2) = W(0,2) + \min\{(C(0,0) + C(1,2)),$

$(C(0,1) + C(2,2))\}$.

$= 12 + \min\{(0+6, 9+0)\} = 12+6$

$= 18$ ft $(0,2) = 1$

Next with i=1, so j=3 as i<k≤j; so the possible value for k=2 and 3.

$W(1,3) = p(3) + Q(3) + W(1,2) = 1+1+6 = 8$

$C(1,3) = W(1,3) + \min\{[C(1,1) + C(2,3)], [C(1,2) + C(3,3)]\}$

$= W(1,3) + \min\{(0+3), (6+0)\}$

$= 8 + 3 = 11$

$ft(1,3) = 2$

Next with i=2, so j=4, as i<k≤j, so the possible value for k=3 and 4.

$W(2,4) = p(4) + Q(4) + W(2,3) = 1+1+3 = 5$

$C(2,4) = W(2,4) + \min\{[C(2,2) + C(3,4)], [C(2,3) + C(4,4)]\}$

$= 5 + \min\{(0+3), (3+0)\}$

$= 5 + 3 = 8$

$ft(2,4) = 3$

Step 3:-

computing all C(i, j) such that j-i=3; j=i+3 and as 0≤i<2; i=0, 1; i<k≤j.

Start with i=0, so j=3 as i<k≤j, so the possible values for k=1, 2 and 3.

$W(0,3) = p(3) + Q(3) + W(0,2) = 1+1+12 = 14$

$C(0,3) = W(0,3) + \min\{[C(0,0) + C(1,3)], [C(0,1) + C(2,3)], [C(0,2) + C(3,3)]\}$

$= 14 + \min\{(0+11), (9+3), (18+0)\}$

$= 14 + 11$

$= 25 \quad ft(0,3) = 1$

Start with $i=1$, so $j=4$; as $i < k \leq j$, so the possible values for $k = 2, 3$ and $4$.

$W(1,4) = p(4) + Q(4) + W(1,3) = 1+1+8 = 10$

$C(1,4) = W(1,4) + \min\{[C(1,1) + C(2,4)], [C(1,2) + C(3,4)], [C(1,3) + C(4,4)]\}$

$= 10 + \min\{(0+8), (6+3), (11+0)\}$

$= 10 + 8 = 18 \quad ft(1,4) = 2$

Step 4:

computing all $C(i,j)$ such that $j-i=4$; $j = i+4$ and as $0 \leq i < 1$; $i=0$; $i < k \leq j$. Start with $i=0$; so $j=4$; as $i < k \leq j$, so the possible values for $k = 1, 2, 3$ and $4$.

$W(0,4) = p(4) + Q(4) + W(0,3)$

$= 1+1+14 = 16$

$C(0,4) = W(0,4) + \min\{[C(0,0) + C(1,4)], [C(0,1) + C(2,4)], [C(0,2) + C(3,4)], [C(0,3) + C(4,4)]\}$

$= 16 + \min\{(0+18), (9+8), (18+3), 25+0)\}$

$= 16 + 17$

$= 33 \quad R(0,4) = 2$

Table for recording W(i,j), C(i,j) and R(i,j)

| Column / Row | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 2,0,0 | 1,0,0 | 1,0,0 | 1,0,0 | 1,0,0 |
| 1 | 9,9,1 | 6,6,2 | 3,3,3 | 3,3,4 | |
| 2 | 12,18,1 | 8,11,2 | 5,8,3 | | |
| 3 | 14,25,2 | 11,18,2 | | | |
| 4 | 16,33,2 | | | | |

→ From the table we see that C(0,4)=33 is the minimum cost of a binary search tree for $(a_1, a_2, a_3, a_4)$.

→ The root of the tree 'T04' is 'a2'.

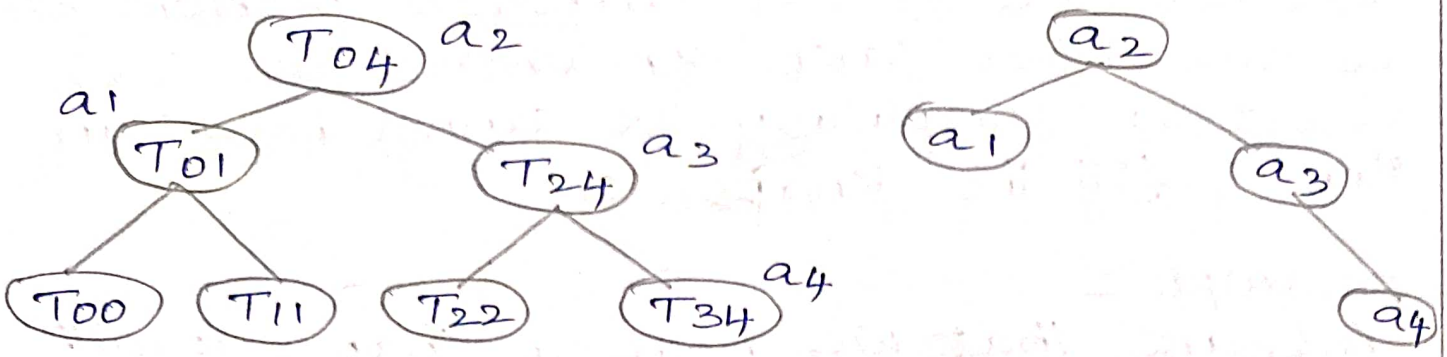
→ Hence the left subtree is 'T01' and right sub tree is T24. The root of 'T01' is 'a1' and the root of 'T24' is a3.

→ The left and right sub trees for 'T01' are 'T00' and 'T11' respectively. The root of T01 is 'a1'. The left and right sub trees for T24 are T22 and T34 respectively.

→ The root of T24 is 'a3'.

→ The root of T22 is 'null'.

→ The root of T34 is 'a4'




## 0/1 KNAPSACK:-

Problem statement:-

→ The profit should be maximum.

Note:-

→ Here fractions are not included.

→ $x_i$ value should be = 0 / 1.

Formula:-

1. max $\sum x_i p_i$ (sum of profits should be maximized).
2. $\sum x_i w_i \leq m$ (weights should be less than (or) equal to the bag capacity)

→ In this problem, we have a knapsack that has a weight limit W.

→ There are items $i_1, i_2, \ldots$ in each having weight $w_1, w_2, \ldots w_n$ and some benefit (value (or) profit) associated with it $v_1, v_2, \ldots v_n$.

→ our objective is to maximize the benefit such that the total weight inside the knapsack is at most W.

→ Since this is 0-1 knapsack problem so we can either take an entire item (or) reject it completely. We cannot break an item & fill the knapsack.

Example:-
Assume that we have a knapsack with max weight capacity W=5. Our objective is to fill the knapsack with items such that the benefit (value (or) profit) is maximum.

Consider the following 4 items & their weights & value.

| Items(i) | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Value(val) | 100 | 20 | 60 | 40 |
| Weight(wt) | 3 | 2 | 4 | 1 |

n=4 & W=5

Solution:-
Create a value table V[i,w]
where i denotes number of item & w denotes the weight of the items (capacity)

| V[i,w] | w=0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| i=0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 100 | 100 | 100 |
| 2 | 0 | 0 | 20 | 100 | 100 | 120 |
| 3 | 0 | 0 | 20 | 100 | 100 | 120 |
| 4 | 0 | 40 | 40 | 100 | 140 | 140 |

→ Rows denote the items.
→ columns denotes the weight.
→ As there are 4 items i=0, 1, 2, 3, 4 Initially i=0 & place remaining 4 items.
→ The weight limit of the knapsack is W=5 so we have 6 columns from 0 to 5.

Initially:-

Step 1:-

i=0 , w=0

Formula:-

wt[i] > w

V[i, w] = V[i-1, w]

wt[0] > 0

wt[0] > 0

0 > 0 [false]

Step 2:-

1) i=1, w=0

wt[1] > 0

3 > 0 [True]

V[1, 0] = V[1-1, 0]

= V[0, 0]

**V[1, 0] = 0**

2) i=1, w=1

wt[1] > 1

3 > 1 [True]

V[1, 1] = V[1-1, 1]

= V[0, 1]

**V[1, 1] = 0**

3) i=1, w=2

wt[1] > 2

3 > 2 [True]

V[1, 2] = V[1-1, 2]

= V[0, 2]

**V[1, 2] = 0**

4) i=1, w=3

wt[1] > 3

3 > 3 [false]

**V[i, w] = max( V(i-1), w), Val[i] + V[i-1, w - wt[i])**

$V(1,3) = \max(V(1-1,3), val[1] + V(1-1, 3-wt[1]))$

$= \max(V(0,3), 100 + V(0, 3-3))$

$= \max(V(0,3), 100 + V(0,0))$

$= \max(0, 100+0)$

$\boxed{V(1,3) = 100}$

5) $i=1, w=4$

$wt[1] > 4$

$3 > 4$ [False]

$V(1,4) = \max(V(1-1,4), val[1] + V(1-1, 4-wt[1])$

$= \max(V(0,4), val[1] + V(0, 4-3))$

$= \max(0, 100 + V(0,1))$

$= \max(0, 100+0)$

$V(1,4) = \max(0, 100)$

$\boxed{V(1,4) = 100}$

6) $i=1, w=5$

$wt[1] > 5$

$3 > 5$ [false]

$V(1,5) = \max(V(1-1,5), val[1] + V(1-1, 5-wt[1])$

$= \max(V(0,5), 100 + V(0, 5-3))$

$= \max(0, 100 + V(0,2))$

$= \max(0, 100+0)$

$\boxed{V(1,5) = 100}$

Step 3 :-

1) $i=2, w=0$

$V(2,0)=0$

2) $i=2, w=1$

$V(2,1)=0$

3) $i=2, w=2$

$V(2,2)=20$

4) $i=2, w=3$

$V(2,3)=100$

5) $i=2, w=4$

$V(2,4)=100$

6) $i=2, w=5$

$V(2,5)=120$

Step 4 :-

1) $i=3, w=0$

$V(3,0)=0$

2) $i=3, w=1$

$V(3,1)=0$

3) $i=3, w=2$

$V(3,2)=20$

4) $i=3, w=3$

$V(3,3)=100$

5) $i=3, w=4$

$V(3,4)=100$

6) $i=3, w=5$

$V(3,5)=120$

Step 5 :-

1) $i=4, w=0$

$V(4,0)=0$

2) $i=4, w=1$

$V(4,1)=40$

3) $i=4, w=2$

$V(4,2)=40$

4) $i=4, w=3$

$V(4,3)=100$

5) $i=4, w=5$

$V(4,5)=140$

6) $i=4, w=5$

$V(4,5)=140$

∴ Maximum value earned

∴ Max value

$= V(n,w)$

$= V[4,5]$

$= 140$

→ Items that were put inside the knap-sack are found using the following rule.

```
set i=n and w=W
while i and w>0 do
  if (V[i,w] != V[i-1,w]) then
    mark the ith item.
    set w=w-wt[i]
    set i=i-1
  else
    set i=i-1
  endif
endwhile.
```

Step 1:-

$V[i,w] \neq V[i-1,w]$

max value = 140 ∵ V(4,5)

$i=4$ , $w=5$

$V[4,5] \neq V[4-1,5]$

$V[4,5] \neq V[3,5]$

$140 \neq 120$ [YES]

∴ Mark the 4th items indicated as '1'.

Step 2:-

set $w=w-wt[i]$

$w=5-wt[4]$

$=5-1$

$\boxed{w=4}$

set $i=i-1$

$i=4-1$

$\boxed{i=3}$

∴ Now i = 3, w = 4

V[3, 4] != V[3-1, 4]

= V[2, 4]

100 != 100 [NO]

Don't mark the 3rd item, indicated as '0'.

Step 3:-

set i = i-1, i = 3-1, **i = 2**

V[2, 4] != V[2-1, 4]

= V[1, 4]

100 != 100 [NO]

Don't mark the 2nd item indicated as '0'.

Step 4:-

Set i = i-1

i = 2-1

**i = 1**

V[1, 4] != V[1-1, 4]

V[1, 4] != V[0, 4]

100 != 0 [True]

So, Mark the 1st item, indicated as '1'.

Conclusion:-

→ So item we are putting inside the Knapsack are 4 & 1.

$x = \{1, 0, 0, 1\}$.

## ALL PAIRS SHORTEST PATH :-

→ Floyd-warshall's algorithm is used for all pairs shortest path.

→ It finds the shortest path between every pair of vertices.

Steps :-

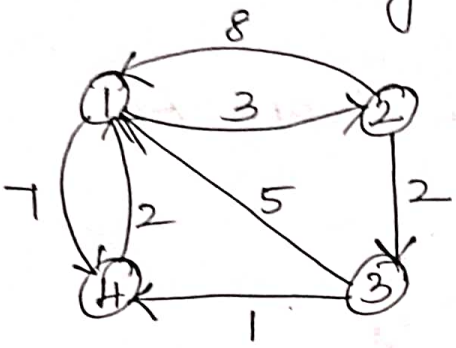
1. To do dynamic programming, we should find middle element.
2. The middle element should traverse to all vertices.
3. It can be done by using matrix.
4. Loops are not possible (self loops) so it indicate as zero in matrix.
5. If an absence of edge then indicate as '$\infty$'.

```
Algorithm AllPaths(cost, A, n)
// cost[1:n, 1:n] is the cost adjacency matrix of
   a graph which n vertices;
// A[i, j] is the cost of a shortest path from
   vertex i to vertex j;
// cost[i, i] = 0.0, for 1 ≤ i ≤ n.
  {
     for i:=1 to n do
          for j:=1 to n do
               A[i, j]:=cost[i, j];
```

```
for K:=1 to n do
    for i:=1 to n do
        for j:=1 to n do
            A[i,j]:=min(A[i,j], A[i,K] +
                                A[K,j]);
}
```

Example:-

consider the graph.




Evaluate the graph in matrix form.

$$A_0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

Now I am considering intermediate vertex, $A_1$

$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & & \\ 5 & & 0 & \\ 2 & & & 0 \end{bmatrix} \end{array}$$

→ Remaining values i need to find out formula.

$$A_0[i,j] = \begin{cases} w, & \text{if there is an edge} \\ \infty, & \text{if there is no edge} \\ 0, & \text{if } i == j \text{ (diagonal element).} \end{cases}$$

→ Here '4' vertices are present, so we will find '4' matrices.

$$A_k[i,j] = \min(A_{k-1}[i,j],\ A_{k-1}[i,k] + A_{k-1}[k,j])$$

step 1:-

copy row 1 and column 1 from $A_0$

$$A_1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & & \\ 5 & & 0 & \\ 2 & & & 0 \end{bmatrix} \end{array}$$

$$A_1[2,2] = \min(A_{1-1}(2,2),\ A_{1-1}(2,1) + A_{1-1}(1,2))$$
$$= \min(A_0(2,2),\ A_0(2,1) + A_0(1,2))$$
$$= \min(0,\ 8+3)$$
$$= \min(0,\ 11)$$

$$A_1(2,2) = 0$$

$$A_1(2,3) = \min(A_0(2,3),\ A_0(2,1) + A_0(1,3))$$
$$= \min(2,\ 8+\infty)$$
$$= \min(2,\ \infty)$$

$$A_1(2,3) = 2$$

$A_1(2,4) = \min[A_0(2,4), A_0(2,1) + A_0(1,4)]$

$= \min(\infty, 8+7)$

$= \min(\infty, 15)$

$A_1(2,4) = 15$

$A_1(3,2) = \min[A_0(3,2), A_0(3,1) + A_0(1,2)]$

$= \min(\infty, 5+3)$

$= \min(\infty, 8)$

$A_1(3,2) = 8$

$A_1(3,4) = \min[A_0(3,4), A_0(3,1) + A_0(1,4)]$

$= \min(1, 5+7)$

$= \min(1, 12)$

$A_1(3,4) = 1$

$A_1(4,2) = \min\{A_0(4,2), A_0(4,1) + A_0(1,2)\}$

$= \min\{\infty, 2+3\}$

$= \min(\infty, 5)$

$A_1(4,2) = 5$

$A_1(4,3) = \min\{A_0(4,3), A_0(4,1) + A_0(1,3)\}$

$= \min\{\infty, 2+\infty\}$

$A_1(4,3) = \infty$

$A_1 =$

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 3 | $\infty$ | 7 |
| 2 | 8 | 0 | 2 | 15 |
| 3 | 5 | 8 | 0 | 1 |
| 4 | 2 | 5 | $\infty$ | 0 |

Step 2:-

copy row2 and column 2 from $A_1$

$$A_2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & & \\ 8 & 0 & 2 & 15 \\ & 8 & & \\ & 5 & & \end{bmatrix} \end{array}$$

$A_2(1,1) = \min[A_1(1,1), A_1(1,2) + A_1(2,1)]$

$= \min(0, 3+8)$

$= \min(0, 11)$

$A_2(1,1) = 0$

$$A_2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array}$$

Step 3:-

copy row3 and column 3 from $A_2$

$$A_3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & & 5 & \\ & 0 & 2 & \\ 5 & 8 & 0 & 1 \\ & & 7 & 0 \end{bmatrix} \end{array}$$

$A_3(1,2) = \min\{$

$$A_3 = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 7 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

Step 4:-

copy row 4 and column 4 from $A_3$

$$A_4 = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 3 & 6 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

## THE TRAVELLING SALES PERSON PROBLEM:-

→ It is one of the algorithm strategy used in dynamic programming.

→ Here the salesman should start at a point and travels all the places and comes back to starting point.

→ The main objective of the problem is to minimize the travelling cost.

→ The main requirement is there should be communication between nodes.

Formula for calculating the cost adjacency matrix in dynamic programming is,

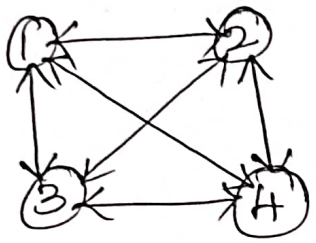
$$g(i,S) = \begin{cases} \min \\ j \in S \end{cases} \{c_{ij} + g(j, S - \{j\})\}$$

$g(i, s)$ → length of shortest path starting at vertex $i$, going through all vertices in $s$ & terminating at vertex 1.

$g\{1, V-\{1\}\}$ is the length of an optimal salesperson tour.

Example:-

For the following graph find minimum cost four for the travelling salesperson problem.




The cost adjacency matrix

$$= \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

Let the cost adjacency matrix

$$C_{ij} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix} \end{array}$$

→ Let us start the tour from vertex 1.

formula:-

$$g(i, s) = \min\{C_{ij} + g(j, s-\{j\})\} \rightarrow (1)$$

clearly,

$$g(1, \phi) = C_{11} = 0$$
$$g(2, \phi) = C_{21} = 5$$
$$g(3, \phi) = C_{31} = 6$$

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

Using equation 1 we obtain

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

$g(2, \{3,4\}) = \min\{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\}$

$g(3, \{4\}) = \min\{c_{34} + g(4, \phi)\}$.

$= \min\{12 + 8\}$

$= 20$

$g(4, \{3\}) = \min\{c_{43} + g(3, \phi)\}$

$= 9 + 6$

$= 15$

Therefore calculate the value for $g(2, \{3,4\})$

$g(2, \{3,4\}) = \min\{c_{23} + g(3, \{4\}), (c_{24} + g(4, \{3\})\}$

$= \min\{9 + 20, 10 + 15\}$

$= \min\{29, 25\}$

$g(2, \{3,4\}) = 25$

Therefore, $g(3, \{2,4\})$

$g(3, \{2,4\}) = \min\{c_{32} + g(2, \{4\}), c_{34} + g(4, \{2\})\}$.

$g(2, \{4\}) = \min\{c_{24} + g(4, \phi)\}$

$= 10 + 8$

$= 18$

$g(4, \{2\}) = \min\{C_{42} + g(2, \phi)\}$

$= 8 + 5$

$= 13$

$g(3, \{2,4\}) = \min\{13+18, 12+13\}$

$= \min\{31, 25\}$

$= 25$

Therefore $g(4, \{2,3\})$

$g(4, \{2,3\}) = \min\{C_{42} + g(2, \{3\}),$

$C_{43} + g(3, \{2\})\}.$

$g(2, \{3\}) = \min\{C_{23} + g(3, \phi)\}$

$= \min\{9+6\}$

$= 15$

$g(3, \{2\}) = \min\{C_{32} + g(2, \phi)\}$

$= 13 + 5$

$= 18$

$g(4, \{2,3\}) = \min\{8+15, 19+18\}$

$= \min\{23, 27\}$

$g(3, \{2,4\}) = 23$

$\therefore g(1, \{2,3,4\}) = \min\{C_{12} + g(2, \{3,4\}),$

$C_{13} + g(3, \{2,4\}),$

$C_{14} + g(4, \{2,3\})\}$

$= \min\{10+25, 15+25,$

$20+23\}$

$= \min\{35, 40, 43\}$

$g(1, \{2, 3, 4\}) = 35$

∴ The optimal tour for the graph has length = 35.

(or)

minimum cost tour for the travelling salesperson problem is 35.

∴ The best optimal toue path is

$1 \to 2 \to 4 \to 3 \to 1$

——— x ———