UNIT-IV Greedy method and Backtracking

=) The General Greedy method:-

-) most straight forward design technique.

a) most problems have n inputs.

b) Solution -) contains subset of inputs that satisfies a given constraint.

c) Feasible solution -) Any subset that satisfies the constraint.

d) Need to find a feasible solution that maximizes/minimizes a given objective function - optimal solution.

-) Used to determine a feasible solution that may/ may not be optimal.

a) At every point, make a decision that is locally optimal; & hope that it leads to a globally optimal solution.

b) leads to a powerful method for getting a solution that works well for a wide range of applications.
(whole)

⇒ $n = 3$, $m = 20$, $(p_1, p_2, p_3) = (25, 24, 15)$,

$(w_1, w_2, w_3) = (18, 15, 10)$.

Four feasible solutions are:-

| | $(x_1, x_2, x_3)$ | $\sum w_i x_i$ | $\sum p_i x_i$ |
|---|---|---|---|
| a) | $(\frac{1}{2}, \frac{1}{3}, \frac{1}{4})$ | 16.5 | 24.25 |
| b) | $(\frac{1}{4}, \frac{2}{15}, 10)$ | 20 | 28.2 |
| c) | $(0, 2/3, 1)$ | 20 | 31 |
| d) | $(0, 1, \frac{1}{2})$ | 20 | 31.5 |

out of these,
Solution 4 yields maximum profit. As we shall soon see, this solution is optimal for the given problem instance.

$m = 15$

| i | a | b | c |
|---|---|---|---|
| P | 10 | 20 | 15 |
| w | 4 | 10 | 5 |

⇒ Change making problem:-

Given unlimited amounts of coins ~~of~~ of denominations $d_1 > - - > d_m$, give change of amount $n$ with the least no. of coins.

Ex:- $d_1 = 25c$, $d_2 = 10c$, $d_3 = 5c$, $d_4 = 1c$ & $n = 48c$

Greedy solution:-

→ Pick

$d_1$ → As it reduces remaining most (to 23)

$d_2$ → remaining amount reduces to 13

$d_2$, amount remaining, reduces to 3

$d_4$, Remaining amount, reduces to 2

$d_4$, remaining amount, reduces to 1

$d_4$, remaining amount, reduces to 0.

⇒ Greedy Technique:-

→ Constructs a solution to an optimization problem piece by piece through a sequence of choices that are:-
a) feasible, b) locally optimal, c) irrevocable.

→ For some problems, yields an optimal solution for every instance.

→ For most, doesn't but can be useful for fast approximations.

⇒ The General method:-

```
Algorithm Greedy(a,n)
// a[1:n] contains the n inputs
{
    Solution := 0; // Initialize the solution.
    for i:=1 to n do
    {
        x := Select(a);
        if Feasible(Solution, x) then
    Solution := union(Solution, x);
```

}

return Solution;

}

Greedy method control abstraction for the Subset paradigm.

⇒ Knapsack problem:-

→ problem definition,

→ Given n objects and a knapsack where object i has a weight $w_i$ & Knapsack has a capacity m.

→ If a fraction $x_i$ of object i placed into Knapsack, a profit $p_i x_i$ is (added) earned.

→ The objective is to obtain a filling of knapsack maximizing the total profit

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i \quad (4.1)$$

$$\text{Subject to} \sum_{1 \le i \le n} w_i x_i \le m \quad (4.2)$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n \quad (4.3)$$

→ A feasible solution is any set satisfying (4.2) & (4.3)

→ An optimal Solution is a feasible Solution for which

(4.1) is maxi

⇒ Algorithm

// p[1:n]-

// of the

// 2 pl

// x[i

{

for

for

{

}

(a.c) is maximized.

=> Algorithm GreedyKnapsack(m, n)

// p[1:n] - profits, w[1:n] - weights,
// of the n objects ordered such that p[i]/w[i]
// ≥ p[i+1] / w[i+1]. m is the Knapsack size &
// x[1:n] is the Solution vector.

```
{
    for i:=1 to n do x[i] := 0.0; //Initialise x
    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];
}
```

$m = 15$

| i | 1 | 2 | 3 |
|---|---|---|---|
| p | 15 | 10 | 20 |
| w | 5 | 4 | 6 |
| p/w | 3 | 2.5 | 2 |

| i | U | w[i] | x[i] |
|---|---|---|---|
| 1 | 15 | 5 | 1 |
| 2 | 10 | 4 | 1 |
| 3 | 6 | 10 | 0.6 |

$P = 15 + 10 + 20 * (0.6) = 37$

Algorithm for greedy strategies for the Knapsack problem.

⇒ Time Complexity :-

→ Sorting :- $O(n \log n)$ using fast sorting algorithm like merge sort.

→ Greedy Knapsack : $O(n)$

→ So, total time is $O(n \log n)$

→ if $P_1/w_1 \geq P_2/w_2 \geq \cdots \geq P_n/w_n$, then Greedy Knapsack generates an optimal solution to the given instance of the Knapsack problem.

⇒ Find an optimal solution to the knapsack instance

$n = 7$
$m = 15$

$(P_1, P_2, \ldots P_7) = (10, 5, 15, 7, 6, 18, 3)$

$(w_1, w_2, \ldots w_7) = (2, 3, 5, 7, 1, 4, 1)$

⇒ Job sequencing with Deadlines :-

| $i$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $P_i$ | 5 | 10 | 25 | 15 | 20 |
| $d_i$ | 2 | 1 | 3 | 3 | 1 |

Q) we are given a set (n Jobs).

b) i - intege
c) Job i
b
d) Eac
e)
f)
Ex:-

b) i - integer, $d_i \rightarrow$ deadline, $d_i \geq 0$ & a profit $P_i \geq 0$.

c) Job i, profit $P_i$ is earned iff the Job is completed by it's deadline.

d) Each Job need one unit of time to be completed & only one machine is available.

e) A feasible solution is a subset J of Jobs such that each Job in this subset can be completed by it's deadline & the total profit is the sum of the Jobs' profits in J.

f) An optimal solution is a feasible solution with a maximum profit.

Ex:- $n=4$, $(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$

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

| | Feasible solution | processing sequence | value |
|---|---|---|---|
| 1. | (1,2) | 2, 1 | 110 |
| 2. | (1,3) | 1,3 / 3,1 | 115 |
| 3. | (1,4) | 4, 1 | 127 |
| 4. | (2,3) | 2, 3 | 25 |
| 5. | (3,4) | 4, 3 | 42 |
| 6. | (1) | 1 | 100 |
| 7. | (2) | 2 | 10 |
| 8. | (3) | 3 | 15 |
| 9. | (4) | 4 | 27 |

→ Begin <u>optimization function</u>
with $J = \phi$.

→ $J_1$ considered, added to $J \rightarrow J = \{1\}$

→ $J_4$ considered, added to $J \rightarrow J = \{1,4\}$

→ $J_3$ considered, but discarded ∵ not feasible → $J = \{1,4\}$

→ $J_2$ considered, but discarded ∵ not feasible → $J = \{1,4\}$

→ Final Solution is $J = \{1,4\}$ with total profit 127

→ It is optimal.

→ Greedy method described above always obtains an optimal Solution to to Job Sequencing problem.

→ High level description of Job Sequencing algorithm.

→ Assuming Jobs are ordered as $p[1] \geq p[2] \geq \ldots \geq p[n]$

```
Greedy_Job(int d[], set J, int n)
// J is a set of Jobs that can be
// Completed by their deadlines.
{
    J = {1};
    for(int i=2; i<=n; i++)
    {
        if (all Jobs in J∪{i} Can be Completed
            by the deadlines)
        J = J∪{i};
    }
}
```

Scanned By Camera Scanner

## ⇒ Single Source shortest path:-

Length = Sum of weights of the edges.

$V_0$ - starting vertex

$V_n$ - destination

Digraphs to allow for one-way Streets.

$G=(V,E)$, $V_q$-Source vertex.

weighting fn cost for the edges 'G'

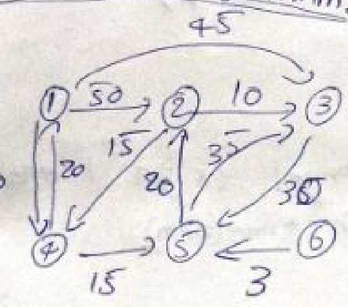
Shortest path from $V_q \rightarrow$ all vertices (G)

weights ⇒ +(ve)

$V_0 \rightarrow$ other node is an ordering among subset of the edges.

This problem fits the ordering (digph) paradigm.

## ⇒ Dijkstra Algorithm:-




a) Graph

| | Path | length |
|---|---|---|
| a) | 1,4 | 10 |
| b) | 1,4,5 | 25 |
| c) | 1,4,5,2 | 45 |
| d) | 1,3 | 45 |

b) shortest path from 1.

⇒ Algorithm ShortPa (v, cost, dist, n) // dist[j], $1 \le j \le n$ is set to
// length of shortest path
// from vertexes v to j
// in digraph G
// with n vertices.
// dis[v] = 0, G is represented by it's
// cost adjacency matrix cost [1:n,1:n].

```
{
    for i:=1 to n do
    {   // Initializes S.
        S[i]:= false; dist[i]:= cost[v,i];
    }
```

S[v] := true; dist[v] = 0.0; // put v in S.

for num := 2 to n-1 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)

if (dist[w] > (dist[u] + cost[u,w])) then

dist[w] := dist[u] + cost[u,w]; // (update) // distances

}

⇒ optimal merge pattern:-

→ 2 sorted files → n, → m records ⇒ merged to get 1 sorted file in time(n+m).

→ >2 sorted files merged together, done by repeatedly merging sorted files in pair.

→ If $x_1, x_2, x_3, x_4$ for merging,

$$x_1 + x_2 = y_1$$

$$y_1 + x_3 = y_2$$

$y_2 + x_4$ → desired sorted file.

Alternatively, $x_1 + x_2 = y_1$, $x_3 + x_4 = y_2$

$y_1 + y_2$ → desired sorted file.

⇒ no. of ways to merge 'n' Sorted files.

→ Diff. pairs, diff. computing's, one best optimal way to merge (we want) that.

Ex:- $x_1 - 30$, $x_2 - 20$, $x_3 - 10$

a) $x_1 + x_2$ ($y_1$) → 50 moves
$y_1 + x_3$ → 60 moves
110 moves needed

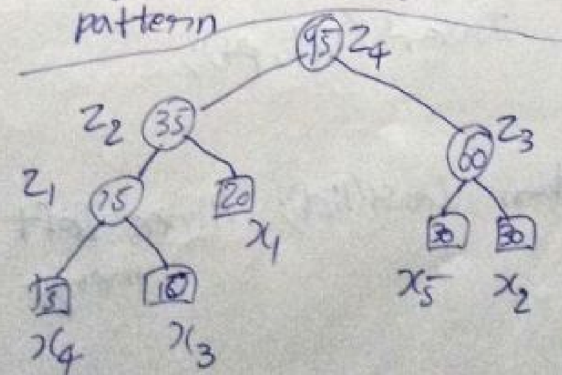
b) $x_2 + x_3$ → 30 moves
$y_1 + x_1$ → 60 moves
90 moves needed

2nd merging pattern is better than 1st.

Ex:-

Binary merge Tree /
for getting optimal merge pattern $(x_1, x_2, x_3, x_4 \& x_5)$
$= (20, 30, 10, 5, 30)$

combining,
Small → Big




$x_3 + x_4 = y_1$
$y_1 + x_1 = y_2$
$y_1 + y_2 = y_3$
$x_2 + x_5 = y_4$
$y_3 + y_4 =$ desired sorted file

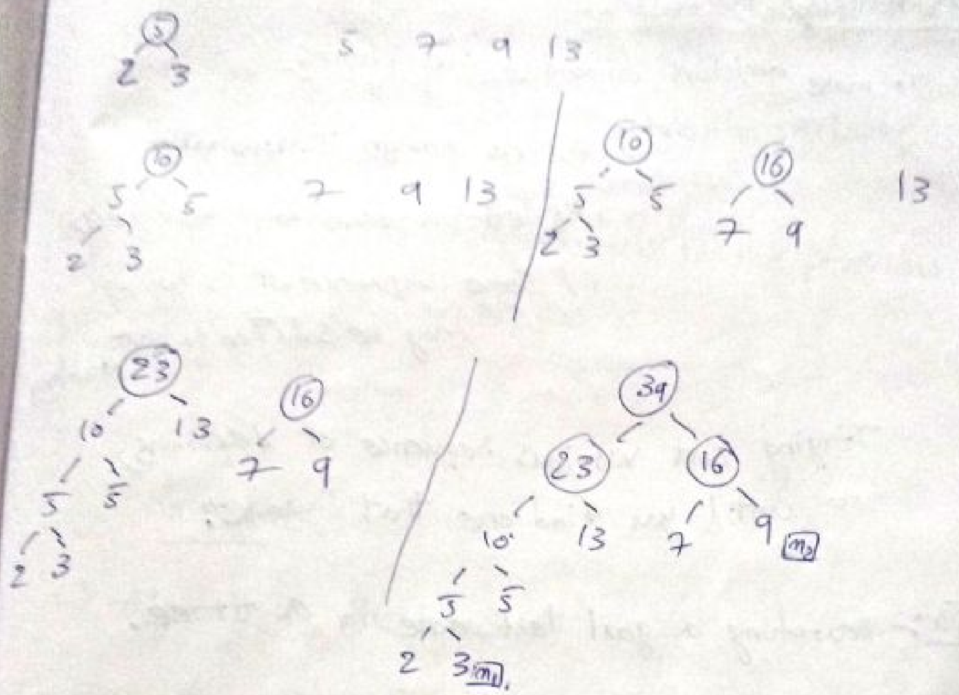
→ Algorithm:

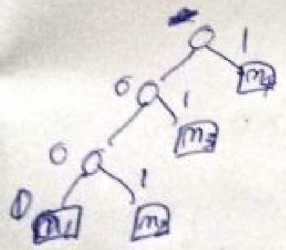
```
treenode = record {
        treenode * lchild;
        treenode * rchild;
        integer weight;
          };

Algorithm Tree(n)      // list is a global of n single node
{                      // binary trees as described below
   for i:=1 to n-1 do
   {
      pt: = new treenode; //get a new tree node.
      (pt→lchild): = Least(list); //merge two trees
      (pt→rchild): =Least(list); //with small lengths.
      (pt→weight): = ((pt→lchild)→weight)
                 + ((pt→rchild)→weight);
      Insert(list, pt);
   }
   return Least(list); //tree left in list is the
                              merge tree
}
```

Time complexity:- $O(n\log n)$

→ Huffman Code:-

2 3 5 7 9 13





$m_1 = 000$

$m_2 = 001$

$m_3 = 01$

$m_4 = 1$

Decoding messages

→ velocity

$\sum_{1 \le i \le n+1} q_i d_i$ → distance from external node to message.

```
{
  if (x[1], ---, x[k] is a path to an answer node)
        then write (x[1:k]);
              k = k+1; // consider the nextset;
}
else k := k-1; // Backtrack to the previous set.
}
}
```

Recursive

=) Algorithm Backtrack(k)

```
   // using Recursion.
   // on entering, first k-1 values x[1], x[2], -,
   // x[k-1] of the solution vector.
   // x[1:n] have been assigned, x[] & n are global.
   {
      for (each x[k] ∈ T(x[1], ---, x[k-1])) do
      { if (B_k(x[1], x[2], - - , x[k]) ≠ 0) then
        {
           if (x[1], x[2], - - -, x[k] is a path to
                         an answer node)
              then write (x[1:k]);
           if (k<n) then Backtrack(k+1);
        }
   } }
```

=) The "8 Queens" problem:-

No Queen must be in Same row & Same column & Same diagonal/not attack other.

Number of Queen's = (Queen * queen) board

Choices / make (or) un-make/Stop

-) Brute Force method / naive Algorithm.

Before placing 1 queen => 64*63*62 _ _ .

After placing 1 Queen => 8*8*8_ _ .

=) Algorithm place(k,i)

// Returns true if a queen can be placed
// in $k^{th}$ row & $i^{th}$ column. otherwise it
// returns false, x[ ] is a global array
// whose first (k-1) values have been set.
//Abs(r) returns the abstract value of r.

```
{
    for j:=1 to k-1 do
        if ((x[j] = i) //Two in the same column
                or (Abs(x[j]-i) = Abs(j-k)))
            then return false;
    return true;
```

}

=) A

=) S

Ad

Add

Add

true; true;

}

=) Algorithm Nqueens(k,n)

//using backtracking) returns all possible
// placements of n queens on an
// n×n chess board by any queen with
non-attacking.

```
{
    for i:=1 to n do
    { if place(k,i) then
        {
            x[k]:=i;
            if (k=n) then write(x[1:n]);
            else Nqueens(k+1,n);
    }   }   }
```

=) <u>Sum of Subsets:-</u>

given, n {$w_1, w_2 \ldots w_n$}, S — ~~$\Sigma x_i$ :-~~    $n=3$, $S=6$, $w_1=2$, $w_2=4$, $w_3=6$

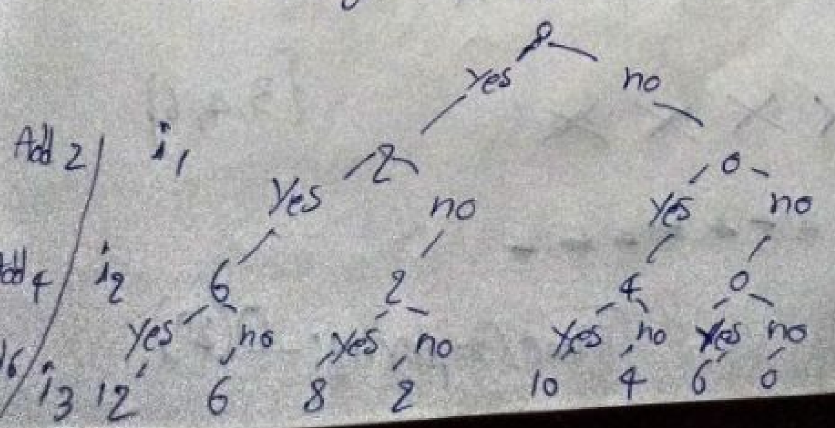
Find subsets $w_1 \ldots w_n$ (+ve) integers.

Sum to S.

Salutions :- {2,4} & {6}

=) binary Full State space tree     All nodes → Promising → Non-promising

(31 nodes)



=> DFS:-

based on height, (No balance to Solution)
checks leaf node & checkes the satisfaction of solution.
Slow process

non-promising node -> not a feasible solution
it is promising node.

A pruned State Space Tree:-

No Non-promising node

$w_1 = 3, w_2 = 4, w_3 = 5, w_4 = 6, S = 13$

0

3     3   0     (15 nodes)

4     4   0   4   0

7   3   4   0

5     5   0   5   0   5   0   5   0

12   7   8   3   9   4   ~~5~~ ~~0~~

6     ~~X~~ ~~X~~   6   0   ~~X~~ ~~X~~ ~~X~~ ~~X~~     {3, 4, 6}

~~X~~ 13 ~~X~~

Time Complexity:- $O(2^n)$

to Solution,
es the
Satisfact

ble Solution,
node.

le

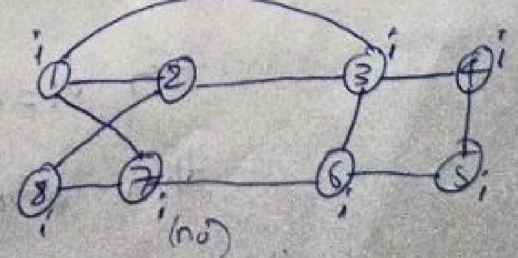
=) Algorithm Sumofsub(s,k,r)

```
{
    x[k] := 1;
    if (s + w[k] = m) then write(x[1:k]);
    else if (s + w[k] + w[k+1] < m)
         then Sumofsub(s + w[k], k+1, r - w[k]);
    if ((s + r - w[k] ≥ m) and (s + w[k+1] ≤ m)) then
    {
        ~~then~~
        x[k] := 0;
        Sumofsub(s, k+1, r - w[k]);
    }
}
```

Recursive backtracking algorithm for sum of Subsets problem.

=> Hamiltonian Cycles:-

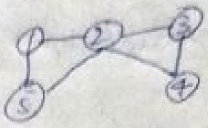
Graph:




1, 3, 4, 5, 6, 7, 8, 2, 1 / 3, 4, 5, 6, 7, 8, 2, 1, 3

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

Graph :-



no hamiltonian cycle

=> Algorithm Nextvalue(k)

// x[1:k-1], path of k-1 vertices, If x[k]=0
// then no vertex has as yet been assigned
// to x[k]. After Execution, x[k] is assigned
// next highest number vertex which been
// already appears in x[1:k-1] & is connected
// by an edge to x[k-1]. otherwise x[k]=0.
// If k=n then in addition x[k] is connected
// to x[1].

```
{
  repeat
  {
    x[k]:=(x[k]+1) mod (n+1); //next vertex.
    if (x[k]==0) then return;
    if (G[x[k-1], x[k]] ≠ 0) then
    {
      //Is there an edge?
        for j:=1 to k-1 do if (x[j]==x[k])
        then break; //check for distinctness.
  if (j==k) then //If true, then the vertex is distinct.
  if ((k<n) or ((k==n) and G[x[n], x[1]] ≠ 0))
```

```
        then return;
    }
} until(false);
}
```

## Generating a next vertex.

```
Algorithm Hamiltonian(k)
// This algorithm uses the recursive formulation
// of backtracking to find all the Hamiltonian
// Cycles, of a graph. The graph is stored as an
// adjacent matrix   G[1:n,1:n]. All cycles begin
//   at node 1.
{
   repeat
   {
     //Generate values for x[k].
     Nextvalue (k); //Assign a legal next value to x[k];
     if(x[k] = =0) then return;
     if (k==n) then write (x[1:n]);
     else  Hamiltonian (k+1);
   }until (false);
}
```

## Finding all Hamiltonian Cycles.

## => Algorithm Tsp using Dynamic programming :-

```
Algorithm Tsp(N,s)
{ Visited[N] = 0;
  cost = 0;
  Visited[s] = 1;
    if |N| = 2 and k≠s then
    {
      cost(N,K) = dist(s,K);
      Return cost;
    }
  else
   {  for j∈N do
       { for i∈N and visited[i] = 0 do
          {
            if j≠i and j≠s then
            {
              cost(N,j) = min(Tsp(N-{i},j) + dist(i,j))
              Visited[j] = 1;
            }
          }
       }
   }.
  Return cost;
}
```