Unit-II
The Greedy Method

1. General method:

- The Greedy method is a most straight forward design technique which can be applied to a wide variety of problems.
- This algorithm works in steps.
- In each step it selects the best available options until all options are finished.
- Most of the problems have n inputs and requires to obtain a subset that satisfies some constraints.
- Any subset that satisfies these constraints is called as a ***feasible solution***.
- A feasible solution that either minimizes or maximizes a given objective function is called as ***Optimal Solution***.
- The Greedy method suggest that one can devise an algorithm that work in stages, considering one input at a time.
- At each stage, a decision is made regarding whether a particular input is an optimal solution or not.
- This is done by considering the inputs in an order determined by some selection procedure.
- If the inclusion of the next input into the partially constructed optimal solution results suboptimal/infeasible solution, then that input is not added to the partial solution. Otherwise, it is added.
- The selection procedure itself is based on some optimization measures.

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

- ***Select*** selects an input from a[] and removes it.
- The selected input's value is assigned to x.
- **Feasible** is a Boolean-valued function that determines whether x can be included into the solution vector or not.
- **Union** combines x with the solution and updates the objective function.

Greedy problems can be classified into 2 types

Subset Paradigm

- To solve a problem (or possibly find the optimal/best solution), greedy approach generate subset by selecting one or more available choices.
  Eg. includes **Knapsack problem, job sequencing with deadlines**.

- Ordering Paradigm
  In this, greedy approach generate some arrangement/order to get the best solution.
  Eg.**Optimal Storage on tapes**

Applications

- ✓ Fractional knapsack algorithm
- ✓ Optimal Storage on tapes
- ✓ Job sequencing with deadline
- ✓ Single source shortest path
- ✓ Activity Selection Problem
- ✓ Minimum Cost Spanning Tree

2. Knapsack Problem

- Given n objects and a Knapsack or Bag.
- This problem also known as fractional knapsack problem because fraction of item/object can be added to knapsack.
- Object i has weight W_i and the Knapsack has a capacity M.
- if a fraction X_i of object i is placed into Knapsack, then a profit of P_iX_i is earned.
- The objective is to obtain a filling of Knapsack that maximizes the total profit earned.
- Problem can be formally defined as

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

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m \qquad 2$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n \qquad 3$$

The profit and weights are the positive numbers.
-Here, A feasible solution is any set (X1, X2, …,Xn) satisfying above rules (2) and (3).
-And an optimal solution is feasible solution for which rule (1) is maximized.

Solution using brute force approach:
Example:
Given N bojects N=3, M=20, (P1, P2, P3)=(25, 24, 15) and (W1, W2, W3)=(18, 15, 10)
Different feasible solutions are:

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1.	(1/2, 1/3, 1/4)	16.5	24.25
2.	(1, 2/15, 0)	20	28.2
3.	(0, 2/3, 1)	20	31
4.	(0, 1, 1/2)	20	31.5

Of these Six feasible solutions, solution 4 yields the maximum profit.
Therefore solution 4 is optimal for the given problem instance.
Consideration 1 - In case the sum of all the weights is ≤ M, then X_i=1, 1 ≤ i ≤n is an optimal solution.
Consideration 2 - All optimal solutions will fill the knapsack exactly.

$n = 3, M = 20, (p_1, p_2, p_3) = (25, 24, 15)$
$(w_1, w_2, w_3) = (18, 15, 10)$
Sol: $p_1/w_1 = 25/18 = 1.39$
$p_2/w_2 = 24/15 = 1.6$
$p_3/w_3 = 15/10 = 1.5$
Optimal solution: $x_1 = 0, x_2 = 1, x_3 = 1/2$
total profit = 24 + 7.5 = 31.5

```
Algorithm GreedyKnapsack(m, n)
// p[1 : n] and w[1 : n] contain the profits and weights respectively
// of the n objects ordered such that p[i]/w[i] ≥ p[i + 1]/w[i + 1].
// m is the knapsack size and x[1 : n] is the solution 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];
    }
    if (i ≤ n) then x[i] := U/w[i];
}
```

Time complexity:

- to sort files according to profit to weight ratio by using any best sorting algorithm it requires nlogn time.
  Optimal profit can be obtained by scanning to find out max take of profit to weight ratio time needed is: n
  Overall Time complexity : nlogn + n = O(nlogn)

3. Optimal Storage on Tapes

- The objective is to find the order of programs to store on the tape so that retrieval time
- for accessing programs is minimum.
- There are n programs that are to be stored on a computer tape of length L.
- Associated with each program i is a length l;
- Clearly, all programs can be stored on the tape if and only if the sum of the lengths of the programs is at most *L*.
- Assumption: whenever a program is *to* be retrieved from this tape, the tape is initially positioned at the front.

- Hence' if the programs are stored in the order I=i_1, i_2' i_3 …. in *the time tj needed to* retrieve program i_j *is proportional to lik* .

- *If* all programs are retrieved equally often then the time t_j needed to retrieve the program.

$$\sum_{1 \le k \le j} l_{i_k}$$

- Expected or mean retrieval time *(MRT) is*

$$(1/n) \sum_{1 \le j \le n} t_j$$

- Minimizing MRT is equivalent to minimizing d(I)=

$$\sum_{1 \le j \le n} \sum_{1 \le k \le j} l_{i_k}$$

Solution using brute force approach

Example 1 Let $n = 3$ and $(l_1, l_2, l_3) = (5, 10, 3)$. There are $n! = 6$ possible orderings. These orderings and their respective D values are:

ordering I	*D(I)*
1,2,3	5 + 5+ 10 + 5+10 + 3 = 38
1,3,2	5 + 5 + 3 + 5+ 3+ 10 = 31
2,1,3	10 + 10 + 5 + 10 + 5 + 3 = 43
2,3,1	10 + 10 + 3 + 10 + 3 + 5 = 41
3,1,2	3 + 3 + 5 + 3+ 5+ 10 = 29
3,2,1	3 + 3+ 10 + 3+10 + 5 = 34

The optimal ordering is 3,1,2.

Greedy Solution:

- The greedy method simply requires us to store the programs in non-decreasing order of their lengths.
- This ordering (sorting) can be carried out in *O(n log n)* time using an efficient sorting algorithm like heap sort.

For the above instance ordering of the programs is 3,1,2

3,1,2 3 + 3 + 5 + 3+ 5+ 10 = 29

```
Algorithm Store(n, m)
// n is the number of programs and m the number of tapes.
{
    j := 0; // Next tape to store on
    for i := 1 to n do
    {
        write ("append program", i,
            "to permutation for tape", j);
        j := (j + 1) mod m;
    }
}
```

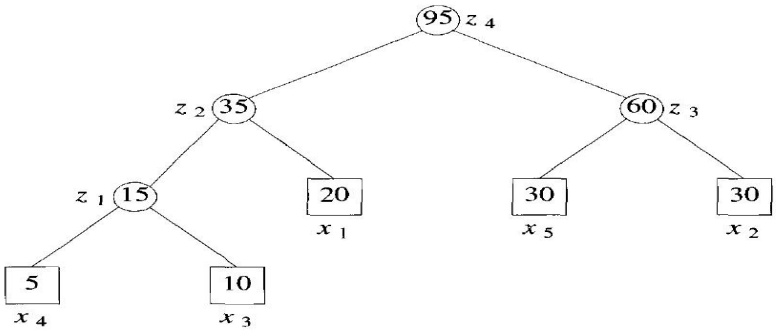
Time complexity:

- Greedy solution requires that the programs are stored in non-decreasing order which can be done in O (nlogn) time by using any best sorting technique like heap sort.

And it requires O(n) time to store/access

4. Optimal Merge Pattern

- Merge a set of sorted files of different length into a single sorted file.
- Given „n‟ sorted files, there are many ways to pair wise merge them into a single sorted file.
- As, different pairings require different amounts of computing time, we want to determine an optimal (i.e., one requiring the fewest comparisons) way to pair wise merge „n‟ sorted files together.
- This type of merging is called as 2-way merge patterns.
- To merge an n-record file and an m-record file requires possibly n + m record moves, the obvious choice choice is, at each step merge the two smallest files together.
- The two-way merge patterns can be represented by binary merge trees.
- Greedy solution to find optimal merge pattern and cost is
  1. Merge two files with minimum number of records.
  2. Add merge file into the list of files.
  3. Repeat the process until single file left.

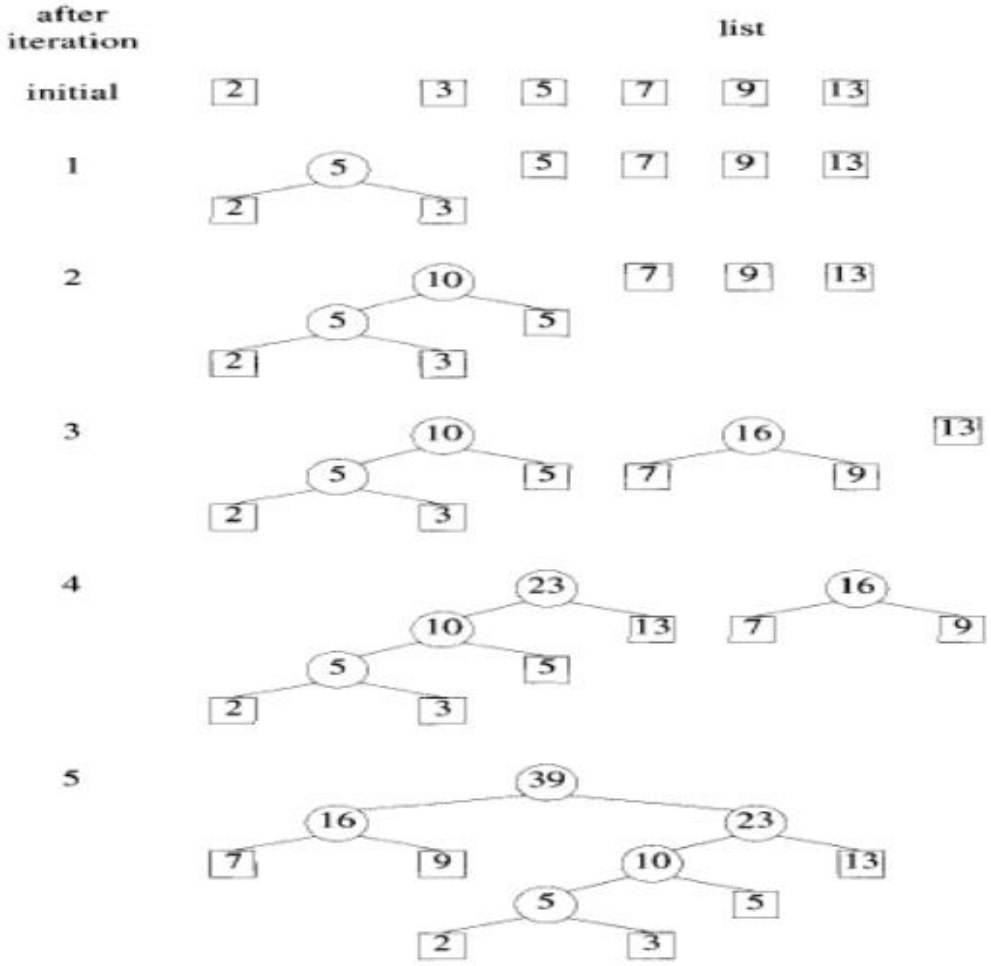


Binary merge tree representing a merge pattern

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

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

Ex:
Given five files (X1, X2, X3, X4, X5) with sizes (20, 30, 10, 5, 30).
Find the optimal merge sequence and cost
Apply greedy rule to find optimal way of pair wise merging to give an optimal solution using binary merge tree representation.



Time complexity:

- if min heap is used to implement priority queue
- Time taken to create min heap = O(n)
  Every time two minimum element will be deleted from min heap in 2 log n time and their sum will be inserted log n after merging.
- It will continue upto n-1 times

$$T(n) = n + (n-1)\,3 \log n$$
$$= n + n \log n$$
$$= O(n \log n)$$

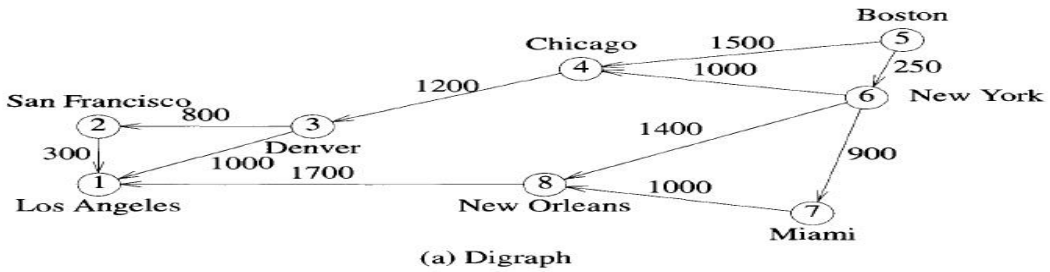
5. Single Source Shortest path Problem

- The problem of finding shortest paths from a source vertex *v* to all other vertices in the graph.
- Dijkstra"s algorithm can be used to solve the problem.
- Works on both directed and undirected graphs.
- However, all edges must have nonnegative weights.

- Given Weighted graph G={E,V} and source vertex $v \in V$, such that all edge weights are non negative
- Algorithm computes the Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices.
- Dijkstra's algorithm initially marks the distance (from the starting point) to every other intersection on the map with infinity.
- Basically, the Dijkstra's algorithm begins from the node to be selected, the source node, and it examines the entire graph to determine the shortest path among that node and all the other nodes in the graph.
- The algorithm maintains the track of the currently recognized shortest distance from each node to the source code and updates these values if it identifies another shortest path.

```
Algorithm ShortestPaths(v, cost, dist, n)
// dist[j], 1 ≤ j ≤ n, is set to the length of the shortest
// path from vertex v to vertex j in a digraph G with n
// vertices. dist[v] is set to zero. G is represented by its
// cost adjacency matrix cost[1 : n, 1 : n].
{
    for i := 1 to n do
    { // Initialize 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) do
            // Update distances.
            if (dist[w] > dist[u] + cost[u, w])) then
                dist[w] := dist[u] + cost[u, w];
    }
}
```

Example:



(a) Digraph

	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	100	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8	1700							0

(b) Length-adjacency matrix

Iteration	S	Vertex selected	Distance							
			LA [1]	SF [2]	DEN [3]	CHI [4]	BOST [5]	NY [6]	MIA [7]	NO [8]
Initial	--	----	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{5}	6	+∞	+∞	+∞	1250	0	250	1150	1650
2	{5,6}	7	+∞	+∞	+∞	1250	0	250	1150	1650
3	{5,6,7}	4	+∞	+∞	2450	1250	0	250	1150	1650
4	{5,6,7,4}	8	3350	+∞	2450	1250	0	250	1150	1650
5	{5,6,7,4,8}	3	3350	3250	2450	1250	0	250	1150	1650
6	{5,6,7,4,8,3}	2	3350	3250	2450	1250	0	250	1150	1650
	{5,6,7,4,8,3,2}									

Time complexity:
The simplest implementation is to store vertices in an array or linked list. This will produce a running time of O(|V|^2 + |E|)
Where |E| is for search needed to find the minimum distance node. //A node may be connected to a maximum of E nodes
For sparse graphs, or graphs with very few edges and many nodes, it can be implemented more efficiently storing the graph in an adjacency list using a binary heap or priority queue.
This will produce a running time of $O((|E|+|V|) \log |V|)$ //inner loop is replaced by a heap