# Design and Analysis of Algorithms

## Unit - III

**Dr. R. Bhuvaneswari**
Assistant Professor
Department of Computer Science
Periyar Govt. Arts College, Cuddalore.

Periyar Govt. Arts College
Cuddalore

# Greedy Method

**Syllabus**

**UNIT - III: THE GREEDY METHOD**

The General Method - Knapsack Problem – Tree Vertex Splitting - Job Sequencing with Deadlines - Minimum Cost Spanning Trees - Optimal Storage on Tapes - Optimal Merge Pattern - Single Source Shortest Paths.

**TEXT BOOK**

Fundamentals of Computer Algorithms, Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran, Galgotia Publications, 2015.

# Greedy Method

## General Method:

- In the method, problems have n inputs and requires to obtain a subset that satisfies some constraints.
- Any subset that satisfies these constraints is called feasible solution.
- A feasible solution should either maximizes or minimizes a given objective function is called an optimal solution.
- The greedy technique in which selection of input leads to optimal solution is called subset paradigm.
- If the selection does not lead to optimal subset, then the decisions are made by considering the inputs in some order. This type of greedy method is called ordering paradigm.

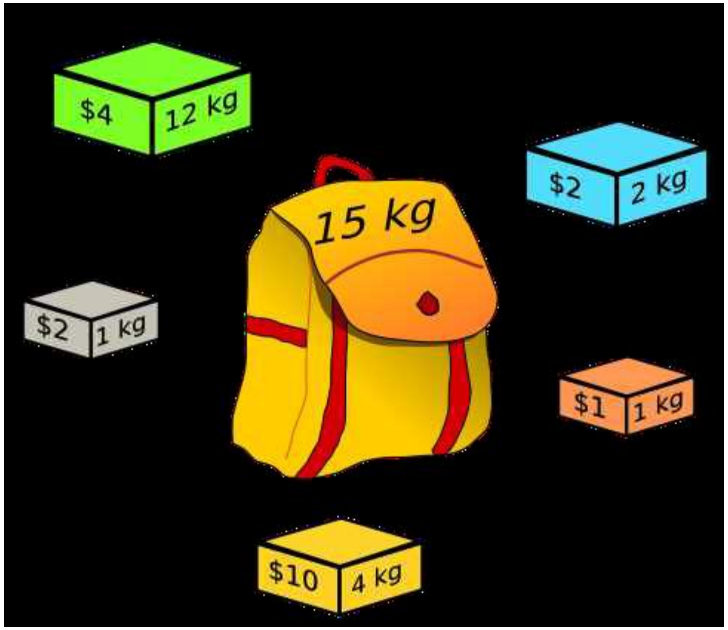
# Greedy Method

## Control Abstraction of Greedy Method

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

# Knapsack Problem

- Given a set of items, each with a weight and a profit, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total profit is as large as possible.
- Items are divisible; you can take any fraction of an item.
- And it is solved using greedy method.



# Knapsack Problem

- Given n objects and a knapsack or bag.
- $w_i$ → weight of object i.
- m → knapsack capacity.
- If a fraction $x_i$, $0 \le x_i \le 1$ of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned.
- Objective is to fill the knapsack that maximizes the total profit earned.
- Problem can be stated as

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

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

$$0 \le x_i \le 1,\ 1 \le i \le n \qquad -----\textcircled{3}$$

- A feasible solution is any set $(x_1 \ldots\ldots x_n)$ satisfying equations ② and ③.
- An optimal solution is a feasible solution for which equation ① is maximized.

# Knapsack Problem

Example: $n = 3$, $m = 20$

| | | | |
|---|---|---|---|
| Weight $w_i$ | 18 | 15 | 10 |
| Profits $p_i$ | 25 | 24 | 15 |

| | $(x_1, x_2, x_3)$ | $\Sigma w_i x_i$ | $\Sigma p_i x_i$ |
|---|---|---|---|
| 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 |
| 5. | (2/3, 8/15, 0) | 20 | 29.5 |
| 6. | (5/6, 1/3, 0) | 20 | 28.8 |

Among all the feasible solutions ④ yields the maximum profit

# Knapsack Problem

**The greedy algorithm:**

Step 1: Sort $p_i/w_i$ into nonincreasing order.

Step 2: Put the objects into the knapsack according to the sorted sequence as possible as we can.

e. g.

| Weight $w_i$ | 15 | 10 | 18 |
|---|---|---|---|
| Profits $p_i$ | 24 | 15 | 25 |

$n = 3, M = 20$

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

$(p_1, p_2, p_3) = (25, 24, 15)$

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:** $x_1 = 0, x_2 = 1, x_3 = 1/2$

# Knapsack Problem

```
Algorithm GreedyKnapsack(m, n)
//n objects are ordered such that p[i]/w[i] ≥ p[i+1]/w[i+1].
{
  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];
}
```

| Weight $w_i$ | 15 | 10 | 18 |
|---|---|---|---|
| Profits $p_i$ | 24 | 15 | 25 |

x[1] = 0.0 **m = 20, n = 3**
x[2] = 0.0
x[3] = 0.0
U = 20
i = 1
x[1] = 1; U = 5
i = 2, 10 > 5
x[2] = 5/10 = 1/2
**x[1] = 1, x[2] = 1/2, x[3] = 0**

# Tree Vertex Splitting

- Weighted directed binary trees are considered.
- The nodes in the tree correspond to the receiving stations and edges correspond to transmission lines.
- The transmission of power from node to another may result in some loss.
- Each edge in the tree is labeled with the loss that occurs in traversing that edge.
- The network may not be able to tolerate losses beyond a certain limit.
- In places where the loss exceeds the tolerance level, boosters have to be placed.

**Given a network and a loss tolerance level, the Tree Vertex Splitting Problem is to determine an optimal placement of boosters.**

- **T = (V, E, W)**
  - ➢ V is the set of vertices
  - ➢ E is the set of edges
  - ➢ w is the weight function for the edges

# Tree Vertex Splitting

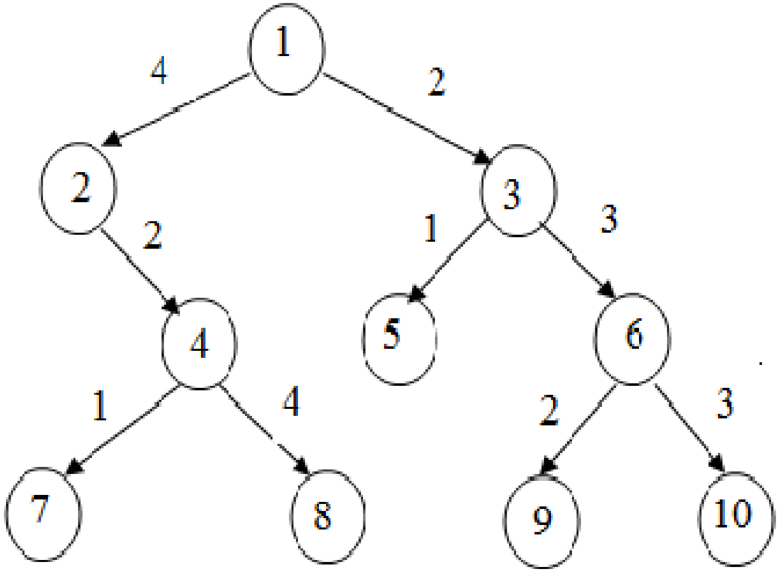
- A vertex with in-degree zero is called a source vertex
- A vertex with out-degree zero is called a sink vertex
- Let T/X be the forest that results when each vertex u is split into two nodes $u^i$ and $u^o$ such that all the edges $\langle u, j\rangle \in E$ ($\langle j, u\rangle \in E$) are replaced by the edges of the form $\langle u^o, j\rangle$ ($\langle j, u^i\rangle$)
- A greedy approach to solve this problem is to compute for each node $u \in V$, the maximum delay d(u) from u to any other node in its subtree.
- If u has a parent v such that

**$d(u) + w(v, u) > \delta$, then the node u gets split and d(u) is set to 0**.

$$d(u) = \max_{v \in C(u)} \{d(v) + W(u, v)\}$$

where C(u) is the set of all children of u.

# Tree Vertex Splitting



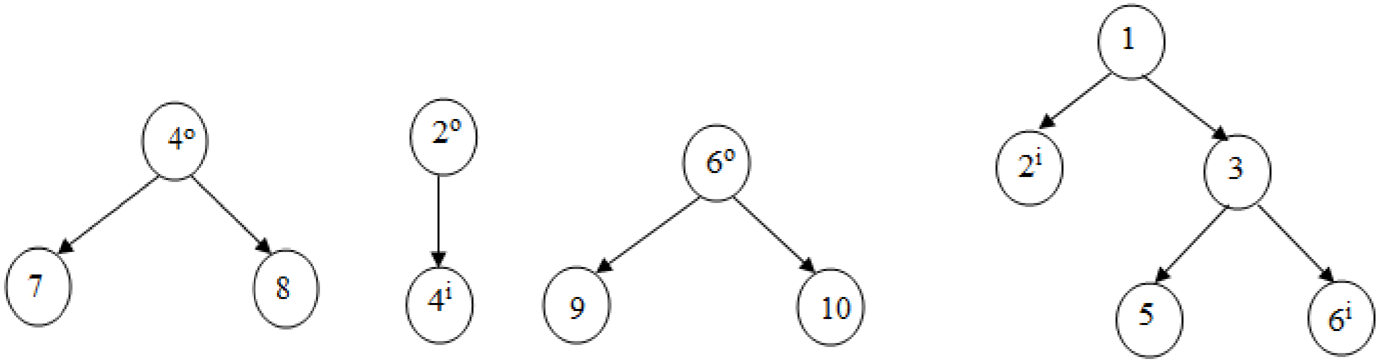
$\delta = 5$

$d(4) = 4$.

since, $d(4) + w(2,4) = 6 > \delta$, node 4 is split and $d(4) = 0$.

since, $d(2) + w(1,2) = 6 > \delta$, node 2 is split and $d(2) = 0$.

since, $d(6) + w(3,6) = 6 > \delta$, node 6 is split and $d(6) = 0$.



# Tree Vertex Splitting

```
Algorithm TVS(T, δ)
{
   if (T ≠ 0) then
   {
      d[T] = 0;
      for each child v to T do
      {
            TVS(v, δ);
            d[T] = max{d[T], d[v]+w[T,v]};
      }
      if ((T is not the root) and (d[T] + w(parent(t), T) > δ)) then
     {
             write(T);
             d[T] = 0;
     }
   }
}
```

# Job sequencing with deadlines

**The problem is stated as below:**

- There are n jobs to be processed on a machine.
- Each job **i** has a deadline $d_i \geq 0$ and profit $p_i \geq 0$ .
- $P_i$ is earned if and only if the job is completed by its deadline.
- The job is completed if it is processed on a machine for unit time.
- Only one machine is available for processing jobs.
- Only one job is processed at a time on the machine.
- A feasible solution is a subset of jobs **J** such that each job is completed by its deadline.

$$\sum_{i \in J} P_i$$

- An optimal solution is a feasible solution with maximum profit value

# Job sequencing with deadlines

## General method of job sequencing algorithm

```
Algorithm GreedyJob(d, J, n)
{
  J := {1};
  for i := 2 to n do
  {
     if (all jobs in $J \cup \{i\}$ can be completed by their deadlines) then
            $J := J \cup \{i\}$;
  }
}
```

# Job sequencing with deadlines

Example: Let n = 4, maximum deadline dmax = 2

$(p_1, p_2, p_3, p_4) = (100,10,15,27)$

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

| | Feasible solution | processing sequence | value |
|---|---|---|---|
| 1. | (1, 2) | 2, 1 | 110 |
| 2. | (1, 3) | 1, 3 or 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 |

# Job sequencing with deadlines

Example 1: Let n = 4, maximum deadline dmax = 2

$(p_1, p_2, p_3, p_4) = (100,10,15,27)$

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

0 1 2

| | |
|---|---|
| J4 | J1 |

27 + 100 = 127

Example 2: Let n = 5, maximum deadline dmax = 3

$(p_1, p_2, p_3, p_4, p_5) = (20, 15, 10, 5, 1)$

$(d_1, d_2, d_3, d_4, d_5) = (2, 2, 1, 3, 3)$

0 1 2 3

| | | |
|---|---|---|
| J2 | J1 | J4 |

15 + 20 + 5 = 40

Example 3: Let n = 6, maximum deadline dmax = 4

$(p_1, p_2, p_3, p_4, p_5, p_6) = (35, 30, 25, 20, 15, 12, 5)$

$(d_1, d_2, d_3, d_4, d_5, d_6) = (3, 4, 4, 2, 3, 1, 2)$

0 1 2 3 4

| | | | |
|---|---|---|---|
| J4 | J3 | J1 | J2 |

20 + 25+ 35+ 30 = 110

# Job sequencing with deadlines

```
Algorithm JS(d, j, n)
// the jobs are ordered such that
p[1] ≥ p[2] ≥ …… ≥ p[n].
{
  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;
}
```

# Minimum Cost Spanning Trees

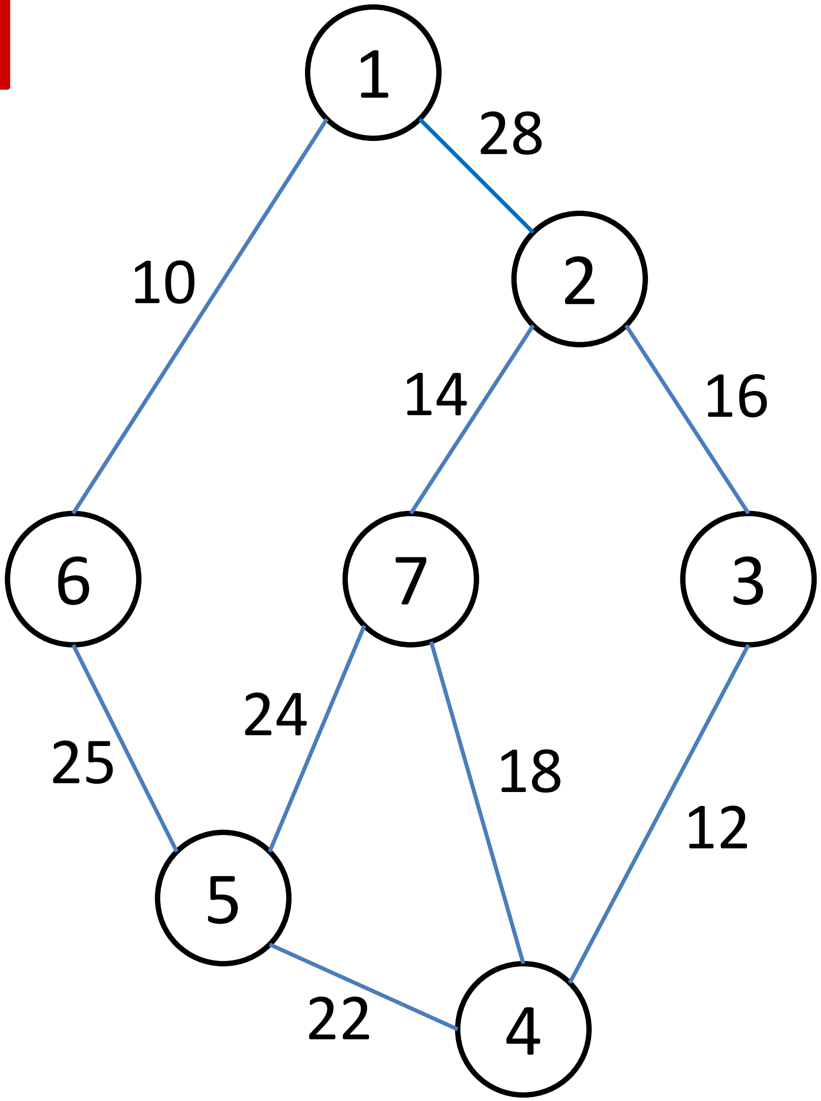
- Given an undirected and connected graph G = (V, E), a spanning tree of the graph G is a subset of graph G, which has all the vertices connected by minimum number of edges.
- The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees.
- A **Minimum Spanning Tree (MST)** is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight.
- There also can be many minimum spanning trees.
- There are two famous algorithms for finding the Minimum Spanning Tree:
  - **Prim's Algorithm**
  - **Kruskal's Algorithm**

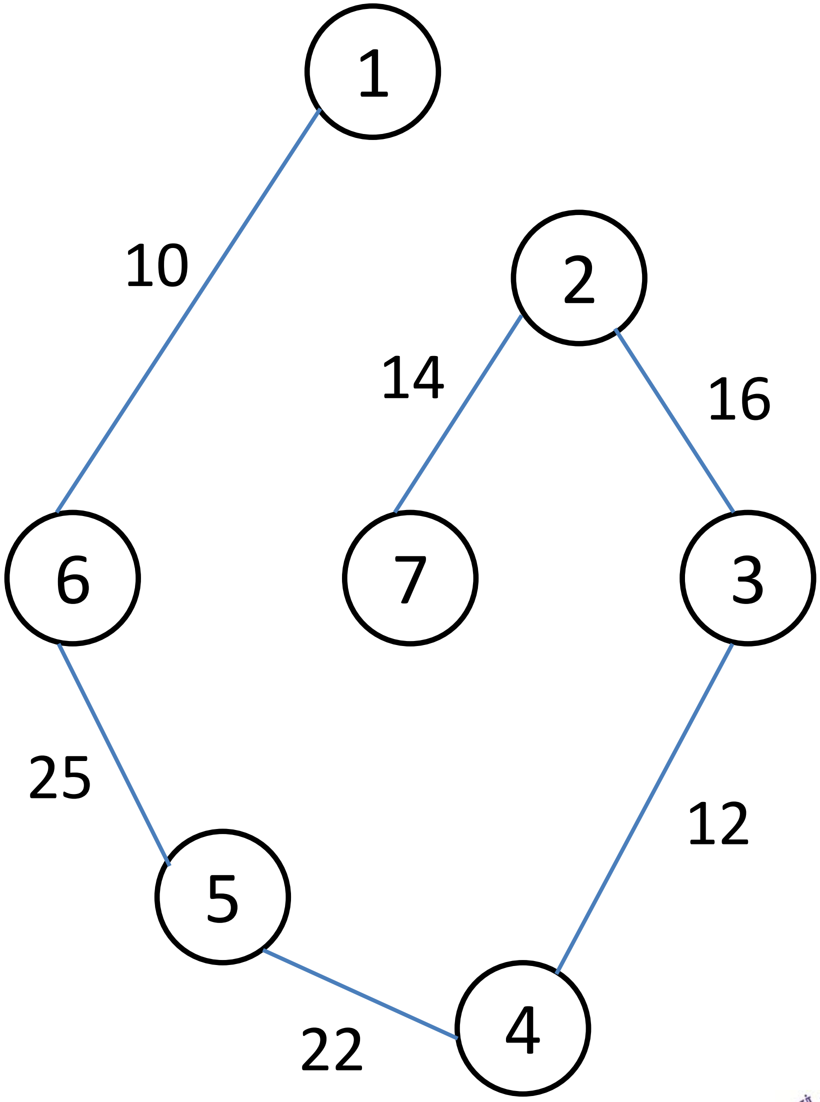
# MST – Prim's Algorithm

- Prim's Algorithm is used to find the minimum spanning tree from a graph.
- Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.
- Prim's algorithm starts with the single node and explore all the adjacent nodes with all the connecting edges at every step.
- The edges with the minimal weights causing no cycles in the graph are selected.
- **Algorithm steps:**

  **Step 1:** Select a starting vertex.
  **Step 2:** Repeat Steps 3 and 4 until there are vertices not in the tree.
  **Step 3:** Select an edge **e** connecting the tree vertex and the vertex that is not in the tree has minimum weight.
  **Step 4:** Add the selected edge and the vertex to the minimum spanning tree **T**
  **Step 5:** Exit

# MST – Prim's Algorithm





# MST – Prim’s 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
//as a set of edges in the array t[1:n-1, 1:2]. The final cost is returned.
{
   Let (k, l) be an edge of minimum cost in E;
   mincost = cost[k, l];
   t[1, 1] = k; t[1, 2] = l;
   for i = 1 to n do
  {
     if (cost[i, l] < cost[i, k]) then near[i] = l;
     else near[i] = k;
  }
  near[k] = near[l] = 0;
```

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

# MST – Prim's Algorithm

```
 for i = 2 to n-1 do
 {
  // find n-2 additional edges for t. Let j be an index such that near[j] ≠ 0
  //and cost[j, near[j]] is minimum;
    t[i, 1] = j;
    t[i, 2] = near[j];
    mincost = mincost + cost[j, near[j]];
    near[j] = 0;
    for k = 1 to n do
   {
        if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j])) then
             near[k] = j;
   }
 }
return mincost;
}
```

# Optimal Storage on tapes

- n programs are to be stored on a computer tape of length l.
- Associated with each program i is a length $l_i$, $1 \leq i \leq n$.
- If the programs are stored in the order $I = i_1, i_2, \ldots\ldots i_n$, the time $t_j$ needed to retrieve the program $i_j$ is $\sum_{1 \leq k \leq j} l_{ik}$
- If all the programs are retrieved equally often, then the Mean Retrieval Time (MRT) is $\frac{1}{n} \sum_{1 \leq j \leq n} t_j$
- Minimizing the MRT is equivalent to minimizing $d(I) = \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq j} l_{ik}$

# Optimal Storage on tapes

**Example:**

$n = 3$,

$(l_1, l_2, l_3) = (5, 10, 3)$

$n! = 6$ possible ordering

| **Ordering I** | **d(I)** | |
|---|---|---|
| 1, 2, 3 | 5+5+10+5+10+3 | = 38 |
| 1, 3, 2 | 5+5+3+5+3+5+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 |

Optimal ordering is 3, 1, 2

Thus the greedy method implies to store the programs in nondecreasing order of their length.

# Optimal Storage on tapes

For more than one tape, **example**,
{12, 34, 56, 73, 24, 11, 34, 56, 78, 91, 34, 45} on three tapes with MRT minimized, store files in non-decreasing length.
{11, 12, 24, 34, 34, 34, 45, 56, 56, 73, 78, 91}

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

| | | | | |
|---|---|---|---|---|
| Tape 0 | 11 | 34 | 45 | 73 |
| Tape 1 | 12 | 34 | 56 | 78 |
| Tape 2 | 24 | 34 | 56 | 91 |

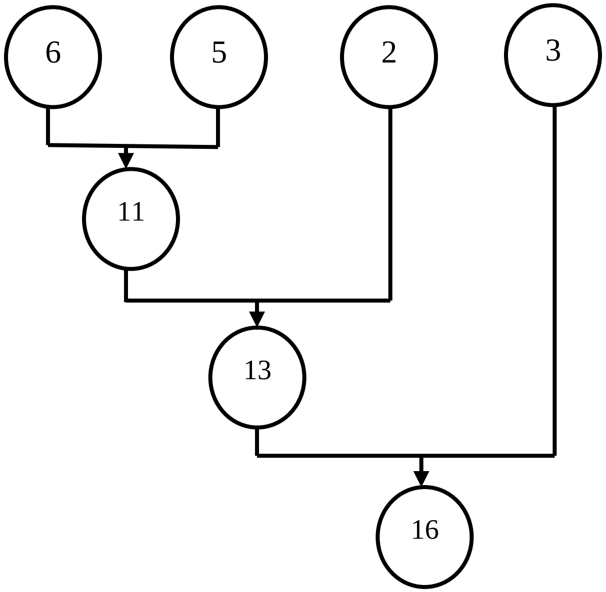
# Optimal Merge patterns

- Merge a set of sorted files of different length into a single sorted file.
- We need to find an optimal solution, where the resultant file will be generated in minimum time.
- If the number of sorted files are given, there are many ways to merge them into a single sorted file. This merge can be performed pair wise. Hence, this type of merging is called as **2-way merge patterns**.
- As, different pairings require different amounts of time, in this strategy we want to determine an optimal way of merging many files together. At each step, two shortest sequences are merged.
- To merge a **m-record file** and a **n-record file** requires possibly **m + n** record moves
- Merge the two smallest files together at each step.
- Two-way merge patterns can be represented by binary merge trees.
- Initially, each element is considered as a single node binary tree.

# Optimal Merge patterns

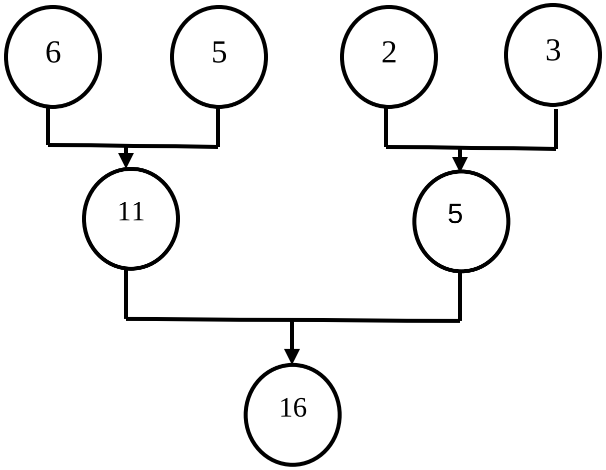
| File/list | A | B | C | D |
|---|---|---|---|---|
| sizes | 6 | 5 | 2 | 3 |

(a)



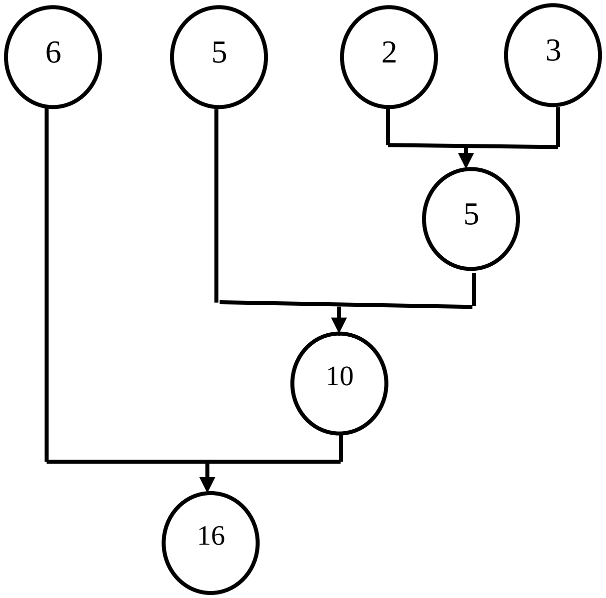
11+13+16 = 40

(b)



11+5+16 = 32

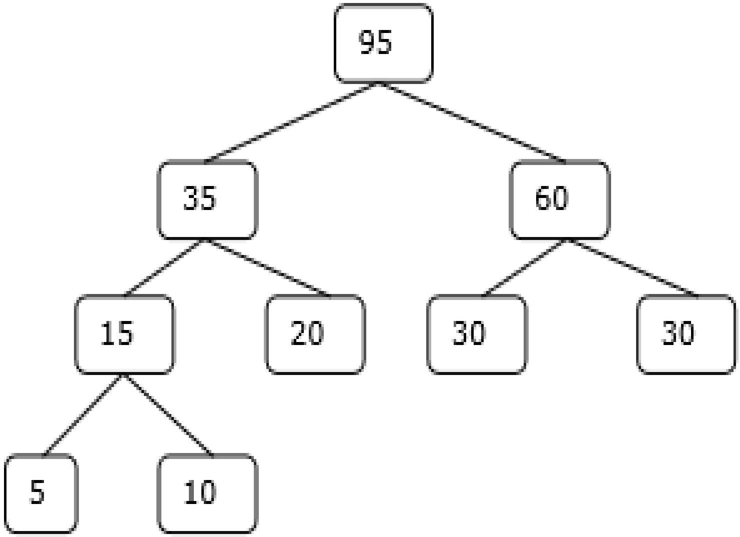
(c)



5+10+16 = 31

# Optimal Merge patterns

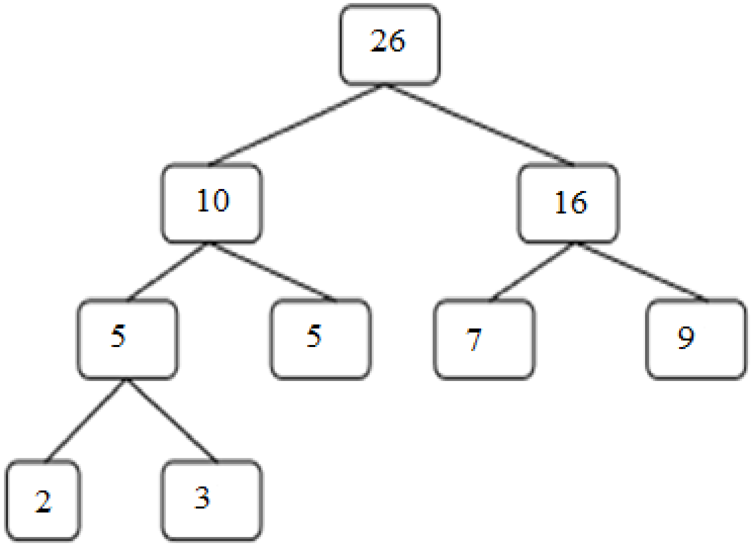
| lists | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|---|
| sizes | 20 | 30 | 10 | 5 | 30 |



15+35+95+60 = 205

$\Sigma d_i x_i$ = 3x5 + 3x10 +2x20 +2x30 + 2x30
= 205

| lists | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|---|
| sizes | 2 | 3 | 5 | 7 | 9 |



5+10+16+26 = 57

$\Sigma d_i x_i$ = 3x2 + 3x3 +2x5 +2x7 + 2x9
= 57

# Optimal Merge patterns

- The algorithm has as input a list *list* of n trees.
- Each node in a tree has three fields, lchild, rchild and weight.
- Initially, each tree in list has exactly one node and has lchild and rchild fields zero whereas weight is the length of one of the n files to be merged.

```
Algorithm Tree(n)
{
   for i = 1 to n-1 do
  {
     pt = new treenode;
     pt→lchild = Least(list);
     pt→rchild = Least(list);
     pt→weight = pt→lchild→weight + pt→lchild→weight;
     insert(list,pt);
  }
 return Least(list);
}
```

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

# Optimal Merge patterns

**Function Tree uses two functions: Least(list) and Insert(list, t).**

- **Least(list)** finds a tree in list whose root has least weight and returns a pointer to the tree. This tree is removed from list.
- **Insert(list, t)** inserts the tree with root t into list.

# Single-source shortest path

- Given a edge-weighted graph $G = (V, E)$ and a vertex $v \in V$, find the shortest weighted path from v to every other vertex in V.
- Dijkstra's Algorithm is a greedy algorithm for solving the single-source shortest-path problem on an edge-weighted graph in which all the weights are non-negative.
- It finds the shortest paths from some initial vertex, say v, to all the other vertices one-by-one.
- The paths are discovered in the order of their weighted lengths, starting with the shortest, and proceeding to the longest.
- For each vertex v, Dijkstra's algorithm keeps track of three pieces of information, $k_v$, $d_v$ and $p_v$.
- The Boolean valued flag $k_v$ indicates that the shortest path to vertex v. Initially, $k_v$ = false for all $v \in V$.
- The quantity $d_v$ is the length of the shortest known path from $v_0$ to v. When the algorithm begins, no shortest paths are known. The distance $d_v$, is a tentative distance.

# Single-source shortest path

- During the course of the algorithm candidate paths are examined and the tentative distances are modified.
- Initially $d_v = \infty$ for all $v \in V$ such that $v \neq v_0$, while $d_{v0} = 0$.
- The predecessor of the vertex v on the shortest path from $v_0$ to v is $p_v$. Initially, $p_v$ is unknown for all $v \in V$.
- The following steps are performed in each pass:
    1. From the set of vertices for with $k_v$ = false, select the vertex v having the smallest tentative distance $d_v$.
    2. Set $k_v \leftarrow$ true.
    3. For each vertex w adjacent to v for which $k_v \neq$ true, test whether the tentative distance $d_v$ is greater than $d_v + C(v,w)$. If it is, set $d_w \leftarrow d_v + C(v,w)$ and set $p_w \leftarrow v$.
- In each pass exactly one vertex has its $k_v$ set to true. The algorithm terminates after |V| passes are completed at which time all the shortest paths are known.

# Single-source shortest path

**Initially:**

$S = \{1\}$; $D[2] = 10$; $D[3] = \infty$; $D[4] = 30$; $D[5] = 100$

**Iteration 1**

Select w = 2, so that S = {1, 2}

$D[3] = \min(\infty, D[2] + C[2, 3]) = 60$

$D[4] = \min(30, D[2] + C[2, 4]) = 30$

$D[5] = \min(100, D[2] + C[2, 5]) = 100$

**Iteration 2**

Select w = 4, so that S = {1, 2, 4}

$D[3] = \min(60, D[4] + C[4, 3]) = 50$

$D[5] = \min(100, D[4] + C[4, 5]) = 90$

**Iteration 3**

Select w = 3, so that S = {1, 2, 4, 3}

$D[5] = \min(90, D[3] + C[3, 5]) = 60$

**Iteration 4**

Select w = 5, so that S = {1, 2, 4, 3, 5}

$D[2] = 10$; $D[3] = 50$; $D[4] = 30$; $D[5] = 60$