Unit - III

Greedy Method

- Knapsack problem
- Minimum spanning Trees
- Single source shortest path
- Job sequencing with deadlines
- Optimal storage on Tapes
- Optimal merge Patterns

Dynamic programming Method

- All pairs shortest paths
- Optimal binary search trees
- 0/1 Knapsack problem
- Reliability Design
- Travelling Salesperson problem

SUMRANA SIDDIQUI
Asst. Professor
CSE, DCET

The Greedy Method – General Method

The 'greedy method' is the most straightforward design technique. Most of these problems have 'n' inputs and require to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a 'feasible solution'. A feasible solution that either maximizes or minimizes a given 'objective function' is to be found and this solution is called an 'optimal function'.

The 'Greedy Method' suggests that an algorithm can be devised that works in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is an optimal solution. 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 will result in an infeasible solution, then this input is not added to the partial solution. Otherwise, it is added. The selection procedure itself is based on some

SUMRANA SIDDIQUI
Asst. Professor
CSE, DCET

optimization measure. This measure may be the objective function. Several different optimization measures may be plausible for a given problem. Most of these, however, will result in algorithms that generate suboptimal solutions. This version of the greedy technique is called the "subset paradigm".

```
1 Algorithm Greedy (a, n)
2 // a[1:n] contains the n inputs.
3 {
4    solution := ∅ ; // Initialize the solution.
5    for i := 1 to n do
6    {
7        x := Select (a);
8        if Feasible (solution, x) then
9            Solution := Union (solution, x);
10   }
11   return solution;
12 }
```

⊙ The function "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.

- The function 'Union' combines 'x' with the solution and updates the objective function.
- The function 'Greedy' describes the essential way that a greedy algorithm will look once a particular problem is chosen and the functions Select, Feasible and Union are properly implemented.

For the problems that do not call for the selection of an optimal subset, in the greedy method the decisions are made by considering the inputs in some order. Each decision is made using an optimization criterion that can be computed using decisions already made. This version of the greedy method is called the "ordering paradigm".

Knapsack Problem

The greedy method is applied to solve the knapsack problem:↓

There are 'n' given objects and a knapsack or bag. Object 'i' has a weight 'w_i' and the knapsack has a capacity 'm'. 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.

The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is 'm', the total weight required of all chosen objects is to be at most 'm'.

Formally, the problem can be stated as

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

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

and $0 \le x_i \le 1$, $1 \le i \le n$

The profits and weights are positive numbers.

A feasible solution is any set $(x_1, \ldots, x_n)$ satisfying the following:

$$\sum_{1 \le i \le n} w_i x_i \le m \quad \text{and} \quad 0 \le x_i \le 1, \; 1 \le i \le n$$

An optimal solution is a feasible solution for which $\sum_{1 \le i \le n} p_i x_i$ is maximized.

The knapsack problem calls for selecting a subset of the objects and hence fits the subset paradigm. In addition to selecting a subset, the knapsack problem also involves the selection of an 'x_i' for each object. Several simple greedy strategies to obtain feasible solutions are:

- Filling the knapsack by including the object with largest profit. If an object under consideration doesn't fit, then a fraction of it is included to fill the knapsack.
- Filling the knapsack with objects in order of non-decreasing (increasing) weights 'w_i', thus trying to be greedy with capacity

and using it up as slowly as possible.

- Filling the knapsack by objects which have the maximum profit per unit of capacity used. This means the objects are considered in order of the ratio p_i/w_i thus acheiving a balance between the rate at which profit increases and the rate at which capacity is used.

```
1 Algorithm GreedyKnapsack (m, n)
2 // p[1:n] and w[1:n] contains the profits and
3 //weights respectively of the n objects ordered such
4 // that p[i]/w[i] ≥ p[i+1]/w[i+1]. m is the
5 // knapsack size and x[1:n] is the solution vector
6 {
7     for i:=1 to n do x[i]=0.0; // Initialize x.
8     U:=m;
9     for i:=1 to n do
10    {
11        if (w[i] >U) then break;
12        x[i]:=1.0 ; U=U-w[i];
13    }
14    if (i≤n) then x[i]:= U/w[i];
15 }
```

On applying the Greedy Method to the solution of the knapsack problem, there are three different measures to optimize when determining which object to include next. These measures are total profit, capacity used, and the ratio of accumulated profit to capacity used. Once an optimization measure has been chosen, the greedy method suggests choosing objects for inclusion into the solution in such a way that each choice optimizes the measure at that time.

→ The greedy-based algorithms using the first two measures do not guarantee optimal solutions for the knapsack problem. A greedy algorithm using strategy 3 (ratio of profit to capacity) always obtains an optimal solution and is proved using the following technique:

c

Compare the greedy solution with any optimal solution. If the two solutions differ, then find the first x_i at which they differ. Next, it is shown how to make the x_i in the optimal solution equal to that in the greedy solution without any loss in total value. Repeated use of this transformation shows that the greedy solution is optimal.'

SUMRANA SIDDIQUI
Asst. Professor
CSE, DCET

Example1: Consider the following instance of Knapsack problem: $n=3$, $m=20$, $(p_1, p_2, p_3)=(25, 24, 15)$ $(w_1, w_2, w_3)=(18, 15, 10)$. Find the optimal solution for : (i) Maximum profit
(ii) Minimum weight
(iii) Maximum profit per unit weight

Solution :

Case(i) : Place an item in the bag with maximum profit $x_1 = 1$ & $w_1 = 18$.

$m = 20$ ∴ Space left after adding x_1 $= 2 \ (20-18=2)$

Next item with highest profit is x_2

x_2 = left out space / weight of item

$= 2/15 \quad (\because w_2 = 15)$

When x_2 is placed no space is left in bag.

So $x_3 = 0$ .i.e. 3rd item cannot be placed.

$$\sum_{1 \le i \le n} p_i x_i = p_1 x_1 + p_2 x_2 + p_3 x_3$$

$$= 25 \times 1 + 24 \times 2/15 + 15 \times 0$$

$$= 25 + 3.2 + 0$$

$$= \underline{28.2}$$

Case (ii) : Place an item in the bag with minimum weight

$x_3 = 1 \quad (\because w_3 = 10)$

∴ Space left after adding $x_3 = 10$ ($\because 20-10=10$)

x_2 = left out space / weight of item

$= 10/15 = 2/3$

No space left for x_1. $\therefore x_1 = 0$

$$\sum p_i x_i = p_1x_1 + p_2x_2 + p_3x_3$$
$$= 25 \times 0 + 24 \times 2/3 + 15 \times 1$$
$$= 0 + 16 + 15$$
$$= \underline{\underline{31}}$$

Case (iii) : Place an item in the bag whose profit per unit weight ratio is maximum.

$$\frac{P_1}{w_1} = \frac{25}{18} = 1.4 \quad \frac{P_2}{w_2} = \frac{24}{15} = 1.6 \quad \frac{P_3}{w_3} = \frac{15}{10} = 1.5$$

$\frac{P_2}{w_2} = 1.6$ is maximum so $x_2 = 1$.

Space left in bag $20 - 15 = 5$.

Next item to be place is $\frac{P_3}{w_3} = 1.5$ i.e. x_3

x_3 = left out space / weight of item

$= 5/10 = 1/2 = 0.5$

No space left for x_1 $\therefore x_1 = 0$.

$$\sum p_i x_i = p_1x_1 + p_2x_2 + p_3x_3$$
$$= 0 \times 25 + 24 \times 1 + 15 \times 0.5$$
$$= 0 + 24 + 7.5$$
$$= \underline{\underline{31.5}} \longrightarrow \boxed{\text{Optimal}}$$

SUMRANA SIDDIQUI
Asst. Professor
CSE, DCET

Example 2: Consider the following instance of Knapsack problem : $n = 7$, $m = 15$

$(p_1, p_2, p_3, p_4, p_5, p_6, p_7) = (10, 5, 15, 7, 6, 18, 3)$

$(w_1, w_2, w_3, w_4, w_5, w_6, w_7) = (2, 3, 5, 7, 1, 4, 1)$

Find the optimal solution for,

(i) Maximum profit

(ii) Minimum weight

(iii) Maximum profit per unit weight

Solution :

Case(i) : Maximum profit

$x_6 = 1 \qquad w_6 = 4$

Space left $(15 - 4 = 11) = 11$

$x_3 = 1 \qquad w_3 = 5$

Space left $(11 - 5 = 6) = 6$

$x_1 = 1 \qquad w_1 = 2$

Space left $(6 - 2 = 4) = 4$

$x_4 = 4/7$ = left out space / weight of item

No Space left $\therefore x_2 = x_5 = x_7 = 0$.

$$\sum p_i x_i = p_1x_1 + p_2x_2 + p_3x_3 + p_4x_4 + p_5x_5 + p_6x_6 + p_7x_7$$

$$= 10 \times 1 + 5 \times 0 + 15 \times 1 + 7 \times 4/7 + 6 \times 0 + 18 \times 1 + 3 \times 0$$

$$= 10 + 0 + 15 + 4 + 0 + 18 + 0$$

$$= 47$$

Case (ii) : Minimum weight

$x_5 = 1$ ($\because w_5 = 1$)

Space left after adding x_5 ($15 - 1 = 14$)

$x_7 = 1$ ($\because w_7 = 1$)

Space left ($14 - 1 = 13$) → after x_7

$x_1 = 1$ ($\because w_1 = 2$)

Space left ($13 - 2 = 11$) → after adding x_1

$x_2 = 1$ ($\because w_2 = 3$)

Space left ($11 - 3 = 8$) → after adding x_2

$x_6 = 1$: ($\because w_4 = 4$)

Space left ($8 - 4 = 4$) → after adding x_6

x_3 = left out space / weight of item

$= 4/5$ ($\because w_3 = 5$)

$x_4 = 0$ ($\because$ No space left)

$$\sum p_i x_i = p_1x_1 + p_2x_2 + p_3x_3 + p_4x_4 + p_5x_5 + p_6x_6 + p_7x_7$$

$$= 10 \times 1 + 5 \times 1 + 15 \times 4/5 + 7 \times 0 + 6 \times 1 + 18 \times 1 + 3 \times 1$$

$$= 10 + 5 + 12 + 0 + 6 + 18 + 3$$

$$= \underline{\underline{54}}$$

SUMRANA SIDDIQUI
Asst. Professor
CSE, DCET

Case (iii) : Maximum profit per unit weight

$\frac{P_1}{w_1} = \frac{10}{2} = 5$ $\frac{P_2}{w_2} = \frac{5}{3} = 1.6$ $\frac{P_3}{w_3} = \frac{15}{5} = 3$ $\frac{P_4}{w_4} = \frac{7}{7} = 1$

Example 2: Consider the following instance of Knapsack problem : $n = 7$, $m = 15$

$(p_1, p_2, p_3, p_4, p_5, p_6, p_7) = (10, 5, 15, 7, 6, 18, 3)$

$(w_1, w_2, w_3, w_4, w_5, w_6, w_7) = (2, 3, 5, 7, 1, 4, 1)$

Find the optimal solution for,

(i) Maximum profit

(ii) Minimum weight

(iii) Maximum profit per unit weight

Solution :

Case(i) : Maximum profit

$x_6 = 1$ $\quad w_6 = 4$

Space left $(15 - 4 = 11) = 11$

$x_3 = 1$ $\quad w_3 = 5$

Space left $(11 - 5 = 6) = 6$

$x_1 = 1$ $\quad w_1 = 2$

Space left $(6 - 2 = 4) = 4$

$x_4 = 4/7$ = left out space / weight of item

No Space left $\therefore x_2 = x_5 = x_7 = 0$.

$$\sum p_i x_i = p_1x_1 + p_2x_2 + p_3x_3 + p_4x_4 + p_5x_5 + p_6x_6 + p_7x_7$$

$$= 10 \times 1 + 5 \times 0 + 15 \times 1 + 7 \times 4/7 + 6 \times 0 + 18 \times 1 + 3 \times 0$$

$$= 10 + 0 + 15 + 4 + 0 + 18 + 0$$

$$= 47$$

Case (ii) : Minimum weight

$x_5 = 1$ ($\because w_5 = 1$)

Space left after adding x_5 ($15 - 1 = 14$)

$x_7 = 1$ ($\because w_7 = 1$)

Space left ($14 - 1 = 13$) → after x_7

$x_1 = 1$ ($\because w_1 = 2$)

Space left ($13 - 2 = 11$) → after adding x_1

$x_2 = 1$ ($\because w_2 = 3$)

Space left ($11 - 3 = 8$) → after adding x_2

$x_6 = 1$ ($\because w_4 = 4$)

Space left ($8 - 4 = 4$) → after adding x_6

x_3 = left out space / weight of item

$= 4/5$ ($\because w_3 = 5$)

$x_4 = 0$ ($\because$ No space left)

$$\sum p_i x_i = p_1x_1 + p_2x_2 + p_3x_3 + p_4x_4 + p_5x_5 + p_6x_6 + p_7x_7$$

$$= 10 \times 1 + 5 \times 1 + 15 \times 4/5 + 7 \times 0 + 6 \times 1 + 18 \times 1 + 3 \times 1$$

$$= 10 + 5 + 12 + 0 + 6 + 18 + 3$$

$$= \underline{\underline{54}}$$

SUMRANA SIDDIQUI
Asst. Professor
CSE, DCET

Case (iii) : Maximum profit per unit weight

$\frac{P_1}{w_1} = \frac{10}{2} = 5$ $\frac{P_2}{w_2} = \frac{5}{3} = 1.6$ $\frac{P_3}{w_3} = \frac{15}{5} = 3$ $\frac{P_4}{w_4} = \frac{7}{7} = 1$

$\frac{P_5}{W_5} = \frac{6}{1} = 6$ $\frac{P_6}{W_6} = \frac{18}{4} = 4.5$ $\frac{P_7}{W_7} = \frac{3}{1} = 3$

$\frac{P_5}{W_5}$ is maximum $\therefore x_5 = 1$

Space left after adding x_5 $(15 - 1 = 14)$

$\frac{P_1}{W_1}$ is next $\therefore x_1 = 1$

Space left after adding x_1 $(14 - 2 = 12)$

$\frac{P_6}{W_6}$ is next $\therefore x_6 = 1$

Space left after adding x_6 $(12 - 4 = 8)$

$\frac{P_7}{W_7}$ is next $\therefore x_7 = 1$

Space left after adding x_7 $(8 - 1 = 7)$

$\frac{P_3}{W_3}$ is next $\therefore x_3 = 1$

Space left after adding x_3 $(7 - 5 = 2)$

$\frac{P_2}{W_2}$ is next $\therefore x_2 =$ Space left / weight of item

$= 2/3$

No space left $\therefore x_4 = 0$

$$\sum p_i x_i = p_1x_1 + p_2x_2 + p_3x_3 + p_4x_4 + p_5x_5 + p_6x_6 + p_7x_7$$

$$= 10 \times 1 + 5 \times 2/3 + 15 \times 1 + 7 \times 0 + 6 \times 1 + 18 \times 1 + 3 \times 1$$

$$= 10 + 3.3 + 15 + 0 + 6 + 18 + 3$$

$$= \underline{55.3} \longrightarrow \boxed{\text{Optimal}}$$

Job Sequencing with Deadlines

On a single processor the arrangement of jobs with deadline constraints is referred as 'job sequencing with deadlines.'

Given a set of 'n' jobs, each job 'i' has an integer deadline $d_i \geq 0$ and a profit $p_i > 0$. For any job i the profit p_i is earned iff the job is completed by its deadline. To complete a job, one has 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, or $\sum_{i \in J} p_i$.

An optimal solution is a feasible solution with maximum value.

Since the problem involves the identification of a subset, it fits the subset paradigm.

To formulate a greedy algorithm to obtain an optimal solution, it is necessary to formulate an optimization measure to determine how the next job is chosen.

SUMRANA SIDDIQUI
Asst. Professor
CSE, DCET

The objective function $\sum_{i \in J} p_i$ is chosen as an optimization measure. Using this measure, the next job to be included is the one that increases $\sum_{i \in J} p_i$ the most, subject to the constraint that the resulting J is a feasible solution. This requires the jobs to be considered in non-increasing order of the p_i's.

Initially $J = \phi$ and $\sum_{i \in J} p_i = 0$.

Next, the job that has the largest profit is added. It forms feasible solution. The next job, to join the set J, has to satisfy the following conditions.

(1) The job should have maximum profit so as to form the optimal solution.

(2) The job to be included should be completed by its deadline to form a feasible solution.

A high-level description of the greedy algorithm below constructs an optimal set J of jobs that can be processed by their due times.

High level description of Job Sequencing

```
1 Algorithm GreedyJob(d, J, n)
2 // J is a set of jobs that can be completed by deadlines.
3 {
4     J := {1};
5     for i := 2 to n do
```

```
6       {
7           if (all jobs in J ∪ {i} can be completed
8               by their deadlines) then J := J ∪ {i}.
9       }
10  }
```

The algorithm below assumes that the jobs are already sorted such that $p_1 \geq p_2 \geq \ldots \geq p_n$. It assumes that $n \geq 1$ and the deadline $d[i]$ of job i is atleast 1. No job with $d[i] < 1$ can ever be finished by its deadline.

Greedy algorithm for Job Sequencing with deadlines

```
1 Algorithm JS(d, j, n)
2 // d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1. The jobs
3 // are ordered such that p[1] ≥ p[2] ≥ ... ≥ p[n]. J[i]
4 // is the ith job in the optimal solution, 1 ≤ i < k.
5 // Also, at termination d[J[i]] ≤ d[J[i+1]], 1 ≤ i < k.
6 {
7     d[0] := J[0] := 0; // Initialize.
8     J[1] := 1; // Include job 1.
9     k := 1;
10    for i := 2 to n do
11    {
12        // Consider jobs in descending order of p[i]. Find
13        // position for i and check feasibility of insertion.
14        r := k;
```

SUMRANA SIDDIQUI
Asst. Professor
CSE, DCET

```
15      while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r) do r := r-1;
16      if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then
17      {
18          // Insert i into J[].
19          for q := k to (r+1) step -1 do J[q+1] := J[q];
20          J[r+1] := i; k := k+1;
21      }
22  }
23  return k;
24 }
```

The above algorithm has two possible parameters in terms of which its complexity can be measured i.e. 'n' the number of jobs, and s, the number of jobs included in the solution J.

The algorithm takes $O(n^2)$ computing time.

The computing time can be reduced from $O(n^2)$ to nearly $O(n)$ by using the disjoint set union and find algorithms.

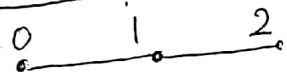
Example1: Solve the following problem of job sequencing with the deadlines specified using greedy method. $n = 4$, $(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$ $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$.

Solution: Maximum deadline = 2 units, maximum of two jobs form the feasible solution.

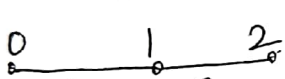
Arranging the jobs in decreasing order of their profits

$(P_1, P_4, P_3, P_2) = (100, 27, 15, 10)$

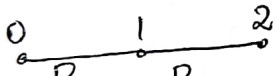
$(d_1, d_4, d_3, d_2) = (2, 1, 2, 1)$



Profit : 100 + 27 = 127
Sequence : {4, 1}

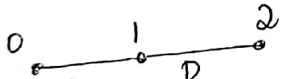


P : 15 + 100 = 115
S : {3, 1}

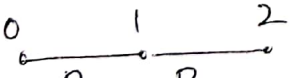


P : 100 + 10 = 110
S : {2, 1}

P : 27 + 15 = 42
S : {4, 3}



P : 27 + 10 = 37
S : {4, 2}



P : 15 + 10 = 25
S : {2, 3}

Maximum Profit = 127 with Jobs P_1 and P_4

∴ jobs $\{P_4, P_1\}$ will give the optimal solution.

SUMRANA SIDDIQUI
Asst. Professor
CSE, DCET

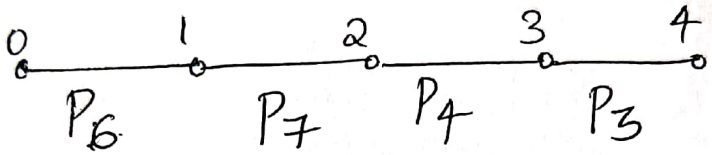
Example 2: Solve the following problem with greedy method

$n = 7$, $(P_1, P_2, P_3, P_4, P_5, P_6, P_7) = (3, 5, 20, 18, 1, 6, 30)$

and $(d_1, d_2, d_3, d_4, d_5, d_6, d_7) = \{1, 3, 4, 3, 2, 1, 2\}$.

Solution: $(P_7, P_3, P_4, P_6, P_2, P_1, P_5) = (30, 20, 18, 6, 5, 3, 1)$

$(d_7, d_3, d_4, d_6, d_2, d_1, d_5) = (2, 4, 3, 1, 3, 1, 1)$.



P_1, P_5, P_2 cannot be done.

$\{P_6, P_7, P_4, P_3\}$ = Profit = 30 + 20 + 18 + 6

= 74

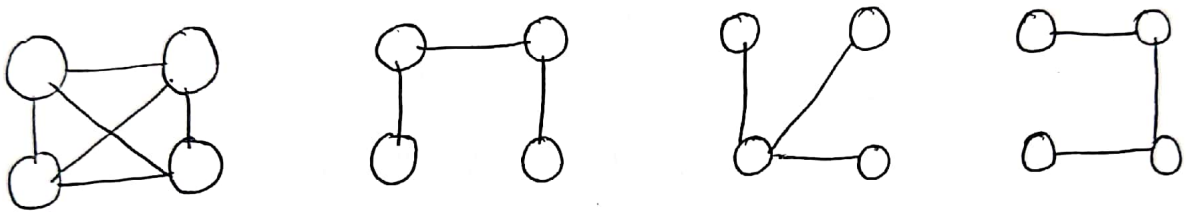
Sequence of processing = $\{6, 7, 4, 3\}$

The jobs P_6, P_7, P_4, P_3 will give the optimal profit.

Minimum - Cost Spanning Trees

Let $G = (V, E)$ be an undirected connected graph. A subgraph $t = (V, E')$ of G is a "spanning tree" iff t is a tree. i.e. a spanning tree is a subgraph of G, containing all the vertices of G. A Graph can have more than one spanning tree.

Example: A Graph with three of its spanning trees.



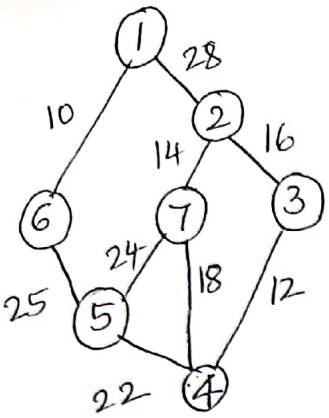
A spanning tree is a minimal subgraph G' of G such that $V(G') = V(G)$ and G'' is connected. A minimal subgraph is one with the fewest number of edges.

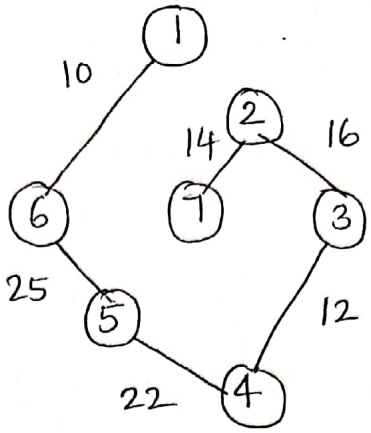
Any connected graph with "n" vertices must have atleast "n-1" edges and all connected graphs with "n-1" edges are trees.

The edges have weights assigned to them. A connected subgraph containing all the vertices such that the sum of the weights on the edges is minimum is called the "minimum cost spanning tree".

SUMRANA SIDDIQUI
Asst. Professor
CSE DCET

Example : A graph and its minimum cost spanning tree





Spanning trees have many applications :

- They can be used to obtain an independant set of circuit equations for an electric network.
- They represent possible communication links connecting two cities, where nodes are the cities and edges, the links.

Since the identification of a minimum-cost spanning tree involves the selection of a subset of the edges, this problem fits the subset paradigm.

If a graph consists of 'n' vertices then all the possible spanning trees are n^{n-2}. It is a time consuming process as 'n' value increases. Two standard algorithms are used for obtaining the minimum cost spanning trees.

→ Prim's Algorithm

A greedy method to obtain a minimum-cost spanning tree builds this tree edge by edge. The next edge to include is chosen according to some optimization criterion.

The simplest criterion is to choose an edge that results in a minimum increase in the sum of the costs of the edges so far included.

If A is the set of edges selected, then A forms a tree. Start with some vertex 'i'. Select a vertex 'j' such that the cost of edge (i,j) is minimum among the edges that satisfy condition that 'i' is a vertex already included in A and 'j' is a vertex not yet included in A. Include 'j' into the solution such that (i,j) is an edge of minimum cost spanning tree. At each step, an edge with smallest weight connecting a vertex in A is added to the tree. It adds only the edges that do not create a cycle.

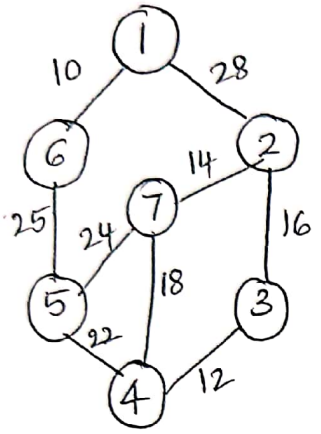
When the algorithm terminates, the edges in A form a minimum spanning tree. This strategy is a greedy strategy since the tree is augmented at each step with an edge that contributes the minimum amount possible to the tree weight.

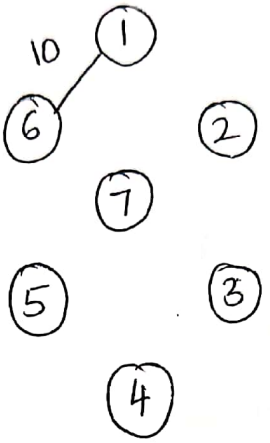
SUMRANA SIDDIQUI
Asst. Professor
CSE, DCET

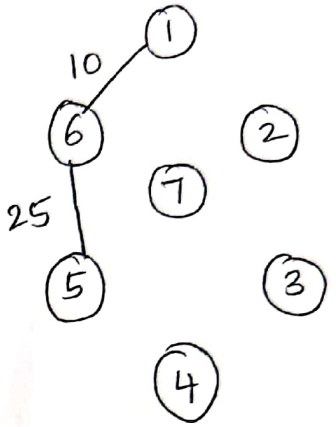
```
1 Algorithm Prim (E, cost, n, t)
2 // E is the set of edges in G. cost[1:n, 1:n] is the cost
3 // adjacency matrix of an 'n' vertex graph such that
4 // cost[i,j] is either a positive real number or ∞ if no
5 // edge (i,j) exists. A minimum spanning tree is computed and
6 // stored as a set of edges in array t. (t[i,1], [i,2]) is an edge
7 // in the minimum-cost spanning tree. The final cost is returned.
8 {
9    Let (k, l) be an edge of minimum cost in E;
10   mincost := cost[k, l];
11   t[1,1] := k; t[1,2] := l;
12   for i := 1 to n do // Initialize near.
13      if (cost[i,l] < cost[i,k]) then near[i] := l;
14      else near[i] := k;
15   near[k] := near[l] := 0;
16   for i := 2 to n-1 do
17   { // Find n-2 additional edges for t.
18     Let j be an index such that near[j] ≠ 0 and
19     cost[j, near[j]] is minimum;
20     t[i,1] := j; t[i,2] := near[j];
21     mincost := mincost + cost[j, near[j]];
22     near[j] := 0;
23     for k := 1 to n do // Update near[].
24       if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k,j])
25          then near[k] := j;
26   }
27   return mincost;
28 }
```

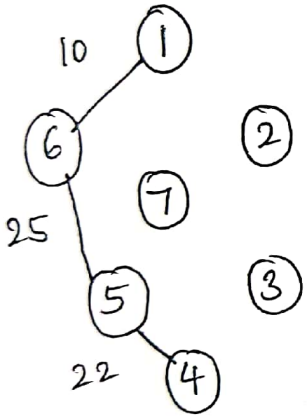
The time required by algorithm 'Prim' is $O(n^2)$, where 'n' is the number of vertices in the graph G.

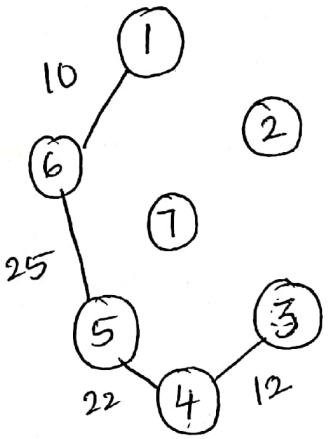
Example: Consider the graph below and find the minimum spanning tree using Prim's Algorithm.

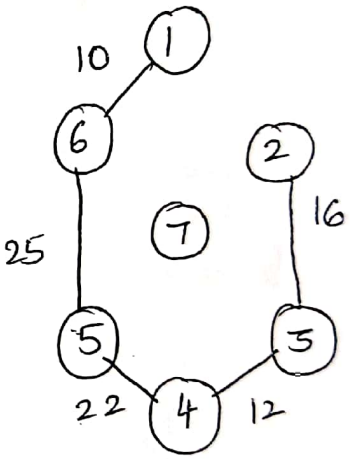








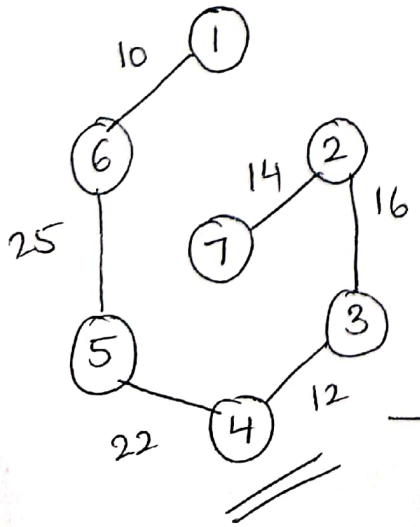




$S = \{1, 6, 5, 4, 3, 2, 7\}$

Total cost = 10 + 25 + 22 + 12 + 16 + 14

= 99



→ Minimum cost Spanning Tree using Prim's Algorithm

SUMRANA SIDDIQUI
Asst. Professor
CSE, DCET

→ Kruskal's Algorithm

In Kruskal's algorithm the optimization criteria chooses the edges of the graph in non decreasing order of cost. The set of edges 't' selected for the spanning tree should be such that it is possible to complete 't' into a tree. Thus 't' may not be a tree at all stages in the algorithm. It will only be a forest since the set of edges 't' can be completed into a tree iff there are no cycles in 't'.

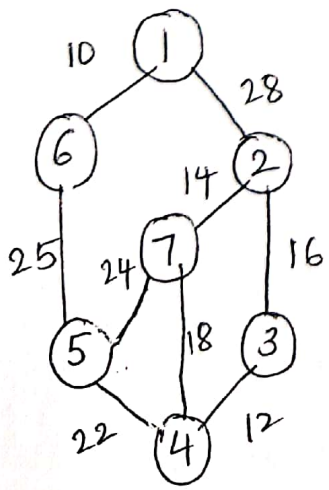
If G is a connected graph, include an edge with minimum cost. This is the first edge of spanning tree to be constructed. Include the next edge with minimum weight that do not form a cycle with the edges already included and so on.

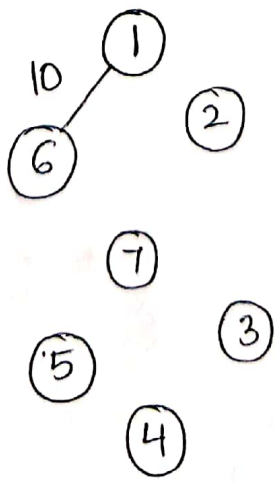
```
1 Algorithm Kruskal (E, cost, n, t)
2 // E is the set of edges in G. G has n vertices, cost[u,v] is the
3 // cost of edge (u,v). t is the set of edges in the minimum
4 // cost spanning tree. The final cost is returned.
5 {
6    Construct a heap out of the edge costs using Heapify;
7    for i := 1 to n do parent[i] := -1;
8    // Each vertex is in a different set.
9    i := 0; mincost := 0.0;
10   while ((i < n-1) and (heap not empty)) do
11   {
12      Delete a minimum cost edge (u,v) from the heap
13      and reheapify using Adjust;
```

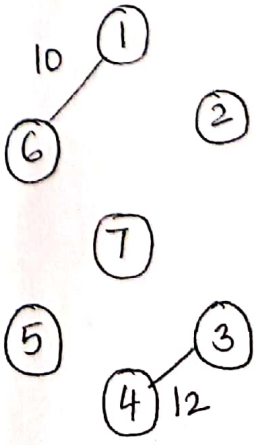
```
14      j := Find(u); k := Find(v);
15      if (j ≠ k) then
16      {
17          i := i + 1;
18          t[i,1] := u ; t[i,2] := v;
19          mincost := mincost + cost[u,v];
20          Union(j,k);
21      }
22  }
23  if (i ≠ n-1) then write ("No spanning tree");
24  else return mincost;
25 }
```

The computing time of algorithm 'Kruskal' is $O(E \log E)$, where E is the edge set of G.

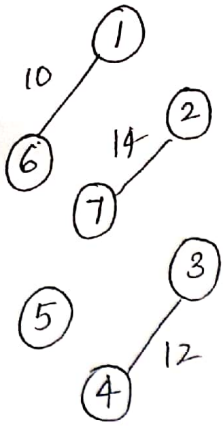
Example: Consider the graph below and find the minimum spanning tree using Kruskal's Algorithm.

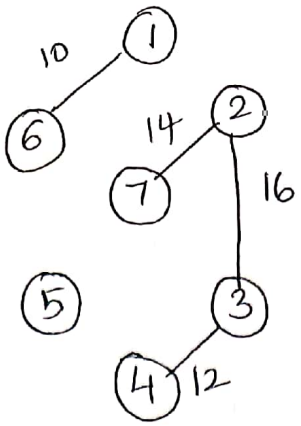


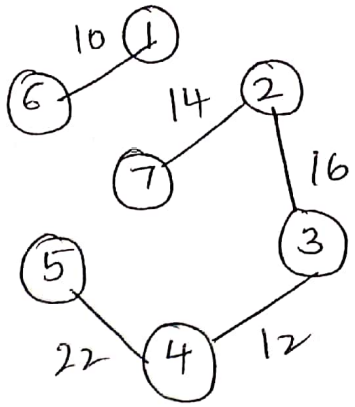


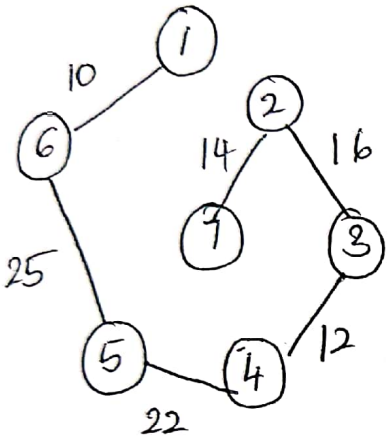


SUMRANA SIDDIQUI
Asst. Professor
CSE, DCET









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

$\{6,5\}\}$

Total cost = $10 + 12 + 14 + 16 + 22 + 25$

$= 99$

→ Minimum cost Spanning Tree using Kruskal's Algorithm.

Single-Source Shortest Paths (Dijkstra's Algorithm)

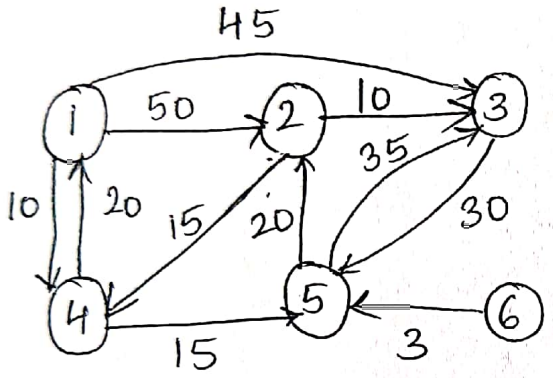
Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. The edges can be assigned weights which may be either the distance between the two cities connected by the edge or the average time to drive along that section of highway.

The Single Source Shortest Path are special cases of the path problem. The length of a path is defined to be the sum of the weights of the edges on that path. The starting vertex of the path is referred to as the 'source', and the last vertex the 'destination'. The graphs are digraphs to allow for one-way streets.

Given a directed graph $G = (V, E)$, with a weighting function 'cost' for the edges of G, and a source vertex 'v_0'. The Single-Source Shortest Path problem is to determine the shortest paths from 'v_0' to all the remaining vertices of G. It is assumed that all the weights are positive. The shortest path between 'v_0'

and some other node 'v' is an ordering among a subset of the edges. Hence, the Single-Source Shortest-Path problem fits the ordering paradigm.

Example: Consider the directed graph below along with the shortest paths from vertex 1 to all other vertices in non decreasing order of path length.



	Path	Length
1)	1,4	10
2)	1,4,5	25
3)	1,4,5,2	45
4)	1,3	45

To formulate a greedy-based algorithm to generate the shortest paths, a multistage solution to the problem and an optimization measure should be conceived. One possibility is to build the shortest paths one by one and use the sum of the lengths of all paths so far generated as an optimization measure. For the measure to be minimized, each individual path must be of minimum length.

The greedy way to generate the shortest paths from 'v_0' to the remaining vertices is to

generate these paths in non decreasing order of path length. First, a shortest path to the nearest vertex is generated. Then a shortest path to the second nearest vertex is generated, and so on. In order to generate the shortest paths in this order, determine

- ⊙ the next vertex to which a shortest path must be generated.
- ⊙ a shortest path to this vertex

Let 'S' denote the set of vertices to which the shortest paths have been generated. Let "dist[w]" be the length of the shortest path starting from "v_0", going through those vertices that are in 'S', and ending at "w". It is observed:

- If the next shortest path is to vertex 'u', then the path begins at "v_0", ends at "u", going through the vertices that are in 'S'.
- The destination of the next path generated must be that of vertex "u", which has the minimum distance, "dist[u]", among all vertices not in S.
- Having selected a vertex "u" and generated the shortest "v_0" to "u" path, vertex "u" becomes a member of S. The length of, the shortest path starting at "v_0", going through vertices in S, ending at vertex "w" may decrease, i.e., the value of "dist[w]" may change.

SUMRANA SIDDIQUI
Asst. Professor
DCET

The above observations leads to the algorithm for the Single Source Shortest Path problem. This algorithm is known as "Dijkstra's Algorithm". It determines only the lengths of the shortest paths from "v_0" to all other vertices in G.

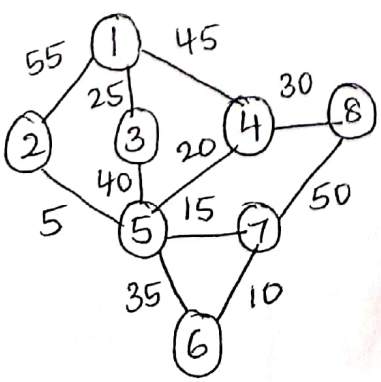
The algorithm assumes that the "n" vertices of G are numbered 1 to n. Set S is a bit array with S[i] = 0 if vertex i is not in S and S[i] = 1 if it is. The graph is represented by its cost adjacency matrix with cost[i,j]'s being the weight of the edge $\langle i,j \rangle$. The weight cost[i,j] is set to large number "∞" in case the edge $\langle i,j \rangle$ is not in E(G). For i=j, cost[i,j] can be set to any nonnegative number.

```
1 Algorithm ShortestPaths (v, cost, dist, n)
2 // dist[j], 1≤j≤n, is set to the length of the shortest
3 // path from vertex 'v' to vertex 'j' in a digraph G
4 // with 'n' vertices. dist[v] = 0. G is represented by
5 // its cost adjacency matrix cost[1:n, 1:n].
6 {
7     for i := 1 to n do
8     { // Initialize S.
9         S[i] := false ; dist[i] := cost[v,i];
10    }
11    S[v] := true ; dist[v] := 0.0 ; // Put v in S.
```

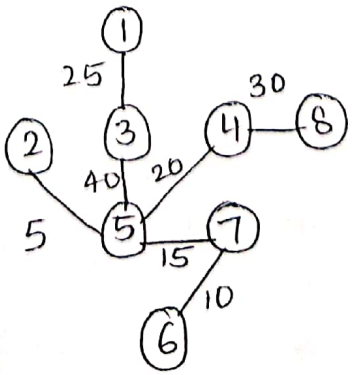
```
12      for num := 2 to n do
13      {
14          // Determine n-1 paths from v.
15          Choose u from among those vertices not
16          in S such that dist[u] is minimum;
17          S[u] := true ; // Put u in S.
18          for (each w adjacent to u with S[w] = false) do
19              // Update distances.
20              if (dist[w] > dist[u] + cost[u,w])) then
21                  dist[w] := dist[u] + cost[u,w];
22      }
23  }
```

The computing time taken by the algorithm on a graph with 'n' vertices is $O(n^2)$.

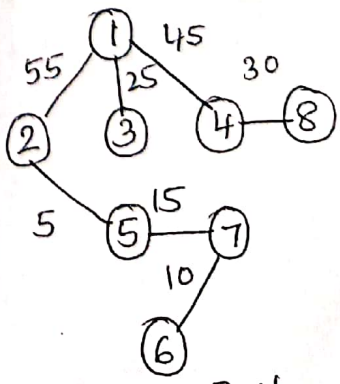
The edges on the shortest paths from a vertex 'v' to all remaining vertices in a connected undirected graph G form a spanning tree of G called a 'shortest-path spanning tree'. This tree may be different for different root vertices 'u'.



A Graph

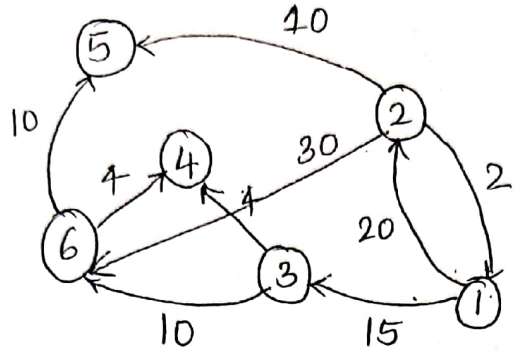


Minimum cost Spanning Tree



Shortest Path Spanning Tree

Example : Using shortest-path algorithm, obtain the shortest path from node 1 to all the remaining nodes in the graph below:



Set S	Vertex Selected	1	2	3	4	5	6
–	–	0	20	15	∞	∞	∞
{1}	3	0	20	15	19	∞	25
{1,3}	4	0	20	15	19	∞	25
{1,3,4}	2	0	20	15	19	30	25
{1,3,4,2}	6	0	20	15	19	30	25
{1,3,4,2,6}	5	0	20	15	19	30	25

Shortest paths from vertex 1 (increasing order)

$1 \to 3 = 15$

$1 \to 4 = 1 \to 3 \to 4 = 19$

$1 \to 2 = 20$

$1 \to 5 = 1 \to 2 \to 5 = 30$

$1 \to 6 = 1 \to 3 \to 6 = 25$

$1 \to 3 = 15$

$1 \to 4 = 19$

$1 \to 2 = 20$

$1 \to 6 = 25$

$1 \to 5 = 30$

Optimal Storage On Tapes

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_i', $1 \le i \le n$. All programs can be stored on the tape if and only if the sum of the lengths of the programs is at most 'L' i.e.

$$L = l_1 + l_2 + l_3 + l_i + \ldots + l_n$$

Assuming, 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 $I = i_1, i_2, \ldots, i_n$, the time t_j needed to retrieve program i_j is proportional to $\sum_{1 \le k \le j} l_{ik}$.

If all programs are retrieved equally, then the expected or 'mean retrieval time' (MRT) is

$$\frac{1}{n} \sum_{1 \le j \le n} t_j .$$

The Optimal Storage on Tape problem requires to find a permutation for the 'n' programs so that they are stored on the tape in the order that minimizes the MRT.

SUMRANA SIDDIQUI
Asst. Professor
CSE, DCET

This problem fits the ordering paradigm.

Minimizing the MRT is equivalent to minimizing

$$d(I) = \sum_{1 \le j \le n} \sum_{1 \le k \le j} l_{ik}.$$

A greedy approach to build the required permutation would choose the next program on the basis of some optimization measure. One possible measure would be the 'd' value of the permutation constructed. The next program to be stored on the tape would be one that minimizes the increase in 'd'. The increase in 'd' is minimized if the chosen program is the one with the least length from among the remaining programs.

The greedy method requires to store the programs in non decreasing order of their lengths. This ordering can be carried out in $O(n \log n)$ time using an efficient sorting algorithm.

Example: 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	
3, 1, 2	3 + (3+5) + (3+5+10) = 29	Minimum
2, 1, 3	10 + (10+5) + (10+5+3) = 43	
2, 3, 1	10 + (10+3) + (10+3+5) = 41	
3, 2, 1	3 + (3+10) + (3+10+5) = 34	

As the ordering 3, 1, 2 has the minimal d(I) value, i.e., 29, this ordering is said to be optimal ordering.

→ The tape storage problem can be extended to several tapes. If there are $m > 1$ tapes, $T_0, \ldots, T_{m-1}$, then the programs are to be distributed over these tapes. For each tape a storage permutation is to be provided as I_j. The 'total retrieval time' (TD) is $\sum_{0 \le j \le m-1} d(I_j)$.

The objective is to store the programs in such a way as to minimize TD.

The solution for one-tape case is to consider the programs in non-decreasing order of l_i's. The program considered is placed on the tape that results in minimum increase in TD. The least amount of tape used will be considered. If there is more than one tape with the above property, then the one with the smallest index can be used.

The general rule is that program 'i' is stored on tape $T_i \bmod m$ (where m is the number of tapes). On any given tape the programs are stored in increasing order of their lengths.

```
1 Algorithm Store (n, m)
2 // n is the number of programs and m the number of tapes
3 {
4     j := 0 ; // Next tape to store on.
5     for i := 1 to n do
6     {
7         write ("append program ", i,
8                "to permutation for tape ", j);
9         j := (j+1) mod m ;
10    }
11 }
```

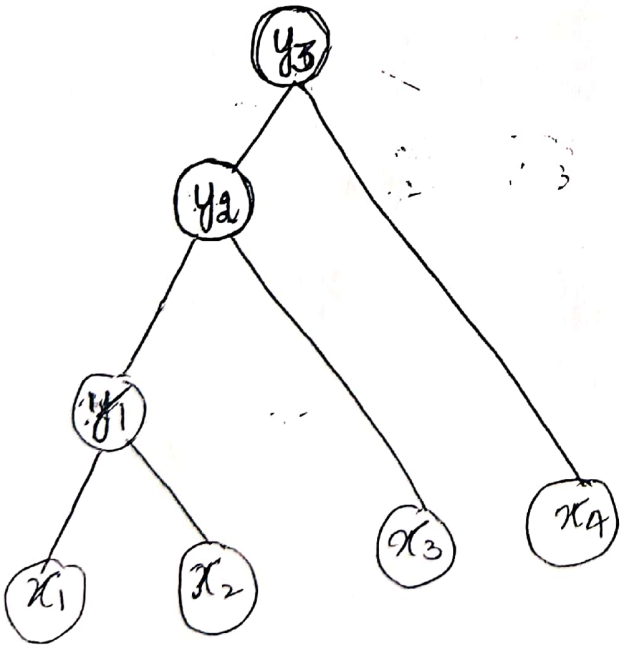
The computing time is $\underline{\underline{O(n)}}$ and does not need to know program lengths.

Optimal Merge Patterns

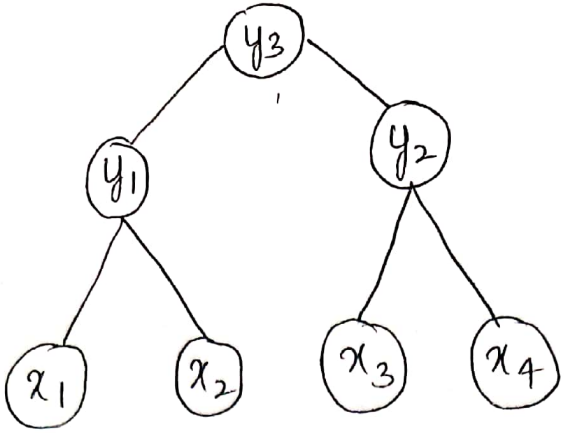
Optimal merge pattern is a method of determining merge patterns that takes minimum amount of compintation time.

When more than two sorted files are to be merged together, the merge can be accomplished by repeatedly merging sorted files in pairs.

If files x_1, x_2, x_3 and x_4 are to be merged, first merge x_1 and x_2 to get a file y_1. Then merge y_1 and x_3 to get y_2. Finally, merge y_2 and x_4 to get the desired sorted file.



Alternatively, first merge x_1 and x_2, so as to get y_1, then merge x_3 and x_4 and get y_2, and finally merge y_1 and y_2 to get the sorted file.



Given 'n' sorted files, there are many ways in which to pairwise merge them into a single sorted file. Different pairings require differing amounts of computing time.

The problem of optimal merge pattern determines an optimal way to pairwise merge 'n' sorted files. This problem calls for an ordering among the pairs to be merged, it fits the ordering paradigm.

Example: The files x_1, x_2 and x_3 are three sorted files of length 30, 20 and 10 records each.

Merge x_1 and x_2 = 50 record moves
Merge result with x_3 = 50 + 10 = 60 record moves
Total number of record moves required to merge the three files is 50 + 60 = 110.

(or)

Faster
- Merge x_2 and x_3 = 30 record moves
- Merge result with x_1 = 30 + 30 = 60 record moves
- Total number of record moves required to merge the three files is 30 + 60 = 90.

A greedy attempt to obtain an optimal merge pattern is:

Merging an 'n'-record file and an m-record file requires 'n+m' record moves; choose a selection criteria: at each step merge the two smallest size files together. This merge pattern is referred to as a "two-way merge pattern".

The two-way merge patterns can be represented by binary merge trees. In a binary merge tree:

- the leaf nodes are drawn as squares and represent the given files. They are called 'external nodes'.
- the remaining nodes are drawn as circles and are called 'internal nodes'.
- each internal node has exactly two children, and it represents the file obtained by merging the files represented by its two children.
- the number in each node is the length (i.e. the number of records) of the file represented by that node.

Example: The five files $(x_1, x_2, x_3, x_4, x_5)$ with sizes (20, 30, 10, 5, 30) would generate the following merge pattern:

Merge x_3 and x_4 to get $z_1 = (10+5) = 15$

Merge z_1 and x_1 to get $z_2 = (15+20)\ 35$

Merge z_2 and x_5 to get $z_3 = (30+30)\ 60$

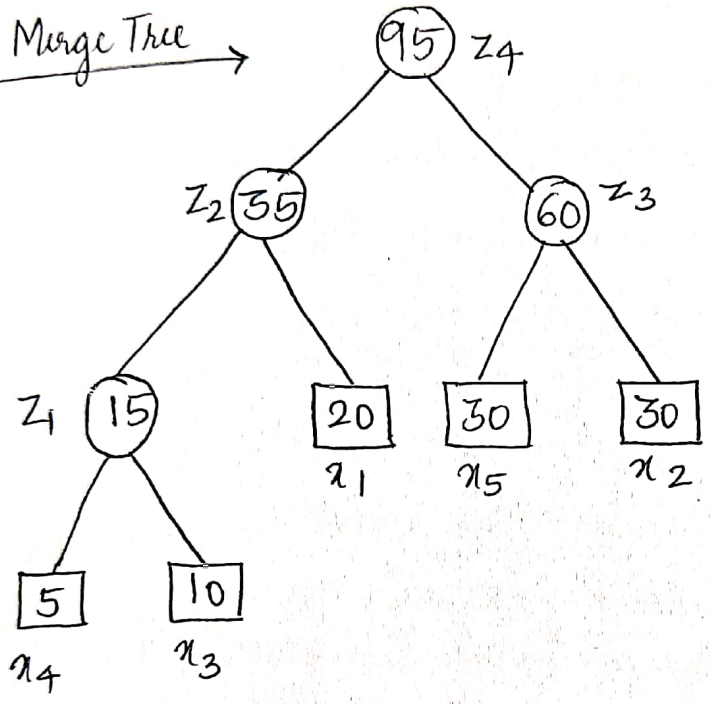
Merge z_2 and z_3 to get $z_4 = (35+60)\ 95$

Total number of record moves $= 15 + 35 + 60 + 95$

$= 205$

This is the optimal merge pattern.

Binary Merge Tree →



The external node x_4 is at a distance of 3 from the root node z_4 (A node at level 'i' is at a distance of $i-1$ from the root). Hence the records of file z_4 are moved three times, $x_4 \rightarrow z_1$, then $z_1 \rightarrow z_2$ and finally $z_2 \rightarrow z_4$.

If 'd_i' is the distance from the root to the external node for file 'x_i' and 'q_i', the length of 'x_i' is the total number of record moves for this binary merge tree is $\sum_{i=1}^{n} d_i q_i$. This sum is called the 'weighted external path length' of the tree.

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

```
1  Algorithm Tree (n)
2  // list is a global list of n single node.
3  // binary tree as described.
4  {
5     for i := 1 to n-1 do
6     {
7        pt := new treenode ; // Get a new tree node.
8        (pt → lchild) := Least (list); // Merge two trees with
9        (pt → rchild) := least (list); // smallest lengths.
10       (pt → weight) := ((pt → lchild) → weight)
11                      + ((pt → rchild) → weight) ;
12       Insert (list, pt) ;
13    }
14    return Least (list); // Tree left in list is the merge
                                                    tree.
15 }
```

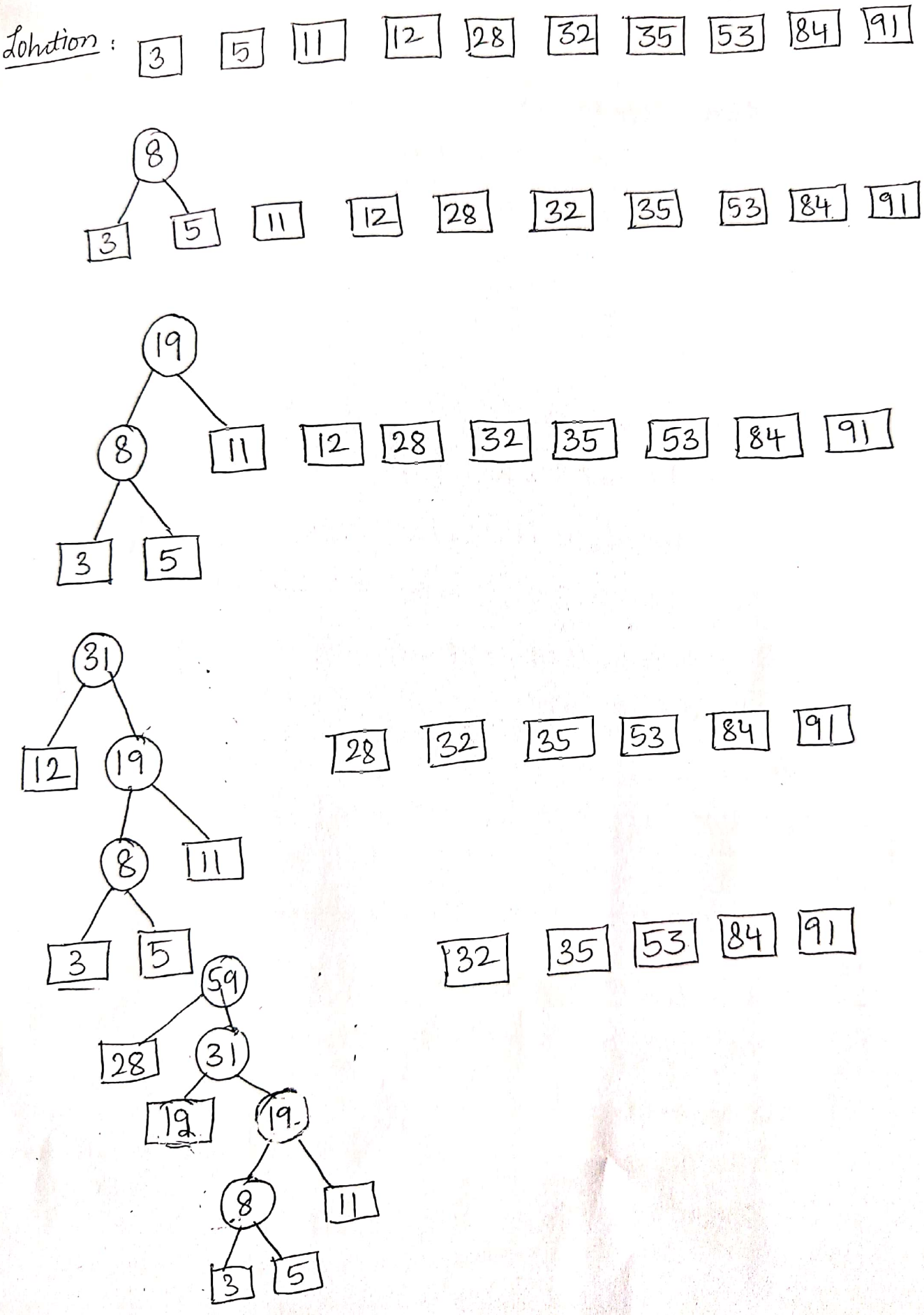
Analysis : If the arrangement of weight value in the roots is increasing order, the computing time is $O(n^2)$.

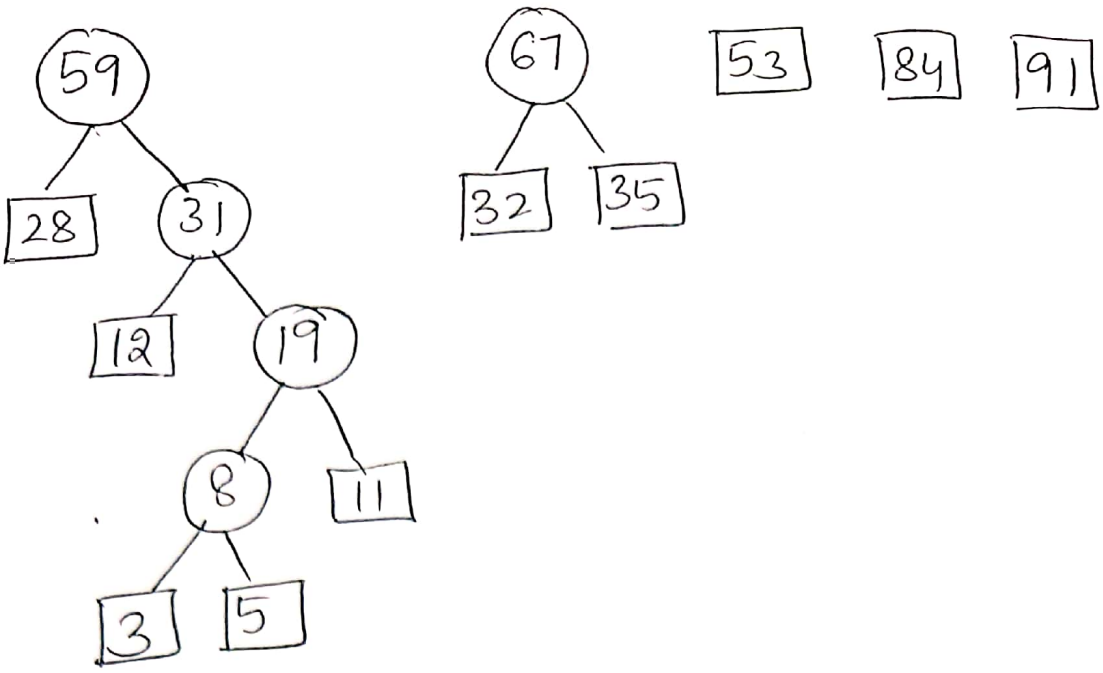
If the arrangement is as a minheap in which the root value is less than or equal to values of its children, the computing time is $O(n \log n)$.

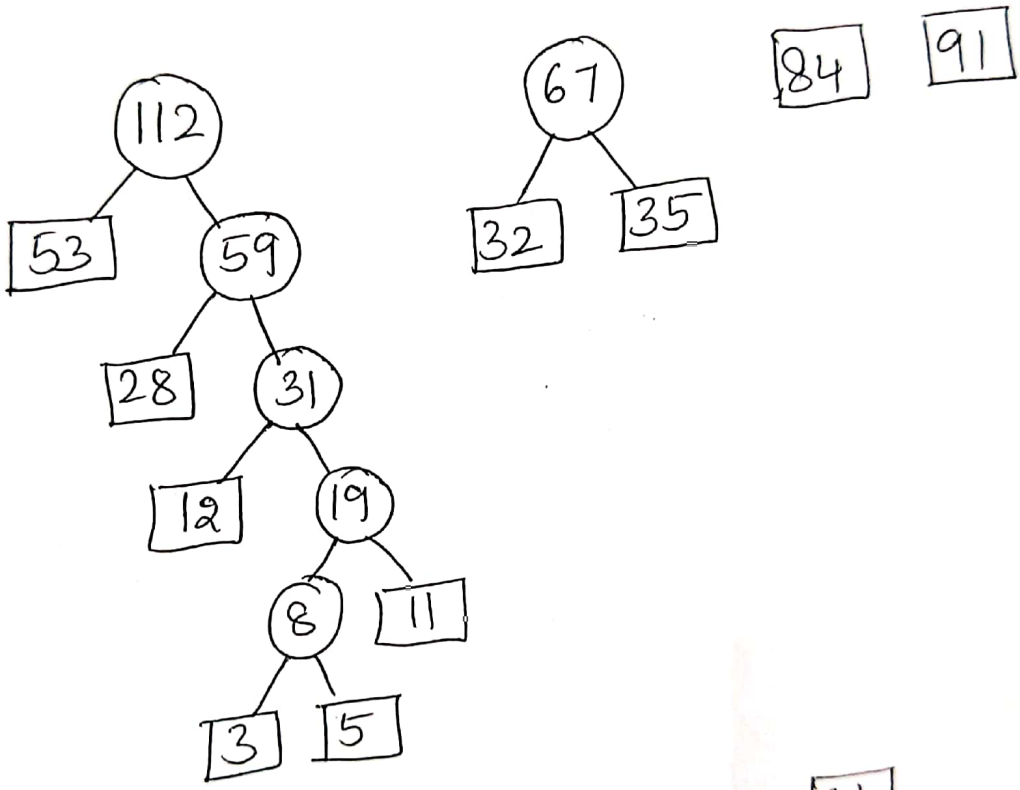
Example: Find an optimal binary merge pattern for ten files whose lengths are:

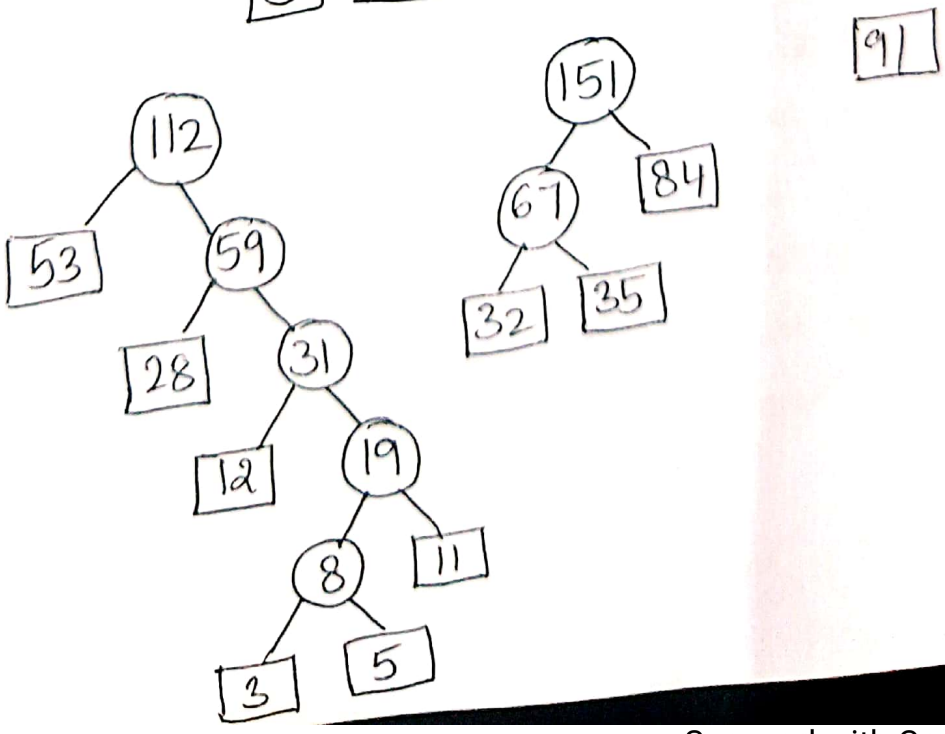
28 32 12 5 84 53 91 35 3 11

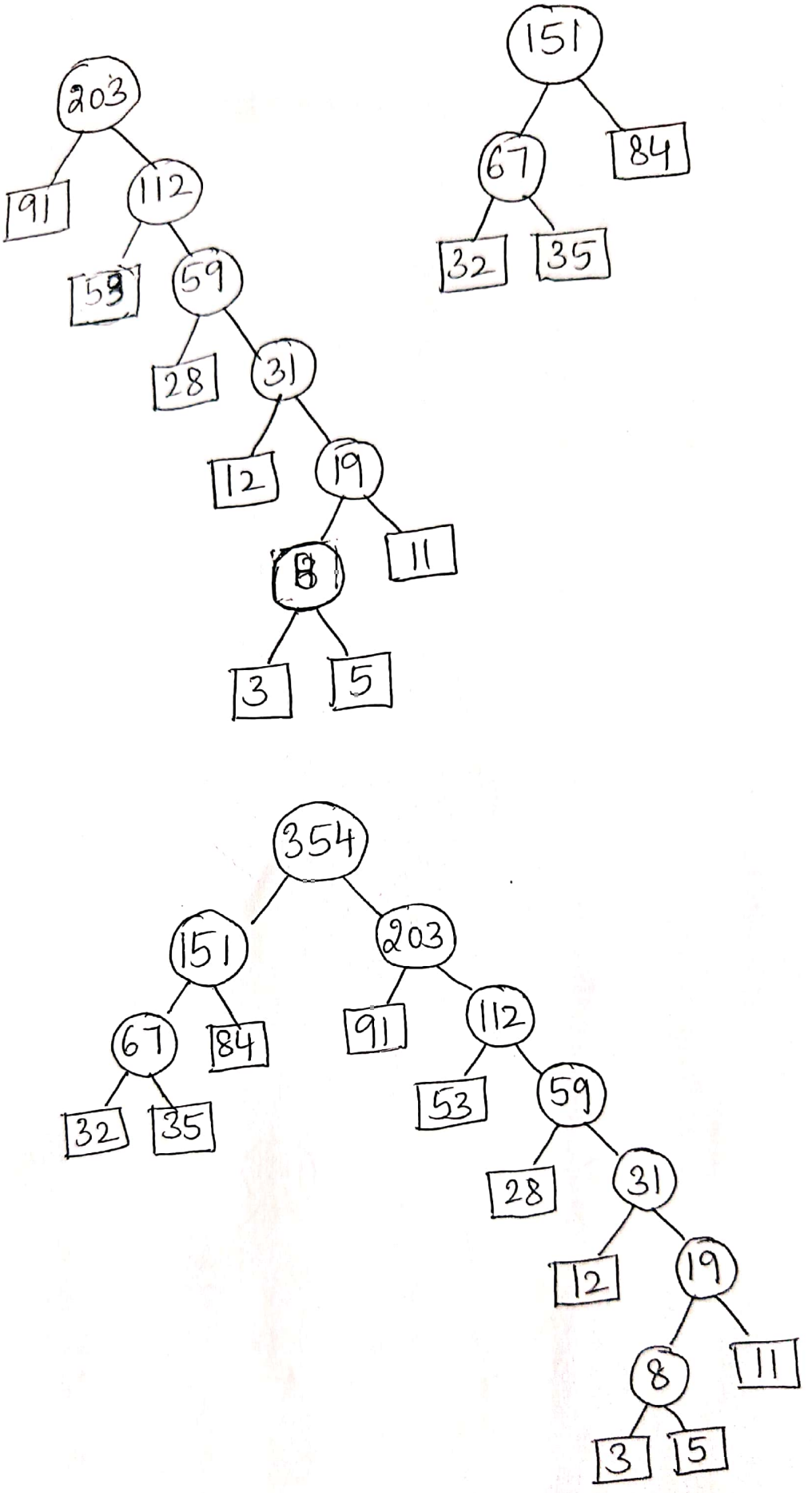
Solution:











This is the required optimal binary merge pattern.