

## Greedy Method

### Introduction

The **Greedy Method** is an important algorithm design technique used for solving optimization problems in computer science. It constructs the solution incrementally and chooses the **best possible option at each stage** based on a certain criterion. The main idea behind the greedy approach is:

“Take the best available choice now and never reconsider previous decisions.”

Greedy algorithms are widely used because they are:

- Simple
- Fast
- Efficient
- Easy to implement

However, greedy algorithms do not always guarantee the globally optimal solution for every problem.

A **Greedy Algorithm** is an algorithmic paradigm that builds a solution piece by piece, always selecting the choice that gives the most immediate benefit. The algorithm hopes that these local optimum choices will lead to the global optimum solution.

### Basic Idea of Greedy Method

At every stage:

1. Select the best available choice.
2. Add it to the solution.
3. Never change the selected choice later.
4. Continue until the complete solution is formed.

### Important Terms

#### 1. Candidate Set

The set of all possible choices available for constructing the solution.

#### Example:

In the knapsack problem, all items are candidates.

#### 2. Selection Function

Chooses the best candidate according to the greedy strategy.

#### Example:

Select item with highest profit/weight ratio.

#### 3. Feasibility Function

Checks whether adding the selected candidate violates constraints.

#### Example:

Knapsack capacity should not exceed limit.

#### 4. Objective Function

Determines the value of the solution.

#### Example:

Maximize profit or minimize cost.

#### 5. Solution Function

Checks whether the complete solution has been obtained.

**General Method of Greedy Algorithm**

The greedy method follows the following procedure:

Step 1: Initialize

Start with an empty solution set.

Step 2: Select Best Candidate

Choose the most promising option according to the greedy criterion.

Step 3: Feasibility Check

Check whether adding the candidate keeps the solution valid.

Step 4: Include Candidate

Add the selected candidate into the solution set.

Step 5: Repeat

Continue until the solution is complete.

**General Structure of Greedy Algorithm**

```
Greedy()
{
    Initialize solution set S = {}

    while (solution not complete)
    {
        x = select_best_candidate()

        if (is_feasible(x))
        {
            add x to S
        }
    }

    return S
}
```

**Characteristics of Greedy Method****1. Incremental Construction**

The solution is built step by step by continuously adding elements to the partial solution until the final solution is obtained.

**Example:** Selecting activities one by one in the Activity Selection Problem.

**2. Local Optimization**

At each step, the algorithm chooses the best immediate option available, hoping it leads to the overall optimal solution.

**Example:** Choosing the nearest vertex in Dijkstra's Algorithm.

**3. Irrevocable Decisions**

Once a choice is made, it cannot be changed later. The algorithm does not backtrack or reconsider previous decisions.

**Example:** Edges selected in Kruskal's Algorithm are not removed later.

**4. Fast Execution**

Greedy algorithms are efficient because they avoid checking all possible solutions and usually involve simple computations and sorting.

**Benefit:** Suitable for large-scale problems and real-time applications.

## 5. Heuristic Nature

Greedy algorithms often provide quick and near-optimal solutions, but they may not always guarantee the best possible result.

**Example:** Greedy method may fail in some Coin Change Problems.

## Applications of Greedy Method

Greedy algorithms are widely used in optimization problems where making the best immediate choice leads to an efficient solution. They are important in areas such as networking, scheduling, data compression, and graph theory.

### 1. Activity Selection Problem

#### Objective

Select the maximum number of non-overlapping activities from a given set of activities.

Each activity has:

- Start time
- Finish time

#### Greedy Strategy

Choose the activity with the **earliest finishing time** first.

After selecting one activity, choose the next activity whose start time is greater than or equal to the finish time of the previously selected activity.

#### Why Greedy Works

Selecting activities that finish earlier leaves more room for remaining activities.

#### Applications

- Classroom scheduling
- Meeting room allocation
- CPU task scheduling
- Event management

### 2. Fractional Knapsack Problem

#### Objective

Maximize total profit while staying within the weight limit of the knapsack.

In this problem:

- Items can be divided into fractions.

#### Greedy Strategy

Select items based on the highest:

$$\frac{\text{Profit}}{\text{Weight}}$$

ratio.

Take the item with maximum ratio first.

#### Why Greedy Works

Choosing items with greater profit per unit weight increases total profit efficiently.

#### Applications

- Cargo loading
- Investment planning
- Resource allocation
- Budget optimization

### 3. Huffman Coding

#### Objective

Reduce storage space and transmission size by compressing data.

#### Greedy Strategy

Repeatedly combine the two symbols with the smallest frequencies.

This creates shorter binary codes for frequently occurring characters.

**Why Greedy Works**

Assigning shorter codes to high-frequency symbols minimizes total encoding length.

**Applications**

- ZIP compression
- JPEG image compression
- MP3 audio compression
- File transmission systems

**4. Dijkstra's Shortest Path Algorithm****Objective**

Find the shortest path from a source vertex to all other vertices in a weighted graph.

**Greedy Strategy**

At each step, choose the unvisited vertex with the minimum distance.

**Why Greedy Works**

The shortest distance once selected cannot be improved later.

**Applications**

- GPS navigation systems
- Internet routing protocols
- Transportation networks
- Network packet transmission

**5. Minimum Spanning Tree (MST)**

A Minimum Spanning Tree connects all vertices of a graph with minimum total edge weight.

**(a) Kruskal's Algorithm****Greedy Strategy**

Select the edge with the smallest weight that does not form a cycle.

**(b) Prim's Algorithm****Greedy Strategy**

Select the minimum weight edge connected to the growing tree.

**Applications of MST**

- Cable network design
- Road construction
- Electrical wiring systems
- Communication networks

**6. Job Sequencing with Deadlines****Objective**

Schedule jobs to maximize total profit while meeting deadlines.

Each job contains:

- Deadline
- Profit

**Greedy Strategy**

Choose jobs with highest profit first and place them before their deadlines.

**Why Greedy Works**

High-profit jobs contribute more to total profit.

**Applications**

- Production scheduling
- Project management
- Manufacturing systems
- Cloud computing task allocation

**7. Coin Change Problem****Objective**

Find the minimum number of coins needed to make a given amount.

**Greedy Strategy**

Always select the largest denomination coin first.

**Example**

To make ₹49 using:

20,10,5,2,1

Choose:

$20 + 20 + 5 + 2 + 2$

**Applications**

- Currency systems
- ATM machines
- Vending machines
- Billing software

**8. Optimal Merge Pattern****Objective**

Merge multiple files with minimum total computation cost.

**Greedy Strategy**

Always merge the two smallest files first.

**Why Greedy Works**

Merging smaller files early reduces total comparisons.

**Applications**

- External sorting
- Database systems
- File merging operations
- Data processing systems

**9. Scheduling Problems**

Greedy algorithms are widely used in scheduling and resource management.

**Examples**

- CPU scheduling
- Disk scheduling
- Airline scheduling
- Exam timetable preparation

**Greedy Strategy**

Select tasks based on:

- Earliest deadline
- Shortest processing time
- Highest priority

**10. Network Routing**

Greedy methods are used in computer networks for efficient data transfer.

**Applications**

- Internet routing
- Wireless sensor networks
- Packet forwarding
- Communication optimization

**Example**

Dijkstra's algorithm is used to determine shortest routes between routers.

**11. Data Compression**

Greedy techniques help reduce storage and transmission costs.

**Example**

Huffman coding assigns:

- Short codes → frequent symbols
- Long codes → less frequent symbols

**Applications**

- Multimedia compression
- Streaming services
- Storage systems

## 12. Resource Allocation Problems

Greedy algorithms allocate limited resources efficiently.

### Applications

- Memory allocation
- Bandwidth allocation
- Budget distribution
- Machine utilization

### Example

Resources are assigned to tasks that provide maximum immediate benefit.

### Advantages of Greedy Applications

1. **Fast Execution:** Greedy algorithms make decisions quickly without checking all possible solutions.
2. **Simple Implementation:** They are easy to design, understand, and implement.
3. **Efficient for Large Problems:** Suitable for handling large datasets and optimization problems efficiently.
4. **Low Memory Usage:** Requires less storage since previous computations are usually not stored.
5. **Useful in Real-Time Systems:** Provides quick solutions, making it useful in networking, scheduling, and routing systems.

### Limitations of Greedy Applications

1. **Does Not Always Give Optimal Solution:** Greedy choice may not produce the best result for all problems.
2. **Depends on Problem Properties:** Works correctly only if the problem satisfies greedy choice property and optimal substructure.
3. **No Backtracking:** Once a decision is made, it cannot be changed later.
4. **Not Suitable for All Problems:** Some complex optimization problems require dynamic programming or other methods instead of greedy algorithms.

### Job sequencing with deadlines

Job Sequencing with Deadlines is a famous optimization problem solved using the **Greedy Method**. In this problem, a set of jobs is given where each job has:

- a **deadline**
- a **profit**

The objective is to schedule jobs in such a way that:

- each job is completed before its deadline
- maximum total profit is earned

Each job requires only *one unit of time* for execution.

The **Job Sequencing with Deadlines Problem** is the process of scheduling jobs with deadlines and profits such that the maximum profit is obtained while completing jobs within their specified deadlines.

## Problem Statement

Given:

- $n$  jobs
- each job has:
  - deadline  $d_i$
  - profit  $p_i$

Find an optimal sequence of jobs that maximizes total profit.

## Characteristics

1. Each job takes one unit of time.
2. Only one job can be executed at a time.
3. A job must be completed before or on its deadline.
4. Goal is to maximize total profit.

## Greedy Strategy

The greedy approach follows:

1. Sort jobs in decreasing order of profit.
2. Select the highest profit job first.
3. Place it in the latest available slot before its deadline.
4. Repeat for remaining jobs.

## Algorithm Steps

1. Arrange jobs based on descending order of profits.
2. Find maximum deadline.
3. Create slots for scheduling jobs.
4. For each job:
  - find a free slot before its deadline
  - assign the job if slot is available
5. Compute total profit.

### Pseudo Code

```

JobSequencing(Jobs, n)
{
    Sort jobs in decreasing order of profit

    for i = 1 to max_deadline
        slot[i] = empty

    for each job j
    {
        for k = min(deadline[j], max_deadline) downto 1
        {
            if slot[k] is empty
            {
                slot[k] = job[j]
                break
            }
        }
    }

    Display scheduled jobs
    Calculate total profit
}
  
```

**Example:**

Job	Deadline	Profit
J1	2	100
J2	1	19
J3	2	27
J4	1	25
J5	3	15

**Step 1: Sort by Profit**

Job	Deadline	Profit
J1	2	100
J3	2	27
J4	1	25
J2	1	19
J5	3	15

**Step 2: Find Maximum Deadline**

Max Deadline = 3

Available slots:

[1][2][3]

**Step 3: Schedule Jobs****Job J1**

Deadline=2

Place in slot 2.

[ ] [J1] [ ]

**Job J3**

Deadline = 2

Slot 2 occupied → place in slot 1.

[J3] [J1] [ ]

**Job J4**

Deadline = 1

Slot 1 occupied → cannot schedule.

**Job J2**

Deadline = 1

Slot 1 occupied → cannot schedule.

**Job J5**

Deadline = 3



Place in slot 3.

$[J3][J1][J5]$

**Final Job Sequence**

$J3 \rightarrow J1 \rightarrow J5$

**Total Profit**

$$27 + 100 + 15 = 142$$

**Time Complexity  
Sorting**

$$O(n \log n)$$

**Slot Allocation**

$$O(n^2)$$

Overall complexity:

$$O(n^2)$$

**Advantages**

1. Simple and efficient method.
2. Maximizes profit effectively.
3. Easy to implement.
4. Suitable for scheduling problems.

**Limitations**

1. Assumes each job takes equal time.
2. Greedy approach may not work for all scheduling problems.
3. Works only when deadlines and profits are properly defined.

**Applications of Job Sequencing with Deadlines**

- CPU Scheduling: Used by operating systems to schedule tasks efficiently.
- Manufacturing Systems: Schedules production jobs to maximize revenue.
- Project Management: Helps prioritize important tasks within deadlines.
- Cloud Computing: Allocates computing tasks to maximize resource utilization.
- Business Management: Used for selecting high-profit activities within limited time.
- Data Processing Systems: Schedules batch jobs and processing tasks efficiently.

**Real-Life Example**

Suppose a company has:

- limited machine time
- multiple customer orders

The company schedules:

- highest profit orders first
- within delivery deadlines

This increases total business profit.

### **Fractional Knapsack Problem using Greedy Method**

The **Fractional Knapsack Problem** is a classic optimization problem that can be efficiently solved using the **Greedy Method**.

In this problem:

- Each item has:
  - Profit (value)
  - Weight
- The knapsack has limited capacity.
- Items can be divided into fractions.

The objective is to maximize total profit without exceeding the knapsack capacity.

#### **Definition**

The Fractional Knapsack Problem is the process of selecting items such that the maximum profit is obtained while keeping the total weight within the knapsack capacity, where fractional parts of items can also be included.

#### **Problem Statement**

Given:

- $n$  items
- profit  $p_i$
- weight  $w_i$
- knapsack capacity  $W$

Find the optimal selection of items to maximize total profit.

#### **Greedy Strategy**

The greedy approach selects items based on the highest:

$$\frac{\text{Profit}}{\text{Weight}}$$

ratio.

This ratio represents profit earned per unit weight.

#### **Rule**

- Select item with highest ratio first.
- If full item cannot fit, take only the required fraction.

#### **Algorithm Steps**

1. Calculate profit-to-weight ratio for all items.
2. Sort items in decreasing order of ratio.
3. Select items one by one.
4. If the item completely fits:
  - include full item.
5. Otherwise:
  - include fractional part of item.
6. Continue until knapsack becomes full.

#### **Characteristics of Fractional Knapsack**

1. Items can be divided.
2. Greedy method gives optimal solution.
3. Based on profit-to-weight ratio.
4. Maximizes profit efficiently.

**Pseudo Code**

```
FractionalKnapsack(W, items, n)
{
    Sort items in decreasing order of (profit/weight)

    totalProfit = 0

    for each item i
    {
        if weight[i] <= W
        {
            take complete item
            totalProfit += profit[i]
            W = W - weight[i]
        }
        else
        {
            fraction = W / weight[i]

            totalProfit += profit[i] * fraction

            break
        }
    }

    return totalProfit
}
```

**Advantages**

1. Simple and efficient algorithm.
2. Produces optimal solution.
3. Faster execution.
4. Easy to implement.

**Limitations**

1. Applicable only when items are divisible.
2. Cannot be used directly for 0/1 knapsack problems.

**Time Complexity****Sorting** $O(n \log n)$ **Selection Process** $O(n)$ 

Overall Time Complexity:

 $O(n \log n)$

YBKR

## Single Source Shortest Path (SSSP)

### Introduction

The **Single Source Shortest Path Problem** is a fundamental problem in graph theory and greedy algorithms.

The objective is to find the shortest path from a single source vertex to all other vertices in a weighted graph.

One of the most commonly used algorithms for solving this problem is **Dijkstra's Algorithm**, which follows the greedy method.

### Definition

The Single Source Shortest Path Problem is the process of determining the minimum distance from one source vertex to every other vertex in a graph.

### Applications

- GPS and navigation systems
- Network routing
- Transportation systems
- Internet packet routing
- Communication networks

### Dijkstra's Algorithm

Dijkstra's Algorithm is a greedy algorithm used to find the shortest path from a source node to all other nodes in a graph with non-negative edge weights.

### Greedy Strategy

At every step:

- Select the unvisited vertex with the smallest distance.
- Update distances of adjacent vertices.

Once a vertex is selected, its shortest distance becomes permanent.

### Algorithm Steps

1. Assign distance 0 to source vertex.
2. Assign infinity to all other vertices.
3. Mark all vertices as unvisited.
4. Select the unvisited vertex with minimum distance.
5. Update distances of neighboring vertices.
6. Mark selected vertex as visited.
7. Repeat until all vertices are visited.

### Advantages

1. Efficient for shortest path problems.
2. Easy to implement.
3. Works well for non-negative weights.
4. Widely used in networking and routing.

### Limitations

1. Does not work for negative edge weights.
2. Greedy choice may fail for negative cycles.
3. Requires connected graph for full traversal.

**Time Complexity  
Using Adjacency Matrix**

$$O(V^2)$$

**Using Priority Queue and Heap**

$$O((V + E)\log V)$$

Where:

- $V$  = number of vertices
- $E$  = number of edges

***Pseudo Code***

```
Dijkstra(Graph, source)
{
  for each vertex v
  {
    distance[v] = infinity
    visited[v] = false
  }

  distance[source] = 0

  for i = 1 to number_of_vertices
  {
    u = vertex with minimum distance

    visited[u] = true

    for each neighbor v of u
    {
      if visited[v] == false AND
        distance[u] + weight(u,v) < distance[v]
      {
        distance[v] = distance[u] + weight(u,v)
      }
    }
  }
}
```

YBKR

## Minimum Spanning Trees

In this chapter, we study an important graph optimization problem called the **Minimum Spanning Tree (MST)**. A Minimum Spanning Tree of an undirected weighted graph is a tree that connects all the vertices of the graph with the minimum possible total edge weight and without forming any cycles.

This chapter first introduces the concepts of **spanning trees** and **minimum spanning trees** in detail. It then explains three important MST algorithms:

- **Kruskal's Algorithm**
- **Prim's Algorithm**
- **Borůvka's Algorithm**

Among these, Kruskal's and Prim's are sequential algorithms, while Borůvka's is a parallel algorithm. All these algorithms are based on an important concept known as the **Cut Property**, which helps in selecting edges that safely belong to the minimum spanning tree.

An undirected graph is called a **forest** if it does not contain any cycles, and it is called a **tree** if it is connected and acyclic.

For a connected undirected graph, we often need to select a subset of edges that:

- connects all the vertices,
- forms a tree,
- and avoids cycles.

Such a tree is called a **Spanning Tree**.

### Definition

For a connected undirected graph:

$$G = (V, E)$$

a spanning tree is defined as:

$$T = (V, E')$$

where:

$$E' \subseteq E$$

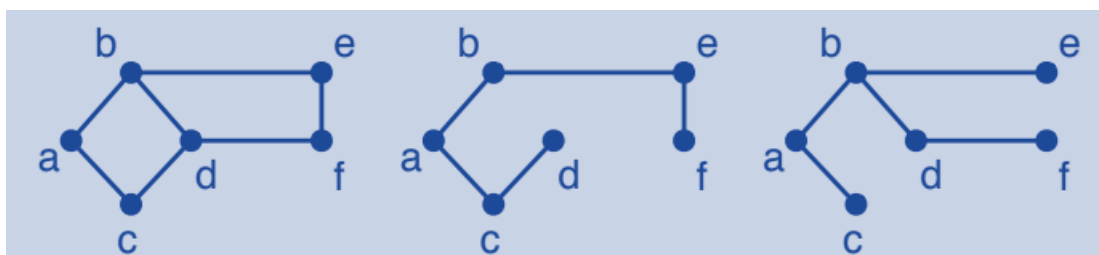
This means the spanning tree contains:

- all vertices of the original graph,
- only some of its edges.

A graph can have multiple spanning trees, but every spanning tree always contains:

- $|V|$  vertices
- $|V| - 1$  edges.

Example: A graph on the left, and two possible spanning trees





Design an algorithm for finding a spanning tree of a connected, undirected graph?

One way to generate a spanning tree is simply to do a graph search. For example the DFS tree of a DFS is a spanning tree, as it finds a path from a source to all the vertices. Similarly, we can construct a spanning tree based on BFS, by adding each edge that leads to the discovery of an unvisited vertex to the tree. DFS and BFS are work-efficient algorithms for computing spanning trees but as we discussed they are not good parallel algorithms.

Given a connected, undirected weighted graph  $G = (V, E, w)$ , the minimum (weight) spanning tree (MST) problem requires finding a spanning tree of minimum weight, where the weight of a tree  $T$  is defined as:

The weight of a spanning tree  $T$  is defined as the sum of the weights of all edges present in the tree.

$$w(T) = \sum_{e \in E(T)} w(e)$$

Where:

- $w(T)$  = total weight (cost) of the spanning tree
- $E(T)$  = set of edges in the spanning tree  $T$
- $w(e)$  = weight of edge  $e$

Thus, the total cost of a spanning tree is obtained by adding the weights of all its edges.

### Example

Suppose a spanning tree contains edges with weights:

2, 4, 5

Then,

$$w(T) = 2 + 4 + 5 = 11$$

So, the total weight of the spanning tree is:

11

### Kruskal's Algorithm

Kruskal's Algorithm is a greedy algorithm used to find the **Minimum Spanning Tree (MST)** of a connected, undirected, weighted graph. The algorithm works by repeatedly selecting the lightest (minimum weight) edge that does not form a cycle with the already selected edges.

Original Idea of Kruskal's Algorithm

As described by Kruskal:

"Choose the shortest edge that does not form a loop with the already chosen edges."

In modern terminology:

- "shortest" → lightest
- "loops" → cycles

### Definition

Kruskal's Algorithm constructs the Minimum Spanning Tree by selecting edges in increasing order of weight while avoiding cycle formation.

### Greedy Strategy

At every step:

Choose the minimum weight edge that does not form a cycle

This greedy choice ensures that the total weight of the spanning tree remains minimum.

**Working Principle**

1. Sort all edges in increasing order of weight.
2. Select the smallest edge.
3. Check whether adding the edge forms a cycle.
4. If no cycle is formed:
  - include the edge in MST.
5. Otherwise:
  - discard the edge.
6. Repeat until:

$$|V| - 1$$

edges are selected.

**Why Kruskal's Algorithm is Correct**

Kruskal's algorithm maintains an important invariant:

The selected edges are always part of some Minimum Spanning Tree.

**Cycle Property**

If an edge forms a cycle:

- it cannot belong to the MST,
- because a spanning tree must not contain cycles.

Therefore, such edges are discarded.

**Light Edge Property**

Among all possible edges connecting two components:

- the minimum weight edge is always safe to include.

This edge is called a **light edge**.

**Pseudo Code**

```

Kruskal(G)
{
    Sort all edges in increasing order of weight

    MST = empty

    for each edge (u,v)
    {
        if adding (u,v) does not form cycle
        {
            add (u,v) to MST
        }
    }

    return MST
}
  
```

**Efficient Implementation**

Checking cycles repeatedly can become expensive.

To improve efficiency, Kruskal's Algorithm uses:

- **Disjoint Set**
- **Union-Find Data Structure**

**Union-Find Operations****1. Find**

Determines which connected component a vertex belongs to.

**2. Union**

Merges two connected components.

**Cycle Detection**

An edge:

$$(u, v)$$

forms a cycle if:

$$Find(u) = Find(v)$$

Otherwise:

- edge is safe,
- perform union operation.

**Time Complexity****Sorting Edges**

$$O(E \log E)$$

**Union-Find Operations**

Nearly:

$$O(1)$$

Overall:

$$O(E \log E)$$

Where:

- $E$  = number of edges

**Advantages of Kruskal's Algorithm**

1. Simple and easy to understand.
2. Efficient for sparse graphs.
3. Produces optimal MST.
4. Greedy approach reduces complexity.

**Limitations**

1. Sorting edges can be costly for dense graphs.
2. Cycle checking requires additional data structures.
3. Less efficient than Prim's algorithm for dense graphs.

**Applications**

- Computer network design
- Road and railway planning
- Electrical wiring systems
- Water pipeline networks
- Telecommunications

Kruskal's Algorithm is an important greedy algorithm for finding Minimum Spanning Trees. By repeatedly selecting the lightest edge that does not form a cycle, the algorithm efficiently constructs a minimum-cost spanning tree. Efficient implementation using Union-Find data structures makes the algorithm practical for large graph problems.

YBKR

## Prim's Algorithm

### Introduction

Prim's Algorithm is a greedy algorithm used to find the **Minimum Spanning Tree (MST)** of a connected, undirected, weighted graph.

The algorithm starts from any vertex and continuously adds the minimum weight edge that connects a visited vertex to an unvisited vertex until all vertices are included in the spanning tree.

### Definition

Prim's Algorithm constructs the Minimum Spanning Tree by growing a single tree step by step, always selecting the minimum weight edge connected to the current tree.

### Greedy Strategy

At every step:

Choose the minimum weight edge connecting visited and unvisited vertices

This ensures that the total cost of the spanning tree remains minimum.

### Working Principle

1. Start with any vertex.
2. Mark it as visited.
3. Find the minimum weight edge connecting:
  - visited vertex
  - unvisited vertex
4. Add the selected edge and new vertex to MST.
5. Repeat until all vertices are included.

### Why Prim's Algorithm is Correct

Prim's Algorithm maintains the property that:

- the selected edges always form a valid part of the MST,
- no cycles are formed,
- and the minimum possible edge is chosen at every step.

The algorithm uses the **Cut Property**:

The minimum edge crossing a cut is always safe to include in the MST.

#### *Pseudo Code*

```
Prim(G, start)
{
    Mark start vertex as visited

    while MST is not complete
    {
        Find minimum edge (u,v)
        where:
        u is visited
        v is unvisited

        Add edge (u,v) to MST

        Mark v as visited
    }

    return MST
}
```

## Efficient Implementation

Prim's Algorithm is efficiently implemented using:

- Priority Queue
- Min Heap

The heap helps quickly select the minimum weight edge.

## Time Complexity

### Using Adjacency Matrix

$$O(V^2)$$

### Using Min Heap and Priority Queue

$$O(E \log V)$$

Where:

- $V$  = number of vertices
- $E$  = number of edges

## Advantages of Prim's Algorithm

1. Simple and efficient.
2. Works well for dense graphs.
3. Produces optimal MST.
4. Easy to implement using heaps.

## Limitations

1. Requires connected graph.
2. Less efficient for sparse graphs compared to Kruskal's Algorithm.
3. Priority queue implementation may increase complexity.

## Applications

- Computer network design
- Electrical wiring systems
- Road and railway networks
- Communication systems
- Water supply networks

## Differentiation between Prim's and Kruskal's Algorithms

Prim's Algorithm	Kruskal's Algorithm
Uses a <b>vertex-based approach</b>	Uses an <b>edge-based approach</b>
Starts from any vertex and grows a single tree	Starts with the smallest edge and builds forest/tree
Selects minimum edge connected to visited vertices	Selects globally minimum edge
Avoids cycles by selecting edges connected to unvisited vertices	Avoids cycles using Union-Find / Disjoint Set
Suitable for <b>dense graphs</b>	Suitable for <b>sparse graphs</b>
Uses <b>Priority Queue / Min Heap</b>	Uses <b>Sorting and Union-Find</b>
Tree grows continuously from one starting vertex	Multiple small trees are merged together
Does not require sorting all edges initially	Requires sorting of all edges
Time Complexity: $(O(E \log V))$ using heap	Time Complexity: $(O(E \log E))$
Easier for adjacency matrix representation	Easier for edge list representation

YBKR

## Optimal Storage on Tapes

### Introduction

The **Optimal Storage on Tapes** problem is an important application of the **Greedy Method**. It deals with storing files on a magnetic tape in such a way that the **Mean Retrieval Time (MRT)** is minimized.

Since magnetic tapes are accessed sequentially, the position of files affects the retrieval time. Therefore, files must be arranged optimally.

### Definition

Optimal Storage on Tapes is the process of arranging files on a tape such that the average or mean retrieval time of files is minimum.

### Basic Idea

In magnetic tape storage:

- files are accessed sequentially,
- files stored earlier are retrieved faster,
- files stored later take more time.

Hence, to minimize retrieval time:

Smaller files should be stored first.

This follows the greedy strategy.

### Greedy Strategy

Arrange files in:

Ascending order of file sizes

This minimizes the cumulative retrieval time.

### Mean Retrieval Time (MRT)

Mean Retrieval Time is defined as:

$$MRT = \frac{\text{Sum of retrieval times of all files}}{\text{Number of files}}$$

### Algorithm Steps

1. Arrange files in ascending order of size.
2. Store files in that order on the tape.
3. Compute retrieval time for each file.
4. Calculate Mean Retrieval Time.

#### *Pseudo Code*

```
OptimalStorage(files, n)
{
    Sort files in ascending order

    retrieval = 0
    total = 0

    for each file i
    {
        retrieval += file[i]
        total += retrieval
    }

    MRT = total / n

    return MRT
}
```



**Example Problem****Given File Sizes**

20, 10, 30, 5

**Step 1: Sort Files**

Ascending order:

5, 10, 20, 30

**Step 2: Compute Retrieval Times****File Size Retrieval Time**

5	5
10	5 + 10 = 15
20	5 + 10 + 20 = 35
30	5 + 10 + 20 + 30 = 65

**Step 3: Total Retrieval Time**

$$5 + 15 + 35 + 65 = 120$$

**Step 4: Mean Retrieval Time**

$$MRT = \frac{120}{4} = 30$$

**Optimal Order**

5 → 10 → 20 → 30

**Why Greedy Method Works**

Placing smaller files first reduces waiting time for remaining files and minimizes cumulative retrieval time.

Thus:

- local optimal choice → global optimal solution.

**Time Complexity**  
**Sorting**

$$O(n \log n)$$

**Retrieval Calculation**

$$O(n)$$

Overall:

$$O(n \log n)$$

**Applications**

- 1. Magnetic Tape Storage:** Efficient file arrangement on backup tapes.
- 2. Data Archiving:** Reducing average retrieval time in storage systems.
- 3. Sequential Access Devices:** Optimization in sequential memory systems.
- 4. Operating Systems:** Efficient scheduling of sequential file access.

**Advantages**

1. Simple greedy solution.
2. Minimizes average retrieval time.
3. Efficient and easy to implement.
4. Improves storage performance.

**Limitations**

1. Applicable mainly to sequential access storage.
2. Assumes file access frequency is equal.
3. Not suitable for random access devices.

Optimal Storage on Tapes is a classical greedy algorithm problem where files are arranged in ascending order of size to minimize mean retrieval time. By storing smaller files first, the greedy method efficiently reduces overall retrieval delay and improves storage performance.