Greedy Approach

▼ **TOC**

The greedy approach is a problem-solving strategy that involves making locally optimal choices at each step in hopes of finding a global optimum. It is simple and efficient but does not always guarantee an optimal solution.

Huffman Code

Huffman code is a lossless data compression algorithm that is widely used to compress data, including text, images, and audio. This algorithm generates a prefix code to compress data by assigning shorter codes to frequently occurring characters and longer codes to less frequent characters.

Advantages

- Huffman code is an efficient algorithm that can achieve a significant reduction in the size of the data.
- It is a lossless compression algorithm that does not compromise the data quality.
- Huffman code is widely used in many applications such as data compression, image compression, and audio compression.

Disadvantages

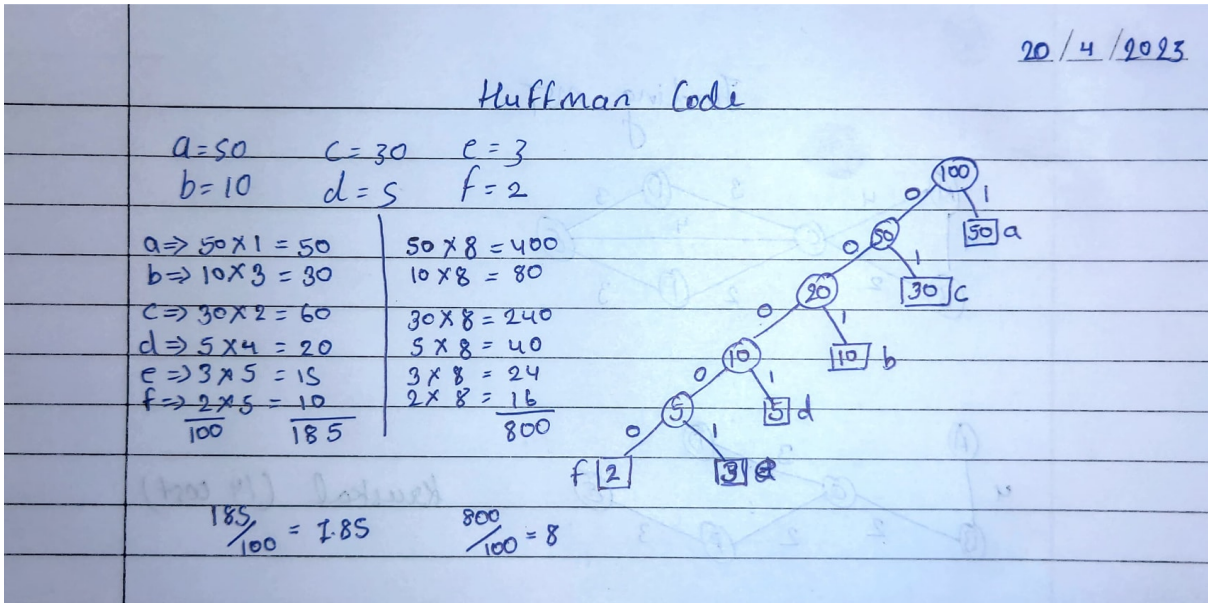
- Huffman code is a variable-length encoding algorithm that requires the use of a decoding table, which can add overhead to the compressed data.
- It requires a pre-processing step to build the Huffman tree, which can be time-consuming for large datasets.
- If the data contains a small number of distinct symbols, Huffman coding may not provide significant compression.

Algorithm

1. Calculate the frequency of occurrence of each character in the input text.
2. Create a binary tree with each character as a leaf node, and assign a weight to each node based on its frequency of occurrence.
3. Merge the two nodes with the lowest weights to create a new parent node, with a weight equal to the sum of the weights of its children.
4. Repeat step 3 until all nodes are merged into a single root node.

5. Traverse the binary tree to assign a binary code to each character, where a left branch represents a 0 bit and a right branch represents a 1 bit.
6. The Huffman code for the input text is the concatenation of the binary codes for each character in the text.

Example



Complexities

The time complexity is **O(n log n)**, where n is the number of symbols in the input data.

The space complexity is **O(n)**, where n is the number of symbols in the input data.

Dijkstra’s Algorithm

Dijkstra's Algorithm is an algorithm used to find the shortest path between a starting node and all other nodes in a graph with non-negative edge weights. The algorithm maintains a priority queue of nodes to visit, starting with the starting node, and updates the distances to all adjacent nodes as it visits each node.

Advantages

- Dijkstra's Algorithm is a well-known and widely used algorithm for

Disadvantages

- Dijkstra's Algorithm cannot handle graphs with negative edge weights.

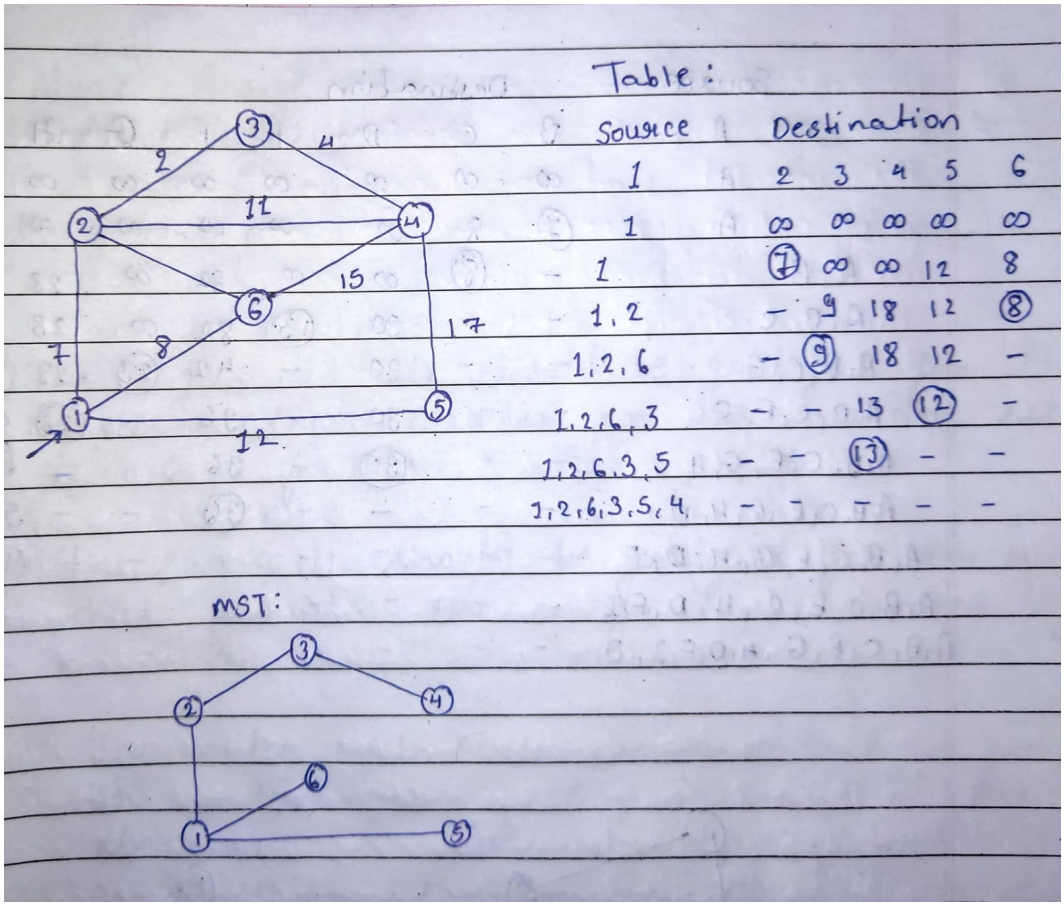
finding the shortest path in a graph.

- The algorithm guarantees finding the shortest path with non-negative edge weights.
- The algorithm requires maintaining a priority queue of nodes to visit, which can be memory-intensive for large graphs.

Algorithm

1. Initialize a distance array for all nodes with infinity, except for the starting node with a distance of 0.
2. Initialize a priority queue with the starting node and its distance as the key.
3. While the priority queue is not empty, dequeue the node with the smallest distance and update the distances to all adjacent nodes if the new distance is smaller than the current distance.
4. Repeat until all nodes have been visited.

Example



Complexities

The time complexity is **O(E log V)**, where E is the number of edges and V is the number of vertices in the graph.

The space complexity is **O(V)**, as it requires maintaining a distance array for all nodes and a priority queue of nodes to visit.

Spanning Tree

A spanning tree is a subset of a connected, undirected graph that includes all of its vertices and enough of its edges to form a tree (a connected graph without cycles).

There are two conditions for a spanning tree:

1. **Connectedness**: A spanning tree must connect all the vertices of the original graph.
2. **Acyclic**: A spanning tree must not contain any cycles.

These two conditions ensure that a spanning tree includes all the vertices of the original graph and that it does not contain any redundant edges.

For n number of vertices in a graph there can be n^{n-2} number of spanning trees.

Kruskal’s Algorithm

Kruskal's Algorithm is a greedy algorithm that finds the minimum spanning tree (MST) of a connected weighted graph. It works by sorting the edges of the graph by weight and adding them to the MST if they do not form a cycle. The algorithm continues until all vertices are connected.

Advantages

- Kruskal's Algorithm is a simple and efficient algorithm that guarantees the construction of a minimum spanning tree.
- It is a distributed algorithm that can be easily parallelized, making it

Disadvantages

- Kruskal's Algorithm can be slow for large graphs, as it requires sorting of all the edges, which can take O(E log E) time.
- The algorithm does not work for disconnected graphs and requires the graph to be connected.

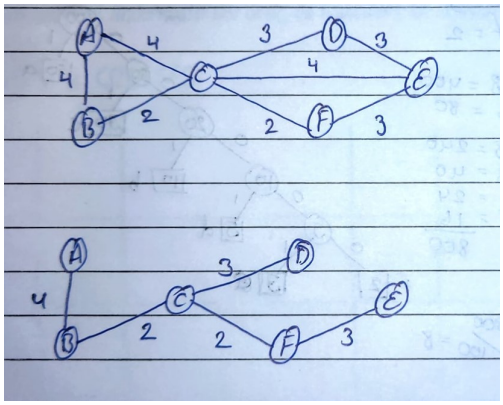
suitable for distributed computing environments.

- The algorithm is suitable for both dense and sparse graphs.

Algorithm

1. Sort the edges of the graph in ascending order based on their weights.
2. Create an empty MST and set the counter of the number of edges added to 0.
3. For each edge in the sorted list, in increasing order of weight, check if adding it to the MST will create a cycle. If adding the edge does not create a cycle, add it to the MST and increment the counter.
4. Continue this process until the MST contains n-1 edges, where n is the number of vertices in the graph.
5. The resulting MST is the minimum spanning tree of the graph.

Example



Complexities

The time complexity is **O(E log E)**, where E is the number of edges in the graph.

The space complexity is **O(V)**, where V is the number of vertices in the graph.

Prim’s Algorithm

Prim's Algorithm is a greedy algorithm that finds the minimum spanning tree (MST) of a connected weighted graph. It works by starting at an arbitrary vertex and adding the edge with the smallest weight that connects to a new vertex until all vertices are connected.

Advantages

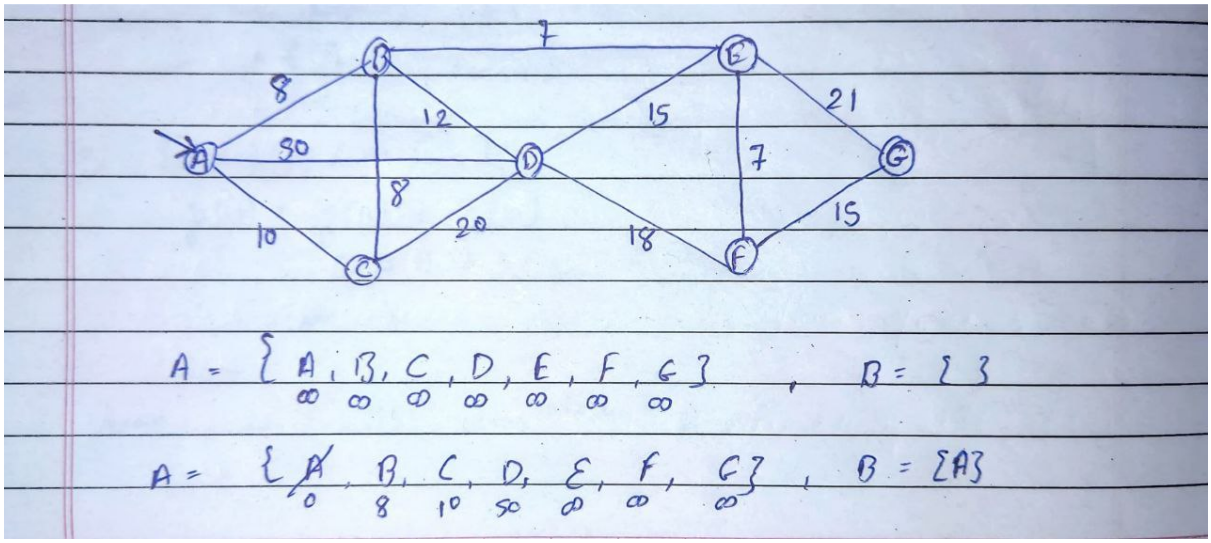
Disadvantages

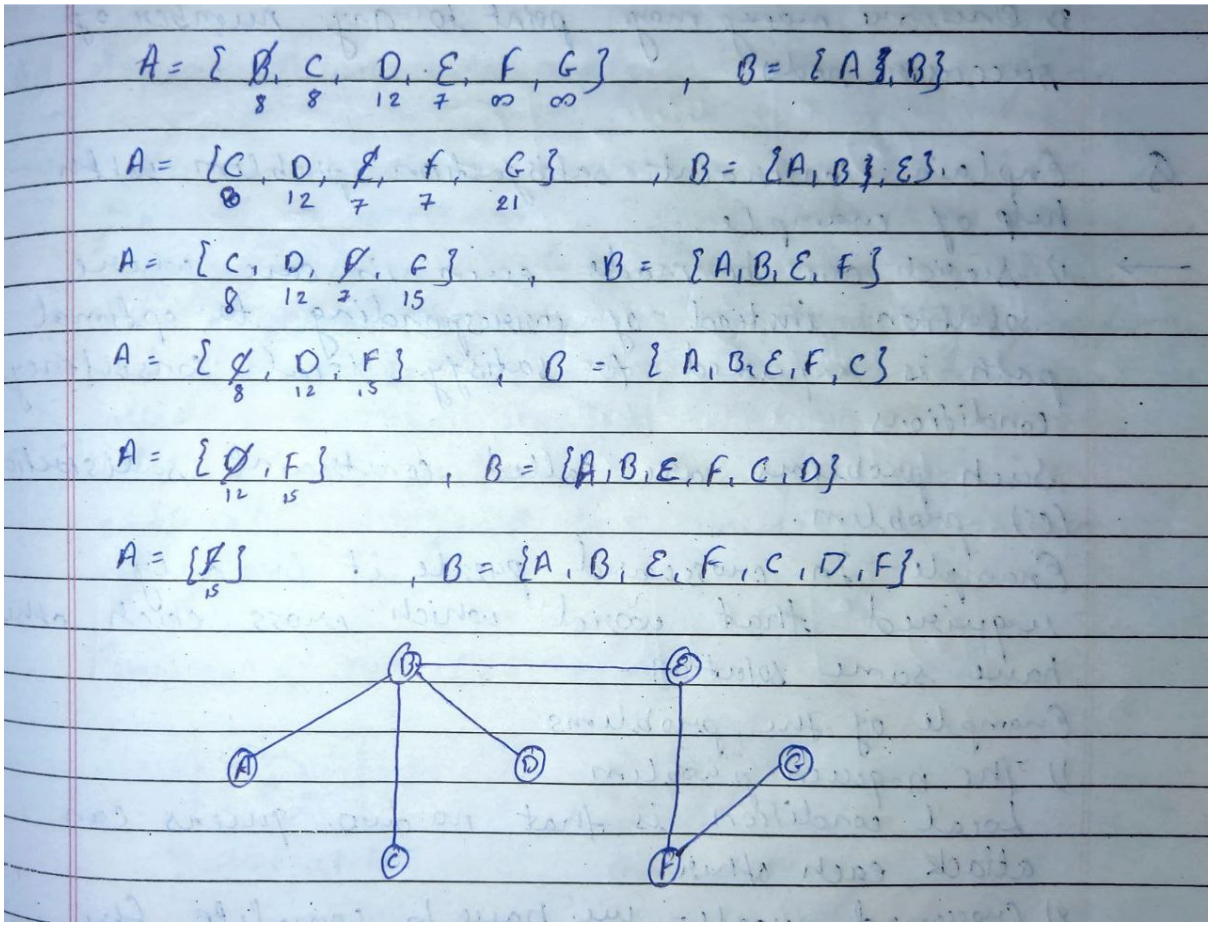
- Prim's Algorithm is a simple and efficient algorithm that guarantees the construction of a minimum spanning tree.
- The algorithm can be easily implemented using a priority queue, making it suitable for sparse graphs.
- Prim's Algorithm is suitable for graphs with negative edge weights.
- Prim's Algorithm can be slow for dense graphs, as it requires scanning all the vertices to find the next minimum weight edge.
- The algorithm does not work for disconnected graphs and requires the graph to be connected.

Algorithm

1. Choose an arbitrary vertex as the starting point.
2. Create a set of visited vertices and a priority queue to hold the edges connected to the visited vertices.
3. Add all edges connected to the starting vertex to the priority queue.
4. While the priority queue is not empty, extract the minimum-weight edge and the vertex it connects to.
5. If the vertex is not in the set of visited vertices, add it to the set and add all its edges to the priority queue.
6. Continue this process until all vertices have been visited.
7. The resulting MST is the set of edges connecting the visited vertices.

Example





Complexities

The time complexity is **O(E log V)**, where E is the number of edges in the graph and V is the number of vertices in the graph.

The space complexity is **O(V)**, where V is the number of vertices in the graph.

Fractional Knapsack

Fractional Knapsack is a problem in which we have a knapsack with a fixed capacity and a set of items with weights and values. The goal is to maximize the total value of the items that can be placed in the knapsack, even if they are only a fraction of an item.

Advantages

- Fractional Knapsack is a simple and efficient algorithm that provides an optimal solution to the problem.
- The algorithm can be easily extended to handle items with different priority levels.
- Fractional Knapsack can handle items that are too large to fit in the knapsack, unlike 0/1 Knapsack.

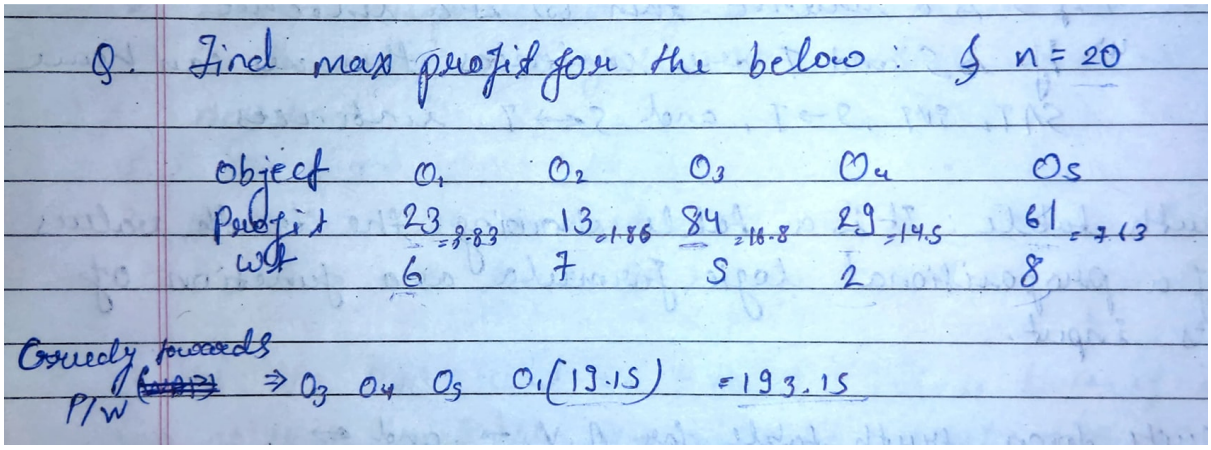
Disadvantages

- Fractional Knapsack can only handle items that can be divided into fractional parts, which is not always practical.
- The algorithm may be unable to handle items with complex dependencies or interrelationships.

Algorithm

1. Calculate the value-to-weight ratio of each item.
2. Sort the items in descending order based on their value-to-weight ratio.
3. Starting from the item with the highest value-to-weight ratio, add as much of the item as possible to the knapsack until it is full.
4. Move on to the next item in the sorted list and repeat step 3 until all items have been considered.
5. Return the total value of the items in the knapsack.

Example



Complexities

The time complexity is **O(n log n)**, where n is the number of items in the knapsack.

The space complexity of the algorithm is **O(1)**.

Job Sequencing

Job Sequencing is a problem in which we have a set of jobs with deadlines and profits. The goal is to schedule the jobs within their deadlines to maximize the total profit earned.

Advantages

- Job Sequencing is a simple and efficient algorithm that provides an optimal solution to the problem.
- The algorithm can be easily extended to handle jobs with different priority levels.
- Job Sequencing can handle jobs that have different processing times.

Disadvantages

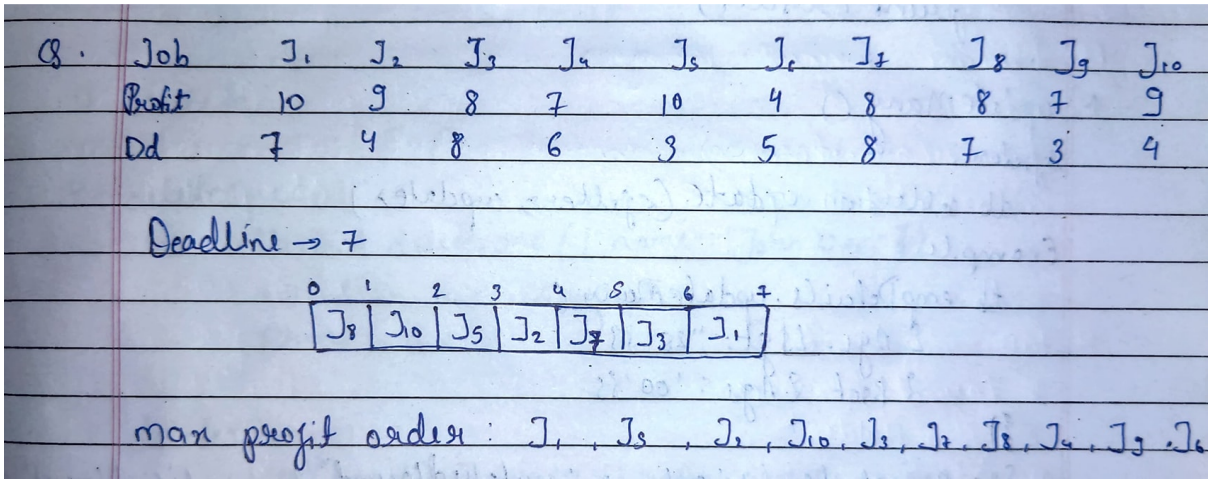
- Job Sequencing assumes that all jobs can be completed within their deadlines, which may not always be practical.
- The algorithm may be unable to handle jobs with complex dependencies or interrelationships.

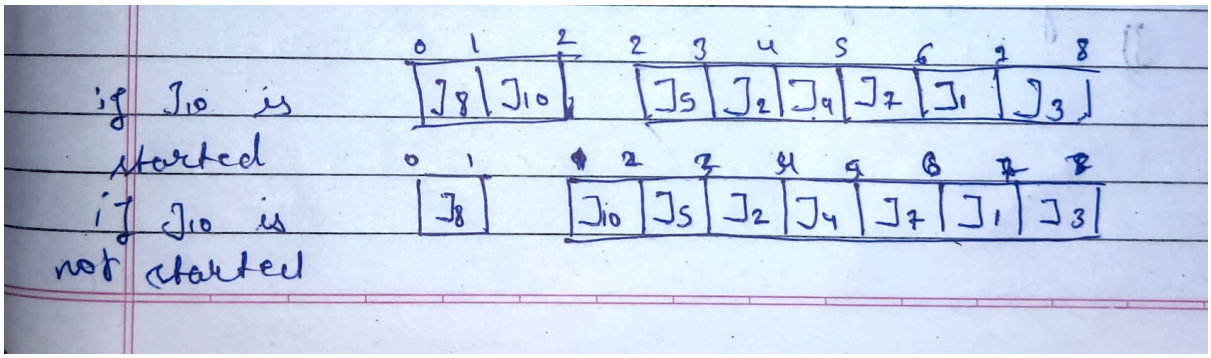
Algorithm

1. Sort the jobs in descending order based on their profit or in ascending order based on their time required for completion.

2. For each job, check if its deadline allows it to be completed within the given time. If it does, schedule the job near the deadline and mark the time slot as occupied.
3. If the time slot is already occupied, check the next available time slot until a suitable slot is found.
4. Repeat steps 2 and 3 until all jobs have been considered.
5. Return the total profit or the total time required to complete all jobs.

Example





Complexities

The time complexity is **O(n log n)**, where n is the number of jobs.

The space complexity is **O(m)**, where m is the maximum deadline.