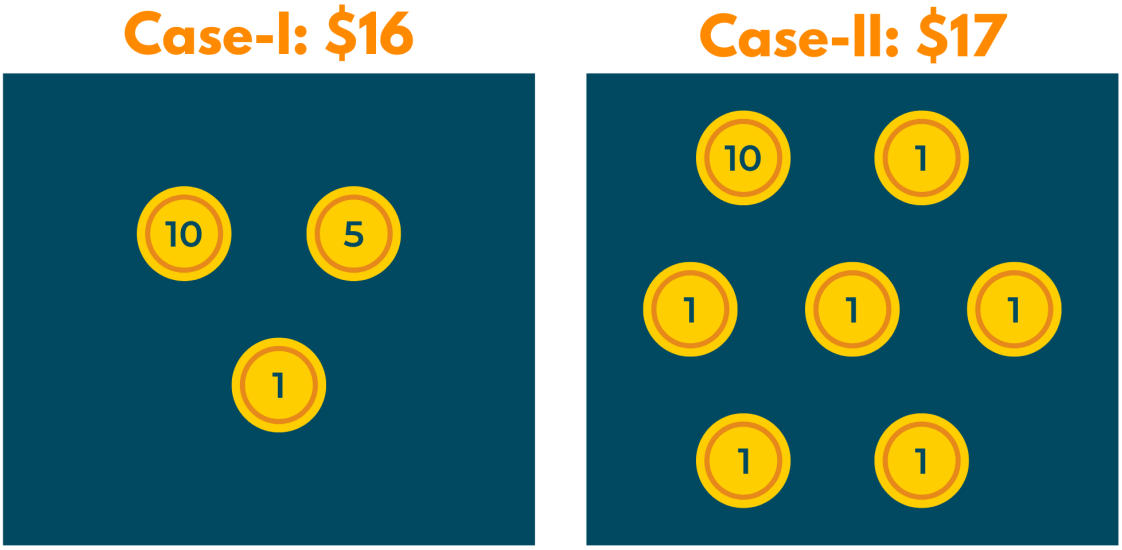
Greedy Algorithm

As the name implies, this is a simple approach which tries to find the **best** solution at every step. Thus, it aims to find the local optimal solution at every step so as to find the global optimal solution for the entire problem.



We are given certain coins valued at $1, $2, $5, and $10. Now you need to count the specified value using these coins.

Case 1: $16

The greedy procedure will be like this.

- Firstly, we'll take 1 coin for $10. So, now we are less by $6.
- Now, we'll take 1 coin of $5 and later, 1 coin of $1.

Why did we choose the $10 coin first, we could have chosen other coins as well. Because we're Greedy. We take the coin that has greater value. In this case, the answer came out to be just fine. What if we take another case?

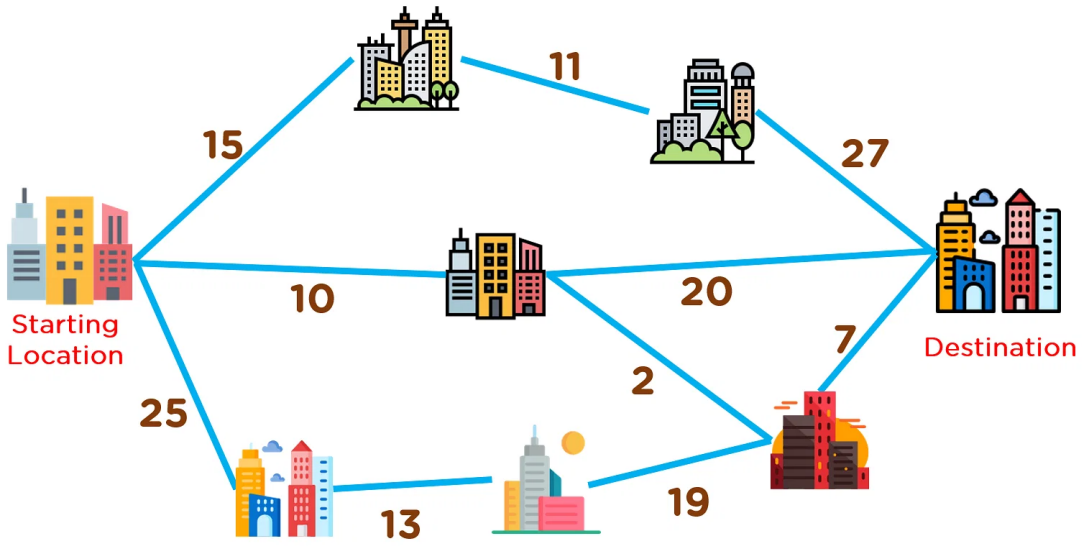
Case 2: $17

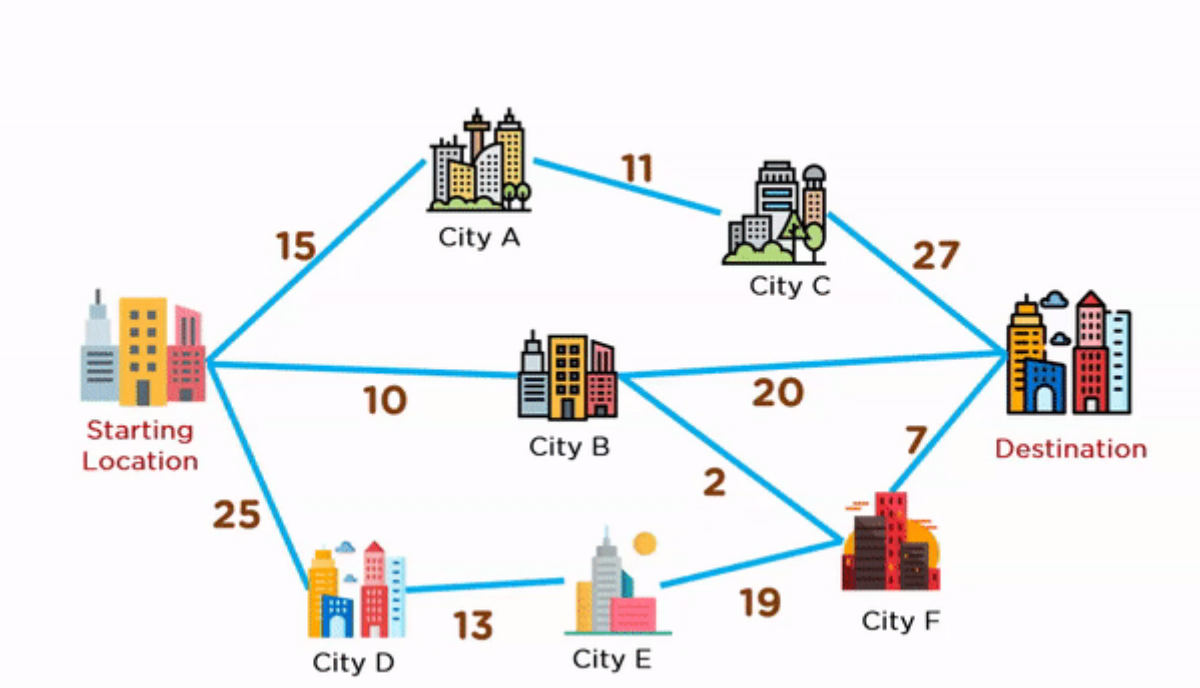
The procedure will be as follows:

- Take 1 $10 coin. Now, we are less by $7.

- We do not have any other choice, than choosing 7 coins of $1 each.

1. We are given an ordered list of resources like coins, tasks, etc.
2. We start with the greatest/largest valued resource.
3. Later, we keep on adding the greater valued resources to reach the optimized solution.

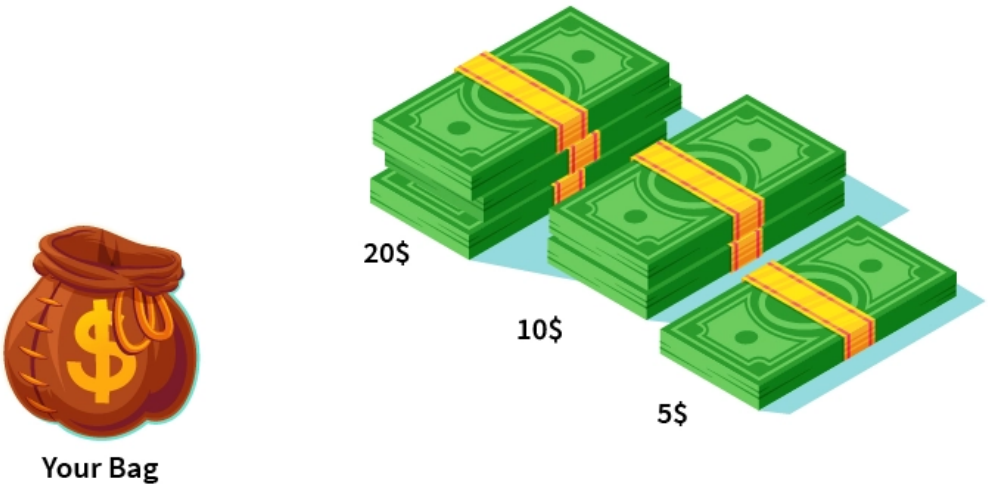




Greedy Solution: In order to tackle this problem, we need to maintain a graph structure. And for that graph structure, we'll have to create a tree structure, which will serve as the answer to this problem. The steps to generate this solution are given below:

- Start from the source vertex.
- Pick one vertex at a time with a minimum edge weight (distance) from the source vertex.
- Add the selected vertex to a tree structure if the connecting edge does not form a cycle.
- Keep adding adjacent fringe vertices to the tree until you reach the destination vertex.

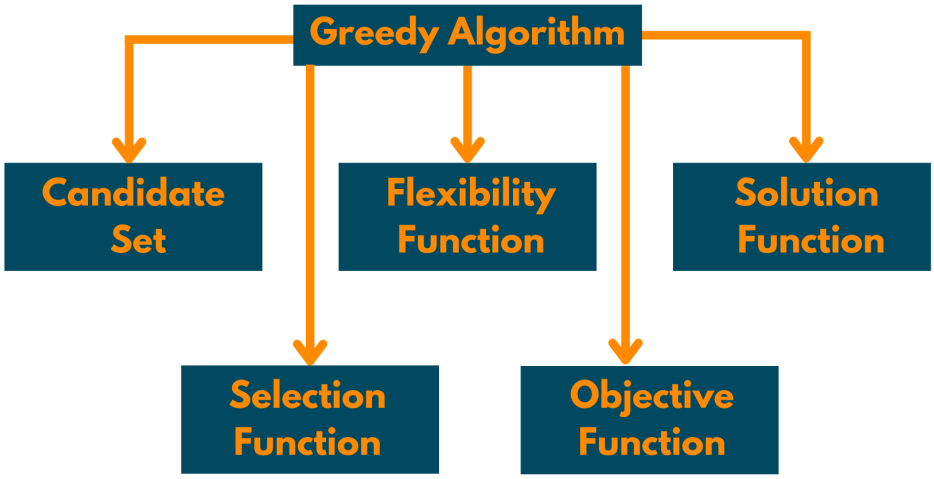
The animation given below explains how paths will be picked up in order to reach the destination city.



Components of Greedy Algorithm

The general components of this approach have been mentioned below:

1. **Candidate Set**: solution to the problem that has been created from the dataset.
2. **Selection Function**: a function that decides whether the element can be added as a solution or not.
3. **Feasibility Function**: checks whether the elements selected in the selection function & candidate set are feasible or not.
4. **Objective Function**: assigns value to the solution.
5. **Solution Function**: used to intimate the reaching of complete function.



Applications of Greedy Algorithm

1. Finding the **shortest path using Dijkstra**.
2. Finding a minimum spanning tree using **Prim's algorithm** or **Kruskal's algorithm**.
3. Solving fractional knapsack problem.

Activity Selection Problem

Activity Selection Problem: Given a set of activities, along with the starting and finishing time of each activity, find the maximum number of activities performed by a single person assuming that a person can only work on a single activity at a time.

```
Input: Following set of activities

(1, 4), (3, 5), (0, 6), (5, 7), (3, 8), (5, 9), (6, 10),
(8, 11), (8, 12), (2, 13), (12, 14)

Output:

(1, 4), (5, 7), (8, 11), (12, 14)
```

The activity selection problem is a problem concerning selecting non-conflicting activities to perform within a given time frame, given a set of activities each marked by a start and finish time. A classic application of this problem is scheduling a room

for multiple competing
events, each having its time requirements (start and end time).

Let's assume there exist `n` activities each being represented by a start time si and finish time `fj`. Two activities `i` and `j` are said to be non-conflicting if $s_i = f_j$ or $s_j = f_i$.

```
import java.util.*;
import java.util.stream.Collectors;

// A simple pair class to store the start and finish time
// of the activities
class Pair
{
    private int start, finish;

    public Pair(int start, int finish)
    {
        this.start = start;
        this.finish = finish;
    }

    public int getFinish() {
        return finish;
    }

    public int getStart() {
        return start;
    }

    @Override
    public String toString() {
        return "(" + getStart() + ", " + getFinish() + ")";
    }
}

class ActivitySelection
{
    // Activity selection problem
    public static List<Pair> selectActivity(List<Pair> activities)
    {
        // `k` keeps track of the index of the last selected activity
        int k = 0;

        // set to store the selected activities index
        Set<Integer> result = new HashSet<>();

        // select 0 as the first activity
        if (activities.size() > 0) {
            result.add(0);
        }

        // sort the activities according to their finishing time
```

```
        Collections.sort(activities, Comparator.comparingInt(Pair::getFinish));

        // start iterating from the second element of the
        // list up to its last element
        for (int i = 1; i < activities.size(); i++)
        {
            // if the start time of the i'th activity is greater or equal
            // to the finish time of the last selected activity, it
            // can be included in the activities list

            if (activities.get(i).getStart() >= activities.get(k).getFinish())
            {
                result.add(i);
                k = i;            // update `i` as the last selected activity
            }
        }

        return result.stream()
                .map(activities::get)
                .collect(Collectors.toList());
    }

    public static void main(String[] args)
    {
        // List of given jobs. Each job has an identifier, a deadline, and a
        // profit associated with it
        List<Pair> activities = Arrays.asList(new Pair(1, 4), new Pair(3, 5),
                new Pair(0, 6), new Pair(5, 7), new Pair(3, 8),
                new Pair(5, 9), new Pair(6, 10), new Pair(8, 11),
                new Pair(8, 12), new Pair(2, 13), new Pair(12, 14));

        List<Pair> result = selectActivity(activities);
        System.out.println(result);
    }
}
```

The time complexity of the above solution is O(n.log(n)), where `n` is the total number of activities. The auxiliary space required by the program is constant.

Find minimum platforms needed to avoid delay in the train arrival

Given a schedule containing the arrival and departure time of trains in a station, find the minimum number of platforms needed to avoid delay in any train's arrival.

```
Trains arrival   = { 2.00, 2.10, 3.00, 3.20, 3.50, 5.00 }
Trains departure = { 2.30, 3.40, 3.20, 4.30, 4.00, 5.20 }

The minimum platforms needed is 2
```

```
The train arrived at 2.00 on platform 1
The train arrived at 2.10 on platform 2
The train departed at 2.30 from platform 1
The train arrived at 3.00 on platform 1
The train departed at 3.20 from platform 1
The train arrived at 3.20 on platform 1
The train departed at 3.40 from platform 2
The train arrived at 3.50 on platform 2
The train departed at 4.00 from platform 2
The train departed at 4.30 from platform 1
The train arrived at 5.00 on platform 1
The train departed at 5.20 from platform 1
```

The idea is to merge the arrival and departure times of trains and consider them in sorted order. Maintain a counter to count the total number of trains present at the station at any point. The counter also represents the total number of platforms needed at that time.

- If the train is scheduled to arrive next, increase the counter by one and update the minimum platforms needed if the count is more than the minimum platforms needed so far.
- If the train is scheduled to depart next, decrease the counter by 1.

One special case needs to be handled – when two trains are scheduled to arrive and depart simultaneously, depart the train first.

```
import java.util.Arrays;

class Plateform
{
    // Function to find the minimum number of platforms needed
    // to avoid delay in any train arrival
    public static int findMinPlatforms(double[] arrival, double[] departure)
    {
        // sort arrival time of trains
        Arrays.sort(arrival);

        // sort departure time of trains
        Arrays.sort(departure);

        // maintains the count of trains
        int count = 0;

        // stores minimum platforms needed
        int platforms = 0;

        // take two indices for arrival and departure time
        int i = 0, j = 0;

        // run till all trains have arrived
```

```
        while (i < arrival.length)
        {
            // if a train is scheduled to arrive next
            if (arrival[i] < departure[j])
            {
                // increase the count of trains and update minimum
                // platforms if required
                platforms = Integer.max(platforms, ++count);

                // move the pointer to the next arrival
                i++;
            }

            // if the train is scheduled to depart next i.e.
            // `departure[j] < arrival[i]`, decrease trains' count
            // and move pointer `j` to the next departure.

            // If two trains are arriving and departing simultaneously,
            // i.e., `arrival[i] == departure[j]`, depart the train first
            else {
                count--;
                j++;
            }
        }

        return platforms;
    }

    public static void main(String[] args)
    {
        double[] arrival = { 2.00, 2.10, 3.00, 3.20, 3.50, 5.00 };
        double[] departure = { 2.30, 3.40, 3.20, 4.30, 4.00, 5.20 };

        System.out.print("The minimum platforms needed is "
                    + findMinPlatforms(arrival, departure));
    }
}
```

The time complexity of the above solution is O(n.log(n)) and doesn't require any extra space, where `n` is the total number of trains.

Job Sequencing Problem with Deadlines

Given a list of tasks with deadlines and total profit earned on completing a task, find the maximum profit earned by executing the tasks within the specified deadlines. Assume that each task takes one unit of time to complete, and a task can't execute beyond its deadline. Also, only a single task will be executed at a time.

For example, consider the following set of tasks with a deadline and the profit associated with it. If we choose tasks 1, 3, 4, 5, 6, 7, 8, and 9, we can achieve a

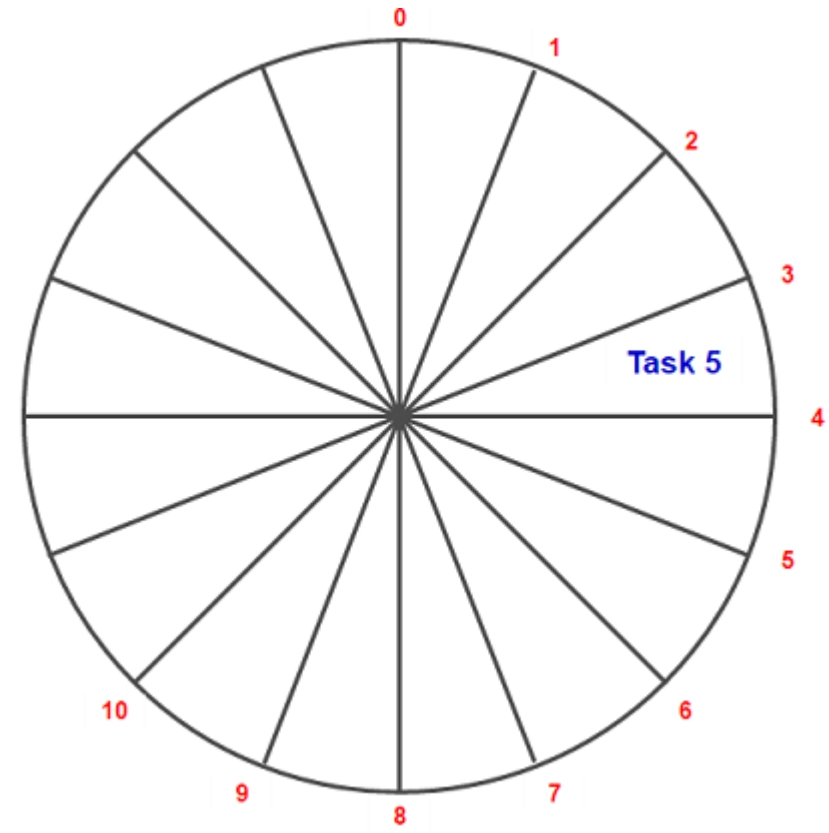
maximum profit of 109. Note that task 2 and task 10 are left out.

Tasks	Deadlines	Profit
1	9	15
2	2	2
3	5	18
4	7	1
5	4	25
6	2	20
7	5	8
8	7	10
9	4	12
10	3	5

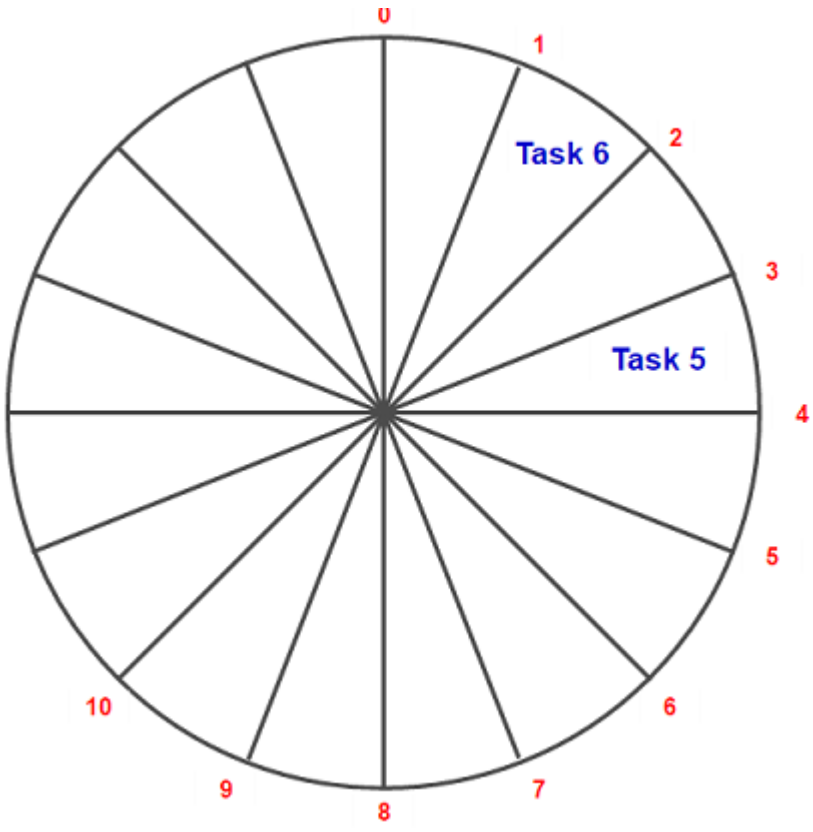
The following table shows the tasks arranged based on their associated profits. Here, task 5 has a maximum priority associated with it as it has a maximum gain of 30. Similarly, task 4 has the least priority. The greedy approach will consider the tasks in decreasing order of their priority.

Tasks	Deadlines	Profit (Maximum first)
5	4	25
6	2	20
3	5	18
1	9	15
9	4	12
8	7	10
7	5	8
10	3	5
2	2	2
4	7	1

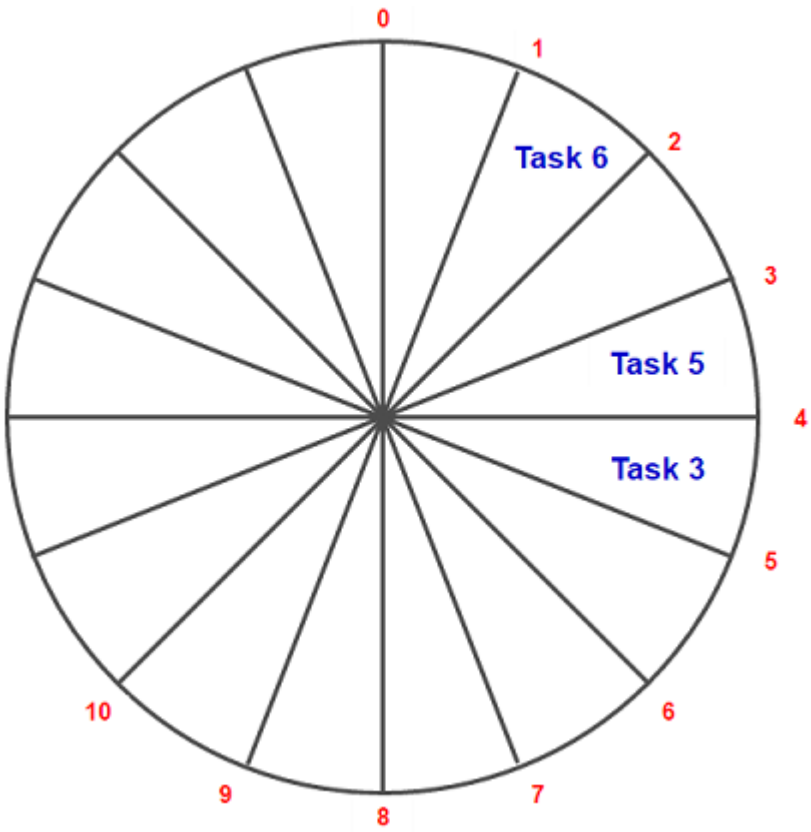
To demonstrate the greedy approach, let's consider the deadlines in the form of a circular structure, as shown below. A given task can fill each slot. Now, let's start allocated the tasks based on the deadlines. We will start with task 5, having a deadline of 4, and fill it empty slot `3-4`.



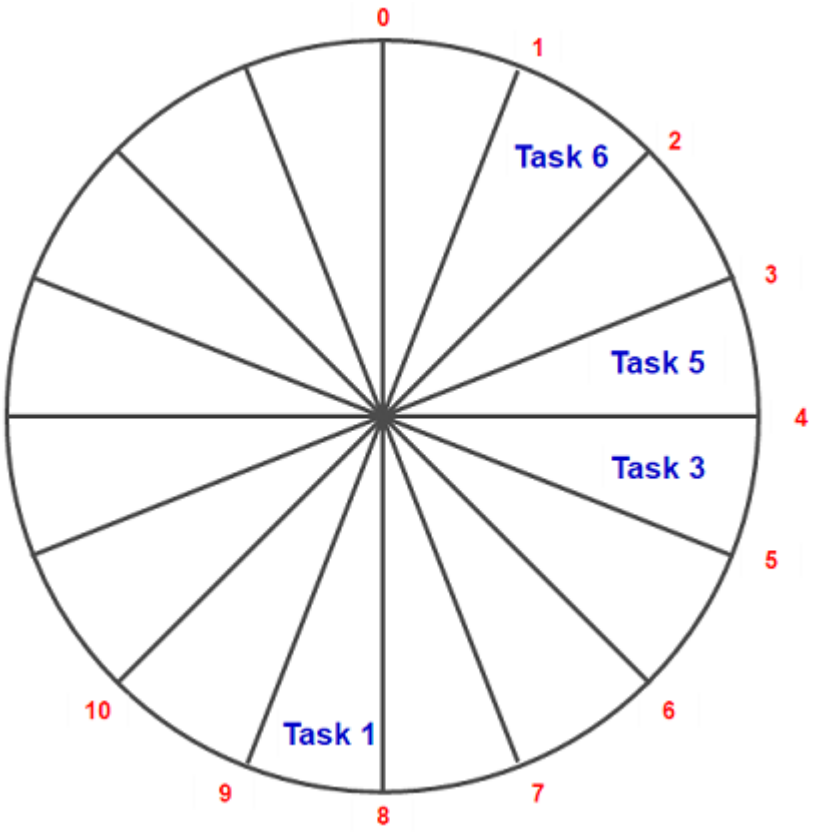
Next, consider task 6 having deadline 2 and fill it empty slot `1-2`.



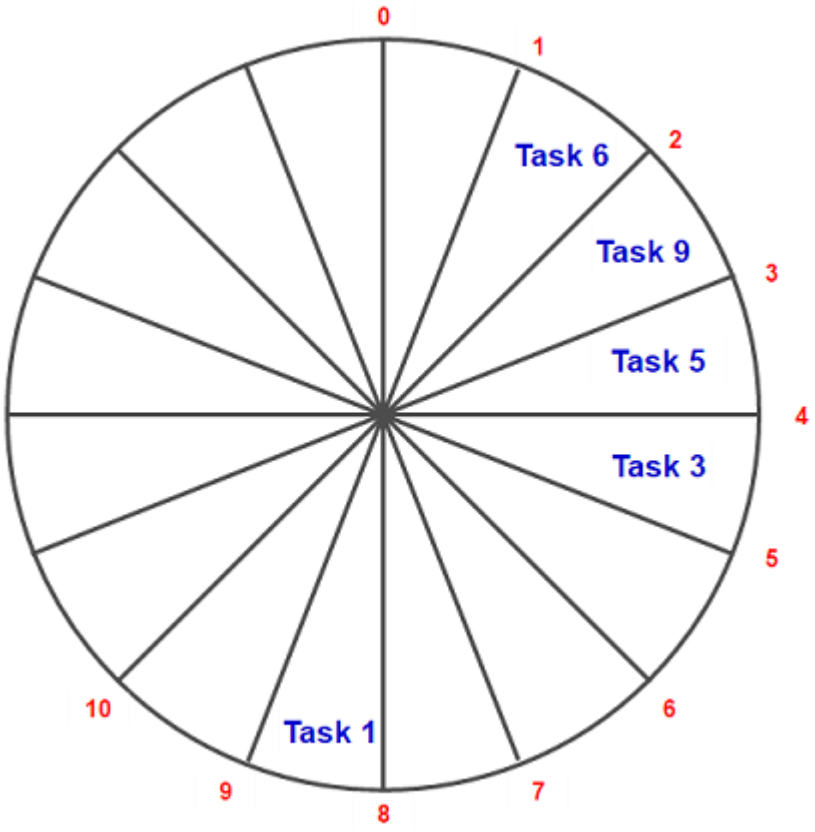
Next, consider task 3 having deadline 5 and fill it empty slot `4-5`.



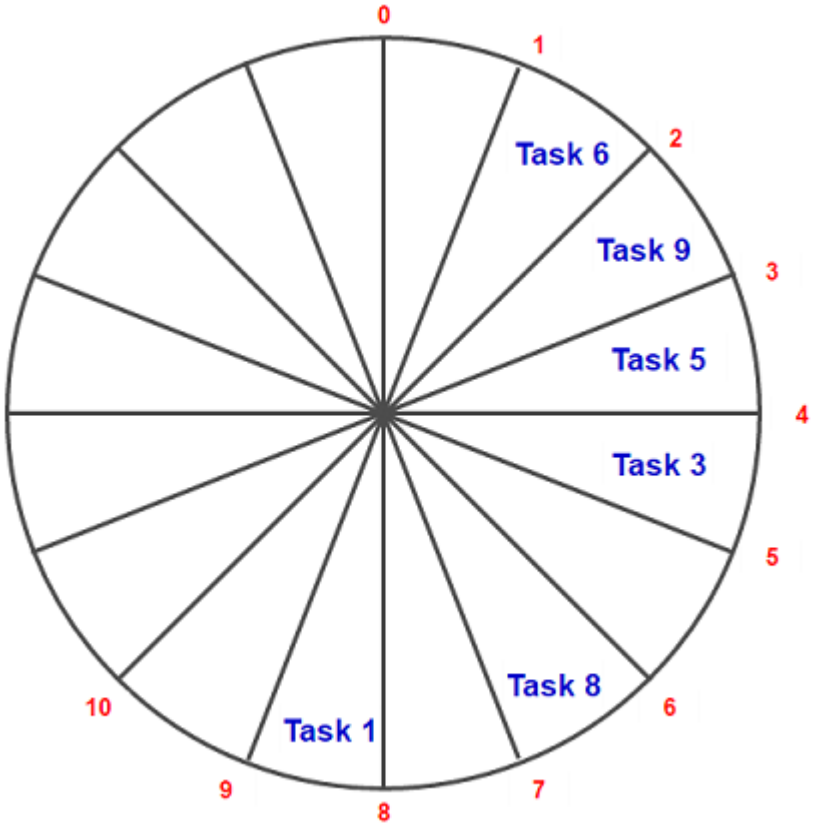
Next, consider task 1 having deadline 9 and fill it empty slot `8-9`.



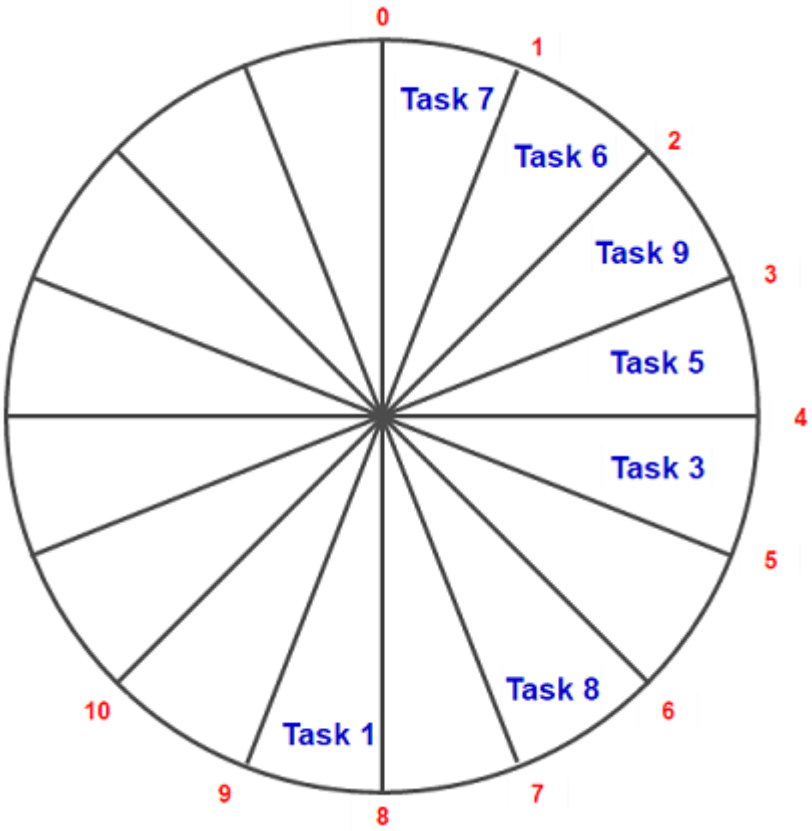
Next, consider task 9 having deadline 4. As slot `3-4` is already filled with task 5, we will consider the next free slot `2-3` and assign task 9.



Next, consider task 8 having deadline 7 and fill it empty slot `6-7`.

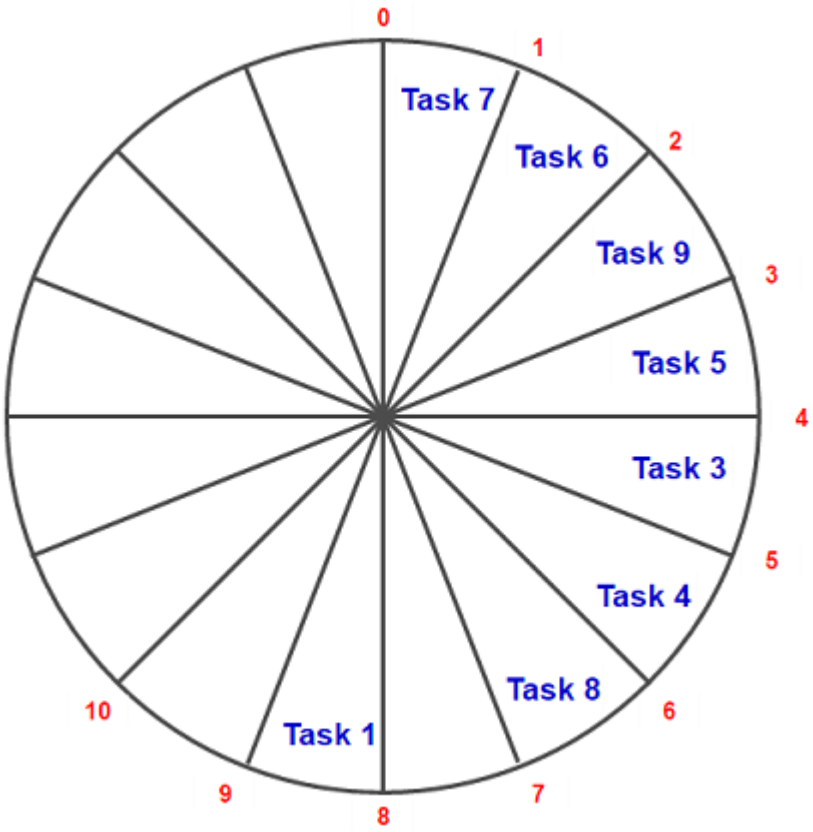


Next, consider task 7 having a deadline of 5. As slot `4-5` is already filled with task 3, we will consider the next free slot `0-1` and assign task 7.



Next, consider task 10 having a deadline of 3. Since all slots before deadline 3 are filled, ignore the task. Similarly, ignore the next task 2 having deadline 2.

The last task is task 4, having deadline 7, gets the next empty slot `6-5`.



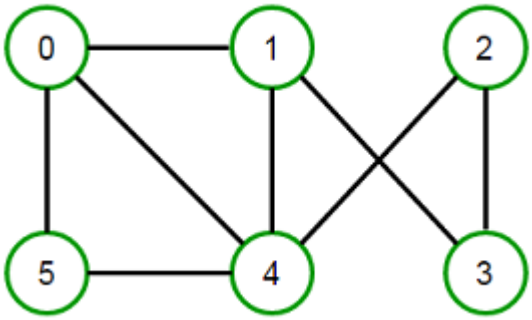

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;

// Data structure to store job details. Each job has an identifier,
// a deadline, and profit associated with it.
class Job
{
    public int taskId, deadline, profit;

    public Job(int taskId, int deadline, int profit)
    {
        this.taskId = taskId;
        this.deadline = deadline;
        this.profit = profit;
    }
}

class JobSchedule
{
    // Function to schedule jobs to maximize profit
    public static void scheduleJobs(List<Job> jobs, int T)
    {
        // stores the maximum profit that can be earned by scheduling jobs
        int profit = 0;

        // array to store used and unused slots info
        int[] slot = new int[T];
```

```
        Arrays.fill(slot, -1);

        // arrange the jobs in decreasing order of their profits
        Collections.sort(jobs, (a, b) -> b.profit - a.profit);

        // consider each job in decreasing order of their profits
        for (Job job: jobs)
        {
            // search for the next free slot and map the task to that slot
            for (int j = job.deadline - 1; j >= 0; j--)
            {
                if (j < T && slot[j] == -1)
                {
                    slot[j] = job.taskId;
                    profit += job.profit;
                    break;
                }
            }
        }

        // print the scheduled jobs
        System.out.println("The scheduled jobs are " +
                Arrays.stream(slot).filter(val -> val != -1).boxed()
                        .collect(Collectors.toList()));

        // print total profit that can be earned
        System.out.println("The total profit earned is " + profit);
    }

    public static void main(String[] args)
    {
        // List of given jobs. Each job has an identifier, a deadline, and
        // profit associated with it
        List<Job> jobs = Arrays.asList(
                new Job(1, 9, 15), new Job(2, 2, 2), new Job(3, 5, 18),
                new Job(4, 7, 1), new Job(5, 4, 25), new Job(6, 2, 20),
                new Job(7, 5, 8), new Job(8, 7, 10), new Job(9, 4, 12),
                new Job(10, 3, 5));

        // stores the maximum deadline that can be associated with a job
        final int T = 15;

        // schedule jobs and calculate the maximum profit
        scheduleJobs(jobs, T);
    }
}
```

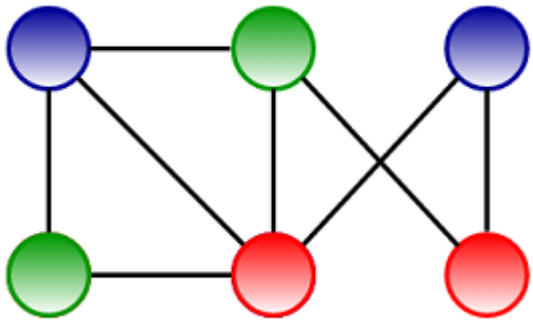
The time complexity of the above solution is O(n2), where `n` is the total number of jobs

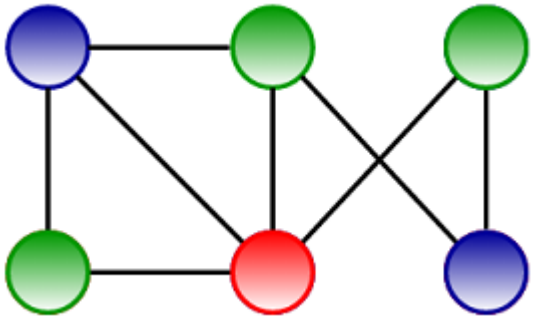
Graph Coloring Problem

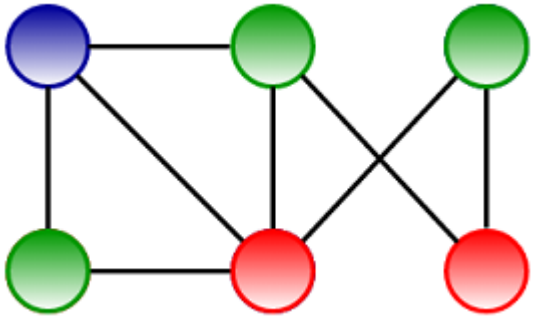
Graph coloring (also called vertex coloring) is a way of coloring a graph's vertices such that no two adjacent vertices share the same color.



We can color it in many ways by using the minimum of 3 colors.







K–colorable graph:

A coloring using at most k colors is called a (proper) *k*–coloring, and a graph that can be assigned a (proper) *k*–coloring is *k*–colorable.

K–chromatic graph:

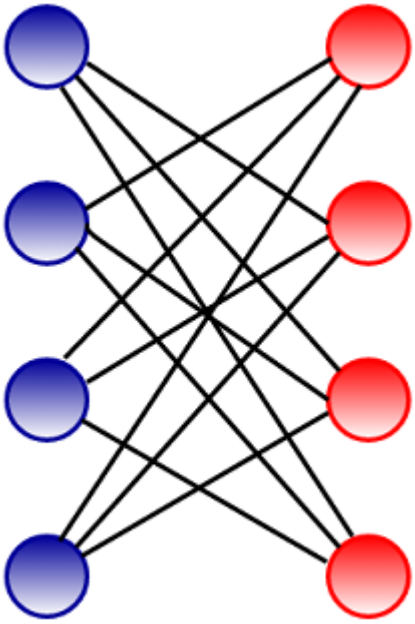
The smallest number of colors needed to color a graph G is called its chromatic number, and a graph that is *k*–chromatic if its chromatic number is exactly k.

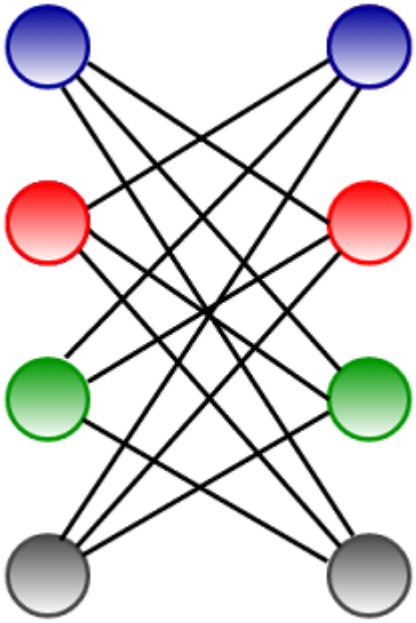
Brooks' theorem:

Brooks' theorem states that a connected graph can be colored with only x colors, where x
 is the maximum degree of any vertex in the graph except for complete graphs and graphs containing an odd length cycle, which requires $x+1$ colors.

Greedy coloring *considers the vertices of the graph in sequence and assigns each vertex its first available color*, i.e., vertices are considered in a specific order $v\,1$, $v\,2$, … $v\,n$, and $v\,i$ and assigned the smallest available color which is not used by any of $v\,i$'s neighbors.

Greedy coloring doesn't always use the minimum number of colors possible to color a graph. For a graph of maximum degree x, greedy coloring will use at most $x+1$ color. Greedy coloring can be arbitrarily bad; for example, the following crown graph (a complete bipartite graph), having n vertices, can be 2–colored (refer left image), but greedy coloring resulted in $n/2$ colors (refer right image).





```
import java.util.*;

// A class to store a graph edge
class Edge
{
    int source, dest;

    public Edge(int source, int dest)
    {
        this.source = source;
        this.dest = dest;
    }
}

// A class to represent a graph object
class Graph
{
    // A list of lists to represent an adjacency list
    List<List<Integer>> adjList = null;

    // Constructor
    Graph(List<Edge> edges, int n)
    {
        adjList = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            adjList.add(new ArrayList<>());
        }

        // add edges to the undirected graph
        for (Edge edge: edges)
        {
            int src = edge.source;
            int dest = edge.dest;
```

```
            adjList.get(src).add(dest);
            adjList.get(dest).add(src);
        }
    }
}

class GraphColoring
{
    // Add more colors for graphs with many more vertices
    private static String[] color = {
            "", "BLUE", "GREEN", "RED", "YELLOW", "ORANGE", "PINK",
            "BLACK", "BROWN", "WHITE", "PURPLE", "VOILET"
    };

    // Function to assign colors to vertices of a graph
    public static void colorGraph(Graph graph, int n)
    {
        // keep track of the color assigned to each vertex
        Map<Integer, Integer> result = new HashMap<>();

        // assign a color to vertex one by one
        for (int u = 0; u < n; u++)
        {
            // set to store the color of adjacent vertices of `u`
            Set<Integer> assigned = new TreeSet<>();

            // check colors of adjacent vertices of `u` and store them in a set
            for (int i: graph.adjList.get(u))
            {
                if (result.containsKey(i)) {
                    assigned.add(result.get(i));
                }
            }

            // check for the first free color
            int color = 1;
            for (Integer c: assigned)
            {
                if (color != c) {
                    break;
                }
                color++;
            }

            // assign vertex `u` the first available color
            result.put(u, color);
        }

        for (int v = 0; v < n; v++)
        {
            System.out.println("The color assigned to vertex " + v + " is "
                    + color[result.get(v)]);
        }
    }

    // Greedy coloring of a graph
    public static void main(String[] args)
    {
```

```
        // List of graph edges as per the above diagram
        List<Edge> edges = Arrays.asList(
                new Edge(0, 1), new Edge(0, 4), new Edge(0, 5), new Edge(4, 5),
                new Edge(1, 4), new Edge(1, 3), new Edge(2, 3), new Edge(2, 4)
        );

        // total number of nodes in the graph (labelled from 0 to 5)
        int n = 6;

        // build a graph from the given edges
        Graph graph = new Graph(edges, n);

        // color graph using the greedy algorithm
        colorGraph(graph, n);
    }
}
```