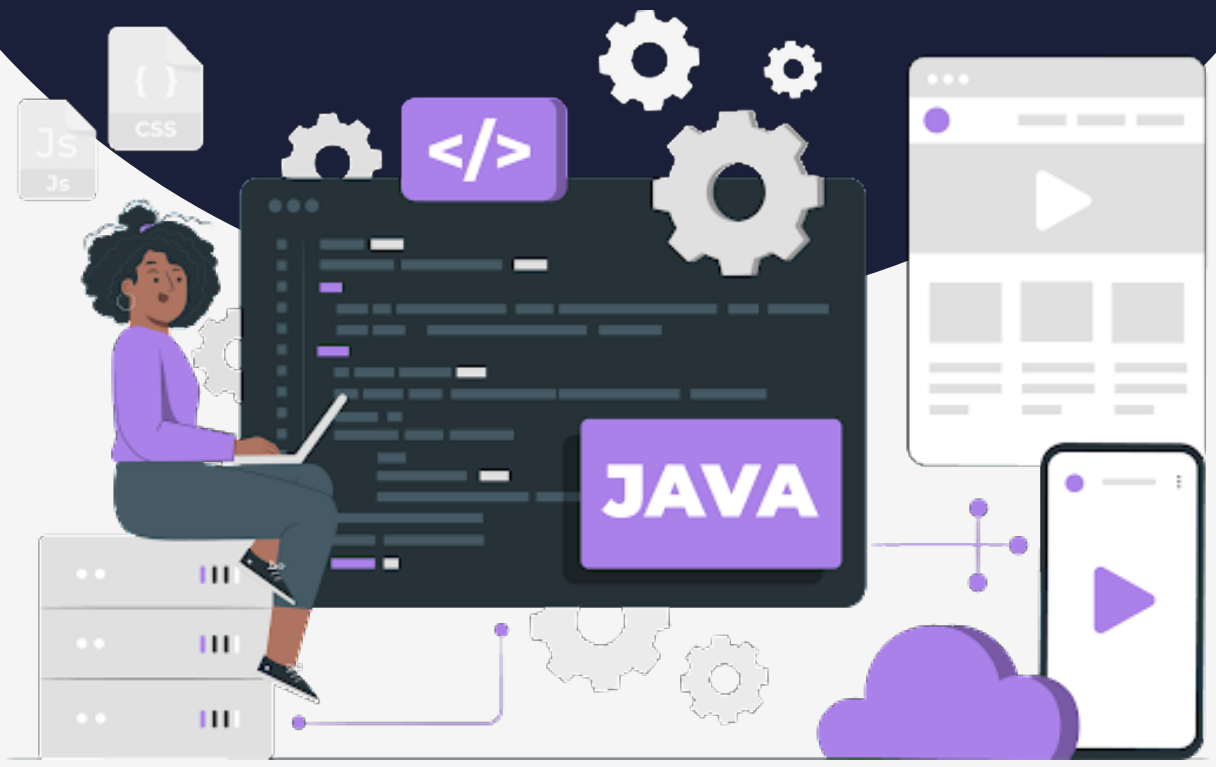
Lesson:

Greedy-1



What is Greedy Algorithm?

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.

A greedy algorithm is a problem-solving approach that involves making the best possible choice at each step, hoping to achieve the overall best solution.
It focuses on the local optimal choice without considering the future consequences.

This algorithm is helpful when a problem can be broken down into smaller parts, and the solution to each part contributes to solving the entire problem. It is commonly used to solve optimization problems where you need to find the best solution from a range of possibilities.

Greedy Choice Property: The Greedy choice property states that a globally optimal solution can be achieved by consistently making locally optimal choices. In other words, a Greedy algorithm selects the best option at each step, considering only the current information and not the future consequences. It iteratively makes these Greedy choices, reducing the original problem into smaller subproblems.

Optimal Substructure: Optimal substructure refers to a property of a problem where finding an optimal solution to the problem involves finding optimal solutions to its subproblems. This means that the overall optimal solution can be constructed by combining the optimal solutions of the smaller subproblems. By solving these subproblems and building up their solutions, we can solve larger and more complex problems.

Greedy Algorithm Structure

The general structure of a greedy algorithm can be summarized in the following steps:

1. Identify the problem as an optimization problem where we need to find the best solution among a set of possible solutions.
2. Determine the set of feasible solutions for the problem.
3. Identify the optimal substructure of the problem, meaning that the optimal solution to the problem can be constructed from the optimal solutions of its subproblems.
4. Develop a greedy strategy to construct a feasible solution step by step, making the locally optimal choice at each step.
5. Prove the correctness of the algorithm by showing that the locally optimal choices at each step lead to a globally optimal solution.

A basic structure:

1. Declare an empty result = 0.
2. We make a greedy choice to select, If the choice is feasible add it to the final result.
3. Return the result.

Characteristics of Greedy Algorithm

To apply the Greedy approach to a problem, the problem should possess the following characteristics:

1. **Ordered List of Resources:** The problem should involve a set of resources, such as profits, costs, or values, that can be arranged in a specific order.
2. **Maximum Resource Selection:** The Greedy algorithm selects the maximum value or resource from the given

ordered list. It aims to choose the option that provides the highest profit, value, or any other relevant measure.

By considering these characteristics, the Greedy algorithm can make locally optimal choices at each step, ultimately leading to an overall optimal or satisfactory solution.

Characteristic Components of Greedy Algorithm:

1. **Feasible Solution:** A subset of the given inputs that satisfies all the specified constraints of a problem is known as a "feasible solution". It is a solution that meets all the requirements of the problem.
2. **Optimal Solution:** The feasible solution that achieves the desired extremum (either minimizes or maximizes) of the objective function specified in the problem is called an "optimal solution". It is the best solution among all the feasible solutions.
3. **Feasibility Check:** It is a process of verifying whether a selected input or solution fulfills all the constraints mentioned in the problem. If it satisfies all the constraints, it is considered a feasible solution and added to the set of feasible solutions. Otherwise, it is rejected.
4. **Optimality Check:** It involves checking whether a selected input or solution produces either the minimum or maximum value of the objective function, while also satisfying all the specified constraints. If an element in the solution set achieves the desired extremum, it is added to the set of optimal solutions.
5. **Optimal Substructure Property:** The globally optimal solution to a problem includes the optimal sub-solutions within it. This means that finding the optimal solution for the overall problem involves combining the optimal solutions of smaller subproblems.
6. **Greedy Choice Property:** The globally optimal solution is constructed by selecting locally optimal choices. The Greedy approach applies certain locally optimal criteria to obtain a partial solution that appears to be the best at that particular moment. It then proceeds to find the solution for the remaining subproblem.

These characteristic components help define and guide the behavior of a Greedy algorithm, enabling it to find efficient and effective solutions for optimization problems.

The local decisions (or choices) must possess three characteristics as mentioned below:

1. **Feasibility:** The selected choice must fulfill the constraints.
2. **Optimality:** The selected choice must be the best at that stage (locally optimal choice).
3. **Irrevocability:** The selected choice cannot be changed once it is made.

Why choose Greedy Approach?

The Greedy approach is often chosen for problem-solving due to several reasons and trade-offs it offers. Here are a couple of significant factors:

1. **Immediate Feasible Solutions:** The Greedy approach allows for quickly obtaining feasible solutions. It focuses on making locally optimal choices at each step, without considering the overall consequences. This can be advantageous in situations where obtaining a feasible solution immediately is more important than finding the globally optimal solution.
2. **Recursive Division:** The Greedy approach can simplify problem-solving by dividing the problem recursively based on a condition, without the need to combine all the solutions. This can be seen in the activity selection problem, where activities are selected based on certain criteria, allowing for efficient scheduling without exhaustive combination calculations.

In summary, the Greedy approach prioritizes immediate feasibility and recursive division, making it a suitable choice for problems where finding a feasible solution quickly is crucial, and combining all solutions is not necessary.

Use of Greedy Algorithm:

The greedy algorithm is a useful method employed in optimization problems. It aims to find the best solution by making locally optimal choices at each step without considering future consequences. Here are some common applications of the greedy algorithm:

1. **Scheduling and Resource Allocation:** The greedy algorithm is effective in efficiently scheduling tasks or allocating resources.
2. **Minimum Spanning Trees:** It can be used to find the minimum spanning tree in a graph, connecting all vertices with the minimum total edge weight.
3. **Coin Change Problem:** The greedy algorithm solves the problem of making a change with the fewest coins by selecting the highest-value coin that fits the remaining amount.
4. **Huffman Coding:** It generates a prefix-free code for data compression by constructing a binary tree based on character frequencies.

It is worth noting that not all optimization problems can be solved using the greedy algorithm, and there may be cases where it produces suboptimal solutions. However, for many problems, it offers a quick and efficient approximation of the optimal solution.

Applications of Greedy Algorithms

1. **Optimal Solution Finding:** Greedy algorithms are commonly used to find optimal solutions in various problems such as activity selection, fractional knapsack, job sequencing, and Huffman coding. These algorithms help determine the best possible outcome based on specific criteria or objectives.
2. **Approximate Solutions for NP-Hard Problems:** Greedy algorithms can be applied to NP-Hard problems like the traveling salesman problem (TSP) to find approximate solutions that are close to optimal. While not guaranteed to find the absolute best solution, greedy algorithms can provide practical and efficient results.
3. **Network Design:** Greedy algorithms are utilized in network design tasks, including constructing minimum spanning trees, finding shortest paths, and optimizing maximum flow networks. These algorithms help in routing, resource allocation, and capacity planning, leading to efficient network designs.
4. **Machine Learning:** Greedy algorithms find applications in machine learning tasks such as feature selection, clustering, and classification. They are used to identify relevant features, optimize cluster or class selection, and improve the accuracy of machine learning models.
5. **Image Processing:** Greedy algorithms play a role in image processing tasks like compression, denoising, and segmentation. For instance, Huffman coding efficiently compresses images by encoding frequently occurring pixels.
6. **Combinatorial Optimization:** Greedy algorithms are employed in combinatorial optimization problems, including graph coloring, scheduling, and the traveling salesman problem. While these problems are usually computationally complex, greedy algorithms can provide practical and efficient solutions that are close to optimal.
7. **Game Theory:** Greedy algorithms are used in game theory applications, aiding in determining optimal strategies for games like chess or poker. By selecting the most promising moves or actions at each turn, based on the current game state, greedy algorithms contribute to decision-making.

8. **Financial Optimization:** Greedy algorithms find applications in financial domains such as portfolio optimization and risk management. They assist in selecting assets for portfolios based on historical data and market trends, aiming to maximize returns and manage risks effectively.

Advantages of the Greedy Approach

1. Easy to implement
2. Often have less time complexity
3. Can be used for optimization or approximate solutions for hard problems
4. Efficient solutions for problems with the greedy choice property
5. Faster than other optimization algorithms
6. Suitable as a heuristic or approximation algorithm
7. Applicable to a wide range of problems
8. Real-time solution for scheduling or resource allocation
9. Provides a starting point for more complex optimization algorithms
10. Can be combined with other algorithms to improve solution quality

Disadvantages of the Greedy Approach

1. Local optimal solution may not always be globally optimal
2. No guarantee of finding optimal solution
3. Relies heavily on problem structure and choice of criteria
4. May require preprocessing or transformation
5. Not suitable for order-dependent problems
6. May struggle with size or composition-dependent problems
7. Constraints on solution space can be challenging
8. Sensitivity to input changes can result in instability and unpredictability

Coding Questions

Q1 Fractional Knapsack

Given the weights and profits of N items, in the form of {profit, weight} put these items in a knapsack of capacity W to get the maximum total profit in the knapsack. In Fractional Knapsack, we can break items for maximizing the total value of the knapsack.

Input: arr[] = {{60, 10}, {100, 20}, {120, 30}}, W = 50

Output: 240

Explanation: By taking items of weight 10 and 20 kg and 2/3 fraction of 30 kg.
Hence total price will be 60+100+(2/3)(120) = 240

Approach:
The algorithm used in the provided code is known as the Fractional Knapsack algorithm. It is a greedy algorithm that selects items based on their profit-to-weight ratios, choosing the items with the highest ratios

first.

The Fractional Knapsack algorithm:

1. Calculate the profit-to-weight ratio for each item.
2. Sort the items in descending order based on their profit-to-weight ratios.
3. Initialize the totalProfit variable to 0.
4. Iterate through the sorted items till knapsack's remaining capacity > 0:
5. If the current item's weight is less than or equal to the remaining capacity of the knapsack, add the entire item to the knapsack and update the remaining capacity and totalProfit.
6. If the current item's weight is greater than the remaining capacity, calculate the fraction of the item that can fit into the knapsack (remaining capacity divided by item weight), add the fractional part of the item to the knapsack, update the remaining capacity, and update the totalProfit.
7. Return the totalProfit, which represents the maximum total profit achievable with the given capacity.

The Fractional Knapsack algorithm works in cases where fractions of items can be included in the knapsack to maximize the total profit. However, it does not guarantee an optimal solution in terms of selecting whole items only (i.e., without fractional parts).

Java Code: https://pastebin.com/Dyz43cFz

Output:

```
Enter the number of items: 3
Enter the capacity of the knapsack: 50
Enter the profit and weight of each item:
Item 1 - Profit: 60
       - Weight: 10
Item 2 - Profit: 100
       - Weight: 20
Item 3 - Profit: 120
       - Weight: 30

Maximum total profit in the knapsack: 240


...Program finished with exit code 0
Press ENTER to exit console.
```

```
Enter the number of items: 1
Enter the capacity of the knapsack: 10
Enter the profit and weight of each item:
Item 1 - Profit: 500
       - Weight: 30

Maximum total profit in the knapsack: 166.66666666666666


...Program finished with exit code 0
Press ENTER to exit console.
```

Time complexity: O(n log n), where n is the number of items. This is because the algorithm involves sorting the items based on their profit-to-weight ratios.

Space complexity: O(1) because it uses a constant amount of additional space for variables and temporary storage, regardless of the input size. The space required does not grow with the number of items, making it a space-efficient algorithm.

Q2 Maximum meetings in one room

There is one meeting room in a firm. There are N meetings in the form of (S[i], F[i]) where S[i] is the start time of meeting i and F[i] is the finish time of meeting i. The task is to find the maximum number of meetings that can be accommodated in the meeting room. Print all meeting numbers.

Constraints:

0 <= S[i], F[i] < 24

Input:

S[] = {1, 3, 0, 5, 8, 5}
F[] = {2, 4, 6, 7, 9, 9}

Output:

1 2 4 5

Approach:

The idea is to solve the problem using the greedy approach i.e. sort the meetings by their finish time and then start selecting meetings, starting with the one with least end time and then selecting other meetings such that the start time of the current meeting is greater than the end time of last meeting selected.

Steps involved:

1. Sort all pairs(Meetings) in increasing order of each pair's second number(Finish time).
2. Select the first meeting of the sorted pair as the first Meeting in the room and push it into the result vector and set a variable time_limit(say) with the second value(Finishing time) of the first selected meeting.
3. Iterate from the second pair to the last pair of the array and if the value of the first element(Starting time of meeting) of the current pair is greater than the previously selected pair's finish time (time_limit) then select the current pair and update the result vector (push selected meeting number into result vector) and variable time_limit.
4. Print the order of meeting from the result vector.

Java Code: https://pastebin.com/7j0kSNar

Output:

```
Enter the number of meetings: 6

Enter the start time of meeting 1: 1
Enter the end time of meeting 1: 2

Enter the start time of meeting 2: 3
Enter the end time of meeting 2: 4

Enter the start time of meeting 3: 0
Enter the end time of meeting 3: 6

Enter the start time of meeting 4: 5
Enter the end time of meeting 4: 7

Enter the start time of meeting 5: 8
Enter the end time of meeting 5: 9

Enter the start time of meeting 6: 5
Enter the end time of meeting 6: 9

The maximum number of meetings that can be accommodated is: 4
Meeting numbers: 1 2 4 5


...Program finished with exit code 0
Press ENTER to exit console.
```

Time complexity: O(n log n), where n is the number of meetings. This is because we sort the meetings based on their finish times, which has the time complexity of O(nlogn). After sorting, we iterate over the sorted meetings once to find the maximum number of meetings that can be accommodated, which takes O(n) time. So, overall time complexity is O(nlogn).

Space complexity: O(n) since we need to store the meetings in a list, and the size of the list is proportional to the number of meetings.

Q3 Activity Selection Problem (Homework)

Given N activities with their start and finish day given in arrays start[ ] and end[ ] respectively. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a given day.
Note: Duration of the activity includes both starting and ending day.

Input:
N = 4
start[] = {1, 3, 2, 5}
end[] = {2, 4, 3, 6}

Output:
3
Explanation:
A person can perform activities 1, 2 and 4.

Approach:
The greedy choice is to always pick the next activity whose finish time is least among the remaining activities and the start time is more than or equal to the finish time of the previously selected activity. We can sort the activities according to their finishing time so that we always consider the next activity as minimum finishing time activity.

1. Sort the activities according to their finishing time
2. Select the first activity from the sorted array and print it.
3. For the remaining activities in the sorted array, if the start time of this activity is greater than or equal to the finish time of the previously selected activity then select this activity and print it.

Java Code: https://pastebin.com/rLP4BCRa

Output:

```
Enter the number of activities: 4

Enter the start and end day of each activity:

Activity 1: 1 2
Activity 2: 3 4
Activity 3: 2 3
Activity 4: 5 6

The maximum number of activities that can be performed is: 4


...Program finished with exit code 0
Press ENTER to exit console.
```

```
Enter the number of activities: 2

Enter the start and end day of each activity:

Activity 1: 2 2
Activity 2: 1 2

The maximum number of activities that can be performed is: 1


...Program finished with exit code 0
Press ENTER to exit console.
```

Time complexity: O(n log n), where n is the number of activities. This is because the code sorts the activities based on their end time, which takes O(n log n) time using a comparison-based sorting algorithm. After sorting, the loop iterates through the activities once, which takes O(n) time. Therefore, the dominant time complexity is O(n log n).

Space complexity: O(n) since it uses an array of Activity objects to store the activities. The size of the array is directly proportional to the number of activities, so the space complexity grows linearly with the input size.

Q4 Check if it is possible to survive on Island

You are a person in an island. There is only one shop in this island, this shop is open on all days of the week except for Sunday. Consider following constraints:

S – Number of days you are required to survive.
N – Maximum unit of food you can buy each day.
M – Unit of food required each day to survive.

Currently, it's Monday, and you need to survive for the next S days.

Find the minimum number of days on which you need to buy food from the shop so that you can survive the next S days, or determine that it isn't possible to survive.

Input:
S = 10 N = 16 M = 2

Output:
Yes
Minimum days to buy food: 2

Explanation: One possible solution is to buy a box on the first day (Monday), it's sufficient to eat from this box up to 8th day (Monday) inclusive. Now, on the 9th day (Tuesday), you buy another box and use the food in it to survive the 9th and 10th day.

Input: 10 20 30

Output: No

Explanation: You can't survive even if you buy food because the maximum number of units you can buy in one day is less than the required food for one day.

Approach:
In this problem, the greedy approach of buying the food for some consecutive early days is the right direction. If we can survive for the first 7 days then we can survive any number of days for that we need to check two things:

1. Check whether we can survive one day or not.
2. (S >= 7) If we buy food in the first 6 days of the week and we can survive for the week i.e. total food we can buy in a week (6*N) is greater than or equal to total food we require to survive in a week (7*M) then we can survive.

Java Code: https://pastebin.com/ySA6qYVa

Output:

```
Enter the following values:
Number of days to survive: 10
Food consumption per day: 16
Food units per purchase: 2
==================================
Yes, you can survive for 10 days.
Minimum days to buy food: 2


...Program finished with exit code 0
Press ENTER to exit console.
```

```
Enter the following values:
Number of days to survive: 10
Food consumption per day: 20
Food units per purchase: 30
==================================
No, you cannot survive for 10 days.


...Program finished with exit code 0
Press ENTER to exit console.
```

Time complexity: O(1) because it performs a fixed number of operations regardless of the input size.
Space complexity: O(1) as it only uses a constant amount of additional space to store the input variables and local variables, regardless of the input size.