Lecture Notes
Data Structures and Algorithms

The Greedy Method for Algorithm Design

Overview

In this lecture, we will discuss the greedy approach to algorithm design. This approach involves making locally optimal decisions at each step, with the hope of finding a global solution.

Intuition: The greedy method might be used to help plan a trip on a budget. For example, we might use the greedy method to choose the cheapest flights and accommodation options, with the hope that these locally optimal decisions will lead to a globally optimal solution in terms of cost of the whole trip.

- **Definition:** A greedy algorithm is an algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.
- **Example:** One well-known example of a greedy algorithm is Dijkstra's algorithm for finding the shortest path between two nodes in a graph. This algorithm works by starting at the source node and continually choosing the next node to visit based on the current shortest known distance from the source.
- **Advantages:**
    - Greedy algorithms can be easy to understand and implement.
    - They can give good solutions in many cases.
    - They can be relatively efficient, since they only consider the current problem and not the entire problem space.
- **Disadvantages:**
    - Greedy algorithms do not always give the optimal solution.
    - They can be inflexible, since they only consider the current problem and not the long-term consequences of their decisions.
- **When to use:**
    - Greedy algorithms can be a good choice when the problem can be divided into subproblems that can be solved independently.
    - They can be a good choice when the locally optimal choice at each step leads to a globally optimal solution.
- **When not to use:**

    - Greedy algorithms may not be a good choice when the locally optimal choice at each step does not lead to a globally optimal solution.
    - They may not be a good choice when the problem requires a more flexible or dynamic approach.
- **Examples of problems that can be solved using a greedy approach:**
    - Scheduling tasks with deadlines
    - Selecting coins to make change
    - Huffman coding (data compression)

The Greedy Method

The greedy method is a strategy for solving optimization problems by making a series of locally optimal decisions, with the hope of reaching a globally optimal solution.

In other words, the greedy method involves making the best decision at each step, without worrying about the consequences of that decision on future steps. The hope is that, by making a sequence of good local decisions, we will end up with a global solution that is also optimal.

It's important to note that the greedy method does not always lead to a globally optimal solution. In some cases, the locally optimal decisions made at each step may not necessarily lead to a globally optimal solution. In these cases, we need to use a different algorithm or method to solve the problem.

There are a few general characteristics of problems that can be solved using the greedy method:

1. The problem has optimal substructures, meaning that the globally optimal solution can be obtained by combining locally optimal solutions.
2. There is a natural ordering among the choices that must be made in the problem, and it is possible to determine which choice is the best at each step.
3. The greedy method usually requires that the input data be presented in a specific order, such as sorted in increasing or decreasing order.

Example 1. Finding the minimum number of coins:

To demonstrate the general approach of the greedy method in Java, let's consider the problem of finding the **minimum number of coins** needed to make a certain amount of change. Here is a simple Java function that uses a greedy approach to solve this problem:

```java
public static int minCoins(int[] coins, int change) {
    // Sort the coins in decreasing order.
    Arrays.sort(coins, Comparator.reverseOrder());

    // Keep track of the number of coins used.
    int numCoins = 0;
```

```
  // Iterate through the coins, starting with the largest.
  for (int coin : coins) {
    // Add as many of this coin as needed to make the change.
    while (change >= coin) {
      change -= coin;
      numCoins++;
    }
  }

  return numCoins;
}
```

This function sorts the available coins in decreasing order and then iterates through the coins, starting with the largest. At each step, it adds as many of the current coin as needed to make the change, and then moves on to the next coin. This greedy approach is guaranteed to find a solution that uses the minimum number of coins, as long as the input coins are sorted in decreasing order.

Proof

To prove that this greedy approach is correct, we need to show that it always leads to an optimal solution.

- Suppose that we have made a series of locally optimal decisions, and we are left with a remaining amount of change R. We will show that there is always a coin C[i] in C such that C[i] <= R and adding C[i] to the solution is optimal.
- Since C is sorted in decreasing order, C[i] is the largest coin that is less than or equal to R. If we add C[i] to the solution, we will reduce the remaining amount of change by C[i], which is the largest possible reduction. This means that adding C[i] to the solution is a locally optimal decision.

Since we have made a series of locally optimal decisions, it follows that the overall solution is also optimal. Therefore, the greedy method always leads to an optimal solution for the coin change problem.

Complexity

The solution to the coin change problem that I provided has a time complexity of *O(n * m),* where n is the number of coins and m is the amount of change.

The function uses two nested loops to iterate through the coins and add up the minimum number of coins needed to make the change. The outer loop runs in O(n) time, and the inner loop runs in *O(m)* time, giving a total time complexity of *O(n * m)*.

The space complexity of the solution is *O(1)*, since it only uses a constant amount of extra space to store a few variables.

It's worth noting that this solution is not the most efficient way to solve the coin change problem. There are faster algorithms that can solve the problem in O(m) time, such as dynamic programming. However, the greedy approach that I provided is a simple and easy-to-understand solution that can be implemented quickly. It is also a good starting point for understanding the greedy method and how it can be applied to solve optimization problems.

Example 2. Non Overlapping Sets:

Suppose we have a set of intervals on the number line, and we want to select a maximum number of non-overlapping intervals. For example, given the following intervals:

```
[1, 3] [2, 4] [5, 7] [6, 8]
```

We can select the intervals [1, 3] and [5, 7] for a total of 2 intervals.

We can use the greedy method to solve this problem by sorting the intervals by their starting point and then iterating through the intervals, selecting the ones that do not overlap with any of the previously selected intervals. Here is a simple Java function that uses this approach:

```java
public static int maxNonOverlapping(int[][] intervals) {
    // Sort the intervals by starting point.
    Arrays.sort(intervals, Comparator.comparingInt(a -> a[0]));

    // Keep track of the current end point.
    int end = Integer.MIN_VALUE;

    // Keep track of the number of intervals selected.
    int count = 0;

    // Iterate through the intervals.
    for (int[] interval : intervals) {
      // If the interval starts after the current end point, select
it.
      if (interval[0] >= end) {
        end = interval[1];
        count++;
      }
    }

    return count;
  }
```

This function first sorts the intervals by their starting point and then iterates through the intervals, keeping track of the current end point. If the current interval starts after the current end point, it selects the interval and updates the end point.

Proof

To prove that this greedy approach is correct, we need to show that it always leads to an optimal solution.

- Suppose that we have made a series of locally optimal decisions, and we are at a point where we need to decide whether to select interval I[j] or not. We will show that selecting I[j] is always optimal.
- Since I[j] starts earliest among all the intervals that have not been selected, it follows that I[j] does not overlap with any of the previously selected intervals. This means that selecting I[j] is a locally optimal decision.

Since we have made a series of locally optimal decisions, it follows that the overall solution is also optimal. Therefore, the greedy method always leads to an optimal solution for the overlapping sets problem.

Complexity

The solution to the overlapping sets problem that I provided has a time complexity of O(n * log n), where n is the number of intervals.

The function first sorts the intervals by their starting point, which takes O(n * log n) time. Then it iterates through the intervals, which takes O(n) time. Therefore, the overall time complexity of the function is O(n * log n).

The space complexity of the solution is O(1), since it only uses a constant amount of extra space to store a few variables.

Example 3. A problem that cannot be solved Greedily:

The Knapsack Problem: Given a set of items, each with a weight and a value, determine the subset of items that maximizes the total value, subject to a constraint on the total weight.

For example, suppose we have the following items:

Item	**Weight**	**Value**
A	2	3
B	3	4
C	4	5
D	5	6

And we have a knapsack with a maximum capacity of 9. We want to maximize the total value of the items that we put in the knapsack.

If we try to use the greedy method to solve this problem, we might make the following locally optimal decisions:

1. Select the item with the highest value-to-weight ratio.
2. Select the item with the next highest value-to-weight ratio, as long as it does not exceed the maximum weight.
3. Repeat until no more items can be selected.

However, this approach does not always lead to an optimal solution.

If we follow the greedy method outlined above, we might select items A and B, for a total value of 7 and a total weight of 5.

However, the optimal solution is to select items C and D, for a total value of 11 and a total weight of 9.

This example shows that the greedy method does not always lead to an optimal solution for the knapsack problem, because the best local decision does not always lead to a globally optimal solution.