Greedy Algorithm

1. Introduction

1.1 What is Greedy Algorithm?

A greedy algorithm is an approach to problem-solving that involves making locally optimal choices at each step with the goal of finding a global optimum. In other words, a greedy algorithm makes the best possible decision at the current moment without considering the long-term effects or consequences of that decision.

These algorithms are used in a variety of applications, from scheduling and resource allocation to graph traversal and data compression.

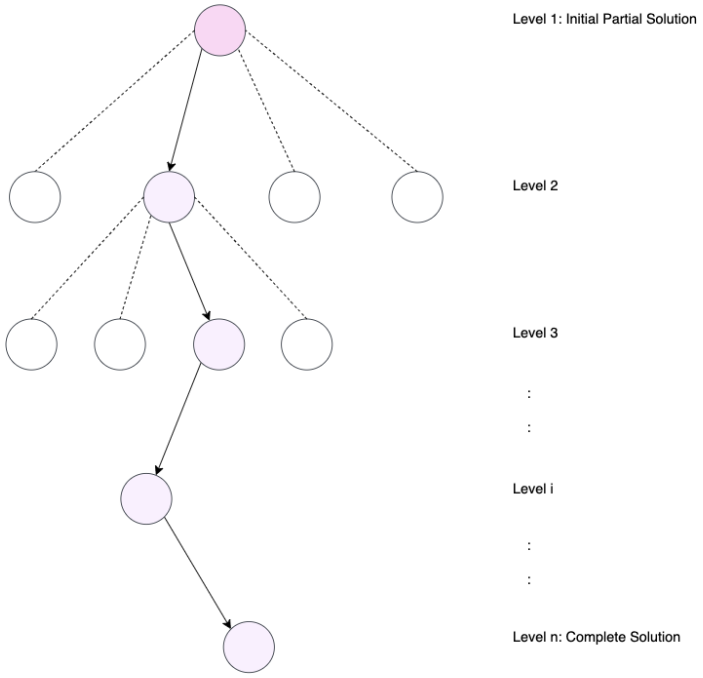


Fig 1: Representation of Basic Greedy Algorithm

1.1.1 Specifics:

1. A candidate set, from which a solution is created
2. A selection function, which chooses the best candidate to be added to the solution
3. A feasibility function that is used to determine if a candidate can be used to contribute to a solution
4. An objective function which assigns a value to a solution, or a partial solution
5. A solution function, which will indicate when we have discovered a complete solution

1.2 How do Greedy Algorithms Work?

At a high level, a greedy algorithm works by making the locally optimal choice at each step, with the hope that this will lead to a globally optimal solution. This means that at each step of the algorithm, the choice that appears to be the best is made without considering the future consequences. The algorithm then proceeds to the next step, using the same principle. This process continues until a solution is found.

The basic steps involved in a greedy algorithm are as follows:

1. **Identify the Optimal Subproblem:** Identify the subproblem that can be solved using a locally optimal choice.
2. **Make a Choice:** Make the locally optimal choice for the identified subproblem.
3. **Reduce the Problem:** Reduce the problem to a smaller subproblem and solve it recursively.
4. **Combine Solutions:** Combine the solutions obtained from each subproblem to obtain the final solution.

1.2.1 Control Abstraction of Greedy Method:

```
Greedy Algorithm(a, n) //a[1:n] contains n inputs {
 solution = Φ //initialize the solution
 for i = 1 to n do {
   x = Select(a)
   if feasible(solution, x) then
     solution = union(solution, x)
   }
 return solution
}
```

1.2.2 Illustration:

To illustrate how a greedy algorithm works, let's consider the following problem:

Problem (Coin Change Problem): You are given a set of coins with different denominations, and you want to make change for a certain amount of money using the minimum number of coins. How do you do it?

One way to solve this problem is to use a greedy algorithm. Here's how it works:

1. Sort the coins in descending order of their denominations.
2. Start with the largest denomination coin and keep adding it to the solution until you can't add it anymore without exceeding the target amount.
3. Repeat this process with the next largest denomination coin until the target amount is reached.

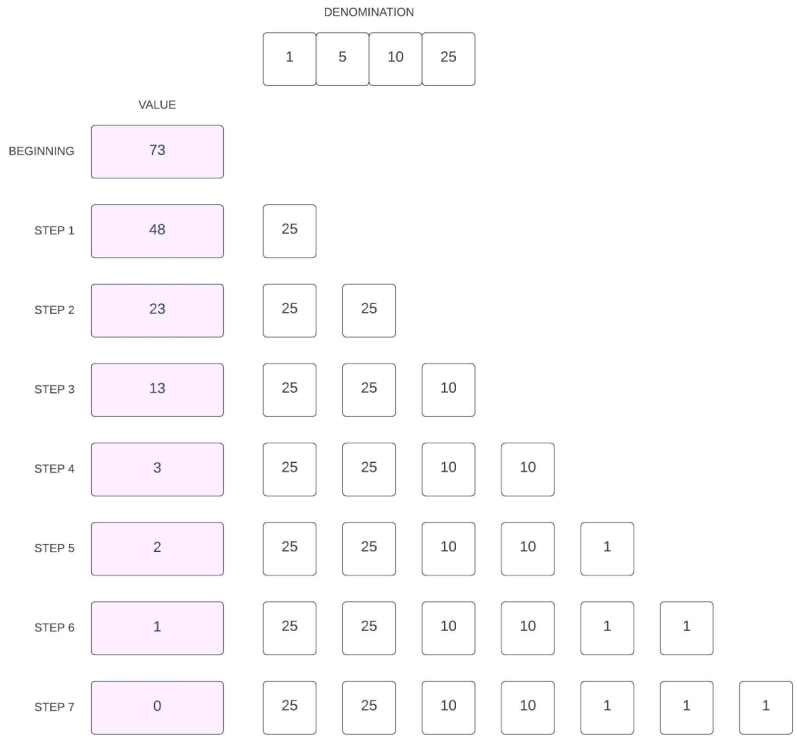
Let's consider an example to understand how the algorithm works in practice.

Example: Suppose you have the following set of coins: {1, 5, 10, 25}. You want to make change for 73 cents using the minimum number of coins. How do you do it?

1. Sort the coins in descending order: {25, 10, 5, 1}.
2. Start with the largest denomination coin (25) and keep adding it to the solution until you can't add it anymore without exceeding the target amount. In this case, you add 2 coins of 25, for a total of 50 cents.

3. Repeat this process with the next largest denomination coin (10). Add 2 coins of 10, for a total of 70 cents.
4. Finally, add 1 coin of 1 to reach the target amount of 73 cents. The solution is {25, 25, 10, 10, 1, 1, 1}.

Below is an illustration that depicts the overall process:



As you can see, at each step we use the highest possible coin from the denominations. At last, we are able to reach the value of 73 just by using 7 coins.

```
[ ]  def CoinChange(coins, target):
       # Sort coins in descending order of denominations
       coins = sorted(coins, reverse=True)

       # Initialize solution list and current amount
       solution = []
       current_amount = 0

       # Greedy approach: add largest coins first until target is reached
       for coin in coins:
         while current_amount + coin <= target:
           solution.append(coin)
           current_amount += coin

       # Return solution list
       return solution
```

```
[ ]  coins = [25, 10, 5, 1]
     target = 73

     print("Solution set of coins is:", CoinChange(coins, target))
```

```
     Solution set of coins is: [25, 25, 10, 10, 1, 1, 1]
```

As we can see, the greedy algorithm provides an optimal solution in this case, using the minimum number of coins.

1.3 When to Use Greedy Algorithms?

While greedy algorithms can be very powerful, they are not always the best choice for solving a given problem. In particular, greedy algorithms are best suited for problems that have the following properties:

1. **Optimal Substructure:** The problem can be broken down into smaller subproblems, each of which can be solved independently. Moreover, the optimal solution to the problem can be constructed from the optimal solutions to the subproblems.
2. **Greedy Choice Property:** The locally optimal choice at each step leads to a globally optimal solution.

If a problem has these properties, a greedy algorithm can be a very efficient and effective way to solve it. However, if the problem lacks one or both of these properties, a different algorithm may be needed.

1.3.1 Greedy Analysis:

1. Greedy's first step leads to an **optimum solution:** Show that there is an optimum solution leading from the first step of Greedy and then use induction.
   Example: Interval Scheduling.
2. **Greedy algorithm stays ahead:** Show that after each step the solution of the greedy algorithm is at least as good as the solution of any other algorithm.
   Example: Interval scheduling.
3. **Structural property of solution:** Observe some structural bound of every solution to the problem, and show that greedy algorithm achieves this bound.
   Example: Interval Partitioning.
4. **Exchange argument:** Gradually transform any optimal solution to the one produced by the greedy algorithm, without hurting its optimality.
   Example: Minimizing lateness.

Here's a list of some common greedy algorithms:

- Activity Selection Problem
- Dijkstra's Shortest Path Algorithm
- Huffman Coding
- Kruskal's Minimum Spanning Tree Algorithm
- Prim's Minimum Spanning Tree Algorithm
- Coin Change Problem
- Fractional Knapsack Problem
- Job Sequencing Problem
- Minimum Vertex Cover Problem

- Interval Scheduling Problem
- Set Cover Problem
- Maximum Discount Problem
- Cluster Analysis
- Set Partitioning Problem
- Shortest Superstring Problem
- Sum of Subset Problem
- Knapsack Problem

2. Time Complexity

The time complexity of a greedy algorithm depends on the specific problem being solved and the implementation of the algorithm. However, in general, the time complexity of a greedy algorithm can be expressed as O(n log n), where n is the size of the input.

This is because most greedy algorithms involve sorting the input data in some way, and sorting takes O(n log n) time in the worst case. However, some greedy algorithms may have a better or worse time complexity depending on the specifics of the problem.

It is important to note that the time complexity of a greedy algorithm does not always correlate with its effectiveness. A greedy algorithm with a higher time complexity may still be more effective than a faster algorithm in certain situations.

2.1 Worst Case Analysis:

In the worst case, a greedy algorithm may not always produce an optimal solution. This is because a greedy algorithm makes locally optimal choices at each step without considering the long-term consequences.

For example, consider the following problem:
Problem (Fractional Knapsack Problem): You have a knapsack with a maximum weight capacity and a set of items with different weights and values. You want to fill the knapsack with items that have the highest total value without exceeding the weight capacity. How do you do it?

A greedy algorithm for this problem would be to sort the items in descending order of their value-to-weight ratio and then add as many items as possible to the knapsack starting from the highest value-to-weight ratio item. However, this algorithm may not always produce an optimal solution.

Consider the following scenario:
Knapsack capacity: 30
Item 1: Weight = 30, Value = 300
Item 2: Weight = 20, Value = 200
Item 3: Weight = 10, Value = 150

The greedy algorithm would select Item 1 first, as it has the highest value-to-weight ratio. However, adding Item 1 to the knapsack would leave only 30 units of weight capacity, which is not enough to add either of the other two items. Therefore, the greedy algorithm would select only Item 1, resulting in a total value of 300. However, the optimal solution would be to select Items 2 and 3, which have a total value of 350.

3. Advantages and Disadvantages of Greedy Algorithms

3.1 Advantages:

1. **Simplicity:** Greedy algorithms are generally simple to understand and implement. They are easy to code and do not require complicated data structures or algorithms.
2. **Speed:** Greedy algorithms are usually very fast because they make locally optimal choices at each step. They do not require exhaustive search of all possible solutions, which can be computationally expensive.
3. **Efficiency:** Greedy algorithms are efficient in terms of both time and memory requirements. They use minimal resources and can be applied to large datasets.
4. **Approximation:** Greedy algorithms can provide an approximate solution to a problem in situations where finding an exact solution is not feasible. This is because greedy algorithms often provide a solution that is very close to the optimal solution.
5. **Flexibility:** Greedy algorithms can be applied to a wide range of problems. They can be used to solve optimization problems, graph problems, and other types of problems that involve finding the best possible solution.
6. **Scalability:** Greedy algorithms are easily scalable to large datasets. They can be used to solve problems with millions of data points in a reasonable amount of time.

3.2 Disadvantages:

1. **Non-Optimal Solutions:** Greedy algorithms do not always provide the optimal solution to a problem. They make locally optimal choices at each step, which may not always lead to the globally optimal solution. Therefore, greedy algorithms may produce suboptimal or even incorrect results in some cases.
2. **Infeasibility:** Greedy algorithms may not be feasible for some problems. For example, if the problem does not have the optimal substructure or the greedy choice property, then greedy algorithms may not be suitable for solving it.
3. **Lack of Flexibility:** Greedy algorithms can be inflexible in their approach to problem-solving. They make decisions based on a fixed set of rules or criteria, which may not be adaptable to different situations or datasets.
4. **Limited Scope:** Greedy algorithms are typically suitable for solving optimization problems, but may not be applicable to other types of problems. For example, they may not be suitable for problems that involve combinatorial optimization or dynamic programming.
5. **Difficult to Design:** Greedy algorithms can be difficult to design, especially for complex problems. It may be challenging to identify the optimal subproblem and make the locally optimal choice at each step.
6. **Difficulty in Analyzing:** It can be difficult to analyze the performance of a greedy algorithm. Unlike dynamic programming or other optimization techniques, there may not be a clear and systematic way to determine the time or space complexity of a greedy algorithm.

4. References

Greedy Algorithms:
https://brilliant.org/wiki/greedy-algorithm/

Greedy Algorithm:
https://www.programiz.com/dsa/greedy-algorithm#:~:text=A%20greedy%20algorithm%20is%20an,in%20a%20top%2Ddown%20approach.

What is Greedy Algorithm: Example, Applications, Limitations and More:
https://www.simplilearn.com/tutorials/data-structure-tutorial/greedy-algorithm

Worst-case analysis of greedy algorithms for the unbounded knapsack, subset-sum and partition problems:
https://www.sciencedirect.com/science/article/abs/pii/016763779390072O

When to Use Greedy Algorithms – And When to Avoid Them [With Example Problems]:
https://www.freecodecamp.org/news/when-to-use-greedy-algorithms/