Learning Objectives

Learners will be able to…

- **Define greedy algorithms**
- **Identify situations where a greedy approach is appropriate**
- **Implement a basic greedy solution for the coin change problem**
- **Define the knapsack problem and differentiate between its 0/1 and fractional versions.**

Greedy Algorithms

Building Up Instead of Breaking Down

Greedy algorithms build up a solution piece by piece, always choosing the next piece that offers the most immediate benefit or profit. This approach is mainly used to solve optimization problems.

Imagine you're at a buffet. A greedy approach would be to always pick the dish that looks most appetizing right now without thinking about what you might choose next or if you're leaving enough room for dessert. You're aiming to get the best immediate satisfaction with each choice.

This can sometimes lead to suboptimal solutions, but it can also be very efficient for problems that have a lot of local optima. Some of the benefits include:

- **Efficiency**: Greedy algorithms often offer a faster solution because they make decisions based on current information without worrying about the consequences. This means they can often reach a solution faster than methods that consider all possibilities before moving forward.
- **Simplicity**: Greedy algorithms are generally easier to describe and implement. They make the most obvious choice at each step of the decision-making process.
- **Optimality**: For some problems, greedy algorithms can find the global optimal solution. However, it's worth noting that while a greedy approach can provide a solution that's good enough, it does not always guarantee the best possible solution for all problems.

Binary Search vs Greedy Algorithm

Binary search **is not** a greedy algorithm. While both binary search and greedy algorithms make decisions at each step, the rationale and purpose behind those decisions are different.

Binary search is a divide-and-conquer algorithm. In binary search, the goal is to find a specific value in a sorted list. At each step, binary search divides the list in half by comparing the target value to the middle element:

- If the target value is equal to the middle element, you have found the value.
- If the target value is less than the middle element, continue the search on the left half.
- If the target value is greater than the middle element, continue the search on the right half.

This division continues until the value is found or the list is exhausted.

A greedy algorithm, on the other hand, makes the best local choice at each step with the hopes of finding a global optimum. It does not divide the problem into subproblems like binary search but instead makes a series of choices.

While both methods can be efficient for their respective problems, their underlying strategies and purposes are distinct. Binary search is specifically designed for searching in sorted collections, while greedy algorithms are a general approach for solving optimization problems. In some cases, binary search can be used as part of a greedy algorithm. However, binary search itself is not a greedy algorithm.

Coin Change Problem

Greedy Algorithm Example

Now that we know what a greedy algorithm is not, lets try to tackle an example of one that is. The main example, that is often introduced when working with a greedy algorithms is the coin change problem.

The **coin change problem** is a problem where you have a certain amount of money and you need to find the minimum number of coins that you need to make up that amount of money. The greedy algorithm for the coin change problem works by choosing the largest coin that does not exceed the amount of money that you need.

Let's walk through the coin change problem using a greedy approach in Java. Note that this greedy method only works when the coin denominations have certain properties. For instance, with US coin denominations (1, 5, 10, 25), the greedy method works. However, it's not always optimal for other denominations.

Code

Given different coin denominations and a total amount, find the minimum number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return `-1`.

Here are the steps to solve this problem with a greedy approach:

- Always select the largest coin denomination less than the remaining total.
- Deduct the value of the coin from the total.
- Repeat the process until the total is zero or no coins can be taken.

Start by creating the `CoinChangeGreedy` class. **Note**, we are importing the `Arrays` utility so we can sort an array. In the class, create the `coinChange()` method that takes an array of integers (the denominations for the coins) and another integer (the target amount). We also need a `main()` method.

```java
import java.util.Arrays;

public class CoinChangeGreedy {

    public static int coinChange(int[] coins, int amount) {

    }

    public static void main(String[] args) {

    }
}
```

In the `main()` method, create an array of integers that has the values for the coin denominations, an integer for the target amount, and a second integer for the result of our greedy algorithm. Finally, print out some context for the results.

```java
    public static void main(String[] args) {
        int[] coins = {1, 5, 10, 25};  // US coin denominations
        int amount = 63;
        int result = coinChange(coins, amount);

        if (result != -1) {
            System.out.println("Minimum coins needed: " +
        result);
        } else {
            System.out.println("The amount cannot be represented
        using the given denominations.");
        }
    }
```

In the `coinChange()` method, first sort the array of coins into ascending order. Create a variable to count the number of coins used and initialize its value to `0`. Then calculate the length of the array of coins.

```java
    public static int coinChange(int[] coins, int amount) {
        // Sort the coins in descending order
        Arrays.sort(coins);
        int count = 0;
        int n = coins.length;

    }
```

Remember, greedy algorithms select for the most immediate profit. In this case, that means the algorithm should as many of the coins with the largest denomination. So, we need to iterate backwards over the array `coins`. If we finish making change before we finish iterating over `coins`, then we want to stop the loop. There is no need to continue if the correct amount has been created.

```
    public static int coinChange(int[] coins, int amount) {
        Arrays.sort(coins);
        int count = 0;
        int n = coins.length;

        // Starting from the largest coin denomination
        for (int i = n - 1; i >= 0 && amount > 0; i--) {

        }

        return amount == 0 ? count : -1;
    }
```

For each coin in `coins`, we are going to keep adding that coin as long as the value of that coin is greater than or equal to the amount remaining. If we can add a coin, subtract the value of the coin from the amount remaining and increment `count`. Finally, after both loops have run, return `count` if the amount remaining is `0`. If not, return `-1` to indicate that this amount cannot be calculated.

```
    public static int coinChange(int[] coins, int amount) {
        Arrays.sort(coins);
        int count = 0;
        int n = coins.length;

        // Starting from the largest coin denomination
        for (int i = n - 1; i >= 0 && amount > 0; i--) {
            while (amount >= coins[i]) {  // while we can still
        use coin[i]
                amount -= coins[i];
                count++;
            }
        }

        return amount == 0 ? count : -1;
    }
```

▼ Code

Your code should look like this:

```java
import java.util.Arrays;

public class CoinChangeGreedy {

    public static int coinChange(int[] coins, int amount) {
        // Sort the coins in descending order
        Arrays.sort(coins);
        int count = 0;
        int n = coins.length;

        // Starting from the largest coin denomination
        for (int i = n - 1; i >= 0 && amount > 0; i--) {
            while (amount >= coins[i]) {  // while we can still
        use coin[i]
                amount -= coins[i];
                count++;
            }
        }

        return amount == 0 ? count : -1;
    }

    public static void main(String[] args) {
        int[] coins = {1, 5, 10, 25};  // US coin denominations
        int amount = 63;
        int result = coinChange(coins, amount);

        if (result != -1) {
            System.out.println("Minimum coins needed: " +
        result);
        } else {
            System.out.println("The amount cannot be represented
        using the given denominations.");
        }
    }
}
```

For an amount of 63, the output will be 6 because two quarters (50 cents), one dime (10 cents), and three pennies (3 cents) make 63 cents.

challenge

Try This Variation

Let's assume that we live in a country who has the following coins: 2, 15, and 25. Modify the `main()` method to reflect these coins, and then try to find the minimum number of coins needed to total 30.

```
    public static void main(String[] args) {
        int[] coins = {2, 15, 25};  // US coin denominations
        int amount = 30;
        int result = coinChange(coins, amount);

        if (result != -1) {
            System.out.println("Minimum coins needed: " +
        result);
        } else {
            System.out.println("The amount cannot be
        represented using the given denominations.");
        }
    }
```

▼ What is happening?

A greedy algorithm will always choose the option that provides the most immediate profit. It does not stop and consider the ramifications of the choice. So, the algorithm selects the coin worth 25, which leaves 5 as the remaining amount. There is no combination of the available coins that produces a total of exactly 5.

Again, it's important to emphasize that this greedy approach does not guarantee an optimal solution for all coin denominations. In cases where it's not suitable, other techniques, such as dynamic programming, would be more appropriate.

Knapsack Problems

0/1 Knapsack Problem

Another classic and simple greedy algorithm is the 0/1 knapsack problem. The problem can be understood in terms of resource allocation with a constraint on total capacity, where items have `values` and `weights`. The goal is to maximize the value stored in the knapsack without going over the weight limit. The “0/1” part means that you can only take entire items. This is similar to binary, where values can be 0 or they can be 1, but they cannot be some decimal between the two.

The greedy algorithm for the knapsack problem works by iteratively adding the item with the highest value-to-weight ratio to the knapsack until the knapsack is full or no more items can be added. This algorithm is not always optimal, but it is often very efficient.

Here is a step-by-step description of the greedy algorithm for the knapsack problem:

1. Sort the items in decreasing order of value-to-weight ratio.
2. Iterate over the items in sorted order.
3. If the item can be fit into the knapsack, add it to the knapsack.
4. If the item cannot be fit into the knapsack, skip it.
5. Repeat steps 3 and 4 until the knapsack is full or no more items can be added.

Fractional Knapsack

This variation of the knapsack problem allows you to take a fraction of an item and place it in the knapsack.

The greedy algorithm for the knapsack problem is not always optimal. For example, consider the following set of items:

- Item 1: Value = 10, Weight = 5
- Item 2: Value = 20, Weight = 10
- Item 3: Value = 30, Weight = 15

Let’s evaluate how the greedy algorithm for the fractional knapsack problem would behave with the given items. Notice how all items have the same value-to-weight ratio.

- Item 1: Value = 10, Weight = 5. (${Value/Weight Ratio} = \frac{10}{5} = 2 $)
- Item 2: Value = 20, Weight = 10. (${Value/Weight Ratio} = \frac{20}{10} = 2 $)

- Item 3: Value = 30, Weight = 15. (${Value/Weight Ratio} = \frac{30}{15} = 2 $)

If the knapsack has a capacity `W`:

- **If W ≥ 30:** The knapsack can fit all items. The maximum value will be $10 + 20 + 30 = 60$.
- **If 15 ≤ W < 30:** The knapsack can fit Item 1 and Item 2 completely, and a fraction of Item 3. The maximum value will be $10 + 20 + (fractionofItem3'svalue)$.
- **If 10 ≤ W < 15:** The knapsack can fit Item 1 completely and a fraction of Item 2. The maximum value will be $10 + (fractionofItem2'svalue)$.
- **If 5 ≤ W < 10:** The knapsack can only fit Item 1 completely. The maximum value will be 10.
- **If W < 5:** The knapsack can fit only a fraction of Item 1.

To provide a concrete example: If the knapsack capacity is $W = 20$, the algorithm will select Item 1 and Item 2 completely (as they both fit). The value in the knapsack is $10 + 20$ so far. The remaining weight is 5 but the weight of Item 3 is 15. If we divide the remaining weight in the knapsack by the weight of Item 3 ($\frac{5}{15}$), we get a ratio of $\frac{1}{3}$. Apply this ratio to the value of Item 3 ($\frac{30}{3}$) and we get a total value of 40 ($10 + 20 + 10$) in the knapsack.

Advantages and Disadvantages

Here are some of the advantages and disadvantages of using the greedy algorithm for the knapsack problem:

Advantages:

- The greedy algorithm is very efficient. It can be used to solve knapsack problems with a large number of items in a reasonable amount of time.
- The greedy algorithm is easy to implement. It does not require any complex data structures or algorithms.

Disadvantages:

- The greedy algorithm is not always optimal. It can sometimes produce suboptimal solutions.
- The greedy algorithm does not work for all knapsack problems. It only works for knapsack problems where the items can be easily sorted in decreasing order of value-to-weight ratio.

Knapsack Code

Fractional Example

Now that we have discussed how the fractional knapsack problem works, let's create a working example. Given weights and values of `n` items, we need to put these items in a knapsack of capacity `W` to get the maximum total value in the knapsack.

The basic idea of the greedy approach is to calculate the value per unit weight for each item, then choose items based on their value-to-weight ratio, starting with the item that has the highest ratio.

In order for our algorithm to work, we need to first need to create the `Item` class, which has `weight` and `value` attributes. We also need to import the `Arrays` utility for sorting our array of `Items`.

```
import java.util.Arrays;

class Item {
    int weight;
    int value;

    Item(int weight, int value) {
        this.weight = weight;
        this.value = value;
    }
}
```

Create the `FractionalKnapsack` class as well as the `main()` and `fractionalKnapsack()` methods. The `fractionalKnapsack()` method takes integers representing the capacity of the knapsack and the number of possible items. It also takes an array containing objects of type `Item`.

```
import java.util.Arrays;

class Item {
    int weight;
    int value;

    Item(int weight, int value) {
        this.weight = weight;
        this.value = value;
    }
}

public class FractionalKnapsack {

    public static double fractionalKnapsack(int W, Item[] arr,
        int n) {

    }

    public static void main(String[] args) {

    }
}
```

In the `main()` method, set a capacity (`W`) for the knapsack and create an array of `Items` with differing weights and values. Then calculate the length of the array. Finally, print some context for the result of the `fractionalKnapsack()` method.

```
    public static void main(String[] args) {
        int W = 50;   // Knapsack capacity
        Item[] arr = {new Item(10, 60), new Item(20, 100), new
        Item(30, 120)};
        int n = arr.length;

        System.out.println("Maximum value in knapsack = " +
        fractionalKnapsack(W, arr, n));
    }
```

For the `fractionalKnapsack()` method, first sort the array based on the value-to-weight ratio for each element. The array should be in descending order. Create a `totalValue` variable that will be used to keep track of the value stored in the knapsack. Finally, iterate over the array. For each element, ask if the entire object will fit in the knapsack. If yes, subtract the weight of the element from the capacity of the knapsack, and add the value of the element to `totalValue`.

```
    public static double fractionalKnapsack(int W, Item[] arr,
        int n) {
        // Sort items based on value-to-weight ratio
        Arrays.sort(arr, (a, b) -> Double.compare((double)
        b.value / b.weight, (double) a.value / a.weight));
        double totalValue = 0;

        for (int i = 0; i < n; i++) {
            // Add entire item if it doesn't exceed capacity
            if (W - arr[i].weight >= 0) {
                W -= arr[i].weight;
                totalValue += arr[i].value;
            }
        }

        return totalValue;
    }
```

If the entire element cannot fit into the knapsack, we need to create an `else` branch for the conditional. Calculate the fraction of the element that could fit into the knapsack. Multiple the `value` and `weight` attributes by the fraction, and add the fractional value to `totalValue` and the fractional weight to the knapsack. After adding the fractional item to the knapsack, break out of the loop. Finally, return `totalValue`.

```
    public static double fractionalKnapsack(int W, Item[] arr,
        int n) {
        // Sort items based on value-to-weight ratio
        Arrays.sort(arr, (a, b) -> Double.compare((double)
        b.value / b.weight, (double) a.value / a.weight));
        double totalValue = 0;

        for (int i = 0; i < n; i++) {
            // Add entire item if it doesn't exceed capacity
            if (W - arr[i].weight >= 0) {
                W -= arr[i].weight;
                totalValue += arr[i].value;
            } else {  // Else add a fraction of it
                double fraction = (double) W / arr[i].weight;
                totalValue += arr[i].value * fraction;
                W = (int) (W - (arr[i].weight * fraction));
                break;
            }
        }

        return totalValue;
    }
```

▼ Code

Your code should look like this:

```java
import java.util.Arrays;

class Item {
    int weight;
    int value;

    Item(int weight, int value) {
        this.weight = weight;
        this.value = value;
    }
}

public class FractionalKnapsack {

    // Function to get the maximum value in the knapsack
    public static double fractionalKnapsack(int W, Item[] arr,
        int n) {
        // Sort items based on value-to-weight ratio
        Arrays.sort(arr, (a, b) -> Double.compare((double)
        b.value / b.weight, (double) a.value / a.weight));

        double totalValue = 0;
        for (int i = 0; i < n; i++) {
            // If adding the entire item doesn't exceed
        capacity, add it
            if (W - arr[i].weight >= 0) {
                W -= arr[i].weight;
                totalValue += arr[i].value;
            } else {  // Else add a fraction of it
                double fraction = (double) W / arr[i].weight;
                totalValue += arr[i].value * fraction;
                W = (int) (W - (arr[i].weight * fraction));
                break;
            }
        }

        return totalValue;
    }

    public static void main(String[] args) {
        int W = 50;   // Knapsack capacity
        Item[] arr = {new Item(10, 60), new Item(20, 100), new
        Item(30, 120)};
        int n = arr.length;

        System.out.println("Maximum value in knapsack = " +
        fractionalKnapsack(W, arr, n));
    }
}
```

For the given example, the output will be 240.0. The maximum value is obtained by choosing the entire first item (60), the entire second item (100), and 2/3 of the third item (80).

Formative Assessment 1

Formative Assessment 2