Solution: Coin Change

Let's solve the Coin Change problem using the Dynamic Programming pattern.

We'll cover the following

Statement

You're given an integer `total` and a list of integers called `coins`. The variable `coins` hold a list of coin denominations, and `total` is the total amount of money.

You have to find the minimum number of coins that can make up the `total` amount by using any combination of the coins. If the amount can't be made up, return `-1`. If the total amount is `0`, return `0`.

> **Note:** You may assume that we have an infinite number of each kind of coin.

Constraints

$1 \leq$ `coins.length` ≤ 12

$1 \leq$ `coins[i]` $\leq 2^{31} - 1$

$0 \leq$ `total` $\leq 10^3$

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

The naive approach is to generates all possible combinations of given denominations such that in each combination, the sum of coins is equal to `total`. From these combinations, choose the one with the minimum number of coins and return the minimum number required. If the sum of any combinations is not equal to `total` then print -1.

In the worst case, the time complexity increases exponentially with the `total` amount, which results in an algorithm edging toward $O(n^{total})$. The space complexity is $O(total)$ because the maximum depth of the

recursion tree grows up to `total`. We can't improve the space complexity, but we can definitely improve our time complexity if we use dynamic programming to solve this problem.

Optimized solution using dynamic programming pattern

If we look at the problem, we might immediately think that it could be solved through a greedy approach. However, if we look at it closely, we'll know that it's not the correct approach here. Let's take a look at an example to understand why this problem can't be solved with a greedy approach.

Let's suppose we have `coins = [1, 3, 4, 5]` and we want to find the `total = 7` and we try to solve the problem with a greedy approach. In a greedy approach, we always start from the very end of a sorted array and traverse backward to find our solution because that allows us to solve the problem without traversing the whole array. However, in this situation, we start off with a `5` and add that to our `total`. We then check if it's possible to get a `7` with the help of either `4` or `3`, but as expected, that won't be the case, and we would need to add `1` twice to get our required `total`.

The problem seems to be solved, and we have concluded that we need maximum `3` coins to get to the `total` of `7`. However, if we take a look at our array, that isn't the case. In fact, we could have reached the `total` of `7` with just `2` coins: `4` and `3`. So, the problem needs to be broken down into subproblems, and an optimal solution can be reached from the optimal solutions of its subproblems.

To split the problem into subproblems, let's assume we know the number of coins required for some `total` value and the last coin denomination is *C*. Because of the optimal substructure property, the following equation will be true:

Min(total) = Min(total - C) + 1

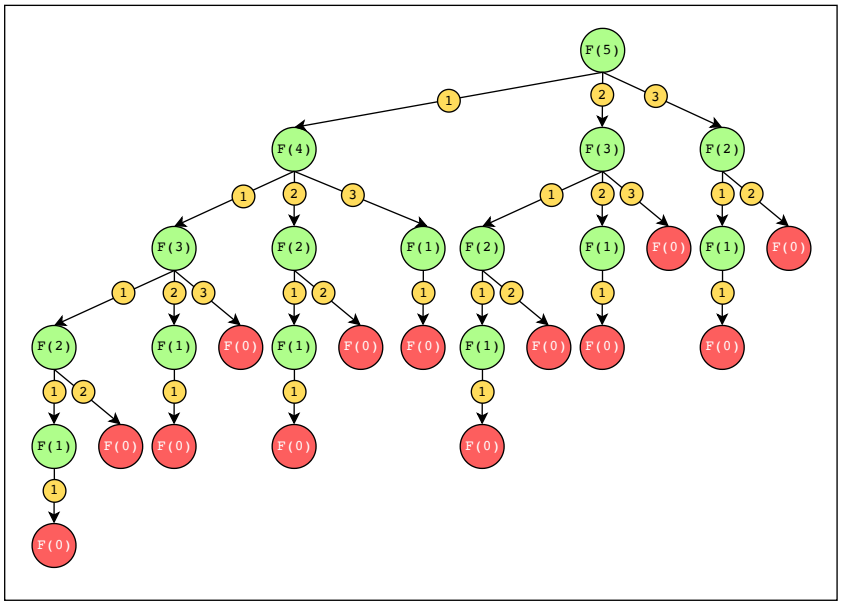
But, we don't know what is the value of C yet, so we compute it for each element of the `coins` array and select the minimum from among them. This creates the following recurrence relation:

$Min(total) = min_{i=0.....n-1}\ Min(total- C_i) + 1$, such that

Min(total) = 0, for total = 0

Min(total) = -1, for n = 0

> **Note:** The problem can also be solved with the help of a simple recursive tree without any backtracking, but that would take extra memory and time complexity, as we can see in the illustration below.



Recursive tree for finding minimum number of coins for the total 5 with the coins [1,2,3]

> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step solution construction

The idea is to solve the problem using top to the bottom technique of dynamic programming. For this, it uses backtracking to deduct the invalid solutions from the recursive tree. If the required `total` is less than the number that's being checked, the algorithm doesn't make any more recursive calls. Moreover, the recursive tree calculates the results of many subproblems multiple times, so if we store the result of each problem in a table, we can drastically improve the algorithm's efficiency by accessing the required value at a constant time. This massively reduces the number of calls we need to reach our result!

This approach doesn't return a viable solution when we try to find the total while working with numbers that are greater than the actual total we're looking for. For example, we have a list with the following numbers: `[2, 4, 8]`. We're attempting to find the total of `6` by utilizing the subcomponents of the number `8`. We start our solution by creating a helper function that assists us in calculating the number of coins we need. It calculates the amount left to be calculated and tells what its value can be. It has three base cases: to cover what to return if the remaining amount is less than zero, if the remaining amount is equal to zero, and if the remaining amount isn't actually either of the former two cases.

Java

```java
class CoinChange {
    public static int calculateMinimumCoins(int [] coins, int rem, int [] counter)
    {
        if(rem < 0)
            return -1;
        if(rem == 0)
            return 0;
        if(counter[rem - 1] != -1)
            return counter[rem - 1];
        int minimum = -1;
        return 0;
    }
    public static int coinChange(int [] coins, int total)
    {
        if(total < 1)
            return 0;
```

```
        System.out.println("The current total is more than zero, hence we need to call the helper functio
        System.out.println("We will pass three parameters to our helper function:\n");
        System.out.println("The first parameter would be the coins we have which are: " + Arrays.toString
        System.out.println("The second parameter would be the total we have to calculate which is: " + to
        System.out.print("The third parameter is the counter which is initially set to:");
        int[] l = new int[total];
        Arrays.fill(l, -1);

        System.out.println(Arrays.toString(l));
        return calculateMinimumCoins(coins, total, l);
    }
    public static void main( String args[] ) {
```

Coin Change

To improve time complexity we store the solutions of the already calculated subproblems in a table. For this, we calculate the result in our helper function and return it. We traverse the `coins` array, and at each element, call the backtracking function, passing `rem` minus the value of the coin to it. We store the return value of the base cases in a variable named `result`. We then add `1` to the `result` variable and assign this value to `minimum`, which is initially set to infinity at the start of each path. At the end of each path traversal, we update the $(rem - 1)^{th}$ index of the `counter` list with `minimum` if it is not equal to infinity, otherwise -1. Finally, we return the value at this index.

Java

```
class CoinChange {
    public static int calculateMinimumCoins(int [] coins, int rem, int [] counter)
    {
        int result = 0;
        if(rem < 0)
            return -1;
        if(rem == 0)
            return 0;
        if(counter[rem - 1] != Integer.MAX_VALUE)
            return counter[rem - 1];
        int minimum = Integer.MAX_VALUE;
        for (int j = 0; j < coins.length; j++)
        {
            result = calculateMinimumCoins(coins, rem - coins[j], counter);
            if(result >= 0 && result < minimum)
                minimum = 1 + result;
        }
        if(minimum != Integer.MAX_VALUE)
            counter[rem - 1] = minimum;
        else
            counter[rem - 1] =  -1;
        return counter[rem -1];
    }
    public static int coinChange(int [] coins, int total)  // main function
    {
        if(total < 1)
            return 0;
        int[] l = new int[total];
```

Coin Change

Just the code

Here's the complete solution to this problem:

Java

```
class CoinChange {
    public static int calculateMinimumCoins(int [] coins, int rem, int [] counter)
    {
```

```
        int result = 0;
        if(rem < 0)
            return -1;
        if(rem == 0)
            return 0;
        if(counter[rem - 1] != Integer.MAX_VALUE)
            return counter[rem - 1];
        int minimum = Integer.MAX_VALUE;
        for (int j = 0; j < coins.length; j++)
        {
            result = calculateMinimumCoins(coins, rem - coins[j], counter);
            if(result >= 0 && result < minimum)
                minimum = 1 + result;
        }
        if(minimum != Integer.MAX_VALUE)
```

```
        return counter[rem -1];
    }
    public static int coinChange(int [] coins, int total)
    {
        if(total < 1)
            return 0;
```

Coin Change

Solution summary

We first check if the total value is either one of the following:

- `0`: No new coins need to be added because we have reached a viable solution.
- `-1`: Our path can't lead to this solution, so we need to backtrack.
- otherwise, using the top-down approach, at each iteration, we either pick a coin or we don't. This way we either solve a new sub-problem or look up the answer from the table if it is already computed.

The algorithm's goal is to construct the problem's answer from top to bottom. We use backtracking and cut the redundant solutions in the recursive tree that don't lead to a viable solution.

Time complexity

The time complexity for the above algorithm is $O(n * m)$. Here, n represents the `total` and m represents the number of `coins` we have. In the worst case, the height of the recursive tree is n as the subproblems solved by the algorithm will be n because we are storing precalculated solutions in a table. Each subproblem takes m iterations, one per coin value. So, the time complexity is $O(n * m)$.

Space complexity

The space complexity for this algorithm is $O(n)$, where `n` is the `total`. The memoization table uses extra