https://leetcode.com/problems/coin-change/

Brute Force Solution:

```
/**
 * @param {number[]} coins
 * @param {number} amount
 * @return {number}
 */
var coinChange = function (coins, amount) {
  // 1. Sort the coins array in descending order
  coins.sort((a, b) => b - a);

  // initialize the total number of coins
  let totalCoins = 0;

  // 2. Iterate the coins array and do the following for every coin
  for (let i = 0; i < coins.length; i++) {
    // 3. for every coin in the coin array
    // iteratively exhaust the number of times we can use the coin
    // to give change to the amount
    while (amount > 0) {
      if (coins[i] <= amount) {
        // it means we can use the coin at coins[i]
        // 4. keep track of the total number of coins for every coin that
is used
        // to update the amount and give change
        // update the amount
        amount -= coins[i];

        // update the number of coins used
        totalCoins++;
      } else {
        break;
      }
    }
  }
  // 5. return the total number of coins
  if (amount > 0) {
    return -1;
  } else {
    return totalCoins;
  }
};
```

```
class Solution:
    # Approach: Greedy Algorithm
    def coinChange(self, coins: List[int], amount: int):
        # sort the array
```

```
        coins = sorted(coins, reverse = True)

        totalCoins = 0

        for coin in coins:
            while amount>0 and coin<=amount:
                totalCoins += 1
                amount -= coin

        return -1 if amount>0 else totalCoins
```

```
import java.util.Arrays;
import java.util.Collections;
class Solution {
    public int coinChange(int[] coins, int amount) {
        Integer[] nums =
Arrays.stream(coins).boxed().toArray(Integer[]::new);
        Arrays.sort(nums, new Comparator<Integer>(){
            public int compare(Integer a, Integer b){
                return b - a;
            }
        });

        int totalCoins = 0;

        for(int i=0; i<nums.length; i++){
            while(amount>0 && nums[i]<=amount){
                totalCoins++;
                amount -= nums[i];
            }
        }

        if(amount>0){
            return -1;
        } else {
            return totalCoins;
        }
    }
}
```

Dynamic Programming Approach

We need to first find the recurrence relation for the problem.

- start with the base case
- from the base case, solve the problem for few values of n
- find the pattern and generalize the solution

Let's say we have an array of coins [1, 2, 5] and we need to give change for 11.

Base Case:

n -> amount

for amount = 0, dp[n] = 0 coins

n=0, dp[0] = 0 n=1, dp[1] = min(1) = 1 coin n=2, dp[2] = min(2, 1) = 1 coin = (1 + dp[0], 1 + dp[1]) = (1 + 0, 1 + 1) = Min(1, 2) = 1 coin n=3, dp[3] = min(3, 2) = 2 coins = (1 + dp[2], 1 + dp[1]) = (1 + 1, 1 + 1) = Min(2, 2) = 2 coins n=4, dp[4] = min(4, 3, 2) = 2 coins = (1 + dp[3], 1 + dp[2]) = (1 + 2, 1 + 1) = Min(3, 2) = 2 coins n=5, dp[5] = min(5, 3, 1) = 1 coin = (1 + dp[4], 1 + dp[3], 1 + dp[0]) = (1 + 2, 1 + 2, 1 + 0) = Min(3, 3, 1) = 1 coin n=6, dp[6] = min(6, 4, 2) = 2 coins = (1 + dp[5], 1 + dp[4], 1 + dp[2]) = (1 + 1, 1 + 2, 1 + 1) = Min(2, 3, 2) = 2 coins

Generalizing the solution:

For n = 0, dp[n] = 0 For n > 0, dp[n] = min(dp[n - coin] + 1) for coin in coins

To demonstrate finding recurrence relation, let's take an example of finding the sum of the first n natural numbers.

Input: n = 5 Output: 15 Explanation: 1 + 2 + 3 + 4 + 5 = 15

Input: n = 4 Output: 10 Explanation: 1 + 2 + 3 + 4 = 10

Recurrence Relation: A function that is defined in terms of itself in a mathematical expression.

Objective: To find the expression S(n) using the same function S(n) in a mathematical expression.

Base Case:

Function to find the sum of the first n natural numbers -> S(n), where n is the number of natural numbers

S(1) = 1 S(2) = 1 + 2 = 3 S(3) = 1 + 2 + 3 = 6 S(4) = 1 + 2 + 3 + 4 = 10 S(5) = 1 + 2 + 3 + 4 + 5 = 15

Let's find the pattern:

S(1) = 1, base case

S(2) = S(1) + 2 = 3 S(3) = S(2) + 3 = 6 S(4) = S(3) + 4 = 10 S(5) = S(4) + 5 = 15 [using backward substitution]

Generalizing the solution:

For n = 1, S(n) = 1 For n > 1, S(n) = S(n - 1) + n

Let's take another example of finding the nth Fibonacci number.

Input: n = 5 Output: 3 Explanation: 0, 1, 1, 2, 3

Input: n = 6 Output: 5 Explanation: 0, 1, 1, 2, 3, 5

Base Case:

Function to find the nth Fibonacci number -> F(n), where n is the number of Fibonacci numbers

For n = 1, F(1) = 0 For n = 2, F(2) = 1

Let's find the pattern:

For n = 3, F(3) = F(2) + F(1) = 1 + 0 = 1 For n = 4, F(4) = F(3) + F(2) = 1 + 1 = 2 For n = 5, F(5) = F(4) + F(3) = 2 + 1 = 3 For n = 6, F(6) = F(5) + F(4) = 3 + 2 = 5

Generalizing the solution:

For n = 1, F(n) = 0 For n = 2, F(n) = 1 For n > 2, F(n) = F(n - 1) + F(n - 2)

Approach: Dynamic Programming - Bottom Up - Tabulation

```
/**
 * @param {number[]} coins
 * @param {number} amount
 * @return {number}
 */
/*
    For n = 0, dp[n] = 0
    For n > 0, dp[n] = min(dp[n - coin] + 1) for coin in coins

    Bottom-Up Approach: Tabulation Method in Dynamic Programming
*/
var coinChange = function (coins, amount) {
  // initialize dp array
  let dp = new Array(amount + 1).fill(amount + 1);

  // base case
  dp[0] = 0;

  // for all the amount > 0
  // let's find the minimum number of coins required on an incremental
basis
  // storing each of the results in the dp table
  // iterate for i = 1 to amount
  for (let i = 1; i <= amount; i++) {
    for (let coin of coins) {
      if (i - coin >= 0) {
        dp[i] = Math.min(dp[i - coin] + 1, dp[i]);
      }
    }
  }

  // return the dp[amount]
  return dp[amount] === amount + 1 ? -1 : dp[amount];
};
```

```
class Solution:
    # Approach: Greedy Algorithm
    def coinChange(self, coins: List[int], amount: int):
```

```
        dp = [amount+1] * (amount+1)

        dp[0] = 0

        for i in range(1, amount+1):
            for coin in coins:
                if i-coin >= 0:
                    dp[i] = min(dp[i], dp[i-coin] + 1)

        return -1 if dp[amount] == amount+1 else dp[amount]
```

```
import java.util.Arrays;
import java.util.Collections;
class Solution {
    public int coinChange(int[] coins, int amount) {
        int dp[] = new int[amount+1];

        for(int i=0; i<=amount; i++){
            dp[i] = amount+1;
        }

        dp[0] = 0;

        for(int i=1; i<=amount; i++){
            for(int coin: coins){
                if(i-coin >= 0){
                    dp[i] = Math.min(dp[i], dp[i-coin] + 1);
                }
            }
        }

        return dp[amount]==amount+1 ? -1 : dp[amount];
    }
}
```

Techniques for Competitive Programming

1. Hashing

Hashing is a technique that uses hash tables (or dictionaries or java script objects or HashMap collections) to store the frequency of elements or to check if an element is present in the array or not.