Recursive Brute Java Code

Let's walk through the brute-force recursive solution for the Coin Change problem. We'll focus on understanding the problem, building the logic, and implementing it step by step in Java.

Problem Description

Given a list of coins with different denominations and a total amount of money, determine the fewest number of coins needed to make up that amount. If that amount cannot be made up by any combination of the coins, return -1.

Recursive Approach

The idea of the recursive approach is to try every possible way to make up the amount by using each coin, and then choose the minimum number of coins needed.

Algorithm Explanation

1. **Base Case**:
   - If `amount` is 0, return 0 because no coins are needed to make 0 amount.
   - If `amount` is negative, return a large value (e.g., Integer.MAX_VALUE) because it's not possible to make a negative amount with positive coin values.
2. **Recursive Case**:
   - Initialize a variable `minCoins` to a large value to keep track of the minimum number of coins needed.
   - Iterate through each coin in the list of coins.
   - For each coin, make a recursive call with the reduced amount (`amount - coin`).
   - Track the minimum number of coins needed among all recursive calls.
3. **Return the Result**:
   - If the minimum number of coins found is still a large value, return -1 indicating it's not possible to make the amount.
   - Otherwise, return the minimum number of coins.

Java Code

```java
public class CoinChange {

    // Main function to find the minimum number of coins
    public int coinChange(int[] coins, int amount) {
        // Starting the recursive function with the given amount
        int result = coinChangeHelper(coins, amount);
```

```
        // If result is Integer.MAX_VALUE, it means amount can't be formed, so return -1
        return result == Integer.MAX_VALUE ? -1 : result;
    }

    // Helper function to perform the recursive search
    private int coinChangeHelper(int[] coins, int amount) {
        // Base case: If amount is 0, no coins are needed
        if (amount == 0) {
            return 0;
        }

        // Base case: If amount is negative, return a large value as it's not possible
        if (amount < 0) {
            return Integer.MAX_VALUE;
        }

        // Initialize the minimum number of coins needed to a large value
        int minCoins = Integer.MAX_VALUE;

        // Iterate through each coin and make the recursive call
        for (int coin : coins) {
            // Recursive call with the reduced amount
            int result = coinChangeHelper(coins, amount - coin);

            // If result is not Integer.MAX_VALUE, update the minimum coins needed
            if (result != Integer.MAX_VALUE) {
                minCoins = Math.min(minCoins, result + 1);
            }
        }

        // Return the minimum number of coins found
        return minCoins;
    }

    // Test the function with a simple example
    public static void main(String[] args) {
        CoinChange solution = new CoinChange();
        int[] coins = {1, 2, 5};
        int amount = 11;
        int result = solution.coinChange(coins, amount);
        System.out.println("Minimum coins needed: " + result); // Expected output: 3
    }
}
```

Key Points and Common Mistakes

1. **Base Cases**: Always define base cases to handle trivial cases and avoid infinite recursion.
2. **Large Value Handling**: Use a large value to represent impossible cases (like negative amounts).
3. **Recursive Depth**: Recursive solutions can be slow for large inputs due to deep recursion and overlapping subproblems.
4. **Memoization**: To optimize, you can add memoization to store results of subproblems, reducing redundant computations.

Dry Run

Let's dry run the example where `coins = [1, 2, 5]` and `amount = 11`.

1. Start with `amount = 11`.
2. Try each coin:
    - Using coin `1`: Recur with `amount = 10`.
    - Using coin `2`: Recur with `amount = 9`.

    - Using coin `5`: Recur with `amount = 6`.
3. Continue recursion until reaching base cases (either `amount = 0` or `amount < 0`).
4. Track the minimum number of coins required among all recursive calls.
5. Return the minimum number of coins needed to make `amount = 11`, which is `3` (using coins `5, 5, 1`).

This solution helps in understanding the recursive approach clearly with detailed comments explaining the logic behind each step.

Let's take a smaller example to clearly understand how the recursive solution backtracks and updates `minCoins` in the Coin Change problem. We'll consider `coins = [1, 2, 5]` and `amount = 5`.

Dry Run Example

Initial Call

- **Function Call**: `coinChangeHelper([1, 2, 5], 5)`
- **minCoins**: Initialized to `Integer.MAX_VALUE`

Recursing with Coin = 1

1. **Function Call**: `coinChangeHelper([1, 2, 5], 4)`
    - **minCoins**: Initialized to `Integer.MAX_VALUE`
    2. **Function Call**: `coinChangeHelper([1, 2, 5], 3)`
        - **minCoins**: Initialized to `Integer.MAX_VALUE`
        3. **Function Call**: `coinChangeHelper([1, 2, 5], 2)`
            - **minCoins**: Initialized to `Integer.MAX_VALUE`
            4. **Function Call**: `coinChangeHelper([1, 2, 5], 1)`
                - **minCoins**: Initialized to `Integer.MAX_VALUE`
                5. **Function Call**: `coinChangeHelper([1, 2, 5], 0)`
                    - **Base Case**: `amount` is 0, return 0.
                - **Backtrack to `amount = 1`**:
                    - **Result from `amount = 0`**: 0

        - `minCoins = min(Integer.MAX_VALUE, 0 + 1) = 1`
        - No other coins to try, return `minCoins = 1`.
    - **Backtrack to** `amount = 2`:
        - **Result from** `amount = 1`: 1
        - `minCoins = min(Integer.MAX_VALUE, 1 + 1) = 2`
        - **Try coin = 2**: `coinChangeHelper([1, 2, 5], 0)`
            - **Base Case**: `amount` is 0, return 0.
        - `minCoins = min(2, 0 + 1) = 1`
        - Return `minCoins = 1`.
    - **Backtrack to** `amount = 3`:
        - **Result from** `amount = 2`: 1
        - `minCoins = min(Integer.MAX_VALUE, 1 + 1) = 2`
        - **Try coin = 2**: `coinChangeHelper([1, 2, 5], 1)`
            - **Result from** `amount = 1`: 1
        - `minCoins = min(2, 1 + 1) = 2`
        - Return `minCoins = 2`.
- **Backtrack to** `amount = 4`:
    - **Result from** `amount = 3`: 2
    - `minCoins = min(Integer.MAX_VALUE, 2 + 1) = 3`
    - **Try coin = 2**: `coinChangeHelper([1, 2, 5], 2)`
        - **Result from** `amount = 2`: 1
    - `minCoins = min(3, 1 + 1) = 2`
    - Return `minCoins = 2`.
- **Backtrack to** `amount = 5`:
    - **Result from** `amount = 4`: 2
    - `minCoins = min(Integer.MAX_VALUE, 2 + 1) = 3`
    - **Try coin = 2**: `coinChangeHelper([1, 2, 5], 3)`
        - **Result from** `amount = 3`: 2
    - `minCoins = min(3, 2 + 1) = 3`
    - **Try coin = 5**: `coinChangeHelper([1, 2, 5], 0)`
        - **Base Case**: `amount` is 0, return 0.
    - `minCoins = min(3, 0 + 1) = 1`
    - Return `minCoins = 1`.

Key Steps and Values

- **Base Case (amount = 0)**: Returns 0.
- **Comparisons**:
  - When using coin 1: Each recursive call adds 1 to the result of the next call.
  - When using coin 2: It skips some steps, leading to a smaller number of coins for certain amounts.
  - When using coin 5: Directly solves `amount = 5` in one step.

Visualization

For `amount = 5`, the recursive tree looks like this (simplified for clarity):

scss

```
coinChangeHelper([1, 2, 5], 5)
   |
   |-- coinChangeHelper([1, 2, 5], 4) + 1
   |    |
   |    |-- coinChangeHelper([1, 2, 5], 3) + 1
   |    |    |
   |    |    |-- coinChangeHelper([1, 2, 5], 2) + 1
   |    |    |    |
   |    |    |    |-- coinChangeHelper([1, 2, 5], 1) + 1
   |    |    |    |    |
   |    |    |    |    |-- coinChangeHelper([1, 2, 5], 0) = 0
   |    |    |    |    + 1 (coin = 1)
   |    |    |    + 1 (coin = 1)
   |    |    + 1 (coin = 1)
   |    + 1 (coin = 1)
   + 1 (coin = 1)

   |-- coinChangeHelper([1, 2, 5], 0) = 0
   + 1 (coin = 5)
```

Result: Minimum coins needed to make `amount = 5` is 1 using coin 5.

Final Java Code with Detailed Comments

java

```
public class CoinChange {

    // Main function to find the minimum number of coins
    public int coinChange(int[] coins, int amount) {
        // Starting the recursive function with the given amount
        int result = coinChangeHelper(coins, amount);

        // If result is Integer.MAX_VALUE, it means amount can't be formed, so return -1
        return result == Integer.MAX_VALUE ? -1 : result;
    }

    // Helper function to perform the recursive search
    private int coinChangeHelper(int[] coins, int amount) {
        // Base case: If amount is 0, no coins are needed
        if (amount == 0) {
            return 0;
```

```
        }

        // Base case: If amount is negative, return a large value as it's not possible
        if (amount < 0) {
            return Integer.MAX_VALUE;
        }

        // Initialize the minimum number of coins needed to a large value
        int minCoins = Integer.MAX_VALUE;

        // Iterate through each coin and make the recursive call
        for (int coin : coins) {
            // Recursive call with the reduced amount
            int result = coinChangeHelper(coins, amount - coin);

            // If result is not Integer.MAX_VALUE, update the minimum coins needed
            if (result != Integer.MAX_VALUE) {
                minCoins = Math.min(minCoins, result + 1);
            }
        }

        // Return the minimum number of coins found
        return minCoins;
    }

    // Test the function with a simple example
    public static void main(String[] args) {
        CoinChange solution = new CoinChange();
        int[] coins = {1, 2, 5};
        int amount = 5;
        int result = solution.coinChange(coins, amount);
        System.out.println("Minimum coins needed: " + result); // Expected output: 1
    }
}
```

This code and dry run should help you understand how the recursive function works, how the `minCoins` variable gets updated, and the final result is calculated.

Sure, let's track the values of `minCoins` and other variables during the dry run in a detailed step-by-step recursive tree format. We'll use the same example: `coins = [1, 2, 5]` and `amount = 5`.

Dry Run with Detailed Tracking

Let's trace the recursive calls and track the variables:

Initial Call

- **Function Call**: `coinChangeHelper([1, 2, 5], 5)`
- **minCoins**: Initialized to `Integer.MAX_VALUE`

Recursing with Coin = 1

1. **Function Call**: `coinChangeHelper([1, 2, 5], 4)`

- **minCoins**: Initialized to `Integer.MAX_VALUE`

2. **Function Call**: `coinChangeHelper([1, 2, 5], 3)`
   - **minCoins**: Initialized to `Integer.MAX_VALUE`
   3. **Function Call**: `coinChangeHelper([1, 2, 5], 2)`
      - **minCoins**: Initialized to `Integer.MAX_VALUE`
      4. **Function Call**: `coinChangeHelper([1, 2, 5], 1)`
         - **minCoins**: Initialized to `Integer.MAX_VALUE`
         5. **Function Call**: `coinChangeHelper([1, 2, 5], 0)`
            - **Base Case**: `amount` is 0, return 0.
         - **Backtrack to** `amount = 1`:
            - **Result from** `amount = 0`: 0
            - `minCoins = min(Integer.MAX_VALUE, 0 + 1) = 1`
            - No other coins to try, return `minCoins = 1`.
      - **Backtrack to** `amount = 2`:
         - **Result from** `amount = 1`: 1
         - `minCoins = min(Integer.MAX_VALUE, 1 + 1) = 2`
         - **Try coin = 2**: `coinChangeHelper([1, 2, 5], 0)`
            - **Base Case**: `amount` is 0, return 0.
         - `minCoins = min(2, 0 + 1) = 1`
         - Return `minCoins = 1`.
   - **Backtrack to** `amount = 3`:
      - **Result from** `amount = 2`: 1
      - `minCoins = min(Integer.MAX_VALUE, 1 + 1) = 2`
      - **Try coin = 2**: `coinChangeHelper([1, 2, 5], 1)`
         - **Result from** `amount = 1`: 1

        - `minCoins = min(2, 1 + 1) = 2`
        - Return `minCoins = 2`.
    - **Backtrack to `amount = 4`:**
        - **Result from `amount = 3`:** 2
        - `minCoins = min(Integer.MAX_VALUE, 2 + 1) = 3`
        - **Try coin = 2:** `coinChangeHelper([1, 2, 5], 2)`
            - **Result from `amount = 2`:** 1
        - `minCoins = min(3, 1 + 1) = 2`
        - Return `minCoins = 2`.

- **Backtrack to `amount = 5`:**
    - **Result from `amount = 4`:** 2
    - `minCoins = min(Integer.MAX_VALUE, 2 + 1) = 3`
    - **Try coin = 2:** `coinChangeHelper([1, 2, 5], 3)`
        - **Result from `amount = 3`:** 2
    - `minCoins = min(3, 2 + 1) = 3`
    - **Try coin = 5:** `coinChangeHelper([1, 2, 5], 0)`
        - **Base Case:** `amount` is 0, return 0.
    - `minCoins = min(3, 0 + 1) = 1`
    - Return `minCoins = 1`.

Recursive Tree with Variable Tracking

Below is the recursive tree with tracking of `minCoins` and other values at each step:

scss

```
coinChangeHelper([1, 2, 5], 5)
   | minCoins = Integer.MAX_VALUE
   |
   |-- coinChangeHelper([1, 2, 5], 4) + 1
   |     | minCoins = Integer.MAX_VALUE
   |     |
   |     |-- coinChangeHelper([1, 2, 5], 3) + 1
   |     |     | minCoins = Integer.MAX_VALUE
   |     |     |
   |     |     |-- coinChangeHelper([1, 2, 5], 2) + 1
   |     |     |     | minCoins = Integer.MAX_VALUE
   |     |     |     |
   |     |     |     |-- coinChangeHelper([1, 2, 5], 1) + 1
   |     |     |     |     | minCoins = Integer.MAX_VALUE
   |     |     |     |     |
   |     |     |     |     |-- coinChangeHelper([1, 2, 5], 0) = 0
   |     |     |     |     + 1 (coin = 1)
   |     |     |     |     | minCoins = 1
   |     |     |     + 1 (coin = 1)
   |     |     |     | minCoins = 1
   |     |     |     |
   |     |     |     |-- coinChangeHelper([1, 2, 5], 0) = 0
   |     |     |     + 1 (coin = 2)
   |     |     |     | minCoins = 1
   |     |     + 1 (coin = 1)
   |     |     | minCoins = 2
   |     |     |
   |     |     |-- coinChangeHelper([1, 2, 5], 1) = 1
   |     |     + 1 (coin = 2)
```

```
|     |     | minCoins = 2
|     + 1 (coin = 1)
|     | minCoins = 3
|     |
|     |-- coinChangeHelper([1, 2, 5], 2) = 1
|     + 1 (coin = 2)
|     | minCoins = 2
+ 1 (coin = 1)
| minCoins = 3
|
|-- coinChangeHelper([1, 2, 5], 3) = 2
+ 1 (coin = 2)
| minCoins = 3
|
|-- coinChangeHelper([1, 2, 5], 0) = 0
+ 1 (coin = 5)
| minCoins = 1
```

Final Values:

- `amount = 0`: Returns 0 (base case).
- `amount = 1`: Returns 1.
- `amount = 2`: Returns 1.
- `amount = 3`: Returns 2.
- `amount = 4`: Returns 2.
- `amount = 5`: Returns 1 (minimum coins).

Java Code with Detailed Comments

java

```java
public class CoinChange {

    // Main function to find the minimum number of coins
    public int coinChange(int[] coins, int amount) {
        // Starting the recursive function with the given amount
        int result = coinChangeHelper(coins, amount);

        // If result is Integer.MAX_VALUE, it means amount can't be formed, so return -1
        return result == Integer.MAX_VALUE ? -1 : result;
    }

    // Helper function to perform the recursive search
    private int coinChangeHelper(int[] coins, int amount) {
        // Base case: If amount is 0, no coins are needed
        if (amount == 0) {
            return 0;
        }

        // Base case: If amount is negative, return a large value as it's not possible
        if (amount < 0) {
            return Integer.MAX_VALUE;
        }

        // Initialize the minimum number of coins needed to a large value
        int minCoins = Integer.MAX_VALUE;

        // Iterate through each coin and make the recursive call
        for (int coin : coins) {
            // Recursive call with the reduced amount
            int result = coinChangeHelper(coins, amount - coin);
```

```
                // If result is not Integer.MAX_VALUE, update the minimum coins needed
                if (result != Integer.MAX_VALUE) {
                    minCoins = Math.min(minCoins, result + 1);
                }
            }

            // Return the minimum number of coins found
            return minCoins;
        }

        // Test the function with a simple example
        public static void main(String[] args) {
            CoinChange solution = new CoinChange();
            int[] coins = {1, 2, 5};
            int amount = 5;
            int result = solution.coinChange(coins, amount);
            System.out.println("Minimum coins needed: " + result); // Expected output: 1
        }
    }
```

This detailed tree and code comments should help you understand the tracking of `minCoins` and the results at each step of the recursion.

Certainly! Let's dry run the recursive solution in a tabular format to track the values of `minCoins` and other variables at each step.

Example: `coins = [1, 2, 5]`, `amount = 5`

Dry Run Table

Recursive Call	Current Amount	Coins Used	Result from Sub-Call	Calculated minCoins	Commen
`coinChangeHelper([1, 2, 5], 5)`	5			`Integer.MAX_VALUE`	Initial call
	5 - 1 = 4	1	2	3	Use coin 1, make sub-call with amount 4
	5 - 2 = 3	2	2	3	Use coin 2, make sub-call with amount 3
	5 - 5 = 0	5	0	1	Use coin 5, make sub-call with amount 0

Recursive Call	Current Amount	Coins Used	Result from Sub-Call	Calculated minCoins	Commen
`coinChangeHelper([1, 2, 5], 4)`	4			`Integer.MAX_VALUE`	Initial call
	4 - 1 = 3	1	2	3	Use coin 1, make sub-call with amount 3
	4 - 2 = 2	2	1	2	Use coin 2, make sub-call with amount 2
	4 - 5 = -1		`Integer.MAX_VALUE`	2	Use coin 5, sub-cal is invalid (amount < 0)
`coinChangeHelper([1, 2, 5], 3)`	3			`Integer.MAX_VALUE`	Initial call
	3 - 1 = 2	1	1	2	Use coin 1, make sub-call with amount 2
	3 - 2 = 1	2	1	2	Use coin 2, make sub-call with amount 1
	3 - 5 = -2		`Integer.MAX_VALUE`	2	Use coin 5, sub-cal is invalid (amount < 0)
`coinChangeHelper([1, 2, 5], 2)`	2			`Integer.MAX_VALUE`	Initial call
	2 - 1 = 1	1	1	2	Use coin 1, make sub-call with amount 1
	2 - 2 = 0	2	0	1	Use coin 2, make sub-call with amount 0
	2 - 5 = -3		`Integer.MAX_VALUE`	1	Use coin 5, sub-cal

Recursive Call	Current Amount	Coins Used	Result from Sub-Call	Calculated minCoins	Commen
					is invalid (amount < 0)
`coinChangeHelper([1, 2, 5], 1)`	1			`Integer.MAX_VALUE`	Initial call
	1 - 1 = 0	1	0	1	Use coin 1, make sub-call with amount 0
	1 - 2 = -1		`Integer.MAX_VALUE`	1	Use coin 2, sub-cal is invalid (amount < 0)
	1 - 5 = -4		`Integer.MAX_VALUE`	1	Use coin 5, sub-cal is invalid (amount < 0)
`coinChangeHelper([1, 2, 5], 0)`	0		0		Base case amount is 0, return

Explanation of the Table

- **Recursive Call**: Shows the current recursive call being executed.
- **Current Amount**: The amount for which the `coinChangeHelper` is calculating the minimum coins.
- **Coins Used**: Shows the coin being used to reduce the amount.
- **Result from Sub-Call**: The result obtained from the recursive sub-call.
- **Calculated minCoins**: The minimum number of coins calculated so far.
- **Comment**: Additional details about the current step.

Final Result

From the table, we can see the recursive process and how `minCoins` is updated at each step. The minimum number of coins needed to make the amount 5 is `1` (using the coin of value 5).

This detailed table should help in understanding how the recursive solution progresses and updates the `minCoins` variable at each step.

Sure, let's track the variables while backtracking. We'll consider the same example: `coins = [1, 2, 5]` and `amount = 5`. We'll use a table to show the values of variables (`currentAmount`, `minCoins`, and results from sub-calls) at each step of recursion and backtracking.

Dry Run Table with Backtracking

Call Stack Depth	Current Amount	Coin Used	Sub-Call Result	minCoins	Backtracking Comments
0	5			`Integer.MAX_VALUE`	Initial call
1	4	1		`Integer.MAX_VALUE`	Use coin 1, call with amount 4
2	3	1		`Integer.MAX_VALUE`	Use coin 1, call with amount 3
3	2	1		`Integer.MAX_VALUE`	Use coin 1, call with amount 2
4	1	1		`Integer.MAX_VALUE`	Use coin 1, call with amount 1
5	0	1	0	1	Base case: amount is 0, return 0
4 (backtrack)	1	1	1	2	Return 1 from sub-call (1 + 1), update minCoins to 2
4	1	2	`Integer.MAX_VALUE`	2	Use coin 2, invalid sub-call (amount < 0)
4	1	5	`Integer.MAX_VALUE`	2	Use coin 5, invalid sub-call (amount < 0)
3 (backtrack)	2	1	2	3	Return 2 from sub-call (1 + 2), update minCoins to 3
3	2	2	0	1	Use coin 2, call with amount 0
3 (backtrack)	2	2	1	1	Return 1 from sub-call (1 + 1), update minCoins to 1
3	2	5	`Integer.MAX_VALUE`	1	Use coin 5, invalid sub-call (amount < 0)
2 (backtrack)	3	1	1	2	Return 1 from sub-call (1 + 1), update minCoins to 2
2	3	2	1	2	Use coin 2, call with amount 1
2	3	5	`Integer.MAX_VALUE`	2	Use coin 5, invalid sub-call (amount < 0)
1 (backtrack)	4	1	2	3	Return 2 from sub-call (1 + 2), update minCoins to 3

Call Stack Depth	Current Amount	Coin Used	Sub-Call Result	minCoins	Backtracking Comments
1	4	2	1	2	Use coin 2, call with amount 2
1	4	5	`Integer.MAX_VALUE`	2	Use coin 5, invalid sub-call (amount < 0)
0 (backtrack)	5	1	2	3	Return 2 from sub-call (1 + 2), update minCoins to 3
0	5	2	2	3	Use coin 2, call with amount 3
0	5	5	0	1	Use coin 5, call with amount 0
0 (backtrack)	5	5	1	1	Return 1 from sub-call (1 + 1), update minCoins to 1

Explanation of the Table

- **Call Stack Depth**: Represents the depth of the recursive call stack.
- **Current Amount**: The amount being processed in the current recursive call.
- **Coin Used**: The coin used to reduce the current amount.
- **Sub-Call Result**: The result obtained from the recursive sub-call.
- **minCoins**: The minimum number of coins calculated so far.
- **Backtracking Comments**: Comments on the state during backtracking.

Final Result

After tracking all recursive calls and backtracking steps, we find that the minimum number of coins needed to make the amount `5` is `1` (using the coin of value `5`).

This table should help you visualize the process of recursion and backtracking, and how `minCoins` and other variables are updated at each step.

Sure! Let's use a detailed step-by-step backtracking format to show how the solution is derived from the base cases. This will help you understand how the variables are updated during the backtracking phase.

Recursive Process and Backtracking Steps

Initial Call:

1. `coinChangeHelper([1, 2, 5], 5)`
    - `minCoins = Integer.MAX_VALUE`

Recursive Calls:
2. Use coin `1`:

- `coinChangeHelper([1, 2, 5], 4)`
    - `minCoins = Integer.MAX_VALUE`

3. Use coin `1` again:

    - `coinChangeHelper([1, 2, 5], 3)`
        - `minCoins = Integer.MAX_VALUE`

4. Use coin `1` again:

    - `coinChangeHelper([1, 2, 5], 2)`
        - `minCoins = Integer.MAX_VALUE`

5. Use coin `1` again:

    - `coinChangeHelper([1, 2, 5], 1)`
        - `minCoins = Integer.MAX_VALUE`

6. Use coin `1` again:

    - `coinChangeHelper([1, 2, 5], 0)`
        - **Base case**: `currentAmount = 0`
        - **Return**: `0`

Backtracking Steps:

Step 1: Backtracking from `coinChangeHelper([1, 2, 5], 1)`

- Received `0` from sub-call with `amount = 0`
- Calculated `minCoins` for `amount = 1`: `minCoins = 1` (using coin `1`)
- Return `1` to previous call

Step 2: Backtracking from `coinChangeHelper([1, 2, 5], 2)`

- Received `1` from sub-call with `amount = 1`
- Calculated `minCoins` for `amount = 2`: `minCoins = 2` (using coin `1`)
- Use coin `2`:
    - `coinChangeHelper([1, 2, 5], 0)`
        - **Base case**: `currentAmount = 0`
        - **Return**: `0`
- Received `0` from sub-call with `amount = 0`
- Calculated `minCoins` for `amount = 2`: `minCoins = 1` (using coin `2`)
- Return `1` to previous call

Step 3: Backtracking from `coinChangeHelper([1, 2, 5], 3)`

- Received `1` from sub-call with `amount = 2`
- Calculated `minCoins` for `amount = 3`: `minCoins = 2` (using coin `1`)
- Use coin `2`:
  - `coinChangeHelper([1, 2, 5], 1)`
    - **Return**: `1`
- Received `1` from sub-call with `amount = 1`
- Calculated `minCoins` for `amount = 3`: `minCoins = 2` (no change)
- Return `2` to previous call

Step 4: Backtracking from `coinChangeHelper([1, 2, 5], 4)`

- Received `2` from sub-call with `amount = 3`
- Calculated `minCoins` for `amount = 4`: `minCoins = 3` (using coin `1`)
- Use coin `2`:
  - `coinChangeHelper([1, 2, 5], 2)`
    - **Return**: `1`
- Received `1` from sub-call with `amount = 2`
- Calculated `minCoins` for `amount = 4`: `minCoins = 2` (using coin `2`)
- Return `2` to previous call

Step 5: Backtracking from `coinChangeHelper([1, 2, 5], 5)`

- Received `2` from sub-call with `amount = 4`
- Calculated `minCoins` for `amount = 5`: `minCoins = 3` (using coin `1`)
- Use coin `2`:
  - `coinChangeHelper([1, 2, 5], 3)`
    - **Return**: `2`
- Received `2` from sub-call with `amount = 3`
- Calculated `minCoins` for `amount = 5`: `minCoins = 3` (no change)
- Use coin `5`:
  - `coinChangeHelper([1, 2, 5], 0)`
    - **Base case**: `currentAmount = 0`
    - **Return**: `0`
- Received `0` from sub-call with `amount = 0`
- Calculated `minCoins` for `amount = 5`: `minCoins = 1` (using coin `5`)
- Return `1` to the main function

Final Result

The minimum number of coins needed to make the amount `5` is `1` (using the coin of value `5`).

Summary

The backtracking process involves:

- Receiving results from sub-calls.
- Updating the `minCoins` variable based on these results.
- Using different coins to explore other possibilities.

- Returning the `minCoins` value up the call stack to the initial function call.

This detailed step-by-step format should help you visualize how the solution is derived during the backtracking phase.