<https://leetcode.com/problems/coin-change/description/>

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return the fewest 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.

You may assume that you have an infinite number of each kind of coin.

**Example 1:**

**Input:** coins = [1,2,5], amount = 11

**Output:** 3

**Explanation:** 11 = 5 + 5 + 1

**Example 2:**

**Input:** coins = [2], amount = 3

**Output:** -1

**Example 3:**

**Input:** coins = [1], amount = 0

**Output:** 0

**Constraints:**

1 <= coins.length <= 12

1 <= coins[i] <= 2^31 - 1

0 <= amount <= 10^4

**Attempt 1: 2025-02-26**

**Solution 1: Native DFS (30 min, TLE if using 'index' 15/189, TLE if using 'i' 39/189, TLE if not using 'index' 15/189)**

Similar to [L39.Combination Sum](note://79A77D14319840B9A134C272EA618600) (different on Combination Sum calculate all possibilities by use backtracking, here just need set up remain < 0 based condition to work with tmp >= 0 and initialize min = Integer.MAX\_VALUE to calculate minimum), refer to <https://leetcode.com/problems/coin-change/solutions/77368/>\*Java\*-Both-iterative-and-recursive-solutions-with-explanations/

**In short:**

**In Leetcode 322:**

**(1) Pass 'index' in recursion call will enable all coins selection in each recursion call as it always keep scanning from index = 0 and the actual effect as find permutation rather than combination.**

**(2) More efficient in this problem is pass 'i' in recursion call which means only coins from index i onward are considered. This restricts the exploration to combinations in a specific order, effectively avoiding permutations.**

**(3) Since we only interested in number of element used in combination or permutation of a result, so recursion call generate combination or permutation doesn't matter, and based on 'index' solution (1), we can directly remove 'index' paramter which is Solution 1 style 2, it has the same effect as enable all coins selection in each recursion call.**

**In Leetcode 39:**

**(1) Since we specifically need to find combination instead of permutation, so pass in 'i' is the only way, should NOT pass in 'index', and if not pass in 'index' also not work.**

**Style 1: With redundant 'index' (since always keep initialized as 0 in each recursion call to enable selection for all coins on each level, TLE 15/189)**

class Solution {

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

        int result = helper(coins, amount, 0);

        return result == Integer.MAX\_VALUE ? -1 : result;

    }

    private int helper(int[] coins, int remain, int index) {

        // One base condition set as negative (-1) for remain < 0

        // later dfs logic 'tmp >= 0' based on check this condition

        if(remain < 0) {

            return -1;

        }

        // Complete, be careful, its return 0 not 1, which means 0 remain amount

        // only require 0 coin to make it

        if(remain == 0) {

            return 0;

        }

        int min = Integer.MAX\_VALUE;

        // Recursive strategy same as Combination Sum I

        // as we can duplicately use the same number

        // so not 'i + 1' or 'i' but still use 'index'

        for(int i = index; i < coins.length; i++) {

            // We are using backtracking on combination sum is because

            // we are calculate all possibilites, but here in coin change

            // we just need to find minimum coins required to reach target,

            // so no need to do backtracking

            int tmp = helper(coins, remain - coins[i], index);

            // tmp >= 0 means valid solution, since in base condition

            if(tmp >= 0 && tmp < min) {

                // Don't forget plus one more coin as in use

                min = tmp + 1;

            }

        }

        return min;

    }

}

Time Complexity: O(n^amount)

Where: n is the number of coins. amount is the target amount.

This is because, at each level of recursion, the function branches into n possibilities,

and the recursion depth can be as large as amount.

Space Complexity: O(amount)

Recursion Stack: The space complexity is dominated by the recursion stack.

In the worst case, the recursion depth is amount (e.g., when the smallest coin is 1).

**Style 2: With 'i' (more efficient as restricts the exploration to combinations in a specific order, effectively avoiding permutations, TLE 39/189)**

class Solution {

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

        int result = helper(coins, amount, 0);

        return result == Integer.MAX\_VALUE ? -1 : result;

    }

    private int helper(int[] coins, int remain, int index) {

        // One base condition set as negative (-1) for remain < 0

        // later dfs logic 'tmp >= 0' based on check this condition

        if(remain < 0) {

            return -1;

        }

        // Complete, be careful, its return 0 not 1, which means 0 remain amount

        // only require 0 coin to make it

        if(remain == 0) {

            return 0;

        }

        int min = Integer.MAX\_VALUE;

        for(int i = index; i < coins.length; i++) {

            // We are using backtracking on combination sum is because

            // we are calculate all possibilites, but here in coin change

            // we just need to find minimum coins required to reach target,

            // so no need to do backtracking

            int tmp = helper(coins, remain - coins[i], i);

            // tmp >= 0 means valid solution, since in base condition

            if(tmp >= 0 && tmp < min) {

                // Don't forget plus one more coin as in use

                min = tmp + 1;

            }

        }

        return min;

    }

}

Time Complexity: O(n^amount)

Passing i: Explores combinations in a specific order, avoiding permutations.

However, the time complexity is still O(n^amount) because the recursion depth and branching factor remain the same.

Space Complexity: O(amount)

Recursion Stack: The space complexity is dominated by the recursion stack.

In the worst case, the recursion depth is amount (e.g., when the smallest coin is 1).

**Style 3: Without redundant 'index' (TLE 15/189)**

class Solution {

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

        int result = helper(coins, amount);

        return result == Integer.MAX\_VALUE ? -1 : result;

    }

    private int helper(int[] coins, int remain) {

        if(remain < 0) {

            return -1;

        }

        if(remain == 0) {

            return 0;

        }

        int min = Integer.MAX\_VALUE;

        for(int coin : coins) {

            int tmp = helper(coins, remain - coin);

            if(tmp != -1) {

                min = Math.min(min, tmp + 1);

            }

        }

        return min == Integer.MAX\_VALUE ? -1 : min;

    }

}

Time Complexity: O(n^amount)

Where: n is the number of coins. amount is the target amount.

This is because, at each level of recursion, the function branches into n possibilities,

and the recursion depth can be as large as amount.

Space Complexity: O(amount)

Recursion Stack: The space complexity is dominated by the recursion stack.

In the worst case, the recursion depth is amount (e.g., when the smallest coin is 1).

**Refer to Deepseek**

**1. Actually i have one doubt regarding the index parameter passing into recursion call, i have below comparison pure solutions need your help to clarify:**

(1) Pure DFS solution for Leetcode 322, where in recursion call we keep pass 'index' rather than 'i':

class Solution {

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

        int result = helper(coins, amount, 0);

        return result == Integer.MAX\_VALUE ? -1 : result;

    }

    private int helper(int[] coins, int remain, int index) {

        // One base condition set as negative (-1) for remain < 0

        // later dfs logic 'tmp >= 0' based on check this condition

        if(remain < 0) {

            return -1;

        }

        // Complete, be careful, its return 0 not 1, which means 0 remain amount

        // only require 0 coin to make it

        if(remain == 0) {

            return 0;

        }

        int min = Integer.MAX\_VALUE;

        // Recursive strategy same as Combination Sum I

        // as we can duplicately use the same number

        // so not 'i + 1' or 'i' but still use 'index'

        for(int i = index; i < coins.length; i++) {

            // We are using backtracking on combination sum is because

            // we are calculate all possibilites, but here in coin change

            // we just need to find minimum coins required to reach target,

            // so no need to do backtracking

            int tmp = helper(coins, remain - coins[i], index);

            // tmp >= 0 means valid solution, since in base condition

            if(tmp >= 0 && tmp < min) {

                // Don't forget plus one more coin as in use

                min = tmp + 1;

            }

        }

        return min;

    }

}

(2) Similar strategy problem Leetcode 39. Combination Sum, where in recursion call we pass 'i' rather than 'index'

class Solution {

public List<List<Integer>> combinationSum(int[] candidates, int target) {

List<List<Integer>> result = new ArrayList<List<Integer>>();

helper(target, candidates, result, new ArrayList<Integer>(), 0);

return result;

}

private void helper(int target, int[] candidates, List<List<Integer>> result, List<Integer> tmp, int index) {

if(target == 0) {

result.add(new ArrayList<Integer>(tmp));

return;

}

if(target < 0) {

return;

}

// Use local variable 'i' start from 'index' to traverse

for(int i = index; i < candidates.length; i++) {

// Not pick --> no need "No pick" branch in for loop style, why ?

//helper(target, candidates, result, tmp, i);

// Pick

tmp.add(candidates[i]);

// Since we can use same number unlimited times, next recursion

// still start with 'i'

helper(target - candidates[i], candidates, result, tmp, i);

tmp.remove(tmp.size() - 1);

}

}

}

The difference in how the index parameter is used in the two solutions stems from the distinct problem requirements and the nature of recursion in each case. Let's break it down:

**1. LeetCode 322: Coin Change (Minimum Coins)**

Goal: Find the minimum number of coins needed to make up an amount.

Key Insight: The order of coins doesn't matter (e.g., 1+2 is the same as 2+1). We just need the smallest count.

**DFS Approach in Your Code:**

Parameter index: The code passes index to the recursive call, but the loop **always** **starts at index = 0.**

for (int i = index; i < coins.length; i++) {

int tmp = helper(coins, remain - coins[i], index); // Passes `index`, not `i`!

}

Why index?: This effectively allows all coins to be re-selected in every recursive call, leading to redundant exploration of permutations (e.g., 1+2, 2+1). However, since the goal is to find the minimal count, all permutations are valid and the minimal result will still be found.

**Issue:**

Inefficiency: By allowing all permutations, the code explores many redundant paths, leading to exponential time complexity.

No Pruning: The index parameter doesn’t restrict the selection of coins in subsequent calls, so it’s functionally redundant. This is why the optimal DP solution for Coin Change uses a 1D array based solely on the remaining amount.

**2. LeetCode 39: Combination Sum (Unique Combinations)**

Goal: Find all unique combinations of coins that sum to a target.

Key Insight: Combinations must be unique (e.g., [2,3] and [3,2] are considered duplicates if the coins are sorted).

**Backtracking Approach:**

Parameter i: The code passes i to the recursive call, ensuring combinations are built in a non-decreasing order (if coins are sorted).

for (int i = index; i < candidates.length; i++) {

helper(target - candidates[i], ..., i); // Passes `i`, not `index`!

}

Why i?: By passing i, the next recursive call starts at the same index, allowing the reuse of the current coin (candidates[i]) but preventing the selection of earlier coins (e.g., once you move past coin 2, you can’t go back to coin 1). This ensures combinations like [2,3] are unique and avoids duplicates like [3,2].

**Efficiency:**

No Redundancy: By controlling the starting index, the algorithm avoids redundant permutations and ensures unique combinations.

Order Enforcement: The use of i enforces a specific order, which is critical for avoiding duplicate combinations.

**3. Key Difference**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Coin Change (LeetCode 322)** | **Combination Sum (LeetCode 39)** |
| **Goal** | Find the minimum number of coins (order doesn’t matter). | Find all unique combinations (order matters). |
| **Index Handling** | index is redundant; all coins are re-selected. | i is critical to enforce non-decreasing order. |
| **Recursion Parameter** | index passed to allow permutations. | i passed to avoid permutations and ensure uniqueness. |
| **Efficiency** | Highly inefficient due to redundant permutations. | Efficient due to controlled exploration. |

**4. Why the Coin Change Code Still Works**

Even though the Coin Change code redundantly explores permutations, the minimal count will eventually be found because:

All permutations are valid (order doesn’t affect the count).

The min variable tracks the smallest count across all paths.

However, this approach is highly inefficient (exponential time). This is why the optimal solution uses 1D DP with O(amount) space and O(n \* amount) time.

**5. How to Fix the Coin Change DFS**

To make the Coin Change DFS more efficient (still exponential but less redundant), remove the index parameter and iterate over all coins in every call:

private int helper(int[] coins, int remain) {

if (remain < 0) return -1;

if (remain == 0) return 0;

int min = Integer.MAX\_VALUE;

for (int coin : coins) { // Iterate all coins, not starting from `index`

int tmp = helper(coins, remain - coin);

if (tmp != -1) {

min = Math.min(min, tmp + 1);

}

}

return (min == Integer.MAX\_VALUE) ? -1 : min;

}

This simplifies the code and makes it clear that all coins are considered at every step.

**Summary**

Coin Change uses index in a way that doesn’t restrict the selection of coins, leading to redundant permutations but still finding the correct minimal count.

Combination Sum uses i to enforce order and avoid duplicate combinations.

The difference arises from the problem requirements: tracking order for uniqueness vs. ignoring order for minimal count.

**So for Leetcode 322 should we change the 'index' to 'i' and pass into recursion method like below ?**

class Solution {

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

int result = helper(coins, amount, 0);

return result == Integer.MAX\_VALUE ? -1 : result;

}

private int helper(int[] coins, int remain, int index) {

// One base condition set as negative (-1) for remain < 0

// later dfs logic 'tmp >= 0' based on check this condition

if(remain < 0) {

return -1;

}

// Complete, be careful, its return 0 not 1, which means 0 remain amount

// only require 0 coin to make it

if(remain == 0) {

return 0;

}

int min = Integer.MAX\_VALUE;

// Recursive strategy same as Combination Sum I

// as we can duplicately use the same number

// so not 'i + 1' or 'i' but still use 'index'

for(int i = index; i < coins.length; i++) {

// We are using backtracking on combination sum is because

// we are calculate all possibilites, but here in coin change

// we just need to find minimum coins required to reach target,

// so no need to do backtracking

int tmp = helper(coins, remain - coins[i], i);

// tmp >= 0 means valid solution, since in base condition

if(tmp >= 0 && tmp < min) {

// Don't forget plus one more coin as in use

min = tmp + 1;

}

}

return min;

}

}

To address your question, let’s clarify the differences and fix the approach for LeetCode 322: Coin Change.

**Key Issue in Your Modified Code**

**In your modified code, you pass i (the current coin index) to the recursive call instead of index. This restricts subsequent recursive steps to only consider coins starting from index i, which incorrectly limits the exploration of valid combinations. For example:**

**If you have coins [1, 2, 5] and amount 3, the minimal solution is 1+2 (2 coins).**

**If you start with coin 2 (index 1), the next call will only consider coins 2 and 5 (starting from index 1), missing the valid combination 1+2.**

**This restriction leads to incorrect results because the order of coins doesn’t matter in Coin Change, but your code artificially enforces an order.**

**Correct Approach for Coin Change**

In Coin Change, every coin can be reused infinitely, and the order of coins doesn’t matter. **The DFS should consider all coins at every step** (not just coins starting from i).

Here’s the corrected DFS and its memoization version:

**1. Corrected Pure DFS (No Index Parameter)**

class Solution {

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

int result = helper(coins, amount);

return result == Integer.MAX\_VALUE ? -1 : result;

}

private int helper(int[] coins, int remain) {

if (remain < 0) return -1;

if (remain == 0) return 0;

int min = Integer.MAX\_VALUE;

for (int coin : coins) { // Iterate all coins at every step

int tmp = helper(coins, remain - coin);

if (tmp != -1) {

min = Math.min(min, tmp + 1);

}

}

return min == Integer.MAX\_VALUE ? -1 : min;

}

}

**2. Memoization (Top-Down DP) Version**

We memoize results based on the remaining amount (no need for an index parameter):

class Solution {

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

int[] memo = new int[amount + 1];

Arrays.fill(memo, -2); // -2 = uncomputed, -1 = invalid, ≥0 = valid count

int result = helper(coins, amount, memo);

return result;

}

private int helper(int[] coins, int remain, int[] memo) {

if (remain < 0) return -1;

if (remain == 0) return 0;

// Return memoized result if computed

if (memo[remain] != -2) return memo[remain];

int min = Integer.MAX\_VALUE;

for (int coin : coins) { // Iterate all coins

int tmp = helper(coins, remain - coin, memo);

if (tmp != -1) {

min = Math.min(min, tmp + 1);

}

}

memo[remain] = (min == Integer.MAX\_VALUE) ? -1 : min;

return memo[remain];

}

}

**Why This Works**

No Index Parameter: At every step, all coins are considered, allowing full exploration of all combinations (including permutations like 1+2 and 2+1).

Memoization: We cache results for each remain value, avoiding redundant computations. This reduces the time complexity to O(amount \* n), where n is the number of coins.

**Comparison with Combination Sum (LeetCode 39)**

Combination Sum: In Leetcode 39 we use 'i' to avoid duplicate combinations (e.g., [2,3] and [3,2]) which if keep sending 'index' in each recursion level in Leetcode 39.Combination Sum will cause each recursion level always start from index 0 same as its initial call and the result actually as finding permutation rather than find combination.

Coin Change: Order doesn’t matter, so we allow all permutations and track only the minimal count.

**Final Answer**

Your modified code (passing i instead of index) is \*\*incorrect\*\* for Coin Change because it restricts valid combinations. The correct approach is to \*\*remove the index parameter\*\* and iterate over all coins at every step. The memoization version should cache results based on remain alone.

**So you mean actually in Leetcode 322 the initial pure DFS solution which we pass the 'index' in each recursion call to enable selection of all coins (since 'index' always start from 0 on each level of recursion, same as initial call as 0) is the correct way ? Although it will turns the finding from combination into permutation, but doesn't impact the minimal count, right ? But do you know if we have any alternative way to still pass in a parameter representing to enable selection of all coins ?**

Yes, you've captured the nuances perfectly! Let's clarify this step-by-step:

**1. Initial Pure DFS with index (Your Original Code)**

**In your initial code, passing index = 0 in every recursive call ensures that all coins are considered at every step, leading to exploring permutations (e.g., 1+2 and 2+1). However, since the goal is to find the minimum number of coins, this redundancy doesn’t affect correctness. The minimal count will still be found, but the solution is highly inefficient due to redundant permutations.**

// Original DFS (passes index = 0)

for (int i = index; i < coins.length; i++) {

int tmp = helper(coins, remain - coins[i], 0); // Always restart at index 0

}

**2. Why This Works (But Is Inefficient)**

**Order Doesn’t Matter: For Coin Change, 1+2 and 2+1 are equivalent permutations but yield the same count (2 coins). The minimal count is unaffected.**

**Redundant Exploration: The code explores all permutations, leading to exponential time complexity. For example, with coins [1,2,5] and amount 5, it explores both 1+1+1+1+1 and 5 but correctly returns 1 as the minimal count.**

**3. Alternative Way: Remove the index Parameter**

**Instead of passing index, simplify the code by iterating over all coins directly in each recursive call. This makes it clear that all coins are considered at every step:**

class Solution {

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

int result = helper(coins, amount);

return result == Integer.MAX\_VALUE ? -1 : result;

}

private int helper(int[] coins, int remain) {

if (remain < 0) return -1;

if (remain == 0) return 0;

int min = Integer.MAX\_VALUE;

// Iterate all coins at every step

for (int coin : coins) {

int tmp = helper(coins, remain - coin);

if (tmp != -1) {

min = Math.min(min, tmp + 1);

}

}

return min == Integer.MAX\_VALUE ? -1 : min;

}

}

**4. Memoization (Top-Down DP) Version**

To optimize, memoize results based on remaining amount (no need for index):

class Solution {

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

int[] memo = new int[amount + 1];

Arrays.fill(memo, -2); // -2 = uncomputed, -1 = invalid, ≥0 = valid count

return helper(coins, amount, memo);

}

private int helper(int[] coins, int remain, int[] memo) {

if (remain < 0) return -1;

if (remain == 0) return 0;

if (memo[remain] != -2) return memo[remain];

int min = Integer.MAX\_VALUE;

for (int coin : coins) { // All coins considered

int tmp = helper(coins, remain - coin, memo);

if (tmp != -1) {

min = Math.min(min, tmp + 1);

}

}

memo[remain] = (min == Integer.MAX\_VALUE) ? -1 : min;

return memo[remain];

}

}

**5. Key Takeaways**

Index Parameter: Passing index = 0 is functionally equivalent to removing the index parameter and iterating all coins. Both approaches explore permutations but find the minimal count.

Memoization Focus: Since the order of coins doesn’t matter, memoization should track only the remaining amount, not the coin index.

Efficiency: The memoized solution reduces time complexity to O(n \* amount), where n is the number of coins.

**Why Not Use index Like in Combination Sum?**

In Combination Sum (LeetCode 39), index enforces a non-decreasing order to avoid duplicate combinations (e.g., [2,3] and [3,2]). However, Coin Change doesn’t care about order, so restricting the coin selection via index would incorrectly limit valid solutions.

**Final Answer**

Your initial approach (passing index = 0) is correct but inefficient. The alternative is to remove index and iterate all coins directly. The memoization version optimizes this by caching results for each remain value. Both approaches work, but the memoized version is far more efficient.

**Compare pass in 'index' solution and pass in 'i' solution**

Let’s analyze the two solutions using the input coins = [1, 2, 5] and amount = 6. The key difference lies in how the index parameter is used in the recursive calls. We’ll walk through the behavior of each solution step by step.

**Solution 1: Use index (Always Pass index = 0)**

In this solution, the recursive call always passes index = 0, meaning all coins are considered at every step. This leads to exploring all permutations of coin combinations.

**Behavior for coins = [1, 2, 5] and amount = 6**

At each step, the function iterates through all coins (1, 2, 5).

For each coin, it subtracts the coin value from the remaining amount and recursively calls itself with index = 0.

This allows the function to reuse coins and explore all possible combinations, including permutations like 1+2+3 and 2+1+3.

**Example Recursion Tree**

Level 1: Start with remain = 6.

Try coin 1: remain = 5, call helper(coins, 5, 0).

Try coin 2: remain = 4, call helper(coins, 4, 0).

Try coin 5: remain = 1, call helper(coins, 1, 0).

Level 2: For remain = 5:

Try coin 1: remain = 4, call helper(coins, 4, 0).

Try coin 2: remain = 3, call helper(coins, 3, 0).

Try coin 5: remain = 0, call helper(coins, 0, 0) (valid solution).

Level 3: For remain = 4:

Try coin 1: remain = 3, call helper(coins, 3, 0).

Try coin 2: remain = 2, call helper(coins, 2, 0).

Try coin 5: remain = -1, invalid.

And so on...

**Result**

The function explores all combinations, including:

1+1+1+1+1+1 (6 coins)

1+1+1+1+2 (5 coins)

1+1+2+2 (4 coins)

2+2+2 (3 coins)

1+5 (2 coins)

The minimal count is 2 (from 1+5).

**Solution 2: Use i (Pass Current Index i)**

In this solution, the recursive call passes i (the current coin index), meaning only coins from index i onward are considered. This restricts the exploration to combinations in a specific order, effectively avoiding permutations.

**Behavior for coins = [1, 2, 5] and amount = 6**

At each step, the function iterates through coins starting from index.

For each coin, it subtracts the coin value from the remaining amount and recursively calls itself with index = i.

This ensures that coins are selected in a non-decreasing order, avoiding permutations like 2+1+3 (since 1 cannot come after 2).

**Example Recursion Tree**

Level 1: Start with remain = 6, index = 0.

Try coin 1: remain = 5, call helper(coins, 5, 0).

Try coin 2: remain = 4, call helper(coins, 4, 1).

Try coin 5: remain = 1, call helper(coins, 1, 2).

Level 2: For remain = 5, index = 0:

Try coin 1: remain = 4, call helper(coins, 4, 0).

Try coin 2: remain = 3, call helper(coins, 3, 1).

Try coin 5: remain = 0, call helper(coins, 0, 2) (valid solution).

Level 3: For remain = 4, index = 0:

Try coin 1: remain = 3, call helper(coins, 3, 0).

Try coin 2: remain = 2, call helper(coins, 2, 1).

Try coin 5: remain = -1, invalid.

And so on...

**Result**

The function explores combinations in a specific order, such as:

1+1+1+1+1+1 (6 coins)

1+1+1+1+2 (5 coins)

1+1+2+2 (4 coins)

2+2+2 (3 coins)

1+5 (2 coins)

The minimal count is still 2 (from 1+5).

**Key Differences**

|  |  |  |
| --- | --- | --- |
| **Aspect** | Solution 1: Use index | Solution 2: Use i |
| **Coin Selection** | All coins are considered at every step. | Only coins from index i onward are considered. |
| **Exploration** | Explores all permutations (e.g.,  1+2 and 2+1). | Explores combinations in a specific order. |
| **Efficiency** | Highly inefficient due to redundant permutations. | More efficient, avoids redundant permutations. |
| **Correctness** | Correct but inefficient. | Correct and more efficient. |

**Why Solution 2 Is Better**

Avoids Redundant Permutations: By restricting the selection of coins to those starting from index i, Solution 2 avoids exploring permutations like 2+1+3 (which are redundant for Coin Change).

More Efficient: Solution 2 explores fewer paths, making it more efficient than Solution 1.

**Conclusion**

Solution 1 is correct but inefficient because it explores all permutations.

Solution 2 is correct and more efficient because it avoids redundant permutations by enforcing a specific order of coin selection.

For LeetCode 322: Coin Change, Solution 2 is the better approach. However, the optimal solution is to use memoization or bottom-up DP, which further improves efficiency by caching results of subproblems.

**Solution 2: DFS + Memoization (10 min)**

class Solution {

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

        Integer[] memo = new Integer[amount + 1];

        int result = helper(coins, amount, memo);

        return result == Integer.MAX\_VALUE ? -1 : result;

    }

    private int helper(int[] coins, int remain, Integer[] memo) {

        if(remain < 0) {

            return -1;

        }

        if(remain == 0) {

            return 0;

        }

        if(memo[remain] != null) {

            return memo[remain];

        }

        int min = Integer.MAX\_VALUE;

        for(int coin : coins) {

            int tmp = helper(coins, remain - coin, memo);

            if(tmp != -1) {

                min = Math.min(min, tmp + 1);

            }

        }

        return memo[remain] = (min == Integer.MAX\_VALUE ? -1 : min);

    }

}

Time Complexity:

Each subproblem (remain) is computed only once.

There are amount subproblems, and each subproblem takes O(n) time (iterating through all coins starting from i).

Total time complexity: O(n \* amount).

Space Complexity:

The memoization table requires O(amount) space.

The recursion stack also requires O(amount) space.

Total space complexity: O(amount).

**Time and Space Complexity analysis from Deepseek**

Let’s analyze the time and space complexity of the DFS solution where the recursive call passes the current index i (instead of always passing index = 0). Here’s the solution again for reference:

class Solution {

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

int result = helper(coins, amount, 0);

return result == Integer.MAX\_VALUE ? -1 : result;

}

private int helper(int[] coins, int remain, int index) {

if (remain < 0) return -1;

if (remain == 0) return 0;

int min = Integer.MAX\_VALUE;

for (int i = index; i < coins.length; i++) {

int tmp = helper(coins, remain - coins[i], i);

if (tmp != -1) {

min = Math.min(min, tmp + 1);

}

}

return min == Integer.MAX\_VALUE ? -1 : min;

}

}

**1. Time Complexity**

**Recursive Tree Structure**

At each step, the function iterates through the coins starting from the current index i.

If there are n coins, the branching factor of the recursion tree decreases as i increases.

The depth of the recursion tree depends on the remaining amount (remain). In the worst case, the recursion goes as deep as the amount itself (e.g., if the smallest coin is 1, the recursion depth is amount).

**Worst-Case Scenario**

The worst case occurs when the smallest coin is 1, and the recursion explores all possible combinations of coins to make up the amount.

For example, if coins = [1, 2, 5] and amount = 6, the recursion explores paths like:

1+1+1+1+1+1

1+1+1+1+2

1+1+2+2

2+2+2

1+5

**Time Complexity**

The time complexity is O(n^amount), where:

n is the number of coins.

amount is the target amount.

This is because, at each level of recursion, the function branches into n possibilities, and the recursion depth can be as large as amount.

**2. Space Complexity**

**Recursion Stack**

The space complexity is dominated by the recursion stack.

In the worst case, the recursion depth is amount (e.g., when the smallest coin is 1).

**Space Complexity**

The space complexity is O(amount) due to the recursion stack.

**3. Why Is This Inefficient?**

The solution is inefficient because:

It explores all combinations of coins in a specific order (starting from index i), but it still explores many redundant paths.

It doesn’t reuse results of subproblems, leading to exponential time complexity.

**4. Comparison with Passing index = 0**

Passing index = 0: Explores all permutations of coins (e.g., 1+2 and 2+1 are treated as separate paths). Time complexity is O(n^amount).

Passing i: Explores combinations in a specific order, avoiding permutations. However, the time complexity is still O(n^amount) because the recursion depth and branching factor remain the same.

**5. Optimized Solution (Memoization)**

To improve efficiency, we can use memoization to cache results of subproblems. This reduces the time complexity to O(n \* amount) and the space complexity to O(amount).

**Memoized Solution**

**Time Complexity (Memoized)**

Each subproblem (remain) is computed only once.

There are amount subproblems, and each subproblem takes O(n) time (iterating through all coins starting from i).

Total time complexity: O(n \* amount).

**Space Complexity (Memoized)**

The memoization table requires O(amount) space.

The recursion stack also requires O(amount) space.

Total space complexity: O(amount).

**6. Summary**

|  |  |  |
| --- | --- | --- |
| **Solution** | **Time Complexity** | **Space Complexity** |
| Pure DFS (Pass i) | O(n^amount) | O(amount) |
| Memoized DFS (DP) | O(n \* amount) | O(amount) |

The pure DFS solution (passing i) is still inefficient due to its exponential time complexity. The memoized version is much more efficient, with polynomial time complexity. Always prefer the memoized or bottom-up DP approach for this problem!

**Solution 3: 2D DP (30 min)**

**在做 pure DFS 到 2D DP 的转换的过程中就会发现 index 这个参数在顶底转换规律中是没有满足规律的 (在 pure DFS 中没有明确的底，实际上最后需要的顶是 n 即所有 coin 类型都用到)，amount 这个参数是满足规律的 (顶是 amount，底是 0，在DP中从 0 发展到 amount)，简单说，index 这个参数是多余的，在额外制造出的需求中 index 被用来表述使用从 0 到 index 类型的 coin 得到特定 amount**

class Solution {

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

        int n = coins.length;

        // Create a 2D DP table

        // dp[i][j] represent the minimum number of coins needed to make

        // up the amount j using the first i coins.

        double[][] dp = new double[n + 1][amount + 1];

        // Base case: 0 coins needed to make up amount 0

        for(int i = 0; i <= n; i++) {

            dp[i][0] = 0;

        }

        // Use a value larger than the maximum possible number of coins

        for(int j = 1; j <= amount; j++) {

            dp[0][j] = Double.POSITIVE\_INFINITY;

        }

        // Fill the table

        for(int i = 1; i <= n; i++) {

            for(int j = 1; j <= amount; j++) {

                // If the current coin's denomination is greater than the

                // remaining amount, not pick up current coin, exclude it

                if(coins[i - 1] > j) {

                    dp[i][j] = dp[i - 1][j];

                // Choose the minimum between including and excluding the current coin

                // Pick up current denomination coin, hence number of coins

                // used will increase 1, then compare with not pick up case

                // to find minimum number of coins required

                } else {

                    dp[i][j] = Math.min(dp[i - 1][j], dp[i][j - coins[i - 1]] + 1);

                }

            }

        }

        // If dp[n][amount] is still greater than amount, it means

        // it's not possible to make up the amount

        return (int)(dp[n][amount] == Double.POSITIVE\_INFINITY ? -1 : dp[n][amount]);

    }

}

Time Complexity: O(n \* amount), where n is the number of coins and amount is the target amount.

Space Complexity: O(n \* amount) for the 2D DP table.

**Refer to Deepseek**

To convert the pure DFS solution into a **bottom-up 2D DP solution**, we need to:

Define the **DP table** and its dimensions.

Identify the **base cases** and initialize the DP table.

Define the **transition** between subproblems.

Iteratively fill the DP table from the smallest subproblems to the largest.

**DP Table Definition:**

Let dp[i][j] represent the minimum number of coins needed to make up the amount j using the first i coins.

**Transition:**

For each coin at index i, we decide whether to **include it** or **exclude it** in the solution:

If we **include** the coin, we reduce the remaining amount by **coins[i-1]** and add 1 to the result.

If we **exclude** the coin, we keep the remaining amount the same and move to the next coin.

**Base Cases:**

dp[0][j] = Infinity for all j > 0: No coins are available, so it's impossible to make up the amount.

dp[i][0] = 0 for all i: No coins are needed to make up the amount 0.

public class CoinChange {

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

int n = coins.length;

// Create a 2D DP table

int[][] dp = new int[n + 1][amount + 1];

// Initialize the DP table

for (int j = 1; j <= amount; j++) {

dp[0][j] = amount + 1; // Use a value larger than the maximum possible number of coins

}

for (int i = 0; i <= n; i++) {

dp[i][0] = 0; // Base case: 0 coins needed to make up amount 0

}

// Fill the DP table

for (int i = 1; i <= n; i++) {

for (int j = 1; j <= amount; j++) {

// If the current coin is greater than the remaining amount, exclude it

if (coins[i - 1] > j) {

dp[i][j] = dp[i - 1][j];

} else {

// Choose the minimum between including and excluding the current coin

dp[i][j] = Math.min(dp[i - 1][j], dp[i][j - coins[i - 1]] + 1);

}

}

}

// If dp[n][amount] is still greater than amount, it means it's not possible to make up the amount

return dp[n][amount] > amount ? -1 : dp[n][amount];

}

}

**Explanation of the Bottom-Up 2D DP Solution**

DP Table Initialization:

dp[0][j] = amount + 1: This represents an impossible scenario (no coins available to make up the amount j).

dp[i][0] = 0: This represents the base case where no coins are needed to make up the amount 0.

Transition:

For each coin i and each amount j:

If the coin value **coins[i-1]** is greater than the amount j, we **exclude** the coin and take the result from the previous row: **dp[i][j] = dp[i-1][j]**.

Otherwise, we **choose the minimum** between:

Excluding the coin: **dp[i-1][j]**.

Including the coin: dp[i][j - coins[i-1]] + 1.

Result:

The final result is stored in **dp[n][amount]**. If it’s greater than **amount**, it means it’s impossible to make up the amount with the given coins.

**Example Walkthrough**

Let’s say **coins = [1, 2, 5]** and **amount = 6**. Here’s how the DP table is filled:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| **0** | 0 | 7 | 7 | 7 | 7 | 7 | 7 |
| **1** | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| **2** | 0 | 1 | 1 | 2 | 2 | 3 | 3 |
| **5** | 0 | 1 | 1 | 2 | 2 | 1 | 2 |

The final result is **dp[3][6] = 2** (using coins 5 and 1).

**Time and Space Complexity**

Time Complexity: **O(n \* amount)**, where n is the number of coins and amount is the target amount.

Space Complexity: **O(n \* amount)** for the 2D DP table.

**Solution 4: 1D DP (30 min)**

Once you have the 2D DP solution, you can further optimize it to 1D DP by observing that each row of the DP table depends only on the previous row. This reduces the space complexity to O(amount).

class Solution {

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

        int n = coins.length;

        // Create a 1D DP array

        // dp[j] represents the minimum number of coins needed to make up amount j

        double[] dp = new double[amount + 1];

        // Initialize the DP array

        Arrays.fill(dp, Double.POSITIVE\_INFINITY);

        // Base case: 0 coins needed to make up amount 0

        dp[0] = 0;

        // Fill the DP array

        for(int i = 1; i <= n; i++) {

            for (int j = 1; j <= amount; j++) {

                // Update dp[j] by choosing the minimum between:

                // 1. Not using the current coin (dp[j])

                // 2. Using the current coin (dp[j - coins[i - 1]] + 1)

                if(j >= coins[i - 1]) {

                    dp[j] = Math.min(dp[j], dp[j - coins[i - 1]] + 1);

                }

            }

        }

        // If dp[amount] is still greater than amount, it means it's not possible to make up the amount

        return (int)(dp[amount] == Double.POSITIVE\_INFINITY ? -1 : dp[amount]);

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to Deepseek**

To convert the 2D DP solution into a 1D DP solution, we need to observe that the dp table only depends on the current row (dp[i][...]) and the previous row (dp[i-1][...]). This allows us to use a 1D array to store the results for the current amount and update it iteratively.

**1D DP Solution**

class Solution {

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

// Create a 1D DP array

// dp[j] represents the minimum number of coins needed to make up amount j

int[] dp = new int[amount + 1];

// Initialize the DP array

Arrays.fill(dp, amount + 1); // Use a value larger than the maximum possible number of coins

dp[0] = 0; // Base case: 0 coins needed to make up amount 0

// Fill the DP array

for (int coin : coins) {

for (int j = coin; j <= amount; j++) {

// Update dp[j] by choosing the minimum between:

// 1. Not using the current coin (dp[j])

// 2. Using the current coin (dp[j - coin] + 1)

dp[j] = Math.min(dp[j], dp[j - coin] + 1);

}

}

// If dp[amount] is still greater than amount, it means it's not possible to make up the amount

return dp[amount] > amount ? -1 : dp[amount];

}

}

**Explanation of the 1D DP Solution**

DP Array Initialization:

dp[j] is initialized to amount + 1 (a value larger than the maximum possible number of coins). This acts as a placeholder for "infinity."

dp[0] = 0 because no coins are needed to make up the amount 0.

Filling the DP Array:

For each coin, iterate through all amounts from coin to amount.

Update dp[j] by choosing the minimum between:

Not using the current coin: dp[j] (the existing value).

Using the current coin: dp[j - coin] + 1 (add 1 coin to the solution for j - coin).

Result:

If dp[amount] is still greater than amount, it means it’s impossible to make up the amount with the given coins. Otherwise, return dp[amount].

**Why Does This Work?**

The 2D DP solution uses dp[i][j] to represent the minimum number of coins needed to make up amount j using the first i coins.

In the 1D DP solution, we reuse the same array (dp[j]) to store the results for the current coin. This works because:

When processing coin i, dp[j] depends only on:

dp[j] (the result without using coin i).

dp[j - coin] + 1 (the result using coin i).

By iterating through the coins and updating dp[j] in place, we effectively simulate the 2D DP table.

**Example Walkthrough**

Let’s use coins = [1, 2, 5] and amount = 6 to illustrate how the 1D DP solution works:

**Initialization**

dp = [0, 7, 7, 7, 7, 7, 7] (where 7 is amount + 1).

**Processing Coin 1**

For j = 1 to 6:

dp[1] = min(7, dp[0] + 1) = 1

dp[2] = min(7, dp[1] + 1) = 2

dp[3] = min(7, dp[2] + 1) = 3

dp[4] = min(7, dp[3] + 1) = 4

dp[5] = min(7, dp[4] + 1) = 5

dp[6] = min(7, dp[5] + 1) = 6

Updated dp = [0, 1, 2, 3, 4, 5, 6].

**Processing Coin 2**

For j = 2 to 6:

dp[2] = min(2, dp[0] + 1) = 1

dp[3] = min(3, dp[1] + 1) = 2

dp[4] = min(4, dp[2] + 1) = 2

dp[5] = min(5, dp[3] + 1) = 3

dp[6] = min(6, dp[4] + 1) = 3

Updated dp = [0, 1, 1, 2, 2, 3, 3].

**Processing Coin 5**

For j = 5 to 6:

dp[5] = min(3, dp[0] + 1) = 1

dp[6] = min(3, dp[1] + 1) = 2

Updated dp = [0, 1, 1, 2, 2, 1, 2].

**Result**

dp[6] = 2 (using coins 1 and 5).

**Time and Space Complexity**

Time Complexity: O(n \* amount), where n is the number of coins and amount is the target amount.

Space Complexity: O(amount) for the 1D DP array.

**Summary**

|  |  |  |
| --- | --- | --- |
| **Solution** | **Time Complexity** | **Space Complexity** |
| 2D DP | O(n \* amount) | O(n \* amount) |
| 1D DP | O(n \* amount) | O(amount) |

The 1D DP solution is more space-efficient and achieves the same time complexity as the 2D DP solution. It’s the preferred approach for this problem!

**Question: I have a link tell us how to downgrade 2D DP to 1D DP:**

<https://afteracademy.com/blog/when-to-convert-a-2d-dp-array-to-1d-dp-array-and-how/>

In our Leetcode 322 2D DP solution, we have below equation:

dp[i][j] = Math.min(dp[i - 1][j], dp[i][j - coins[i - 1]] + 1);

which means row i column j's status depends on previous row i - 1 column j - coins[i - 1]'s status, its a column on the left of column j, as the theory in the word doc, when we downgrade 2D DP to 1D DP, we should update(loop) backwards, since the elements of the previous row whose column index > j will not affect the update of dp[i][j] as we will not touch them, and these elements keep stable when we update dp[i][j].

But why in our 1D DP solution here, it still update(loop) forwards ?

**Refer to Deepseek**

Okay, so I'm trying to understand why the 1D DP solution for the LeetCode 322 Coin Change problem uses a forward loop even though the article suggests updating backwards when converting from 2D to 1D. Let me break this down step by step.

First, let's recall the 2D DP approach. The recurrence relation is:

dp[i][j] = min(dp[i-1][j], dp[i][j - coins[i-1]] + 1)

This means that for each coin (i) and each amount (j), the value depends on two things: not using the current coin (from the previous row, dp[i-1][j]) or using the current coin (from the current row, dp[i][j - coins[i-1]] + 1).

In the article, it mentions that when dependencies are on the previous row or left columns, we can optimize space. Specifically, if the current row depends only on the previous row and left elements of the current row, we can update the array in reverse to prevent overwriting values that are yet to be used.

But in the provided 1D solution, the loop is forward. Here's the code snippet:

for(int i = 1; i <= n; i++) {

for (int j = 1; j <= amount; j++) {

if(j >= coins[i - 1]) {

dp[j] = Math.min(dp[j], dp[j - coins[i - 1]] + 1);

}

}

}

The inner loop runs from `j = 1` to `amount`, which is a forward traversal. According to the article, if dependencies are on the left (like j - coins[i-1]), we should iterate backwards to avoid using already updated values. So why is this solution using forward iteration?

Let me analyze the dependencies again. In the 2D version, dp[i][j] depends on dp[i-1][j] (previous row, same column) and dp[i][j - coins[i-1]] (current row, left column). When converting to 1D, the previous row's value (dp[i-1][j]) is represented by the current dp[j] before any updates in the current iteration. The current row's left value (dp[i][j - coins[i-1]]) is now dp[j - coins[i-1]] in the 1D array.

Wait, in the 1D version, we are essentially overwriting dp[j] in place. For each coin, when processing j from left to right, dp[j - coins[i-1]] would have already been updated if we processed a smaller j earlier. But in the 2D case, the current row's left value is part of the same row. However, in the 1D array, since we are reusing the same array for all rows, processing left to right would mean that dp[j - coins[i-1]] might have already been updated in the current iteration (i.e., for the current coin), which is not the case in the 2D version where it's still the previous row's value.

**Ah, there's a mistake here! In the original 2D recurrence, when using the current row's left value (dp[i][j - coins[i-1]]), it's part of the same row.** But in the 1D version, if we process j in forward order, dp[j - coins[i-1]] would have been updated in the current iteration for the same coin. This actually allows using the current coin multiple times, which is correct for the Coin Change problem where coins can be reused infinitely.

Wait, but the article suggested backward iteration to prevent overwriting. But in this case, forward iteration allows multiple uses of the same coin, which is exactly what we need. Let me think again. The Coin Change problem allows unlimited use of coins. So when building up the dp array, each time we process a coin, we want to update the amounts in such a way that using the same coin multiple times is allowed. **Forward iteration allows this because once you update dp[j], when you get to a higher j, you can reuse the already updated dp[j - coins[i]] which includes the current coin's contribution.**

In contrast, if we used backward iteration (processing j from amount down to 1), then when calculating dp[j], dp[j - coins[i]] would refer to the state before processing the current coin, which corresponds to the previous row in the 2D version. This would prevent multiple uses of the same coin. But in Coin Change, we do want to allow multiple uses. Therefore, forward iteration is correct here because it allows using the same coin multiple times, which aligns with the problem's requirement.

**So the key difference is whether the problem allows unlimited use of items (like Coin Change) or only once (like the Partition Equal Subset Sum problem). For unlimited use (unbounded knapsack), you iterate forward. For single use (0/1 knapsack), you iterate backward. The article's example (Partition Equal Subset Sum) is a 0/1 knapsack problem, hence the backward iteration. But Coin Change is an unbounded knapsack problem, so forward iteration is appropriate.**

Therefore, the direction of the loop (forward or backward) depends on whether items can be reused. In the Coin Change problem, since coins can be reused, we use forward iteration to allow multiple uses, and that's why the 1D DP solution loops forward.

The direction of the loop (forward or backward) in the 1D DP solution depends on whether the problem allows unlimited use of items (unbounded knapsack) or single use (0/1 knapsack). Here’s why the LeetCode 322 Coin Change solution uses a forward loop:

**Key Insight**

Coin Change allows unlimited use of coins (unbounded knapsack).

The recurrence in the 2D DP is:dp[i][j] = min(dp[i-1][j], dp[i][j - coins[i-1]] + 1).

dp[i][j - coins[i-1]] refers to the current row (allowing reuse of the current coin).

**1D DP Conversion**

When converting to 1D DP:

dp[j] depends on dp[j] (previous row) and dp[j - coins[i-1]] (current row, updated in the same iteration).

Forward iteration (from j = 1 to amount) ensures that dp[j - coins[i-1]] includes the current coin’s contribution, allowing unlimited reuse.

**Why Not Backward Iteration?**

Backward iteration (from j = amount to 1) would reference dp[j - coins[i-1]] from the previous iteration (i.e., the previous row), which would restrict the solution to single use of coins (incorrect for Coin Change).

**Summary**

|  |  |  |
| --- | --- | --- |
| **Problem Type** | **Direction** | **Reason** |
| Unbounded Knapsack (Coin Change) | Forward | Allows reuse of the current coin by referencing updated values in the same iteration. |
| 0/1 Knapsack (Partition Equal Subset Sum) | Backward | Prevents reuse by referencing values from the previous iteration. |

**Final Answer**

The 1D DP solution for LeetCode 322 uses forward iteration because the problem allows unlimited reuse of coins. Forward iteration ensures that dp[j - coins[i-1]] reflects the current coin’s contribution, enabling multiple uses. This aligns with the unbounded knapsack nature of the problem. In contrast, backward iteration is used for 0/1 knapsack problems (like Partition Equal Subset Sum) to restrict item reuse.

Code Validation:

The provided 1D DP solution is correct and efficient for Coin Change.

**Refer to**

[When to Convert a 2-D DP Array To 1-D DP Array And How](note://74A2DD972C0F4106B2A5D870091EBE07)

[L39.Combination Sum](note://79A77D14319840B9A134C272EA618600)

[L40.Combination Sum II (Refer L39.Combination Sum and L90/P11.2.Subsets II)](note://7895D4AD64EC43CA903579F3C8F803AE)

[L46.P11.3.Permutations (Ref.L77)](note://0506470DCA494FCCB9BC6CC154718ED0)

[L216.Combination Sum III](note://WEB322b73c0787cd12b4eed700838bed14d)

[L377.P16.4.Combination Sum IV(Backpack VI) (Refer L39.Combination Sum)](note://8000F8241A0B4F3AA39E5CA16B6A7BB8)