Design and Analysis of Naive and Optimized Algorithms for the Coin Change Problem

Problem Description

Given a set of coin denominations coins = {c1, c2, ..., ck} and a target amount A (a non-negative integer), the objective is to determine the minimum number of coins required to form exactly the amount A, assuming an unlimited supply of each coin denomination.

If it is possible to form the amount A, the algorithm outputs the minimum number of coins required. If it is not possible to form A using the given denominations, the algorithm outputs -1.

This problem is an optimization problem, where the goal is to minimize the number of coins used.

Input

- A list/array of integers coins (coin denominations), each greater than 0.
- An integer amount where amount ≥ 0.

Output

- An integer representing the minimum number of coins needed to form amount, or -1 if it is impossible.

Three Illustrative Input–Output Examples

Example 1 (Normal case – multiple options exist)

Input:
coins = [1, 3, 4]
amount = 6

Output:
2

Explanation:
The amount 6 can be formed as 3 + 3, which uses the minimum number of coins.

Example 2 (Impossible case)

Input:
coins = [2, 4]
amount = 7

Output:
-1

Explanation:
All possible combinations of the given coins produce even sums, so forming 7 is impossible.

Example 3 (Edge case)

Input:
coins = [5, 10]
amount = 0

Output:
0

Explanation:
No coins are needed to form an amount of 0.

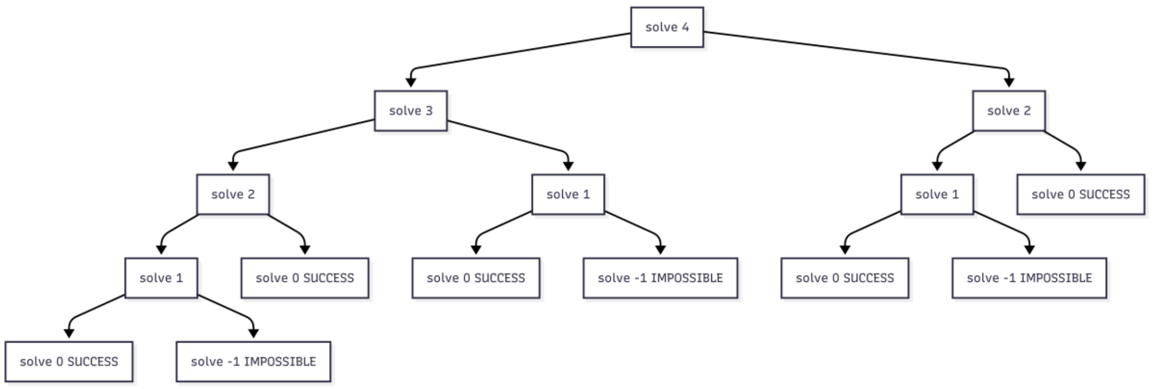
Naive Algorithm (Recursive Approach)

The naive approach explores all possible combinations of coins recursively. For each amount, it tries every coin denomination and solves the remaining amount. This approach does not store intermediate results, which leads to repeated computations.

Naive Algorithm – Pseudocode

```
FUNCTION CoinChangeNaive(coins, amount)
    result ← Solve(amount)
    IF result == IMP:
        RETURN -1
    ELSE:
        RETURN result

FUNCTION Solve(rem)
    IF rem == 0:
        RETURN 0
    IF rem < 0:
        RETURN IMP
    minCoins ← IMP
    FOR each coin c in coins:
        sub ← Solve(rem - c)
        IF sub ≠ IMP:
            minCoins ← MIN(minCoins, sub + 1)
    RETURN minCoins
```



Asymptotic Time and Space Complexity (Naive Algorithm)

We consider the naive recursive algorithm for Coin Change and let:
- A = target amount
- k = |coins| (number of coin denominations)
- c_min = minimum coin denomination

The recurrence relation in the worst case can be written as:

$$T(A) = \begin{cases} \Theta(1), & A \leq 0 \\ \sum\limits_{c \in coins} T(A-c) + \Theta(1), & A > 0 \end{cases}$$

If we assume that all branches recursively subtract the smallest coin denomination, then in the worst case we approximate the recurrence as:

T(A) ≈ k · T(A – 1) + Θ(1)

To analyze the recursion tree, we note that each level multiplies the number of calls by k (branching factor), and the depth h is approximately: h ≈ A / c_min

Thus, the total time is:

T(A) ≈ Θ(1 + k + k^2 + ... + k^h)

Since this is a geometric series:

1 + k + k^2 + ... + k^h = Θ(k^h)

Therefore:

T(A) = Θ(k^h) = Θ(k^(A / c_min))

If c_min = 1 (common case):T(A) = Θ(k^A)

This indicates exponential time complexity in terms of the amount A.

Space Complexity (recursion depth):
Space grows linearly with the recursion depth. Since the maximum depth occurs when the algorithm repeatedly subtracts the smallest coin denomination, the worst-case recursion depth is A/c_{min}. Therefore, the worst-case space complexity is:

$$S(A) = \Theta\left(\frac{A}{c_{\text{min}}}\right)$$

If c_min = 1: S(A) = Θ(A)

Optimized Algorithm (Dynamic Programming)

The optimized solution uses dynamic programming to avoid recomputing subproblems. It builds the solution bottom-up by storing the minimum number of coins required for each intermediate amount.

Optimized Algorithm – Pseudocode

```
FUNCTION CoinChangeOptimized(coins, amount)
    IMP ← amount + 1
    CREATE array mc[0 … amount]
    mc[0] ← 0
    FOR x ← 1 TO amount:
        mc[x] ← IMP
        FOR each coin c in coins:
            IF x - c ≥ 0:
                mc[x] ← MIN(mc[x], mc[x - c] + 1)
    IF mc[amount] == IMP:
        RETURN -1
    ELSE:
        RETURN mc[amount]
```

Asymptotic Time and Space Complexity (Optimized Algorithm)

he optimized Coin Change algorithm employs a bottom–up dynamic programming strategy to compute the minimum number of coins required for all values from 0 up to the target amount A. Let $k = | coins |$represent the number of coin denominations. For each value $x$in

the range $1 \leq x \leq A$, the algorithm iterates over all $k$coins and performs constant–time operations (comparisons and updates). As a result, the total running time can be expressed as:

$$T(A) = \sum_{x=1}^{A} \Theta(k) = \Theta(A \cdot k)$$

The space complexity is dominated by the dynamic programming table, which stores

$A + 1$ entries. Aside from this table, only constant additional space is required. Therefore:

$$S(A) = \Theta(A)$$

Empirical Analysis

Both algorithms were implemented in Python to measure performance. For the naive recursive version, we counted recursive calls. For the optimized dynamic-programming version, we counted DP table updates.

Results:

- coins = [1, 3, 4], amount = 6 → naive: 46 calls | optimized: 18 operations
- coins = [2, 4], amount = 7 → naive: 15 calls | optimized: 14 operations
- coins = [1, 3, 4], amount = 10 → naive: 313 calls | optimized: 30 operations

These results show that both algorithms return correct answers, but the number of operations in the naive method grows very rapidly, while the optimized approach grows slowly and remains consistent.

Results Comparison

Theoretical time complexity:

$$T_{naive}(A) = \Theta(k^A) \text{ and } T_{opt}(A) = \Theta(A \cdot k)$$

The empirical data supports this: as the amount increases from 6 to 10, recursive calls in the naive algorithm jump from 46 to 313, while operations in the optimized algorithm rise only from 18 to 30.

For a large input such as coins = [1, 2] and amount = 1,000,000, the naive approach would require around Θ(2^{1,000,000}) operations (not realistic), while the optimized solution would need Θ(1,000,000 × 2), which is easily manageable.

Conclusion

Both algorithms correctly solve the Coin Change problem, but their performance differs greatly. The naive recursive method becomes exponentially slower as the amount grows, while the optimized dynamic-programming approach scales efficiently and remains practical even for very large inputs. This confirms that algorithmic design choices have a major impact on real-world performance.