INDIAN INSTITUTE OF TECHNOLOGY, KHARAGPUR

SCRIBE ON COIN SELECTION PROBLEM

Instructor: Dr. Partha P. Chakraborty

Krishna Biswakarma

24CS60R71

Introduction to the Coin Selection Problem

The Coin Selection Problem is a well-known dynamic programming problem, typically divided into two main types based on coin availability:

1. Unlimited Supply of Coins: Given a set of coins and a target amount, determine the number of ways to achieve the target amount using the coins, where each coin can be used as many times as needed.
2. Limited (Single Use) Supply of Coins: In this variation, each coin can be used only once. The goal is to determine whether it is possible to reach the target amount with the given set of coins.

Problem Statement

Given a set "C" of "n" coins with denominations {c1, c2, ..., cn} and a target value "V", the goal is to find the minimum number of coins needed from "C" to sum exactly to "V".
Example -
C = {8, 6, 5, 2, 1, 3}, V = 11
Possible Solutions:
S1 = {8, 3, 1}
S2= {6, 5} → Minimum
The optimal solution is S2, requiring the minimum number of coins.

Terms Required

Coins (S, T, X, Z, N)
S: The set of coins selected so far.
T: The set of coins is still available to be chosen.
X: The total value of the coins in set S.
Z: The remaining value needed to reach the target V from the set T.
N: The number of coins selected so far.

RECURSIVE SOLUTION

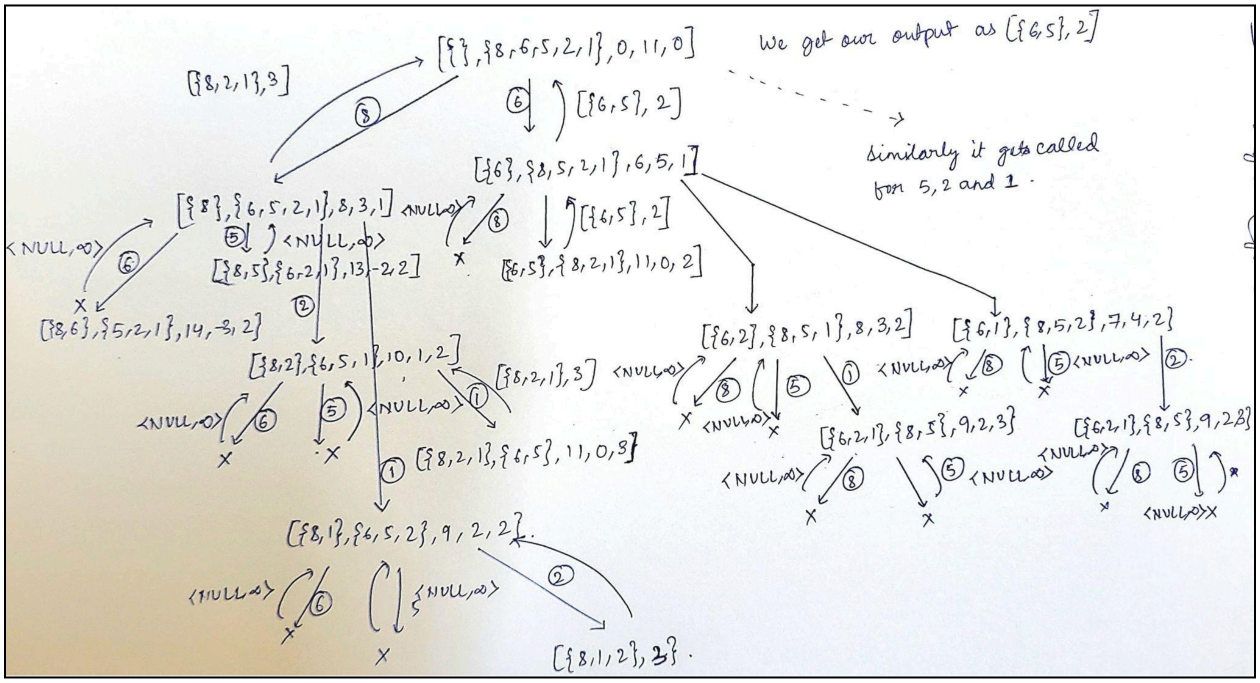
The initial approach to solving the Coin Selection Problem involves using a recursive method to explore all possible combinations of coins to reach the target value. While this method is simple and direct, it is not very efficient because it may evaluate the same subproblems multiple times.

Recurrence Relation

We define the recursive function f(T, Z) to determine the minimum number of coins required to make the value Z using the remaining coins in set "T". The recurrence relation is given by:

f(T,Z) = min{1 + f(T - t[i], Z - t[i]) | t[i] ∈ T and Z - t[i] > 0}

Recursion Tree



Proof of Induction

Base Case:
For “z = 0”, the minimum number of coins required is “0”, which is trivially correct.

Inductive Hypothesis:
Assume that the function “f(T, k)” correctly determines the minimum number of coins needed for any value “k < z”.

Inductive Step:
To find the minimum number of coins for “z”, the recurrence relation examines each coin “t[i]” that can be used (z - t[i] ≥ 0) and calculates “1 + f(T - {t[i]}, z - t[i])”. According to the inductive hypothesis, “f(T - {t[i]}, z - t[i])” provides the correct result, so by considering the minimum value across all valid t[i], we obtain the correct number of coins needed for “z”.
Conclusion:
By induction, the recurrence relation is validated to correctly compute the minimum number of coins required for any value “z”.

Pseudo Code

```
function coins(S, T, x, z, n):
  if z == 0:
    return <S, n>  # Solution found with current set S
  if z < 0:
    return <NULL, ∞>  # Impossible solution
  if T is empty:
    return <NULL, ∞>  # No coins left to consider

  Pmin = NULL
  min = ∞  # Initialize min to a large value

  for i = 1 to |T| do:
    ti = T[i]  # Current coin
    W = S + {t[i]}  # Add current coin to selected set
    U = T - {ti}  # Remaining coins to consider
    <p', d'> = coins(W, U, x + ti, z - ti, n + 1)  # Recursive call

    if d' < min:
        min = d'
        Pmin = p'

  return <Pmin, min>
```

Time Complexity

The time complexity of this approach is **O(n!)** due to the factorial growth in the number of subproblems being solved.

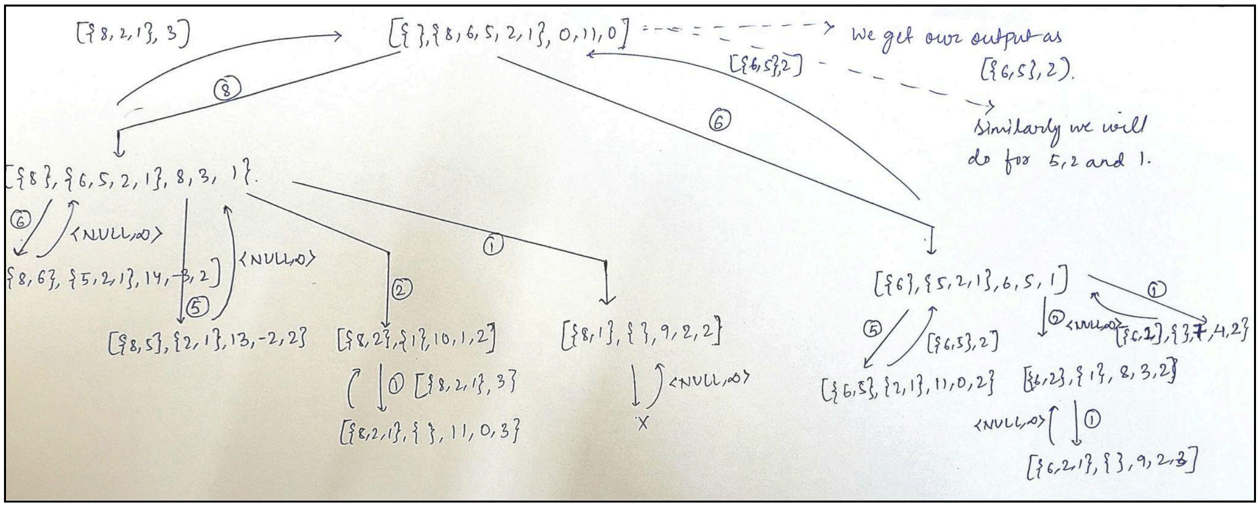
IMPROVED RECURSIVE SOLUTION

The initial recursive approach generates a recursion tree where many subproblems are solved multiple times. To enhance efficiency, an improved method ensures that each coin is considered only once per problem instance, thereby minimizing redundant computations.

Recurrence Relation

$f(T,x) = \min\{1 + f(T-\{t[1], t[2], \ldots, t[i]\}, z-t[i]) \mid t[i] \in T \text{ and } z - ti > 0\}$

Recursion Tree



Proof of Correctness

Base Case:

- For z = 0, the minimum number of coins needed is 0, which is trivially correct.

Inductive Hypothesis:

- Assume that the function f(T, k) correctly computes the minimum number of coins for any value k < z.

Inductive Step:

Consider the value z. The function f(T,z) considers each coin tit_iti in T that can be used (z-t[i]≥ 0) and calculates 1+f(T-{t1,t2,...,ti},z-t[i]). By the inductive hypothesis, f(T-{t1,t2,..., ti},z-ti) correctly computes the minimum number of coins for the remaining value z-t[i]. By minimising all possible coins, the function ensures that the minimum number of coins is chosen to achieve z. Therefore, by induction, the recurrence relation correctly computes the minimum number of coins needed for any value z.

Pseudo Code

```
function coins(S, T, x, z, n):
  if z == 0:
    return <S, n>  # Solution found with current set S
  if z < 0:
    return <NULL, ∞>  # Impossible solution
  if T is empty:
    return <NULL, ∞>  # No coins left to consider

  min = ∞
  for i = 1 to |T|:
    ti = T[i]  # Current coin
    remaining_coins = T - {t1, t2,..., ti}
    <p', d'> = 1 + coinChange(remaining_coins, z - ti)
    if d' < min:
      min = d'
      Pmin = p'

  return <Pmin, min>
```

Time Complexity

The improved recursive method has a time complexity of O(2^n), which is more efficient than the original factorial approach.

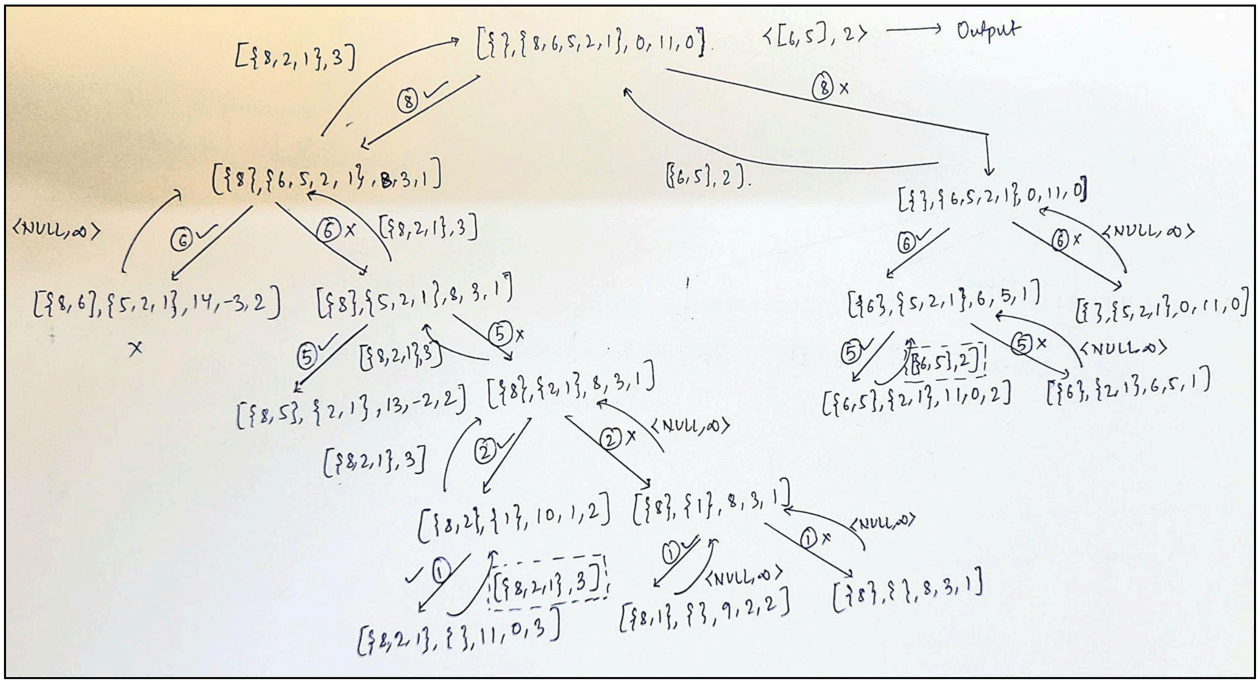
ALTERNATE RECURSIVE SOLUTION

An alternative recursive approach to the Coin Selection Problem can be formulated using the **inclusion-exclusion** principle. This method evaluates two main scenarios: including the current coin and excluding it. By combining these recursive solutions, it identifies the optimal result.

Recurrence Relation

f(T, z) = min{1 + f(T - t[i], z - t[i]),f(T - {t[i], z)}

Recursion Tree



Proof of Correctness

Base Case:

- If z = 0, the function returns <S, n> because no coins are needed to achieve a target sum of 0.
- If z < 0 or T is empty, the function returns <NULL, ∞>, indicating that it's impossible to achieve the target.

Inductive Hypothesis:

- Assume that the function correctly computes the minimum number of coins for any value k < z.

Inductive Step:

For a given value “z”, the function examines both scenarios: including the current coin and excluding it. It then chooses the minimum number of coins required between these two cases. This approach ensures that the optimal solution is found by systematically exploring all possible combinations, thereby guaranteeing that no potential solutions are missed.

Pseudo Code

```
function coins2(S, T, x, z, n):
  if z == 0:
    return <S, n>  # Solution found with current set S
  if z < 0:
    return <NULL, ∞>  # Impossible solution
  if T is empty:
    return <NULL, ∞>  # No coins left to consider

  # Recursive cases: include or exclude the first coin
  <P1, d1> = coins2(S + {t1}, T - {t1}, x + t1, z - t1, n + 1)
  <P2, d2> = coins2(S, T - {t1}, x, z, n)

  if d1 <= d2:
    return <P1, d1>
  else:
    return <P2, d2>
```

Time Complexity

This method also has a time complexity of $O(2^n)$, similar to the improved recursive approach.

DYNAMIC PROGRAMMING SOLUTION

The most effective solution to the Coin Selection Problem is to use dynamic programming. This approach efficiently calculates the minimum number of coins required to achieve a given value “z”, avoiding the exponential time complexity associated with naive recursive methods and delivering an optimal solution in polynomial time.

Recurrence Relation

For each coin c[i] in the set C, and for each value j from c[i] to V:
Dp[j] = min(dp[j], 1 + dp[j - c[i]])

Proof of Correctness

Base Case: We initialize dp[0] = 0 correctly, as no coins are needed to achieve the value 0.

Inductive Step: For each value j from 1 to V, the dynamic programming array is updated to reflect the minimum number of coins needed to reach that value, based on the available coin denominations. At the end of this process, dp[V] will show the minimum number of coins required to make V, thereby ensuring the solution's correctness.

Optimal Substructure: The value for dp[j] is calculated using the solution for dp[j - c[i]], where c[i] represents a coin denomination. This method guarantees that the final solution is built from optimal solutions to smaller subproblems, maintaining the property of optimal substructure.

Pseudo Code

```
function coinChange(C, V):
  dp = array of size (V + 1)
  dp[0] = 0
  for i = 1 to V:
    dp[i] = ∞

  for each coin ci in C:
    for j = ci to V:
      dp[j] = min(dp[j], 1 + dp[j - ci])

  if dp[V] == ∞:
    return -1  # or any other indication that it's impossible to make the value V
  else:
    return dp[V]
```

Time Complexity

The time complexity is O(nV), where n is the number of coins and V is the target value. This is due to the nested loop structure.

Space Complexity

The space complexity is O(V), as it only requires an array of size V+1.

BRANCH AND BOUND

Branch and Bound is a well-known algorithmic approach for tackling combinatorial optimization problems. It systematically explores potential solutions through a state space search, conceptualizing the set of candidate solutions as a rooted tree with the complete set at the root. The algorithm then examines various branches of this tree, each representing different subsets of the solution space. To understand the technique more clearly, let's break down the fundamental concepts and steps involved in the Branch and Bound method.

Basic Concepts

- Bounding: Calculate an upper or lower bound for the objective function within a subproblem. If the bound indicates that the subproblem cannot contain a better solution than the best solution found so far, the subproblem is discarded.
- Branching: Divide the problem into smaller subproblems and solve them individually.
- Pruning: Use the bounds to discard subproblems that cannot possibly contain the optimal solution.

PRUNING

Pruning is a key concept in the Branch and Bound method, focusing on eliminating subproblems that cannot surpass the best-known solution. By using bounds (such as upper or lower bounds), this technique allows for the exclusion of branches in the search tree that are unlikely to yield better results, thus minimizing the number of nodes that need to be explored.

Maintain a global current best CB = α (initially)

Recursion is evaluated in a depth-first manner

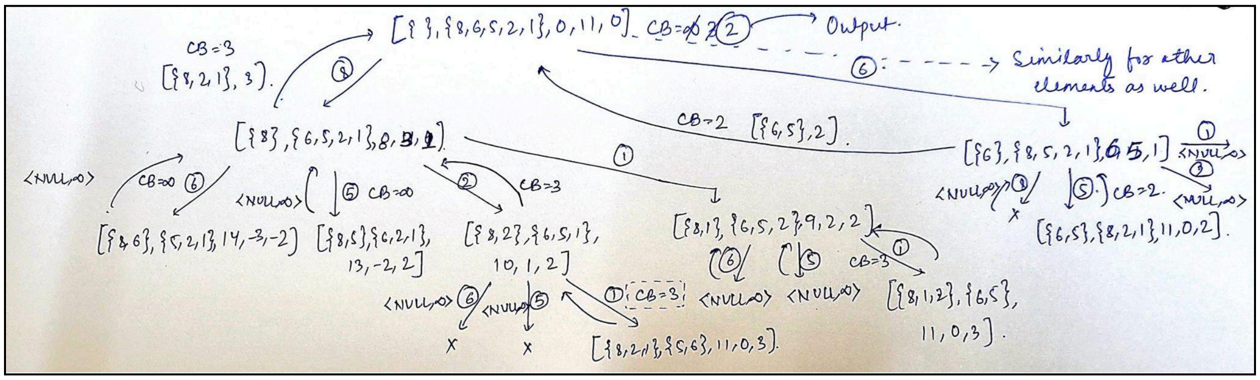
Base Conditions are revised for Pruning

```
1. If (z=0) {
       if (n < CB), CB = n [update the current best]
     return (S,n)
   }
2. If (z<0)
   return (NULL, α)
3. If (T = NULL)
   return (NULL, α)
4. If (n > CB)
   return (NULL, α)
```

Notes:

- "CB" stands for "Current Best".
- "α" represents an Initial Value.
- z, n, S, and T are variables used within the conditions.

Recursion Tree



Recurrence Relation

f(n)=min c∈C, c ≤ n (1 + f(n - c))

Time Complexity

T(n)=O(n*c) (with memoization)