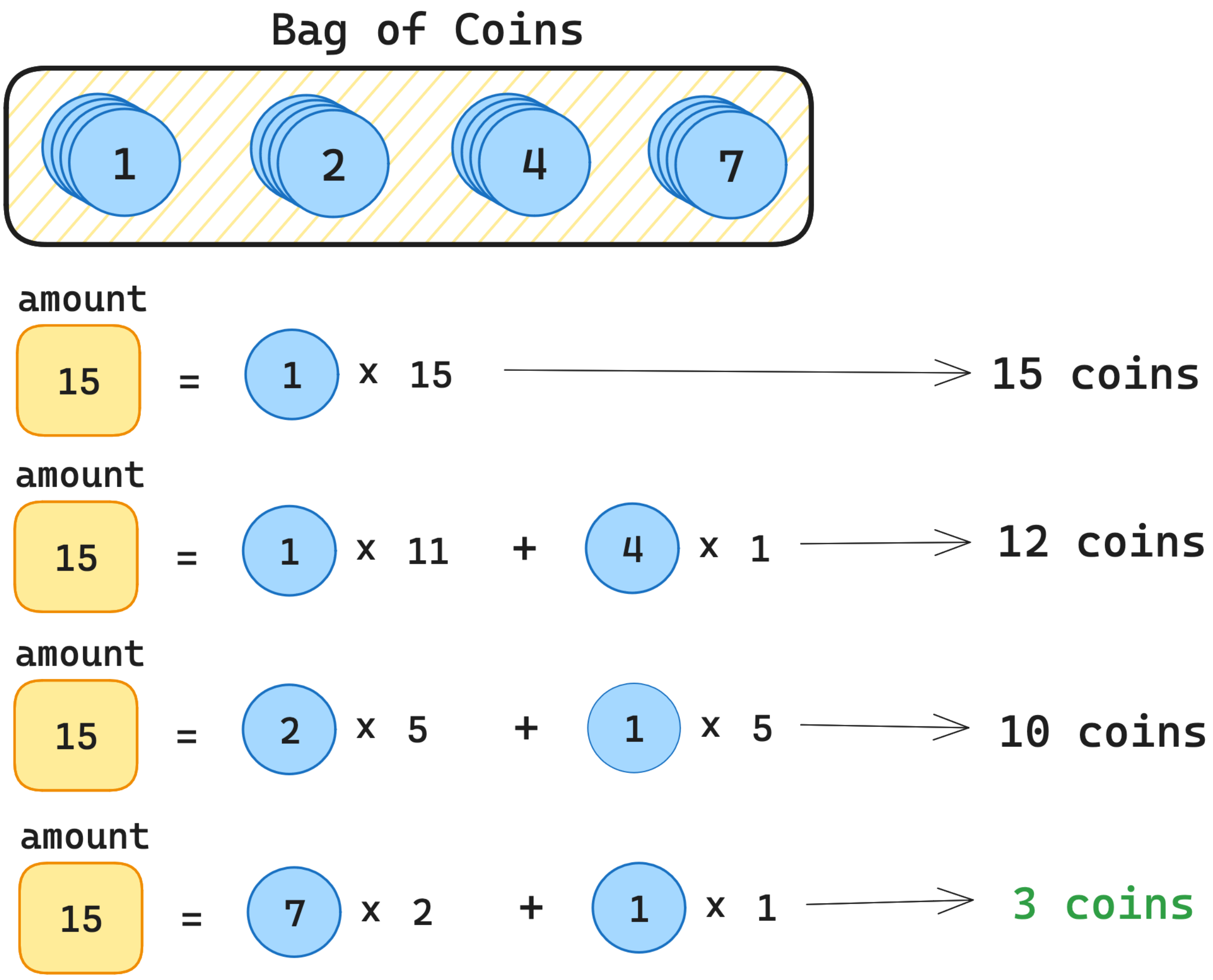
Lecture 26: Dynamic Programming

Date: 11/16/2023

Coin Change



Equation

⇒ We want to write an equation that can generate the answer for our main problem based on smaller subproblems.

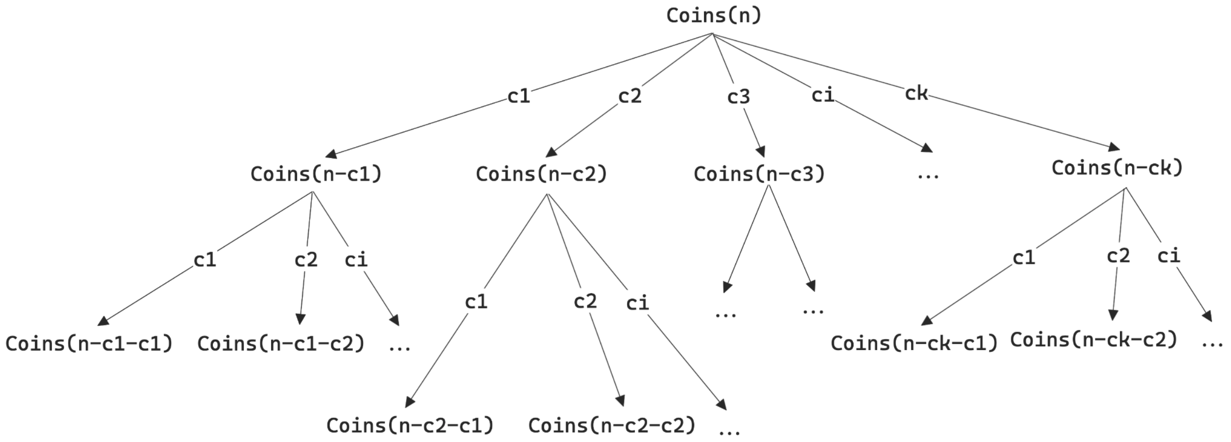
Intuition: we want to use coins so that we can get the target amount n.
→ Each time, we can pick any coin, and the amount we have to meet will decrease by that value.
i.e. after picking a coin with value c1, the remaining amount will be n - c1. And the total coins used will be,
amount of coins needed to get the amount + n - c1 + 1.
→ If we have k coins, then each time we will have k options to choose from.

T_n = min(
    T_{n-c1} + 1,
    T_{n-c2} + 1,
    .
    .
)

Recursive

```
int solve(vector<int>& coins, int remainingAmount){
      // base case
      if(remainingAmount == 0)
      return 0;

      // equation
      vector<int> answers;
      for(int coin: coins){
            // make sure that next remainingAmount is not less than 0
            if(remainingAmount - coin < 0)
            continue;

            // finding answer for subproblem
            int coinsForSubproblem = solve(coins, remainingAmount - coin);
            // if choosing 'coin' later gives us deadend
            if(coinsForSubproblem == -1) continue;

            int answer = coinsForSubproblem + 1;
            answers.push_back(answer+1);
      }

      if(answers.size() == 0)
      return -1;

      return *min_element(answers.begin(), answers.end());
}

int coinChange(vector<int>& coins, int amount) {
      return solve(coins, amount);
}
```

⇒ Recursion tree,



⇒ Time Complexity: O(k^n)

Top Down

```
map<int,int> memo;

int solve(vector<int>& coins, int remainingAmount){
      // base case
      if(remainingAmount == 0){
```

```
      memo[remainingAmount] = 0;
      return 0;
      }

      // remainingAmount was in the memo
      if(memo.find(remainingAmount) != memo.end())
      return memo[remainingAmount];

      // equation
      vector<int> answers;
      for(int coin: coins){

            // make sure that next remainingAmount is not less than 0
            if(remainingAmount - coin < 0)
            continue;

            // finding answer for subproblem
            int coinsForSubproblem = solve(coins, remainingAmount - coin);

            // if choosing 'coin' later gives us deadend
            if(coinsForSubproblem == -1) continue;

            int answer = coinsForSubproblem + 1;
            answers.push_back(answer+1);
      }

      if(answers.size() == 0){
            memo[remainingAmount] = -1;
            return -1;
      }

      int answer = *min_element(answers.begin(), answers.end());
      memo[remainingAmount] = answer;
      return answer;
}

int coinChange(vector<int>& coins, int amount) {
      memo.clear();
      return solve(coins, amount);
}
```

⇒ Time Complexity: O(n*k)

Bottom Up

```
int coinChange(vector<int>& coins, int amount) {
      // need to find answer for all possible amounts
      vector<int> dp(amount+1, 0);

      // base case
      dp[0] = 0;

      for(int _n = 1; _n <= amount; _n ++){
            vector<int> answers;
            for(int coin: coins){

                  // make sure that next remainingAmount is not less than 0
                  if(_n - coin < 0)
                  continue;

                  // finding answer for subproblem
                  int coinsForSubproblem = dp[_n - coin];

                  // if choosing 'coin' later gives us deadend
                  if(coinsForSubproblem == -1) continue;

                  int answer = coinsForSubproblem + 1;
                  answers.push_back(answer);
            }

            if(answers.size() == 0){
                  dp[_n] = -1;
            }
            else{
                  dp[_n] = *min_element(answers.begin(), answers.end());
            }
      }

      return dp[amount];
}
```

⇒ Time Complexity: O(n*k)