Applied Algorithms

Written Assignment-4

Jaya Sandeep Ketha

November 2023

Q1. k distinct pirates (call them p1, p2, etc) have raided a ship, and found a chest containing n gold pieces. They are wondering how many ways there are to split this gold amongst themselves. Find a recursive formula that computes this number, then explain how you would compute this quantity using dynamic programming. What is the asymptotic complexity of this computation?

Answer:

The recursive formula for this problem can be expressed as follows:

$$D(N, k) = D(N-1, k-1) + D(N-1, k)$$

where:

- $D(N, k)$ represents the number of ways to distribute N identical items into k distinct groups.
- $D(N-1, k-1)$ represents the case where the last item is placed into a new group, decreasing both N and k by 1.
- $D(N-1, k)$ represents the case where the last item is placed into one of the existing groups, decreasing N but keeping k the same.
- The code implements this recursive formula using the `ncr_memoization` function with memoization to avoid redundant calculations.
- The base cases are handled when r becomes 0 or n equals k, returning 1 in both cases.

Code:

```
def ncr_memoization(n, k, memo):
    if r == 0 or n == k:
        return 1
    if (n, k) in memo:
        return memo[(n, k)]
    result = ncr_memoization(n - 1, k - 1, memo) + ncr_memoization(n - 1, k, memo)
    memo[(n, k)] = result
    return result

def NoOfDistributions_combined(N, k):
    memo = {}
    return ncr_memoization(N + k - 1, k - 1, memo)

N = 8
R = 3
print(NoOfDistributions_combined(N, k))
```

To explain how this approach can compute ways:

1. **Define the Problem and Subproblems:**
   - The problem is to find the number of ways to distribute N identical items into k distinct groups.
   - We want to calculate the number of ways to distribute a certain number of items into a certain number of groups, which can be represented as $D(N, k)$.
2. **Identify the Recursive Formula:**
   - As mentioned in the code, we can use the following recursive formula:

$$D(N, k) = D(N - 1, k - 1) + D(N - 1, k)$$

   - This formula is based on two cases: either placing the next item into a new group ($D(N-1, k-1)$) or placing it into an existing group ($D(N - 1, k)$).
3. **Create a Memoization Table (or Array):**
   - In dynamic programming, we use a data structure (in this case, a dictionary called `memo`) to store the results of subproblems we've already solved to avoid redundant calculations. The keys of the dictionary are pairs (N, k) representing the state of the problem, and the values are the precomputed results for that state.
4. **Base Cases:**
   - The base cases for the recursion are when k is 0 or when N equals k, which are the situations where there is only one way to distribute the items (either by not placing any or placing all items in one group). These cases are handled by returning 1.
5. **Use Bottom-Up Approach (Dynamic Programming):**
   - Starting from the base cases, we work our way up to the original problem. We use the recursive formula and memoization to compute the values of $D(N, k)$ for all relevant subproblems.
6. **Return the Final Result:**
   - Once the dynamic programming process is complete, the result for the original problem is found in $D(N, k)$ and can be returned as the final answer.
7. **Running Time and Space Complexity:**
   - Using dynamic programming and memoization, we avoid redundant calculations and significantly reduce the time complexity of solving this problem, as opposed to pure recursion.

The asymptotic complexity of this computation can be analyzed as follows:

1. **Time Complexity:**
   - The dynamic programming approach evaluates and memoizes the results for each unique pair (N, k).
   - The number of subproblems to be computed is proportional to the product of N and k.
   - Therefore, the time complexity is $O(N \cdot k)$.
2. **Space Complexity:**
   - The space complexity is determined by the memoization table (the `memo` dictionary) used to store previously computed results.
   - In the worst case, there can be $N \cdot k$ unique pairs (N, k) in the table.
   - Thus, the space complexity is $O(N \cdot k)$.

Q2. Grid pathway problem. Given a grid of size n $\times$ m, find an algorithm that calculates the number of shortest pathways that starts from the bottom-left vertex and ends at the top-right vertex. See the above question for the steps in solving this.
Example: For a 1 $\times$ 1 grid, there are two shortest pathways of length 2 from the bottom-left vertex to the top-right vertex.
Answer:
The 'shortest_paths_count' function is designed to find the number of shortest paths from the bottom-left corner of a grid to the top-right corner. The grid has 'n' rows and 'm' columns, and are allowed to move only up and right. The function uses memoization to store previously computed results, which helps avoid redundant calculations and improves efficiency.

Let's break down the code and explain the recursive formula:

1. The function 'shortest_paths_count' takes three parameters:
- 'n': The current row position (0-based index).
- 'm': The current column position (0-based index).
- 'memo': A dictionary used for memoization to store computed results.

2. First, the function checks if it has already computed the result for the current position '(n, m)' by looking it up in the 'memo' dictionary. If it finds a value, it returns that value immediately, which is the number of shortest paths from '(n, m)' to the top-right corner.

3. The base case is when 'n' or 'm' reaches zero. When either 'n' or 'm' becomes zero, it means we've reached the top or left edge of the grid, respectively. In this case, there is only one path available because we can only move right (when 'n == 0') or up (when 'm == 0') to reach the top-right corner. So, 'result' is set to 1.

4. In the recursive case, we calculate 'result' by adding two components:
- 'shortest_paths_count(n - 1, m, memo)': This represents the number of paths from the cell one step above the current cell (moving up).
- 'shortest_paths_count(n, m - 1, memo)': This represents the number of paths from the cell one step to the left of the current cell (moving right).

5. The computed result is then stored in the 'memo' dictionary with the key '(n, m)' to avoid redundant calculations.

6. The function returns the result.

The idea behind dynamic programming and memoization is to avoid redundant work by storing and reusing already computed results. In this specific problem, when we're calculating the number of paths for a particular cell '(n, m)', we break down the problem into smaller subproblems by considering paths to the cells above and to the left of the current cell. By storing and reusing the results of these subproblems, we reduce the number of recursive calls and avoid recalculating the same paths, making the algorithm much more efficient. This approach ensures that each subproblem is solved only once, leading to a significant speedup in calculating the total number of shortest paths.

Recursive Formula:

$$\text{result} = \text{shortest\_paths\_count}(n-1, m, \text{memo}) + \text{shortest\_paths\_count}(n, m-1, \text{memo})$$

Code:

```
def shortest_paths_count(n, m, memo):
    # Check if the result for the current (n, m) has already been computed and stored in memo.
    if (n, m) in memo:
```

```
        return memo[(n, m)]

    # Base case: If we are at the top-left corner, there's only one path.
    if n == 0 or m == 0:
        result = 1
    # Recursive case: Calculate the result recursively and store it in the memo dictionary.
    else:
        result = shortest_paths_count(n - 1, m, memo) +
                    shortest_paths_count(n, m - 1, memo)

    # Store the result in the memo dictionary and return it.
    memo[(n, m)] = result
    return result

n = 7  # Number of rows
m = 8  # Number of columns
memo = {}  # Memoization dictionary to store computed results
result = shortest_paths_count(n-1, m-1, memo)  # Adjusted for 0-based indexing
print("Number of shortest paths (Combined):", result)
```

Q3. Describe and prove an efficient greedy algorithm that makes change for n cents, using the least number of quarters, dimes, nickels and pennies.

Answer:
Approach:
The greedy algorithm for making change using quarters, dimes, nickels, and pennies works as follows:

1. Start with n cents to make change for.
2. Begin with zero quarters, zero dimes, zero nickels, and zero pennies.
3. Repeatedly do the following until we have made change for the entire amount:
a. If n is greater than or equal to 25 (the value of a quarter), add one quarter to the change, subtract 25 from n.
b. If n is greater than or equal to 10 (the value of a dime), add one dime to the change, subtract 10 from n.
c. If n is greater than or equal to 5 (the value of a nickel), add one nickel to the change, subtract 5 from n.
d. If n is less than 5, add the remaining n pennies to the change.

Algorithm 1 Make Change(coins, v)

$n \leftarrow \text{length}(coins)$
numcoins$[0, \ldots, n-1] \leftarrow 0$
surplus $\leftarrow v$
for i in $0, \ldots, n-1$ **do**
  numcoins$[i] \leftarrow \lfloor \frac{\text{surplus}}{coins[i]} \rfloor$
  surplus $\leftarrow$ surplus $-$ numcoins$[i] \cdot coins[i]$
end for
return numcoins

The greedy algorithm essentially tries to give out as many quarters as possible first, then dimes, then nickels, and finally pennies. It selects the largest denomination that is less than or equal to the remaining amount of change. The algorithm continues this process until it reaches the desired change amount.

To prove that this greedy algorithm yields an optimal solution, we can use the greedy-choice property and the optimal substructure property:

1. Greedy-Choice Property:
The greedy algorithm's choice of using the largest denomination coin that is less than or equal to the remaining amount of change is the optimal choice at each step. This is because it minimizes the number of coins used to make change for that particular amount.

2. Optimal Substructure Property:
If the greedy algorithm makes the optimal choice at each step, it can be shown that the overall solution is optimal. Suppose we have made change using the greedy algorithm up to a certain point, and we are left with a remaining amount of change. We can extend this argument by induction. The solution for the remaining amount is also optimal because the greedy algorithm's choice is the best possible choice at each step.

Example:
Let us consider having 67 cents. For making a change the best possible way to return least number of coins as change is by offering 2 quarters, 1 dimes, 1 nickel and 2 pennies.

Explanation:

- We start with $v = 67$ cents.
- First, we use quarters (25 cents). We can use 2 quarters, which totals 50 cents. The remaining surplus is $67 - (2 \cdot 25) = 17$ cents.
- Next, we use dimes (10 cents). We can use 1 dimes, which totals 10 cents. The remaining surplus is $17 - (1 \cdot 10) = 7$ cents.
- Next, we use 1 nickel (5 cents), which totals 5 cents. The remaining surplus is $7 - (1 \cdot 5) = 2$ cents.
- Finally, we use pennies (2 cent) to make up the remaining 2 cents, using 2 pennies.
- The result is numcoins = [2, 1, 1, 2], which represents the use of 2 quarters, 1 dimes, 1 nickels, and 2 pennies, yielding the least number of coins to make change for 67 cents.

The greedy algorithm for making change using quarters, dimes, nickels, and pennies is guaranteed to produce an optimal solution because it always selects the largest denomination coin that can be used without exceeding the remaining amount. This approach minimizes the number of coins used, and the proof of optimality is based on the greedy-choice property and the optimal substructure property.