Süleyman KORAMAZ

1901042615 CSE321 HOMEWORK-3

①

```
procedure find_best_combination(index, combination, stores):
    if index == stores.length:
        return calculate_discount(combination), combination
    end if
    with_store = find_best_combination(index+1, combination + [stores[index]], stores)
    without_store = find_best_combination(index+1, combination, stores)

    if with_store[0] > without_store[0]:
        return with_store
    else:
        return without_store
    end if
end
```

$T(n) = 2T(n-1) + 1$

$2T(n-1)$ => two recursive calls (with and without store)

$+1$ => calculate_discount() function

Average Case Time Complexity:

We consider the average number of nodes visited. Each level of the trees has twice as many as nodes as the level above it.

Total # of nodes : $2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^n = 2^{n+1} - 1$

Average # of nodes visited is $\approx \frac{2^{n+1}}{2} = 2^n$

Average case = $O(2^n)$

②

```
procedure find_optimal_assignment(users, processes, processors, matrix):
    permutations = list(permutation(users))
    best = ∅
    minimum_cost = ∞
    for permutation in permutations:
        cost = 0
        for i in range(processors.length):
            cost = cost + matrix[processors[i]][users[user_index]][processes[process_index]]
        end for
        if cost < minimum_cost:
            minimum_cost = cost
            best = permutation
        end if
    end for
    return best, minimum_cost
end
```

Worst Case : $O(n.n!)$

Program generates all permutations of user-process pairs. Which is $n!$. For each it calculates the cost. ($O(n)$)

Best Case : $O(n.n!)$

Algorithm needs to explore all permutations. There is no scenario where it can terminate early.

Average Case : $O(n.n!)$

On average, algorithm needs to explore a significant portion of the solution space.

③

```
procedure find_optimal_energy(position, remain, sequence, matrix):
    if remain == ∅:
        return calculate_energy(sequence, matrix)
    end if
    min_energy = ∞
    for part in remain:
        sequence.append(part)
        position = part
        energy = find_optimal_energy(position, remain - {part}, sequence, matrix)
        if energy < min_energy:
            min_energy = energy
        end if
        sequence.pop()
    end for
    return min_energy
```

```
procedure calculate_energy(sequence, matrix):
    total = 0
    for i in range(sequence.length - 1)
        total = total + matrix[sequence[i]][sequence[i+1]]
    end for
    return total
end
```

CamScanner ile tarandı

③ Time Complexities : (calculate_energy( ) => $\in \Theta(n)$)

Worst case : Algorithm needs to explore all possible combinations of assembling parts. This corresponds to the scenario where the algorithm has to check all permutations. It is $(n-1)!$ times, where n is # of parts. → $\Theta(n!)$

Best case : The nature of the exhaustive search algorithm implies that it needs to explore a significant portion of the solution space. $\Theta(n!)$

Average case : The algorithm need to explore all, similar to worst and best case. $\Theta((n-1)!.n) = \Theta(n!)$

④

```
procedure minCoins(coins, target):
    if target == 0 :
        return 0
    end if
    min_coins = ∞
    for coin in coins :
        if coin <= target :
            sub_coins = minCoins(coins, target-coin)
            if sub_coins + 1 < min_coins :
                min_coins = sub_coins + 1
            end if
        end if
    end for
    return min_coins
end
```

Worst Case :

The algorithm has to explore all possible combinations of coins for each recursive call before finding the optimal solution. Time complexity is $\Theta(2^n)$, where n is the target amount.

Best Case :

Algorithm still needs to explore all possible combinations

Time complexity is $\Theta(2^n)$

Average Case :

Algorithm still needs to explore all similar to worst and best.

Time complexity is $\Theta(2^n)$

Function Steps :

1- Base case check => Checks if the target amount is 0. If it is, we don't need any more coin.

2- Initialize minimum coins => Start from ∞. This will be used to keep track of the min number of coins needed.

3- Loop through coins => Iterate each coin in the list

a. Check coin validity => Checks if the coin is smaller than or equal to the remaining amount. Then it is valid.

b- Recursive call => Makes a recursive call with an updated target amount. Finding minimum in remain.

c- Update minimum coins => Compare return value of recursive call and min coins. Update min coins if needed.

d. Return result.

⑤

$$T(n) = 2\,T\left(\frac{n}{2}\right) + 2$$

Recursively calls of left half and right half => $2 * T\left(\frac{n}{2}\right)$

Comparison and assignment = '2' (constant time)

Master Theorem :

$$T(n) = a.T\left(\frac{n}{b}\right) + f(n)$$

$a = 2$ $\left.\begin{array}{l} a > b^d \\ 2 > 1 \end{array}\right\}$ so, $\Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$

$b = 2$

$f(n) = 2$

$d = 0$