Assignment 2 — AD

Schmidt, Victor Alexander, rqc908 Ibsen, Helga Rykov, mcv462
Larsen, Christian Vadstrup, ljq919

Monday 08:00, February 27th

Task 1

$$N(C,i) = \begin{cases} 0 & i < 1 \\ 1 + N(C, i-1) & C = p_i \\ N(C, i-1) & 0 < C < p_i \\ N(C - p_i, i-1) + N(C, i-1) & C > p_i \end{cases}$$

Task 2

When the algorithm goes out of bounds (when $i < 0$), the formula returns 0, which is equivalent to saying the beer combination did not use up all of the money.

When selecting a beer, one can view the problem as turning on a bit for a price p_i. Depending on how much money you have left, you have different options:

1. You have more than the amount of money you need to turn on the bit, the formula should consider both subproblems of turning on the bit or leaving it off, as both subproblems may contain valid solutions. This can be seen in $C > p_i$ case 4.

2. You have too little money to turn on the bit/buy the beer. You may however have enough money for the next one (or another one down the line), so continue using the rest of the beer as a subproblem. This can be seen in $0 < C < p_i$ case 3.

3. You have exactly enough money for the bit/beer. This means there is a valid set, but we also have to consider other possible solutions. Consider then the subproblem of not buying the beer together with the decision to spend the last DKK on the last beer. This can be seen in $C = p_i$ case 2.

... as every recursive call is called on $i - 1$, and i is a finite collection of beer, $N(C, i - 1)$ must be a subproblem of $N(C, i)$.

Task 3

Comments: Algorithm 1 calculates number of ways to spend C DKK that runs in $\mathcal{O}(N * C)$.
Comments:
// int C = the total amount to spend on beers
// found_comb [][] is a 2D-array to save/momoize the sum of possible beer combinations
// At 1st iteration int sum = 0
// At 1st iteration, each cell of found_comb[][] = -1
// At 1st iteration n = N
// N is the total amount of available beers in beers[]

Algorithm 1 WAYS-TO-SPEND-C(beers, C, n, sum, found_comb)

```
if (sum == C) then                              // No money left
    return 1
else if (sum < C & n < 1) then                  // & no more beers to choose
    return 0
else if (sum > C) then                          // no more money to spend
    return 0
else
    if (found_comb[sum][n] == -1)  then                 // result not in the lookup table
        if ((C - sum) >= beers[n]) then
            // Case 1: beers[n] NOT added to beers
            a = WAYS-TO-SPEND-C(beers, C, n - 1, sum, foudn_comb)
            // Case 2: beers[n] added to beers
            beers[] = beers.append(beers[n])
            b = WAYS-TO-SPEND-C(beers, C, n - 1, sum + beers[n], found_comb)
            found_comb[sum][n] = a + b                  // result is saved
            return a + b
        end if
        if ((C - sum) < beers[n]) then
            a = WAYS-TO-SPEND-C(beers, C, n - 1, sum, foudn_comb)
            found_comb[sum][n] = a
            return a
        end if
    else
        return found_comb[sum][n]                       // result is in the lookup table
    end if
end if
```

Task 4

2. Prove the running time of your algorithm

Analysing the complexity of **Algorithm 1**:

`WAYS-TO-SPEND-C()` returns the total number of ways the students can spend all their C DKK (line 16). Its input size is determined by the number of beers (N) and the total amount to spend on beers (C).

If the `if-statement` in line 1 is TRUE, the function also assigns 1 to either a or b in the `if-statement` (lines 9 - 16).

If the two `elseIf-statements` are TRUE, then the function assigns 0 to either a or b.

In this respect, the first three `if-statements` depend on the number of recursive calls of `WAYS-TO-SPEND-C()` in lines 9 - 16. They check if the base cases have been reached and determine whether the function should return a value or continue with the recursive calls.

At each recursive call in lines 9-16 , the algorithm either adds or does not add a beer to the basket. Therefore, the number of recursive calls made by the code is at most 2^N. This means that if we consider only the recursive calls, the time complexity is exponential in N.

However, the algorithm also performs constant time operations at each recursive call, such as checking if a particular value is in the lookup table or appending an element to an array. These operations are not dependent on N and take a constant amount of time, so we can ignore them when analyzing the time complexity.

The main factor that affects the running time of **Algorithm 1** is the number of recursive calls. We can bound this by considering the maximum possible value of sum, which is C, and the maximum number of times we can subtract a beer price from C before we reach zero, which is N. Therefore, the maximum number of recursive calls is bounded by $\mathcal{O}(N * C)$.

In conclusion, the time complexity of **Algorithm 1** is not exponential in N, but it is also not $\mathcal{O}(N * C)$. Instead, it is somewhere in between, with an upper bound of $\mathcal{O}(N * C)$.

3. What is the memory usage of your algorithm?

The memory usage of **Algorithm 1** depends on the size of the input as well as the size of the lookup table.

The input to the algorithm is an array of N beer prices, an integer C representing the total amount of money to spend on beers, and two integer variables *sum* and n used in the recursion. The size of these variables is constant and does not depend on the input size, so their memory usage is $\mathcal{O}(1)$.

The lookup table `found_comb[][]` stores the previously computed results of the algorithm, so that we can avoid redundant computation by looking up previously computed results rather than recomputing them. The size of the lookup table is $(C+1) \cdot (N+1)$, because the maximum possible value of sum is C and the maximum value of n is N. Therefore, the memory usage of the lookup table is $\mathcal{O}(NC)$.

In addition, **Algorithm 1** uses an array to store the beers that have been added to the basket. The maximum size of this array is N, because we can add at most N beers to the basket. Therefore, the memory usage of this array is $\mathcal{O}(N)$.

Overall, the memory usage of **Algorithm 1** is $\mathcal{O}(NC)$, because the memory usage of the lookup table dominates the memory usage of the input array and the array used to store the beers in the basket.