KØBENHAVNS UNIVERSITET

Assignment 2

Alex (zhc522), Adit (hjg708), Usama (mhw630), Nikolaj (bxz911)

AD 2023

10. april 2023

Task 1

Write a recursive formula for the number of ways the students can spend all their C DKK.

Hint: You can look at a formula, where N (C, i) denotes the number of ways to spend exactly C DKK on beers with prices p1, . . . , pi.

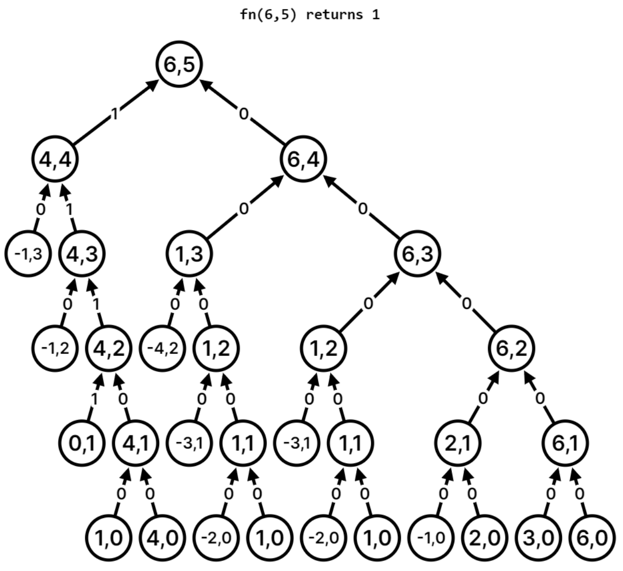
The recursive formula can be written as:

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

The $C = 0$ case need to be first, or else during the condition of $i = 0$ and $C = 0$ it would return 0 instead of 1.

Task 2

Prove that your recursive formula is correct and that it consists of overlapping subproblems.



Figur 1: Recusion tree over the formula

Our recursive formula works in a binary way. We have two base cases that can be either 1 or 0. The first case is if $C = 0$, meaning we are able to purchase the beer. The second case is for $C < 0$ meaning we are not able to purchase a beer because we have run out of money. Also in the second case we do not buy any beer when there are no more beers left, meaning $i = 0$. The third part of the formula calls the formula again recursively. Here we are repeating the binary options again with C being the money left AFTER we purchase a beer and one less beer since we choose to buy that beer. In the formula it looks like $N(C - p_i, i - 1)$, that is our current purchase. We need to keep a record of the beers that we have bought, so we add $N(C, i - 1)$ meaning the money we have with one less beer. This is to keep record of all the beers we have bought.
Figure 1 is an example of how our code runs with an input of $i = 5$ and $C = 6$, with different beer prices between 2 and 8 costs. In the example tree we have a recurring substructure of 1,2 in the third degree where it gets calculated twice. This is an example in which our recursive formula are not taking in account for previous substructures and instead recalculates them instead. Here will Dynamic Programming do better.

Task 3

Turn your recursive formula into an O(nC) dynamic programming algorithm. Provide pseudocode for the algorithm. You can use either memoization or bottom-up DP.

In this exercise we have chosen to write a program making use of memoization to solve the task.

Algorithm 1 beerCount$(p, C, i, memo)$

```
// memo[C][i] is a table instanced with None's
// Check if we have already calculated this value
if $memo[C][i] \neq$ None then
    return $memo[C][i]$
// Base cases
if $C == 0$ then
    $memo[C][i] \leftarrow 1$
else if $C < 0$ or $i == 0$ then
    $memo[C][i] \leftarrow 0$
else
    $memo[C][i] \leftarrow (beerCount(p, C - p[i], i - 1, memo) + beerCount(p, C, i - 1, memo))$
return $memo[C][i]$
```

Task 4

Prove the correctness and running time of your algorithm.

Line 3-4 of the algorithm states if the table spot has already been calculated, then we return it since we do not have to calculate it again.

The base cases in line 6 to 9 states that:
When C becomes 0, it means that the money was used to purchase a beer, meaning that it would be 1. When C is negative or when i is 0 it is not possible to purchase a beer with negative money, and it is also impossible to purchase a beer when there are no beers remaining.

The recursive part of the algorithm in line 11 is the inductive step of the proof.
If C is positive and i is positive, the algorithm makes a recursive call to itself with two different parameters:
The first recursive call subtracts the value of the current coin p[i] from C, and decreases i by 1. This represents the case where the current coin is used in the formation of C.
The second recursive call keeps C and i the same, and only decreases i by 1. This represents the case where the current coin is not used in the formation of C.
The sum of the two recursive calls is assigned to memo[C][i] as the total number of ways to form C using coins p.
By using memoization, the algorithm ensures that each recursive call is only computed once, avoiding redundant computations.
With the induction proof we have proven that our algorithm can calculate the number of ways the students can spend all their C DKK.

Overall, the algorithm works by building a table of memoized results as it recursively computes solutions to subproblems. The memoization allows the algorithm to avoid recomputing the same subproblems, resulting in a faster overall runtime.

The memory usage of the algorithm is then $O(nC)$ as each instance of the table requires a constant space to store and the table is of size $n \cdot C$.