# Algorithms and Datastructures assignment 1

Thomas Broby Nielsen (xlq119)
Tobias Overgaard (vqg954)
Christian Buchter (zvc154)

14. maj 2015

## Indhold

# 1 Task 1

$$|N(c,i)| = \begin{cases} 0 & \text{if } c \leq 0 \\ 0 & \text{if } i = 0 \\ 1 + N(c, i-1) & \text{if } p_i = c \\ N(c - p_i, i-1) + N(c, i-1) & \text{otherwise} \end{cases}$$

# 2 task 2

In order to prove the correctness of our formula, for calculating the amount of unique combinations of unique beers you can buy with a specified amount of money, we will have to tjeck the three properties of a correct formula.

First we will first prove the correctness of the initialization of our formula.
This we have summarized into showing that for a given value of $1 \leq c$ and $1 \leq i$ that our base case holds true for the first iteration. For the example above, it is trivial to see that if $N(1,1)$ then the base cases for the formula, won't be triggered. From this we can see that initialization will hold for any value of $1 \leq c$ and $1 \leq i$.

The second property that our formula must hold, is that if the invariant before an iteration is true, must remain true after the iteration. To show that our

To further prove the correctness of our formula, we have see if the formula holds true before and after an iteration.

# 3 Task 3

Bellow we have written the pseudo-code for MemoizedBeerComp using the DP-method

```
MemoizedBeerComp(c, i, beerList, holderMatrix)
1 if c <= 0 or i==0
2    return 0
3 if array(c, i) > -1
4    return holderMatrix(c, i)
5 if c - beerList[i] == 0
6     array(c,i) = 1 + MemoizedBeerComp(c, i - 1)
7     return holderMatrix(c,i)
8   else
9        holderMatrix(c,i) =
            MemoizedBeerComp(c, i - 1, beerList, holderMatrix) +
            MemoizedBeerComp(c - beerList[i], i - 1, beerLis, holderMatrix)
10       return holderMatrix(c, i)
```

Since we weren't able to calculate the running time for our memoizedBeerComp for task 4, with our top-down implementation of memoizedBeerComp, we decided to also implement our original formula as a bottom-up method instead, as seen below.

```
Bottom-Up-Beers (P,c) =
1   let r[1,2, ... ,c][0,1,2,...,P.length] be a new Matrix
2   for j=1 to c{
3      r[j][0] = 0;
4      for k = 1 to P.length {
5         q = 0;
6         if (P[k] == j) {q+=1}
7            q += r[j][k-1];
8         if (j-P[k] > 0) {
9            q += r[j-P[k]][k-1];
10         }
11     r[j][k] = q;
12     }
13  }
14 return r[c][P.length];
```

NOTE: we later figured out how to calculate the running time for the top-down method implementation after talking with a TA, but we decided to keep both of the implementations.

# 4 Task 4

For this task we went ahead and analysed the bottom-up implementation in task 3 made from our original formula in task 1. Because of the nested forloop nature in our implementation, it is easy to see that the running time for our implementation is $O(cn)$ where $n = p.length$ is the amount of individual beers. Therefore our running time, that is based off of the upper bound, is equal to $\theta(cn)$.