Algorithms and Datastructures assignment 2

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

26. maj 2015

Indhold

1 Task 1 **2**

2 task 2 **2**

3 Task 3 **3**

4 Task 4 **3**

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 show that our formula devides a problem into overlapping sub-problems, and solves those subproblems correctly. First we have to establish the correctness of our base cases. To Illustrate this, if the algorithm starts out with $c \leq 0$ or if it starts out with $i = 0$ where $i \in \mathbb{Z}$, the algorithm will return 0. If at some point $c = 0$ occurs and c started out as $0 \leq c$ our algorithm will returns 1. If the same scenario happens but c goes bellow 0, $c < 0$, our algorithm will return 0. If both $c \geq 0$ and and $i > 1$ then our algorithm will make a tree like structure and divide the problem into overlapping sub-problems. These sub-problems will then be solved by the algorithm in accordance with the base cases.

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     holderMatrix(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 = R[j][k-1];
6         if (P[k] == j){
7             q+=1}
8         if (P[k] < j) {
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, Bottom-Up-Beers. From the nested for-loop in our implementation, first seen at line 2 and then again inside the first for-loop at line 4, we can see that our algorithm first checks through all possible combinations of k $>$ p.lenght where j $>$ c first tiebacks though

every we can see that our algorithm first fores though anywe can see that our algorithm first Because of the nested forloop, first at line 2 in our Bottum-Up-Beers 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 $\theta(cn)$.

Since our matrix has to be stored for the algorithm to access it our memory usage is ||R|| = cn.

We prove the correctness by using the loop invariants:

- At the start each iteration of the inner loop the value of R[j]|[$k-1$] is the number of combinations of the p[1..$k-1$] beers achievable with j kr.
- At the start of each iteration of the outer loop the value of R[$j-1$]|[n], where n is P.length, is the number of combinations of all the beers achievable with $j-1$ kr.

Initialisation
Before the first inner loop the value of R[j]|[0] is 0, which is the number of combinations of 0 beers achievable with j kr.

Before the first outer loop the value of R[0]|[n] is 0, which is the number of combinations of all the beers achievable with 0 kr.

Maintenance

If R[j]|[$k-1$] is true before the $k'th$ inner loop R[j]|[k] will be true after it since if $k = j$ we add that as a standalone combination to R[j]|[$k-1$], and otherwise if $j >$ P[k] we add the combinations from R[$j - P[k]$]|[$k-1$] which gives us all the combinations of R[j]|[k] as long as earlier iterations were Initialised correctly.

If R[$j-1$]|[n] is true before the $j'th$ iteration of the inner loop R[j]|[n] will be true after it since if the loop invariant for the inner loop holds after termination the $n'th$ iteration of the inner loop R[j]|[n] will contain the number of combinations of all the beers achievable with j kr.

Termination

The condition causing the inner loop to terminate is $k > n$. because each loop iteration increases k by one and because $n \neq \infty$ the loop must terminate. Since Maintenance of the inner loop gives us that R[j]|[n] will be true after the $n'th$ iteration we have that the loop will terminate and give us the loop invariant of the outer loop after the $n'th$ iteration.

The condition causing the outer loop to terminate is $j > c$. because each loop iteration increases j by one and because $c \neq \infty$ the loop must terminate. Since Maintenance of the outer loop gives us that R[c]|[n] will be true after the $c'th$ iteration we have that the loop will terminate after the $c'th$ iteration and give us the total number

of combinations of the n beers achievable with c kr.

Litteratur