AD - Assigment 2

pwn274, npd457, kgt356

February 2022

Contents

TODO: Lav python til pseudo Lav correctness lav individuel

1 Group part

1.1 Write pseudocode

$$f(C,i) = \begin{cases} 1 & C = 0 \\ 0 & i > \text{len(P)} \\ D(C - P[i], i+1) + f(C, i+1) & \text{otherwise} \end{cases}$$

Here, the recursive definition can be seen as combining two cases: Either buying or skipping an item before moving on to the next. This means that for every call the problems splits into two branches where, as we are not allowed to buy the same item twice, the possibilities are exhausted once the last item has been reaches by all branches.

Unsure what you mean by formula, I choose to also write this in pseudocode:

```
def D(prices,  C,  i):
    if (C == 0):
        return 1
    if (i >= len(prices) or C < 0):
        return 0

    _self = D(prices, C - prices[i], i + 1)
    _next = D(prices, C, i + 1)

    return _self + _next
```

1.2 Correctness proof

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

Base case: Only one item. Here three recursions are needed. The first time, none of the if-statement pass, and the `_self` and `_next` will start a new level of recursion depth. Now, if the price is correct for the first item it will trigger the first-if statement and return 1, if not, the second if-statement will trigger and 0 is returned. For the other branch C is unchanged and 0 is also returned. So the algorithm behaves as expected for a trivial input.

Looking at any larger input N, the splitting is the crux of the problem. As we try both of possibilities, leaving us with $N-1$ beers and $C - P(N)$ kroner in the first and case $N-1$ beers and C kroner in the second. If no base conditions are met, this now simply corresponds to looking at a new smaller input. This can of course be done recursively until an input size of 1 is hit where we have already seen that our algorithm is correct.

For the problem to have overlapping sub-problems, one or both of the following criteria must be fulfilled: Either some pair p_i and p_j must sum to some prior element p_k, where $i, j < k$ and/or a pricing must be repeated in the pricing list. This creates scenarios where the same value of `D[C,i]` will have to be calculated numerous times, and the problem therefore has overlapping sub-problems.

1.3 Making the algorithm dynamic

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.

I used memoization:

```
def D_dynamic(prices,  C,  i):
    if memory[C,i] >= 0:
        return memory[C,i]

    if (C == 0):
        memory[C,i] = 1
        return memory[C,i]
    if (i >= len(prices) or C < 0):
        memory[C,i] = 0
        return memory[C,i]

    _self = D_dynamic(prices, C - prices[i], i + 1)
    _next = D_dynamic(prices, C, i + 1)
    memory[C,i] = _self + _next
    return memory[C,i]
```

1.4 Prove the correctness and running time of your algorithm

What is the memory usage of your algorithm? There is almost no need to prove the correctness of the algorithms as it is unchanged in functionality, but working on a lookup-table instead. As all the entries in the table are filled by

Asumming a RAM with trivial data-lookup, it is easy to see that the running time is $\mathcal{O}(nC)$, as the main computational sinner is the time taken to fill the memory (an $n \times C$ matrix) and everything else are simple lookups.

A final proof is visual by virtue of running the algorithms. As can clearly be seen, the computation time are improved by multiple orders of magnitude and the asymptotic scaling goes from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$:

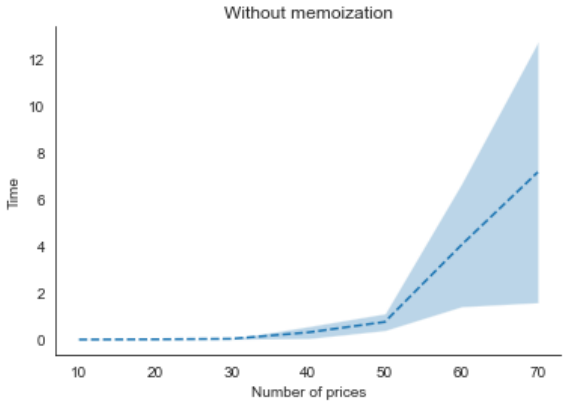

Figure 1: The naive solution

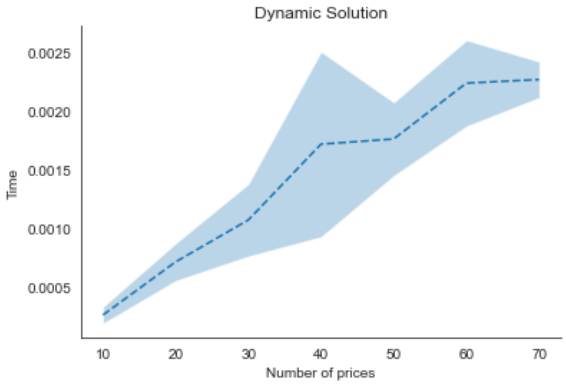

Figure 2: The optimized solution

2 Individual parts

2.1 pwn274 - Jakob Hallundbæk Schauser

- Dynamisk programmering
  - Bevidst om, at der er meget der gentages
  - Overlapping subproblems
- Manglerne ved dynamisk programmering
- Hvordan designer man en dynamisk algoritme?
- Eksempel: Rod Cutting (eller måske Knapsack?)

- Top-down og bottom-up

2.2 npd457 - Sebastian Ø. Utecht

Dynamisk programmering

- Dynamisk programmering overordnet
  - Optimeringsopgave
  - Overlappende sub-problems struktur
  - Optimal delstruktur
- Metoder:
  - Memoization
  - Bottom-Up
- Eksempel: Rod-cutting:
  - Gennemgang af memoization løsning + kort tale om bottom up.

2.3 kgt356 - Christoffer A. Ankerstjerne

dynamic programming

- -optimerer gentagende sup-problemer
- -optimal substruktur

cases: rod-cutting

- (dynamic) vs (naive) $O(2^{n-1})$ to $O(n^2)$
- Top-down vs Bottom up

A Why do we need an appendix?