Name: Grade:

Quick Test: Dynamic Programming

Master MoSIG — Algorithms and Program Design

Florent Bouchez Tichadou — Gwenaël Delaval

October 23rd, 2015

Duration: 30 minutes.
This is the document with solutions.

Note: In this document, we try to give the shortest possible answers that would guarantee the best grade. These solution are written in plain black color. However, since we want to give more details, text written in this color provide additional information on a solution, or alternative answers.

Exercise 1 (Change)

We consider a currency system composed of a set of n different coins, of values $v_1, \ldots, v_n$, with $v_1 = 1$ and $\forall i \in \{1, \ldots, n-1\}, v_i < v_{i+1}$. We want to write an algorithm to compute the minimal number of coins for the change of an amount of money W.

More formally, given the values of $v_1, \ldots, v_n$ and W, we want to find the values of $x_1, \ldots, x_n \in \mathbb{N}^n$ such that

$$\sum_{i=1}^{n} x_i.v_i = W \text{ and } \sum_{i=1}^{n} x_i \text{ is minimal.}$$

Question 1.1 Let $v_1 = 1$, $v_2 = 3$, $v_3 = 4$. What are the values of x_i for $W = 10$?

Solution to 1.1: $x_1 = 0$, $x_2 = 2$, $x_3 = 1$.

Note: The purpose here is to see that greedy does not work. Greedy will produce $x_1 = 2$, $x_2 = 0$, $x_3 = 2$, i.e., use 4 coins while our solution is 3.

Our solution is optimal as it is impossible to do it with only 2 coins (the best we could attain is $8 = 2 \times 4$ which is not enough for $W = 10$.

Question 1.2 Write a recursive function MinChange(W, i) that computes the values of $x_1, ..., x_i$ for the change of W, using coins of values $v_1, \ldots, v_i$.

Solution to 1.2: Note: After Question 1.1, we know that a greedy algorithm will not be optimal. So we use a complete space exploration algorithm with a recursive function. There are multiple ways to do it; we propose two algorithms here but only one was asked.

Note that we consider here also the case when a partial solution cannot be completed (for instance, if the coins are 2 and 3, and $W = 4$, if we start taking 3, it is impossible to "finish" this solution, as MinChange is called with $W = 1$).

We consider the solution is an array from 1 to n: sol[i] contains x_i.

Algorithm 1:

We can either take one v_i or no. If we don't take one, we recurse on $i - 1$, otherwise we recurse on i (as we might need more than one).

```
MinChange (W, i)
  if i = 0
    if W = 0
      return [0, 0, 0, ..., 0] /* n times zero */
```

```
    else
      return NULL      /* no solution possible */

  sol <- MinChange (W, i-1) /* take no coin v_i */

  if v_i >= W
     sol' <- MinChange (W - v_i, i) /* take at least one coin v_i */
     if sol' != null
        sol'[i] += 1

  if cost(sol) < cost(sol')  /* cost(null) returns MAXINT */
    return sol
  else
    return sol'
```

Solution 2:

We try every coin possible, in that case, we always recurse on i, so we can leave the i parameter out.

```
MinChange (W)
  if W = 0
    return [0, 0, 0, ..., 0] /* n times zero */

  sol <- null
  for i <- 1 to n
    sol' <- MinChange (W - v_i)
    if sol' != null
      sol'[i] += 1
    if cost(sol') < cost(sol)
      sol <- sol'

  return sol
```

Solution to 1:

Question 1.3 What is the complexity of your function?

Solution to 1.3: Note: Here, we are performing a complete space exploration so the complexity will be exponential. Depending on the solution chosen, the analysis will change.

Algorithm 1:

MinChange(W, i) will trigger (indirectly) W/v_i calls to MinChange$(\ldots, i)$, each of them making also one call to MinChange$(\ldots, i-1)$.

If we draw all calls with same i on the same level, we obtain a "tree" with, on level $i-1$, $\frac{W}{v_i}$ more nodes than on the level i. The height of this tree is n, so the complexity is $O((\frac{W}{v_1})^n)$.

Algorithm 2: Each call will make n recursive calls. The height of the tree is $O(\frac{W}{v_1})$, so we have complexity $O(n^{\frac{W}{v_1}})$.

Question 1.4 Use a dynamic programming technique to improve your function by avoiding multiple re-computations. (If you need to modify your function you can directly modify the initial function: then, annotate or identify clearly the modifications.)

Solution to 1.4: Note: In the two proposed algorithms, there are multiple re-computations of the same values. The cache method proposed depends on the solution.

Algorithm 1: We use a 2D cache of size $W \times n$. $cache[w][i]$ will contain the **total number of coins** for the optimal solution for $w \leqslant W$ using only coins $v_1, \ldots, v_i$. All cells of the cache are initialized to -1.

We modify the algorithm as follows: when entering `MinChange`(w, i), we check if $cache[w][i]$ is $\neq -1$, in which case we return it directly. Otherwise, we compute the solution recursively, and store the cost of the solution in $cache[w][i]$ before returning it.

Note: If we want to keep the actual solution as well, we use also a matrix $coins$ where $coins[w][i]$ contains $true$ if we decided to take one v_i in MinChange;

To print the solution, we use the following code:

```
i <- n
sol <- [0, 0, ..., 0]
while W > 0
  if coins[W][i]
    sol[i]++
    W <- W - v_i
  else
    i--

for i in 1 to n
print ‘‘Number of coins of value v_i: sol[i]’’
```

Algorithm 2:

We use an array of size W. $cache[w]$ will contain the **total number of coins** for the optimal solution for $w \leqslant W$ using all coins $v_1, \ldots, v_n$. All cells of the cache are initialized to -1.

We modify the algorithm as for algorithm 1 (checking at entry, and storing at before returning).

Note: Again, if we want to keep track of the actual solution, we can do this in an array $coins$, where $coins[w]$ holds the index i of the coin chosen v_i. To print the solution, we use the following code:

```
sol <- [0, 0, ..., 0]
while W > 0
  i <- coins[W]
  sol[i]++
  W <- W - v_i
for i in 1 to n
  print ‘‘Number of coins of value v_i: sol[i]’’
```

Note: In all cases, there are many subproblems that are not reachable (even more so for algorithms 1), so we do not consider sequential algorithms.

Question 1.5 What is now the complexity of the algorithm?

Solution to 1.5: Algorithm 1: We use a $W \times n$ matrix, and to compute each cell, we need only constant time (we can keep track and update the number of coins in a solution instead of calling `cost` every time). So the complexity is $O(W \times n)$.

Algorithm 2: We use an array of size W, and for each cell we loop on all n coins; on a coin the computation is $O(1)$, hence the complexity is $O(W \times n)$.