# Solutions to ADC practice questions: Dynamic programming

**Question 1.**

A tree trunk of length $L$ is to be cut into logs of different lengths in a manner that minimizes waste. The lengths of $n \geq 0$ logs are $k_1, \ldots, k_n$, where $k_i \geq 0$ for all $0 < i \leq n$ (assume that both $L$ and $k_i$ are integers, for example centimeters). Find a subcollection of $\{ k_1, \ldots, k_n \}$ such that the logs can be cut from the trunk, and with the minimum possible amount of waste. Within this context 'waste' is understood as a remaining section of trunk from which it is no longer possible to cut a log in the specified subcollection that has not already been cut. You may assume, for convenience, that the blades of the chainsaw used to cut the logs have a width of 0 cm.

An alternative and slightly more precise specification of the problem is: minimize $L - \sum_{i=1}^{n} b_i \cdot k_i$ subject to the condition that $\sum_{i=1}^{n} b_i \cdot k_i \leq L$ en $b_i \in \{ 0, 1 \}$ for all $0 < i \leq n$.

Questions:

1. Determine which of the following 8 lengths of logs can be cut from a tree trunk of a length 100 cm with a minimum amount of waste. Explain why your solution minimizes waste. The lengths of the logs are: $k_1 = 10, k_2 = 12, k_3 = 20, k_4 = 30, k_5 = 53, k_6 = 10, k_7 = 5$ and $k_8 = 42$ (all dimensions are in centimetres).
2. Give a recurrence relation $C(i, l)$ for $i \geq 0$ that that describes the length of the minimum amount of waste left on cutting logs from the collection $\{ 1, \ldots, i \}$ from a tree trunk of length $l$. *Hint: It will be useful to assume that $C(i, l) = \infty$ for $l < 0$.*
3. Give an algorithm that determines the length of the minimum amount of waste from a tree trunk of length $L \geq 0$ and $n \geq 0$ logs of lengths $k_1, \ldots, k_n$. *Hint: use dynamic programming.*
4. Determine the worst case time complexity and spatial complexity of your algorithm.

**Solution**

1. Select logs 5, 7 and 8. The total length of these logs is 100 centimetres. There will be no waste at all, so this must be an optimum solution.
2. Formulate a recurrent relation $C(i, l)$ for the maximum length of waste, in centimetres, left when a tree trunk of length $l$ can only be cut into the remaining logs of $k_1, k_2, \ldots, k_i$. There is no choice when no logs remain ($i = 0$). The length of waste is then simply equal to the remaining length of tree trunk, $C(0, l) = l$. The exception is $l < 0$, as a solution cannot be based on a log of a negative length. For this reason the costs are set to infinite, so
$$C(0, l) = \begin{cases} l & \text{, if } l \geq 0 \\ \infty & \text{, if } l < 0 \end{cases}$$
When logs are still available then the problem can be simplified by expressing the problem in terms of one less log to choose from. This can be achieved by either cutting the last length ($k_i$) from the tree trunk (after which the remaining choice of logs will be from $i - 1$ and the tree trunk will have a length of $l - k_i$), or by doing nothing (when the length of the tree trunk remains $l$). Obviously, the required solution is the result with the minimum lengths of waste. So:
$$C(i, l) = \min(C(i - 1, l), C(i - 1, l - k_i))$$
3. Both $L$ and all $k_i$ need to be integers (in centimetres) for this solution. Given the assumptions, it is now possible to generate a matrix of dimensions $(n + 1) \times (L + 1)$ containing the values of $C(i, l)$ for all $0 \leq i \leq n, 0 \leq l \leq L$.

The algorithm used to fill the matrix is as follows (whereby it is assumed that the `lengths` array is filled from `lengths[1]` through `lengths[n]` and, consequently, has length n+1 ):

```
def hakker(lengths,L):

    n=len(lengths)-1
    C=[[0 for l in range(L+1)] for i in range(n+1)]

    for l in range(0,L+1):
        C[0,l]=l           # wast equal to l

    for i in range(1,n+1):
        for l in range(0,L+1):
            if (l-lengths[i])<0:
                C[i,l]=C[i-1,l]
            else:
                 C[i,l]=min(C[i-1,l],C[i-1,l-lengths[i]])

    return C[n,L];
```

First, fill all positions in the matrix for which no remaining logs are available to be cut and, as noted above, the length of waste is equal to the length of the tree trunk.
The other positions in the matrix are filled as follows: first, determine all elements for which only 1 more log can be cut. This only needs the values pursuant to the recursive comparison for which no further log is available, and these are already known. This is followed by working up to 2 ogs and then 3 logs, etc.
The number of logs asked in the question is given in `C[n,L]`, as all logs are still available and no logs have been cut from the tree trunk.

4. The worst case time complexity of the algorithm is $O(n \cdot L)$, due to the nested for loop with a constant number of iterations for each element of the matrix.
The spatial complexity is also $O(n \cdot L)$, due to the storage of the $(n+1) \times (L+1)$-matrix. However, a reduction is feasible, as all that is needed to calculate the row of elements $C[i, \dots]$ is the row $C[i-1, \dots]$. Consequently, the algorithm always needs to generate only two rows to arrive at a spatial complexity $O(L)$.

**Question 2.**

Suppose that there are four matrices $A$, $B$, $C$ and $D$ with dimensions of 20x2, 2x15, 15x40 and 40x4. Give the *cost* matrix as computed by the algorithm in slide 27.

**Solution**

This relates to

$$d_1 = 20 \quad d_2 = 2 \quad d_3 = 15 \quad d_4 = 40 \quad d_5 = 4$$

A matrix is to be filled in which element $(i, j)$ is the minimum number of multiplications for the matrices demarcated by the dimension $d_i$ and $d_j$ (for example, $(2, 5)$ relates to the multiplication of $B \times C \times D$).

As the main diagonal has no significance, this is not completed. First, fill the 'diagonal' to the right of and above the main diagonal: as these are all 'multiplications' of just one matrix, the costs are 0. Next, fill the 'diagonal' to the right of and above the first diagonal. These elements are the product of the multiplication of two matrices, and as these can only be multiplied one way the costs are given by the product of the dimensions of the matrices.

Filling a next diagonal requires the calculation of a minimum for the possible positions of the last multiplication. Filling the diagonals yields the following matrix:

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | $\times$ | 0 | 600 | 2800 | 1680 |
| 2 | | $\times$ | 0 | 1200 | 1520 |
| 3 | | | $\times$ | 0 | 2400 |
| 4 | | | | $\times$ | 0 |
| 5 | | | | | $\times$ |

The following calculation of $(1, 5)$ illustrates how these figures are obtained:

$$\begin{aligned} &\text{Last multiplication at position 2: } 0 + 1520 + 20 \cdot 2 \cdot 4 && = 1680 \Leftarrow \\ &\text{Last multiplication at position 3: } 600 + 2400 + 20 \cdot 15 \cdot 4 && = 4200 \\ &\text{Last multiplication at position 4: } 2800 + 0 + 20 \cdot 40 \cdot 4 && = 6000 \end{aligned}$$

$(1, 4)$ was minimized in the same manner between 12600 and 2800, and $(2, 5)$ between 2520 and 1520.

**Question 3.**

Suppose that you have the following keys, together with the probability that a search will be carried out for the relevant key: $A$ .20, $B$ .24, $C$ .16, $D$ .28, $E$ .04, $F$ .08. Give the *cost* matrix as computed by the algorithm in slide 33. What is the optimum binary search tree?

**Solution**

The following recurrent relationship is used:

$$A(T^r_{(l,h)}) = p(l, h) + A(T_{(l,r-1)}) + A(T_{(r+1,h)})$$

The `cost` matrix is now filled such that `cost[i,j]` contains the weighted average number of comparisons needed to find a key in the tree containing elements $i$ through $j$ (so: $A(T_{(i,j)})$.

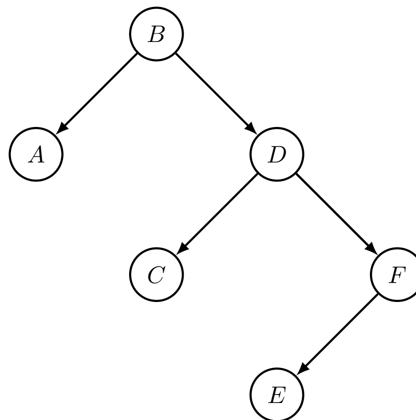
The main diagonal is simple: for a sub-tree $T^r_{(l,h)}$ with $l = h$, $A(T_{(l,r-1)})$ and $A(T_{(r+1,h)})$ are equal to 0, so it follows that $A(T^r_{(l,h)}) = p(l, h)$ in that case. Minima once again need to be calculated for the diagonals above this main diagonal in the same manner as for the matrix multiplications.

| | $A$ | $B$ | $C$ | $D$ | $E$ | $F$ |
|---|---|---|---|---|---|---|
| $A$ | 0.20 ($A$) | 0.64 ($B$) | 0.96 ($B$) | 1.68 ($B$) | 1.80 ($B$) | 2.08 ($B$) |
| $B$ | | 0.24 ($B$) | 0.56 ($B$) | 1.20 ($C$) | 1.32 ($C$) | 1.52 ($D$) |
| $C$ | | | 0.16 ($C$) | 0.60 ($D$) | 0.68 ($D$) | 0.88 ($D$) |
| $D$ | | | | 0.28 ($D$) | 0.36 ($D$) | 0.56 ($D$) |
| $E$ | | | | | 0.04 ($E$) | 0.16 ($F$) |
| $F$ | | | | | | 0.08 ($F$) |

The following calculation of $AE$ illustrates how these figures are obtained:

$$\begin{aligned}
A(T^A_{(A,E)}) &= p(A,E) + A(T_{(B,E)}) = 0.92 + 1.32 && = 2.24 \\
A(T^B_{(A,E)}) &= p(A,E) + A(T_{(A,A)}) + A(T_{(C,E)}) = 0.92 + 0.20 + 0.68 && = 1.80 \Leftarrow \\
A(T^C_{(A,E)}) &= p(A,E) + A(T_{(A,B)}) + A(T_{(D,E)}) = 0.92 + 0.64 + 0.36 && = 1.92 \\
A(T^D_{(A,E)}) &= p(A,E) + A(T_{(A,B)}) + A(T_{(D,E)}) = 0.92 + 0.64 + 0.36 && = 1.92 \\
A(T^E_{(A,E)}) &= p(A,E) + A(T_{(A,D)}) = 0.92 + 1.68 && = 2.60
\end{aligned}$$

The tree is now simple to create. The root is apparently node $B$, as this needs to be at the top to place keys $A$ through $F$ in the most efficient position. $A$ will then need to be to the left of $B$, and $C$ through $F$ will then need to be placed in the best possible position to the right of $A$. rom the table it is apparent that $D$ will then need to be at the top. This then results in the following tree:




It is now also possible to check to verify that the average number of comparisons needed to find a key is actually 2.08 :

$$T = 1 \cdot 0.24 + 2 \cdot 0.20 + 2 \cdot 0.28 + 3 \cdot 0.16 + 3 \cdot 0.08 + 4 \cdot 0.04 = 2.08$$

No other configuration could yield a lower value than this.

**Question 4.**

Suppose that the currency of a given country is made up of the series $c_1 > c_2 > \ldots > c_n$ (for example 50, 25, 10, 5, 1 for the USA). The *coin change problem* is: given a currency and change of $a$ cents, what is the minimum number of coins that add up to $a$ (assume that $c_n = 1$)?

1. Formulate a greedy algorithm for this problem. Show how this works for \$ 1.43.

2. State a currency of a fictitious country for which the greedy algorithm does not always yield the minimum.
3. State an algorithm (with met dynamic programming) that solves the problem with every currency.

**Solution**

1. Use the largest denomination of coin for as far as is possible and then of the next largest denomination of coin, etc. For \$1.43 this works as follows:
   - Use two 50 cent coins (43 cents remain);
   - Use one 25 cent coin (18 cents remain);
   - Use one 10 cent coin(8 cents remain);
   - Use one 5 cent coin(3 cents remain);
   - Use three 1 cent coins.

   This method results in the use of 8 coins.
2. Take, for example, 1, 3 and 4 cent coins and try to give 6 cents change. The greedy approach results in $4+1+1$ (three coins), whilst $3+3$ is more efficient (two coins).
3. The approach needed here is comparable to the approach to the earlier tree trunk problem. Formulate an algorithm $M(i,j)$ for the minimum number of coins adding up to an amount of $j$ cents, where a choice can be made from coins $c_i,\ldots,c_n$. Should, for example, it be decided not to use coin $i$ at all, then $M(i+1,j)$. However, when coin $i$ is chosen then a coin has been used and the remainder is $j-c_i$ cents and if so wished coin $i$ can be chosen again: at least $1+M(i,j-c_i)$ coins will then be required. As the objective is to use a minimum number of coins, the minimum of these two options must be selected:
$$M(i,j)=\min(M(i+1,j),1+M(i,j-c_i))$$
   The base cases are given by $M(n,j)=j$ for every $j$, as giving change of $j$ cents would require precisely $j$ coins if the choice was restricted to 1 cent coins. In addition, $M(i,0)=0$ holds for $i$, and for the sake of simplicity $M(i,j)=\infty$ for $j<0$. So:
$$M(i,j)=\begin{cases}\min(M(i+1,j),1+M(i,j-c_i)) & \text{, als } i<n \text{ en } j>0\\ j & \text{, if } i=n \text{ en } j>0\\ 0 & \text{, if } j=0\\ \infty & \text{, als } j<0\end{cases}$$
   The matrix can now be filled with the following algorithm:

```
def change(coins,amount):

    n=len(coins)-1
    M=[[0 for j in range((amount+1)] for i in range(n+1)]

    for j in range(0,amount+1):
        M[n,j]=j                    # choose only 1 cents

    for i in range(1,n+1):
        M[i,0]=0                    # no more coins needed

    for i in range(n-1,0,-1):
```

```
    for j in range(1,amount+1):
        if coins[i]<=j:
            M[i,j]=min(M[i+1,j],1+M[i,j-coins[i]])
        else:
             M[i,j]=M[i+1,j]

return M[1,amount]
```