CentraleSupélec

Algorithmics and complexity

TD 4/7 – Dynamic Programming

Training exercises

Note: The solution elements given here are not complete. Their purpose is only to guide you. We encourage you to write a proper answer, as you would do for the exam. If any question remains, feel free to ask your tutorial supervisor for help.

Exercice 1 : Pyramid of numbers

A pyramid of numbers is a set of integers stacked as in Pascal's triangle. In such a pyramid, we want to find the path (strictly descending) from the top to the base such that the sum of the numbers along this path is maximum. Example, in the pyramid below, the maximum path is $3 + 7 + 4 + 9 = 23$.

```
   3
  7 4
 2 4 6
9 5 9 3
```

Question 1

For a pyramid of a base of size n, how many paths are there? Is it possible to use a brute force algorithm to enumerate all paths to find the maximum?

Solution elements :

A pyramid of base n has a height n. By building the path from the top, at each level (with the exception of the base), it is possible to extend the path down to the left or to the right. There are therefore 2^{n-1} possible paths. It is not an option to enumerate them.

Question 2

We note x a position in the pyramid, $v(x)$ the value of the integer at this position, $g(x)$ the integer below x to the left, $d(x)$ the integer below x to the right and we are looking to compute $c(x)$ the total of the numbers traversed by a maximum path from x.

Using these definitions, propose a recursive function to compute the $c(x)$ value for any x.

Solution elements :

One possible definition:

- $c(x) = v(x)$ for any x at the base of the pyramid ;
- $c(x) = v(x) + max(c(g(x))), c(d(x)))$ for any other position.

Question 3

If we directly implement this recursive function, what would be the complexity of our algorithm for a pyramid of base n?

Solution elements :

Implementing the recursive function directly, correspond to enumerate all the paths, the complexity is therefore exponential.

Question 4

For the implementation, we want to set up a technique of memoization, i.e. to store the values $c(x)$ already calculated. Propose an algorithm for this. What is its complexity?

The simplest data structure to use to store the pyramid is a list of lists.

```
[
[3],
[7,4],
[2,4,6],
[9,5,9,3]
]
```

Solution elements :

Complexity in $\mathcal{O}(n^2)$.

```
pyramid = [
    [3],
    [7, 4],
    [2, 4, 6],
    [9, 5, 9, 3]
]


def c(pyramid):
    mem = {}

    def c_rec(i, j):
        if i == len(pyramid) - 1:
            return pyramid[i][j]
        else:
            if (i + 1, j) not in mem:
                mem[(i + 1, j)] = c_rec(i + 1, j)

            if (i + 1, j + 1) not in mem:
                mem[(i + 1, j + 1)] = c_rec(i + 1, j + 1)

            return pyramid[i][j] + max(mem[(i + 1, j)], mem[(i + 1, j + 1)])

    return c_rec(0, 0)


print(c(pyramid))
```

Exercice 2 : Traveling salesman problem

Let S be a set of sites and $d : S \times S \to \mathbb{N}$ a total function giving distances between sites. The traveling salesman problem is, starting from a given site, to find a minimum distance cycle passing exactly once through each site, and returning to the departure site.

In general, we identify the n sites of S with integers $\{1, 2, \ldots, n\}$, and we suppose that site 1 is the departure site.

For $S' \subseteq \{2, \ldots, n\}$ and $x \in S \setminus S'$, we note $D(S', x)$ the length of the shortest path starting at the departure site 1, visiting all sites within S' (exactly one time) and ending at site x. We clearly have $D(\emptyset, x) = d(1, x)$ and $D(\{2, \ldots, n\}, 1)$ corresponding to the optimal solution of the problem.

Question 1

Give a recursion formula for $D(S', x)$. Justify your answer.

Élements de correction :

$$D(S', x) = min_{y \in S}(D(S' \setminus \{y\}, y) + d(y, x))$$

Question 2

Write a recursive function $D(S', x)$ (in Python or in pseudo-code) implementing the recursion formula.

Élements de correction :

```
def D(S,x):
    if S == []:
        return d(1,x)
    minD = sys.maxsize
    for i in range(len(S)):
        y = S[i]
        S.remove(y)
        minD = min(minD, D(S,y) + d(y,x))
        S.insert(i,y)
    return minD
```

Below is a random intput generation to test the previous code and the code of Question 6.

```
####### Données à tester ##########

import random
import math

def genere_carte(lx, ly, n):
    carte = []
    for i in range(n):
        while True:
            p = (random.randint(0, lx-1), random.randint(0, ly-1))
            if p in carte:
                continue
            carte.append(p)
            break
    return carte

def distance(carte, i, j):
    return round(math.hypot(carte[j][0]-carte[i][0], carte[j][1]-carte[i][1]))

def generer_matrice_distances(carte):
    return [[distance(carte, i, j) for j in range(len(carte))] for i in range(len(carte))]

matrice_distances = generer_matrice_distances(carte)

def d(x, y):
    return matrice_distances[x-1][y-1]
```

Question 3

What is the complexity of the function $D(S', x)$ in terms of the size of S'?

Élements de correction :

We define $C(n)$ the complexity of $D(S', x)$ where n is the size of S'. We have $C(0) = O(1)$ and $C(n) = n \times (C(n-1) + O(1))$. We deduce that $C(n) = O(n!)$

Question 4

Unroll the execution of $D(\{x_1, x_2, x_3, x_4, \ldots, x_m\}, x)$ in the form of a tree of recursive calls. What do you observe?

Élements de correction :

In the call tree, we find redundant computations (same case computation on several branches). for example the following 2 branches result in the same computation:

- $D(\{x_1, x_2, x_3, x_4, \ldots, x_m\}, x) \longrightarrow D(\{x_2, x_3, x_4, \ldots, x_m\}, x_1) \longrightarrow D(\{x_3, x_4 \ldots, x_m\}, x_2) \longrightarrow D(\{x_4 \ldots, x_m\}, x_3)$
- $D(\{x_1, x_2, x_3, x_4, \ldots, x_m\}, x) \longrightarrow D(\{x_1, x_3, x_4, \ldots, x_m\}, x_2) \longrightarrow D(\{x_3, x_4 \ldots, x_m\}, x_1) \longrightarrow D(\{x_4 \ldots, x_m\}, x_3)$

Question 5

Apply the principles of dynamic programming to propose a table to save intermediate computations. Give its dimensions and its contents.

Élements de correction :

To avoid redundant computation when running $D(\{2, \ldots, n\}, 1)$, we save the intermediate results of $D(S', x)$ where $S' \subset \{2, \ldots, n\}$ and $x' \in \{2, \ldots, n\} - S'$.

We need a 2-dimensions table of size $(n-1) \times (2^{n-1} - 1)$.

- one dimension (for example columns) corresponds to subsets S' of $\{2, \ldots, n\}$ of size $2^{n-1} - 1$ as we do not consider the whole set $\{2, \ldots, n\}$
- the other dimension (the rows) of size $n - 1$ corresponds to the different values of x.

We can see that for each line (i.e. each value of x) only the half will be filled because of the condition $x \notin S'$. This may serve for an accurate calculation of the complexity in the following.

Question 6

Write in Python or in pseudo-code an algorithm to solve the traveling salesman problem.

Élements de correction :

```
# cette fonction sauvegarde dans la table
# une table T est un dictionnaire (clés : les x) de dictionnaires (clés: les sous-ensembles de S)
def save(S, x, T, v):
    if not x in T:
        T[x] = {}
    # on transforme la liste S en un String hashable pour l'utiliser comme clé du dico
    T[x][str(S)] = v
    return v


def D_dyn(S, x, T = {}):
    # verifier la présence du calcul dans la table
    if x in T and str(S) in T[x]:
        return T[x][str(S)]

    # sinon on applique la formule de récurence (partie inchangée)
    # on sauvegarde dans la table avant de retourner
    if S == []:
        return save(S, x, T, d(1, x))

    minD = sys.maxsize
    for i in range(len(S)):
        y = S[i]
        S.remove(y)
        minD = min(minD, D_dyn(S, y, T) + d(y, x))
        S.insert(i, y)

    return save(S, x, T, minD)
```

Question 7

Give its complexity and compare with the recursive version.

Élements de correction :

The complexity of this algorithm is the size of the table ($\approx n \times 2^n$) multiplied by the computation time of a cell in $O(n)$. We therefore have a complexity in $O(n^2 \times 2^n)$ which is exponential but much less than $O(n!)$ The complexity of the recursive algorithm.