IE Algorithmics and Programmation 2

SCAN - June 2022

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Duration: 2h
Documents and calculator forbidden

Warning : A program that is **badly indented**, **badly commented** or with the **wrong choice of variable names** will be **penalized** *(up to -1 point)*.

« Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. » *Martin Golding*

Exercise 1 Code Reading - Recursion (3 pts)

Given the following code :

```
1  def function (k,n):
2      res = 1
3      if n == k or k == 0 :
4          res = 1
5      else :
6          res = function(k,n-1) + function(k-1,n-1)
7      return res
8
9  total = 4
10 for n in range(total+1) :
11     for k in range(n+1) :
12         print(f"{function(k,n)} ", end="")
13     print()
```

(Q1.1) What is displayed on the terminal?

(Q1.2) What is the base case in this example? Specify the lines of the program and their interpretation

(Q1.3) What is the recursive form in this example?

Exercise 2 Code Reading - 2D arrays (3 pts)

Given the following code :

```
def function1(t,a,b,c):
    for i in range(len(t[a])):
        t[a][i] = t[a][i] + c*t[b][i]

def function2(t,l):
    t2=[]
    for i in range(len(t)):
        t2.append(t[i].copy())
        t2[i].append(l[i])
    return t2


t=[[-1,1,2],[2,1,1],[1,3,3]]

function1(t,1,0,2)
print(t)

print(function2(t,[2,3,1]))

print(t)
```

(Q2.1) What is displayed on the terminal?

(Q2.2) Briefly explain what the functions `function1` and `function2` do?

Exercise 3 Functional decomposition - Game of Nim (5 pts)

The game of Nim is a two player pure strategy game. There are many versions of it (like the game of sticks in Fort-Boyard), but here we will focus on the version with multiple heaps. Consider the following code that implements this version of the game :

```python
from random import randint

nb_heat = 3
np_piece_max = 10
num_player = 1
fini = False
the_heaps = []
for i in range(nb_heat):
    the_heaps.append(randint(1,np_piece_max))
print(f"{len(the_heaps)} heap : ", end="")
for ele in the_heaps:
    print(f" {ele} ", end="")
print()
while fini == False :
    print(f"Player {num_player}, which heap?")
    num_heap = int(input(f"Enter a value between 0 and {nb_heat-1} : "))
    while num_heap < 0 or num_heap >= nb_heat :
        num_heap = int(input(f"Enter a value between 0 and {nb_heat-1} : "))
    while the_heaps[num_heap] == 0 :
        print("Heap empty!")
        print(f"Player {num_player}, which heap?")
        num_heap = int(input(f"Enter a value between 0 and {nb_heat-1} : "))
        while num_heap < 0 or num_heap >= nb_heat :
            num_heap = int(input(f"Enter a value between 0 and {nb_heat-1} : "))
    print(f"Player {num_player}, how many pieces?")
    np_piece = int(input(f"Enter a value between 1 and {the_heaps[num_heap]} : "))
    while np_piece < 1 or np_piece > the_heaps[num_heap] :
        np_piece = int(input(f"Enter a value between 1 and {the_heaps[num_heap]} : "))
    the_heaps[num_heap] -= np_piece
    print(f"{len(the_heaps)} heap : ", end="")
    for ele in the_heaps:
        print(f" {ele} ", end="")
    print()
    if num_player == 1 :
        num_player = 2
    else :
        num_player = 1
    fini = True
    i = 0
    while fini and i < len(the_heaps):
        if the_heaps[i] != 0 :
            fini = False
        else :
            i += 1
print(f"Player {num_player} has lost !")
```

In this version, we consider N heaps of pieces, numbered from 0 to $N-1$. The players play alternately. When it is his turn, a player indicates the number i of the heap from which he wants to remove pieces. If n_i is the number of pieces in this heap, the player can take between 1 and n_i pieces in a single heap (but he can change the heap on the next turn). The winner is the one who removes the last piece, all heaps combined. The loser is therefore the one who must play while all the heaps are empty.

(Q3.1) Propose a functional decomposition with at least four functions from the proposed code, according to the following guidelines :

You will write the main part of this program using the functions you propose. For each function, you will write its signature as well as its content. The proposed functions must perform a meaningful subtask and/or avoid repeating lines of code in the main program. Regarding the content of the functions, you can if you wish write the number of lines identical to the program above and add only the lines of code that differ. For instance :

```
def function1 (param1) :
    lines 8 to 21
    return var
```

Exercise 4 2D Lists - Prim's Algorithm (10.5 pts)

A - Warming up

(Q4.1) Propose a function `matrix_empty` which initializes an empty square matrix. This function takes the size of the matrix as a parameter and returns a square matrix filled with -1.

B - Prim's Algorithm

We want to build a network of high speed train lines which connects all the major cities of the country to the capital at the lowest possible cost. We already know the lines that could be built between different cities and their cost. An example configuration is shown in Figure 1. Your goal is to **find a set of lines that allows all cities to be connected by train to the capital, with minimal total construction cost** (an example of solution is shown in blue in Figure 1).

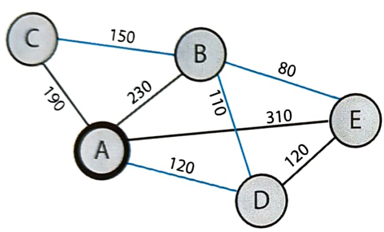

FIGURE 1 – An example of possible train lines (black and blue lines) between cities A (the capital), B, C, D and E. The set of lines drawn in blue is a solution to the problem.

	A	B	C	D	E
A	-1	230	190	120	310
B	230	-1	150	110	80
C	190	150	-1	-1	-1
D	120	110	-1	-1	120
E	310	80	-1	120	-1

FIGURE 2 – The matrix representing the configuration in Figure 1.

To solve this problem, you will implement Prim's algorithm. It consists of adding the train lines one by one, choosing each time the cheapest line that connects a city not yet connected to the capital to a city already connected. It can be written with the following pseudo-code :

```
# the capital is considered connected
# while there are still cities not connected to other cities:
#     we find the cheapest train line that connects a
#           new city
#     we update the set of cities connected to the other cities
```

Running this algorithm on the example in Figure 1 would add the following train lines (in order) :

- the line from A to D (cost 120)
- the line from D to B (cost 110)
- the line from B to E (cost 80)
- the line from B to C (cost 150)

To implement this algorithm, we will represent the train lines that are possible to build as a matrix `matrix_lines` of size $n \times n$ (n is the number of cities to connect, including the capital) where `matrix_lines[i][j]` represents the cost of a line between the cities `i` and `j`. If it is not possible to build a line between the cities `i` and `j`, then `matrix_lines[i][j]` is -1. The corresponding matrix to the example is shown in Figure 2. Note that `matrix_lines[i][j] == matrix_lines[j][i]` and `matrix_lines[i][i] == -1`.

To find out which cities have already been connected, we will use a list of booleans `list_cities` of length n which indicates for each city if it is already connected to the others : if the city i is already connected to another city, then `list_cities[i]` is `True`, `False` otherwise.

(Q4.2) Propose a function `all_cities` which checks if all cities are connected.
This function takes as a parameter the list of booleans `list_cities` defined above and returns `True` if all the cities are connected to the network, `False` otherwise. Make your loop stop as soon as possible.

(Q4.3) Propose a function `new_city` which tests whether the line connecting the cities `i` and `j` allows to connect a new city to the network (connects a city already connected and a city not yet connected).
This function returns `True` if the line connects a new city, `False` otherwise. It takes as parameters the indices `i` and `j` of the cities linked by the line and the list `list_cities`.

To ease the rest of the algorithm, we want to extract from the matrix **`matrix_lines`** a list of possible lines in the form of a 2D list **`list_lines`** of size $m \times 3$ (m the number of possible lines) where :

- `list_lines[i][0]` represents the starting point of the line i
- `list_lines[i][1]` represents the end point of the line
- `list_lines[i][2]` represents the cost of the line

From the matrix given as an example, we would obtain the following list :
`[1,0,230],[2,0,190],[2,1,150],[3,0,120],[3,1,110],[4,0,310],[4,1,80],[4,3,120]`

(Q4.4) Propose a function `list_from_mat` which creates the list `list_lines` of possible train lines from the matrix `matrix_lines`. You will ensure that the list contains each train line only once (the direction of the line does not matter).

(Q4.5) Propose a function `next_line` which finds the next line to add : the line with the lowest cost that allows to reach a new city.
This function takes as parameters the lists `list_cities` and `list_lines` defined above and **returns the indices of the two cities connected by the chosen line.**

(Q4.6) Propose a function `algorithm_prim` that implements Prim's algorithm described above. **You will reuse the functions defined previously (in part 4.B).**
This function takes as parameters the matrix `matrix_lines` and the index of the starting city (the capital) and returns the list of pairs of cities connected by the chosen lines and the total cost of construction. It displays the chosen line at each turn of the loop.

You can use the function **`init_cities`** which initializes the list **`list_cities`** : it takes as a parameter the number of cities and returns a list filled with `False`. **We will assume that this function is already defined : you don't have to write it.**

Executing the function `algorithm_prim(matrix_lines,0)` on the example defined above would return the list `[[3,0],[1,3],[4,1],[2,1]]` and a total cost of 460, and would output at runtime :

```
New line from 0 to 3
New line from 3 to 1
New line from 1 to 4
New line from 1 to 2
```

(Q4.2) Propose a function `all_cities` which checks if all cities are connected.
This function takes as a parameter the list of booleans `list_cities` defined above and returns `True` if all the cities are connected to the network, `False` otherwise. Make your loop stop as soon as possible.

(Q4.3) Propose a function `new_city` which tests whether the line connecting the cities `i` and `j` allows to connect a new city to the network (connects a city already connected and a city not yet connected).
This function returns `True` if the line connects a new city, `False` otherwise. It takes as parameters the indices `i` and `j` of the cities linked by the line and the list `list_cities`.

To ease the rest of the algorithm, we want to extract from the matrix **`matrix_lines`** a list of possible lines in the form of a 2D list **`list_lines`** of size $m \times 3$ (m the number of possible lines) where :

- **`list_lines[i][0]`** represents the starting point of the line i
- **`list_lines[i][1]`** represents the end point of the line
- **`list_lines[i][2]`** represents the cost of the line

From the matrix given as an example, we would obtain the following list :
`[1,0,230],[2,0,190],[2,1,150],[3,0,120],[3,1,110],[4,0,310],[4,1,80],[4,3,120]`

(Q4.4) Propose a function `list_from_mat` which creates the list `list_lines` of possible train lines from the matrix `matrix_lines`. You will ensure that the list contains each train line only once (the direction of the line does not matter).

(Q4.5) Propose a function `next_line` which finds the next line to add : the line with the lowest cost that allows to reach a new city.
This function takes as parameters the lists `list_cities` and `list_lines` defined above and returns the indices of the two cities connected by the chosen line.

(Q4.6) Propose a function `algorithm_prim` that implements Prim's algorithm described above. You will reuse the functions defined previously (in part 4.B).
This function takes as parameters the matrix `matrix_lines` and the index of the starting city (the capital) and returns the list of pairs of cities connected by the chosen lines and the total cost of construction. It displays the chosen line at each turn of the loop.

You can use the function **`init_cities`** which initializes the list **`list_cities`** : it takes as a parameter the number of cities and returns a list filled with **`False`**. **We will assume that this function is already defined : you don't have to write it.**

Executing the function **`algorithm_prim(matrix_lines,0)`** on the example defined above would return the list `[[3,0],[1,3],[4,1],[2,1]]` and a total cost of 460, and would output at runtime :

```
New line from 0 to 3
New line from 3 to 1
New line from 1 to 4
New line from 1 to 2
```