Verónica Gaspes
School of Information Technology
December 21, 2022

Python: a gateway to machine learning DT8051
Written Exam

January 3, 2023 from 09:00 to 14:00

- No aids or tools are allowed.
- **Grading criteria:** You can get at most 26 points.

  To pass you need at least 50% of the points.

  For the highest grade you need more than 90% of the points.
- **Responsible:** Verónica Gaspes (telephone: 0722 39 14 16)

- **Read carefully!** Some exercises might include explanations, hints and/or some code. What you have to do in each exercise is marked with the points that you can get for solving it (as **(X pts)**).
- **Write clearly!** You must write complete sentences in your answers and when you provide an example you should explain in what way it is an example of what was asked for.
- **Motivate your answers!**

Good Luck!

1. Consider the following Python function that helps in simulations for buying and selling stocks of shares. The function has one argument: an array with the price of a share over a number of past days. The indices of the array are taken to be the consecutive days 0, 1, etc. The elements are the price of the share on those days. The function returns the buying day and the selling day for maximum gain and this maximum gain. The algorithm implemented in the function assumes that prices are non-negative and that you sell after buying.

```
def buy_sell(share_price):

    n = len(share_price)

    b_s = (-1,-1)
    gain = 0

    for buy_day in range(n):
        for sell_day in range(buy_day + 1, n):
            diff = share_price[sell_day] - share_price[buy_day]
            if diff > gain:
                b_s  = (buy_day,sell_day)
                gain = diff
    return (b_s, gain)
```

   (a) **(1 pts)** What does the function calculate when it is called with the argument `[9, 1, 5]`?

   (b) **(1 pts)** Provide an argument for which the result is `((-1,-1), 0)`. How would you interpret this result?

   (c) **(1 pts)** What aspect of the input would you consider the size of the input?

   (d) **(1 pts)** Are there inputs that can be considered worst cases? Explain how you came up with your answer.

   (e) **(1 pts)** Explain how you calculate the Big-O for the execution time of this function. What do you get as a result?

2. The algorithm in Exercise 1 is a so called *Brute Force* algorithm: it tests all possible pairs of buying days and selling says and identifies one pair with maximum gain. In this exercise you will develop a *Divide & Conquer* solution to the same problem.

   (a) **(1 pts)** What aspect of the input will you use to determine the sub-problems?

   (b) **(1 pts)** Explain what subproblems you consider and what you have to do to obtain the subproblems.

   (c) **(1 pts)** Explain what you have to do to combine the solutions to the subproblems, possibly doing some extra work, to solve the problem.

   (d) **(2 pts)** Write down the algorithm (or Python function) that implements your ideas from parts 2(a), 2(b) and 2(c).

   (e) **(1 pts)** The Master Method helps calculate the Big-O for recursive algorithms designed using Divide & Conquer. Use it to calculate the Big-O for your algorithm.

   Here is the Master Method:

$$T(n) \text{ is } \begin{cases} O(n^d \log n), & \text{if } a = b^d \\ O(n^d), & \text{if } a < b^d \\ O(n^{log_b a}) & \text{if } a > b^d \end{cases} \quad (1)$$

   with a the number of recursive calls, b the shrinking factor for the size of the input and d the exponent of the running time of the combine part.

3. The function `revenue` defined below solves the problem of calculating the maximum revenue of placing billboards along a road. There are positions along the road given as distances to the start of the road where the billboards can be placed. This is given as an array `dist`. For each position there is an associated revenue. This is given as another array `rev` of the same length.

   Billboards are not allowed to be too close to each other: there has to be a distance of at least of 5 between two consecutive billboards.

   For any position `i` in the array of distances, the pre-defined function `e(i, dist)` calculates the closest position to the left of `i` that is at a distance bigger than 5 from position `i`.
   Consider the example where

   - `dist` is `[6, 7, 12, 14]` indicating that billboards can be placed at distances 6, 7, 12, and 14 from the start of the road.
   - `rev` is `[5, 6, 5, 1]` indicating the revenue of placing a billboard at each of these positions.

   In this case the result of the predefined function `e(3, dist)` is `1` because the distance between billboard at distance 14 (position 3 in the array) and billboard at distance 7 (position 1 in the array) is 7 (bigger than 5) but the distance between billboard at distance 14 and billboard at distance 12 (position 2 in the array) is 2 (smaller than 5).

   The first argument to the function is `n`: the number of elements considered from the start of the array. So that to solve a problem it should be called with the length of the two arrays with the distances and the revenues as in `revenue(4, [6, 7, 12, 14], [5, 6, 5, 1])`.

```
def revenue(n, dist, rev):
    if n == 0: return 0
    if n == 1: return rev[0]
    k = e(n-1, dist)
    return max(revenue(k+1, dist, rev) + rev[n-1],
               revenue(n-1, dist, rev))
```

   Unfortunately, the program takes too much time due to overlapping recursive calls.

   (a) **(1 pts)** What is the base case and what are the recursive calls?

   (b) **(1 pts)** Enumerate the recursive calls made by the program when used with input
   - 4 for `n`
   - `[6, 7, 12, 14]` for `dist`
   - `[5, 6, 5, 1]` for `rev`

   as in `revenue(4,[6,7,12,14],[5,6,5,1])`.

   What is the result?

   (c) **(1 pts)** What are the smaller problems that need to be solved in order to calculate the revenue using up-to the first n billboards?

   (d) **(1 pts)** Suggest a way of storing results to smaller problems so that they are available when needed.

   (e) **(2 pts)** Use your suggestion to write a dynamic programming algorithm and show how to obtain the result to the problem. You can express your algorithm as a Python program.

   (f) **(1 pts)** Explain how you calculate the Big-O for your algorithm. What is the Big-O of your algorithm?

4. Consider the following program that implements a sorting algorithm called *selection sort*.

```
def selection_sort(a):
    for i in range(0,len(a)):
        min_pos = i
        for j in range(i, len(a)):
            if (a[j] < a[min_pos]): min_pos = j
        a[min_pos], a[i] = a[i], a[min_pos]
```

Unfortunately this algorithm is quadratic in the size of the array! More formally, $T(n) = O(n^2)$ for n the number of elements in the array to be sorted. This is because the algorithm has to identify the minimum element in the rest of the array for every position i, and this takes linear time.

A heap is a data structure for inserting items and retrieving the minimum element. In Python heaps are lists. The operations on a heap that has n items have the following execution times:

- Inserting an item is $O(\log n)$. In Python the operation is `heapq.heappush(hp, item)`.
- Retrieving the minimum item is $O(\log n)$. In Python the operation is `heapq.heappop(hp)`
- Heapifying a list is $O(n)$. In Python the operation is `heapq.heapify(lst)`. After heapifying a list the list can be considered a heap.

(a) **(2 pts)** Explain how you could improve the execution time of selection sort using a heap. Write an algorithm that implements your idea.

(b) **(2 pts)** Explain how to calculate the Big-O for the execution time of your algorithm. What is the Big-O for your algorithm?

5. Consider the problem of identifying the *majority* element in an array. A majority element is an element of the array that appears more that half of the total number of elements. For example, in the array `[1,2,3,4,2,2,2]` the value 2 is the majority element: there are 7 elements and 2 occurs 4 times. It is easy to see that if there is a majority element it is unique.

(a) **(2 pts)** Your task is to write an algorithm (or a Python program that implements the algorithm) that works in linear time on the size of the array.
The input to your algorithm is an array that has a majority element. The output is the majority element.

(b) **(2 pts)** Explain how to calculate the Big-O for the execution time of your algorithm. If you use a data structure you need to state what the execution times of the operations you use are.