Algorithms: Assignment 5

Due on May 31, 2018 at 20:00

Prof. Antonio Carzaniga

A. Romanelli

Exercise 1

In this exercise we want to help our friend Antonio which has been studying so called *irregular* subsequences. His very imaginative definition states that an irregular subsequence is such if and only if the following holds within a given sequence of integers:

$$a_{i_1} > a_{i_2} < a_{i_3} > a_{i_4} < a_{i_5} > \dots$$

Firstly I misinterpreted what Antonio meant, as I interpreted *subsequences* as *substrings* which are not really the same thing, since the possible substrings of `abc` are [`a`, `b`, `c`, `ab`, `bc`, `abc`] whereas its subsequences are all the possible combinations also of non successive characters: [`a`, `b`, `c`, `ab`, `ac`, `bc`, `abc`]

Once I realised what the problem was actually asking, I felt pulled toward a Dynamic Programming approach, which would exploit recursion and the storing of partial results as I'll show you later in **Exercise 3**, but instead of going for the recursion I decided to take a easier approach in terms of both asymptotic complexity and writability.

Solving Strategy

The idea behind my solution is to linearly scan the given sequence and to compose a new sequence of operators that can be either `>` or `<` . Once this sequence is compiled, I can count how many times the operator flips and add one to count the last element considered. Let's take a look at some more concrete examples:

The sequence: $[0,\ 4,\ 2,\ 3,\ 3]$ has the following relationships:

$0 < 4 > 2 < 3 = 3,$ thus our operators' sequence will be:

$$[>, <] \implies 3$$

We can now just count the elements in our operators' sequence and add one to find what's the longest possible subsequence that satisfies such a property like the *irregular* one.

$[1,\ 2,\ 3,\ 4]$	$[2,\ 1,\ 2,\ 1,\ 2]$	$[0,\ 4,\ 3,\ 0,\ 0,\ 2,\ 2,\ 3,\ 4]$
$1 < 2 < 3 < 4$	$2 > 1 < 2 > 1 < 2$	$0 < 4 > 3 > 0 = 0 < 2 = 2 < 3 < 4$
$[<] \implies 2$	$[>, <, >, <] \implies 5$	$[>, <] \implies 3$

Now that the strategy is clear, I should better explain how I can get from a given sequence to the correct operators' sequence.

Implementation

This time around I don't have to go about how my data structure works so I'll try to be as brief as possible, the code you're about to see is the product of condensing and refactoring to minimal terms the code to avoid redundancies.

```
1 A = []
2 for n in input().split():
3   A.append(int(n))
```

As always we need to parse the input from a string into sensible data.

Once we are done with the parsing we can get to work:

```
4  def lenOfIrregEx(A):
5    if len(A) < 2:
6        return len(A)
7    signs = []
8    for i in range(1,len(A)):
9        if A[i-1] > A[i]:
10           if len(signs) == 0 or signs[-1] != ">":
11               signs.append(">")
12       elif A[i-1] < A[i]:
13           if len(signs) != 0 and signs[-1] != "<":
14               signs.append("<")
15   return len(signs) + 1
```

In the first two lines of this function I handle the case whether our array consists of less than two elements, in which cases we really can't compare anything but we know already the solutions to such problems.

Else the first thing to do is — after allocating the `signs` array — to linearly scan the given sequence. I do not want to append to the `signs` array every operator that I'm able to find, but only those that make sense to count, thus if I have `>` and `>` as its successive, it would not make sense to append the second one. What I'm interested in is to append an operator only when it's different from the last one I have appended. Moreover I do not want my operator list to start with a `<` but exclusively with `>` .

The conditions on which I decide to append (or not) an operator to my array should now be clearer. The very last thing we are left to do is to simply count how many operators I have at the end of the scan and add 1.

Time Complexity Analysis

As I previously described, the only thing we are doing is a linear scan of the array and for each element we'll append it to our operators' array if that's the case. Our best case scenario is that we don't have anything to append and thus we only take n time.

Our worst case happens when the whole sequence is an irregular one. But what does this imply?

If our whole sequence is irregular, then we will need to allocate $n-1$ operators within the operators array and — as we all know — appending an element to an array takes $\mathcal{O}(c)$ and since we would do it for $n-1$ elements this gives us a worst case complexity of: $c(n-1)$.

Thus we can conclude that this code runs in $\mathcal{O}(n)$

Exercise 2

This time around, our pal Antonio has given us a $n \times m$ grid, just like the one we had created in **Assignment 4**, where we were looking for the shortest path out of a maze. According to his ever so imaginative vocabulary, he decided to denote a specific sequence of adjacent cells of this grid as *interesting* path if every value in a cell is greater than its preceding one. Our goal is to return the length of the longest *interesting* path.

Assets

As in the previous assignment, I recycled the structures of a `Cell` and `Grid`. The latter stayed the same where I slightly modified the `Cell` structure to fit my needs:

```
class Cell:
    def __init__(self, key):
        self.x = 0
        self.y = 0
        self.key = key
        self.longest = 0
    def __lt__(self, other):
        if self.key == other.key:
            return self.y < other.y
        else:
            return self.key < other.key

class Grid:
    def __init__(self, n, m):
        self.matrix = []
        self.x = m
        self.y = n
        self.longest = 1
    def addRow(self, n, row):
        self.matrix.append(row)
```

Data Structure for the `Grid` and `Cell`

I also recycled the input parser, which is going to allocate the grid depending on the first two integers passed from the standard input and then allocate one row at a time:

```
n,m = [int(x) for x in input().split()]
g = Grid(n,m)
for i in range(n):
    line = input().split()
    row = []
    for j in range(m):
        cell = Cell(int(line[j]))
        cell.x = j
        cell.y = i
        row.append(cell)
    g.addRow(i, row)
```

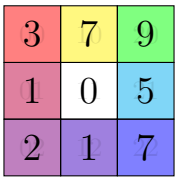
Input parser for a $n \times m$ `Grid` allocation

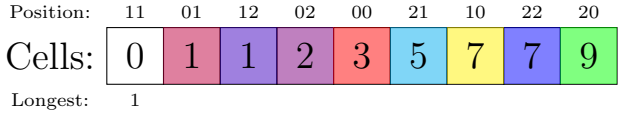
Solving Strategy

The intuition that was most helpful in solving this problem came to me when I took a look at the proposed complexity of $\mathcal{O}(nm \log(nm))$, this pointed me towards the fact that I could exploit a sorting algorithm and the fact that the *interesting* path must account for a number being greater than its predecessor.
The idea behind my solution is to allocate a linear array of cells and to sort them in ascending order using

the built-in function provided by **Python**, then setting the base case to be the first element of this newly created array, which will have a longest length of 0: the *interesting* path that terminates in the considered cell.

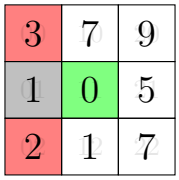
Then from left to right I'm going to compute the length of the longest path exploiting the base case that we just set and checking whether there are adjacent cells that have lesser key than the one we are considering.

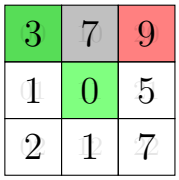


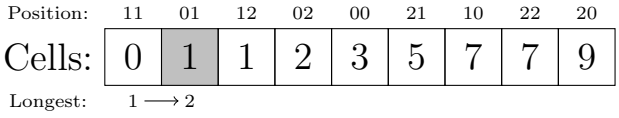


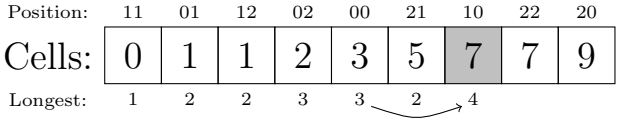
Now that we have our linear array and our base case that the cell 0 at position 11, besides having four adjacent cells, since it's the minimum of the array, it cannot be the ending cell of a sequence longer than 1.

If we want to compute the longest sequence for a given cell, it's enough to consider its adjacent cells that have a key lesser than the considered one — thus implying that we already computed the adjacent cell longest path — and simply add one to the maximum longest path of the adjacent cells that we managed to find.









As we keep computing the longest distance for every cell, it's going to be easy to keep track of the longest one we found so far and to return at the end of our process, solving the problem.

Implementation

My implementation is quite straight forward and should not require any further explanation as it follows 1:1 my solving strategy:

- Generate array sorted array of cells;
- Compute distance by watching adjacent cells that were already computed;
- Return the longest distance ever computed.

```
def findLongestInterestingPath(Grid):
    if Grid.x < 1 and Grid.y < 1:
        return 1

    cells = []

    for i in range(Grid.y):
        for j in range(Grid.x):
            cells.append(Grid.matrix[i][j])

    cells.sort()

    cells[0].longest = 1
    for j in range(1, len(cells)):
        cell = cells[j]
        adjacents = []
        if cell.x - 1 >= 0:
            adjacents.append(Grid.matrix[cell.y][cell.x-1])
        if cell.x + 1 < Grid.x:
            adjacents.append(Grid.matrix[cell.y][cell.x+1])
        if cell.y -1 >= 0:
            adjacents.append(Grid.matrix[cell.y-1][cell.x])
        if cell.y +1 < Grid.y:
            adjacents.append(Grid.matrix[cell.y+1][cell.x])
        longest = 0
        for adj in adjacents:
            if adj.key < cell.key and adj.longest > longest:
                longest = adj.longest
        cell.longest = longest + 1
        if cell.longest > Grid.longest:
            Grid.longest = cell.longest
    return Grid.longest
```

Implementation of the solving strategy

Time Complexity Analysis

In this exercise sorting the array came at a cost of complexity, which as we know is $\mathcal{O}(n \log n)$, but since we sorted an array of cells of our $n \times m$ grid, we effectively sorted nm cells, thus our asymptotic complexity so far is going to be: $\mathcal{O}(nm \log(nm))$. The rest of our implementation can be trivially seen to be linear over the number of cells, which does not affect our complexity. This means that our complexity is going to match the requirements for this assignment:

$$\mathcal{O}(nm \log(nm))$$

Exercise 3

Antonio's obsession this exercise is about palindromic subsequences and more specifically he wants to know, given a string, what's the longest palindromic subsequence present in the string. Just like in the first exercise, there was a possible misunderstanding of the difference between substring and subsequence. Specifically:

abracadabra

Has a palindromic subsequence of length 7, you don't believe me?

abracadabra
abadaba

Solving Strategy

The initial idea would be to solve this problem by means of recursion, checking all the possible subsequences that there are but we would soon run out of time complexity, as the numbers of subsequences to check grows exponentially.

The concept of dynamic programming that we recently studied in class could turn out useful. We know that our base case will be for the subsequence of length 1, which must also have length 1. If we then have a tree which has at its root our string and as its children all the possible subsequences:

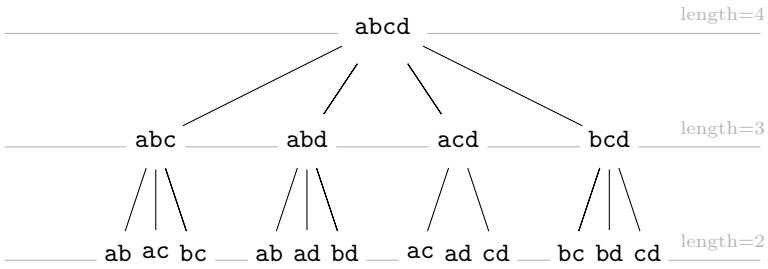


Figure 1: All possible subsequences for `abcd`

Something worth noticing is how we considered multiple times the same subsequences which means that we are going to spend time trying to compute something we already did previously, which hint the fact that we should store partial subproblem solutions using the tabulation method.

If we consider an actual palindrome subsequence like:

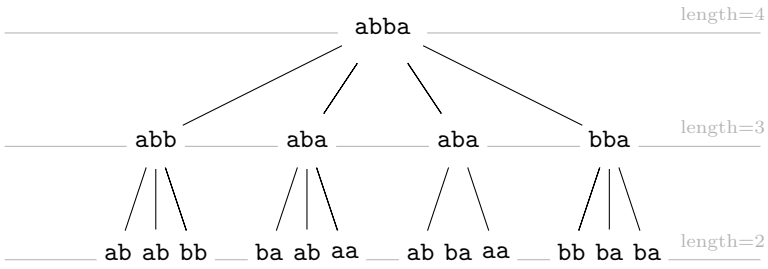


Figure 2: All possible subsequences for `abba`

We can find out the length of the longest palindromic subsequence by starting from our leaves and doing solving the subproblems bottom-upwards.

If the first and last characters of a subsequence of length n match, then we take the value stored in the sub-subsequence of length n-2 and add 2 to it, whereas if they don't match we just carry on the bottom highest value from the children of the node. We should also consider the case where if we are considering subsequences of length 2 and we find a match, since we cannot look for a subsequence of length 2-2 = 0, we need to explicitly set the new highest value to be 2.

Starting from the individual characters — thus the length of the longest palindromic subsequence is 1 — the program will build a tree bottoms up, starting from the base cases:

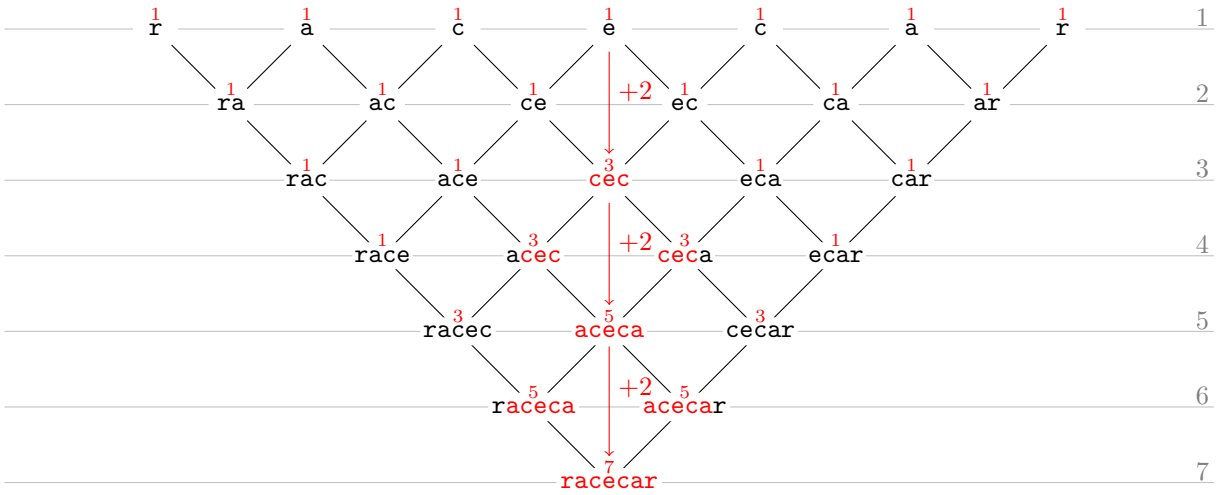


Figure 3: Diagram of the bottom-up execution, from the shortest subsequences to the complete string

Now that we have found a way to link sub-solutions together we have pretty much solved the problem. I'll use a HashMap which is going to keep track of the sub-solutions I have found so far but what I still miss is some kind of coordinate system to keep track where I am at in the tree.

My best intuition was to use the length of the subsequence — having set the base case already for subsequences of length 1 — that you can see in the graph above on the right hand side.

This would act as the first component of my "cursor" and for the second component I could use the index of the first character of the substring (which would implicitly tell me the index of the last character by knowing the length of the subsequence).

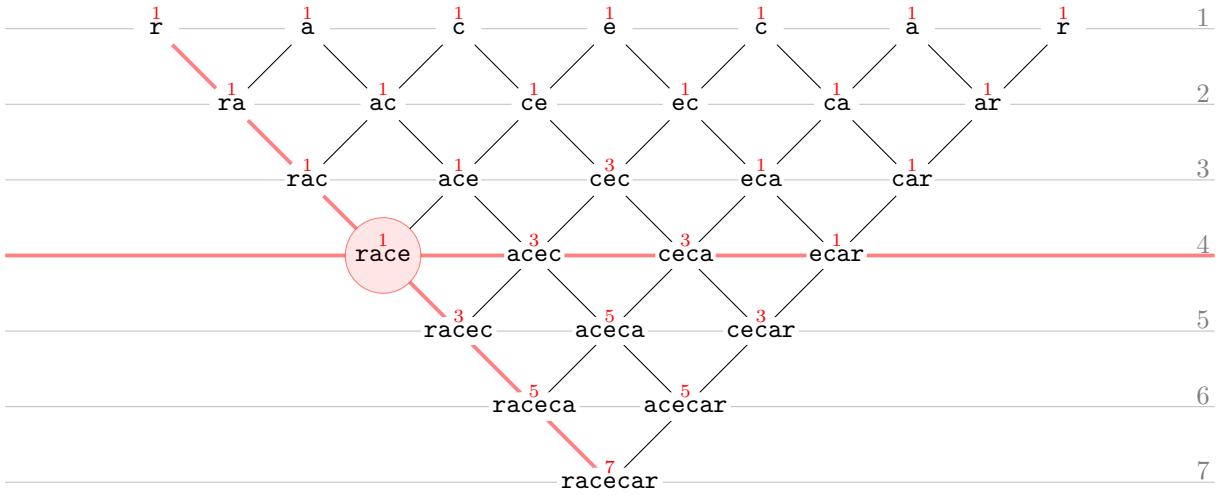


Figure 4: Identification of the substring defined by starting index 0 and depth 4

Now that we also have a system of coordinates in place we are 100% set to find the solving implementation.

Implementation

```
def lenOfMaxPalindrome(string):
    solutionsMap = {} #Allocate an empty hashmap
    #for each possible sublength from 1 to len(string) included
    for sublength in range(1, len(string)+1):
        #for each starting character index from 0 to len(string)-sublength included
        for i in range(len(string) - sublength + 1):
            #find the ending character index
            j = i+ sublength - 1
            #if the sublength is 1 ==> longest palindromic sequence is 1
            if sublength == 1:
                solutionsMap[str(i)+" 1"] = 1
            #else if the length is 2 and the two characters match assign 2
            elif sublength == 2 and string[i] == string[j]:
                solutionsMap[str(i)+" 2"] = 2
            #else if the start and end characters match and the sublength > 2 we need to retrieve
            #the subsolution of the subsequence of the same length-2 starting with index i+1
            elif string[i] == string[j]:
                solutionsMap[str(i)+" "+str(sublength)] = solutionsMap[str(i+1)+" "+str(sublength-2)] + 2
            #else we just carry on the best subsolution from the previous level of sublength-1
            else:
                l_subsolution = solutionsMap[str(i)+" "+str(sublength-1)]
                r_subsolution = solutionsMap[str(i+1)+" "+str(sublength-1)]
                solutionsMap[str(i)+" "+str(sublength)] = max(l_subsolution, r_subsolution);
    #we return the result stored at the root of our tree, which must contain the length of the
    #longest palindromic sequence we were able to find.
    return solutionsMap["0 "+str(len(string))]
```

Implementation of the solving strategy

Because I already commented my code extensively, I feel it would be useless to further explain it. However, two remarks about the structure of my code:

- I chose to use an hash-map rather than a bi-dimensional array (matrix) because I felt it would have been a wasteful use of space, since I would use only half of the space I allocate to solve this problem.
- The keys of the hash-map are formatted like so `[x y]` with a non-empty space intentionally left between `x`, which is the starting index and `y` which is the length of the subsequence. This was introduced to avoid potential key conflicts and/or unwanted overwrites, as without the space between `x` and `y`, the key `111` could map to the solution for index 1 length 11 **AND** index 11 length 1. The space solves this problem, as `"1 11"` $\neq$ `"11 1"`

Time Complexity Analysis

Since all our operations happen on a hash-map, we can assume all our put and get operations happen in $\mathcal{O}(c)$ (although it cannot be always 100% guaranteed due to possible hash collisions, which are still possible, even though highly unlikely). Having n be defined as the length of the given string, we'll run two for loops, one for traversing the tree horizontally and one to traverse the different tree depths, corresponding to the subsequence lengths. The horizontal span of the tree is going to shrink as the length of the subsequence increases, until we just have to process a single node: the root.

We can be fairly sure to say that the complexity of this algorithms is $\mathcal{O}(n^2)$.

Credits

I worked on my own throughout the whole assignment and developed all my solution autonomously. I had some exchange of opinions as usual with **Cristian Buratti** without never seeing or trading code among ourselves.

Exercise 1: Cristian actually made me realise that the problem was actually asking for subsequences, whereas I initially developed a solution to find subsets, implying that the numbers had to be consecutive.

Exercise 3: The website **GeeksForGeeks** provided truly inspiring insights on how to solve this problem[1], providing also a possible **Python** solution which I decided to overlook, focusing more on the idea of having the subproblem tree and starting from the known cases to reach its root.

Attachments

- `assignment.pdf` — This document, did I really have to say it?
- `ex1.py` — Python file, solution to **Exercise 1**
- `ex2.py` — Python file, solution to **Exercise 2**
- `ex3.py` — Python file, solution to **Exercise 3**

[1] `https://www.geeksforgeeks.org/dynamic-programming-set-12-longest-palindromic-subsequence/`