Assignment 2

Irene Avezzu' - 20142 - IAvezzu@unibz.it
Worked with: Massimo Marcon, Ivana Nworah Bortot and Emmanuel Scopelliti

1. Recursion

1. Implement your algorithm in Java

Pseudo-code:

```
boolean isPalindrome (String s)
    l := 1
    r := s.length
    while l>=r //empty array or left bound is bigger than right bound
        if s[l] != s[r]
            return false
        r--
        l++
    return true
```

2. Explain why your algorithm is correct. To do so, try to answer the following questions:

a. In which way do you reduce the problem with each recursive call?

b. Why can you be sure that the original problem is solved if the recursive call returns a positive answer? Hint: Think about partial solutions and termination.

a) Each time we iterate the recursive call (the loop in the pseudo-code) the algorithm either returns an output if one of the two base cases occurs (l>=r ➔ true or s.charAt(l) != s.charAt(r) ➔ false) or proceeds to reduce the problem (r-- and l++) and reiterate the loop.

b) Throughout the recursion of the algorithm (Java code) the partial solution is represented by the prefix [0 … l(excluded)] and the suffix [r(excluded) … A.length] of the string which are the chars that have already been compared and found to be equal. The algorithm terminates when either all chars have been scanned, compared, and found to be equal (l>=r) or when a comparison returns false.

I. *Loop invariant*: In this algorithm (Java code) the loop invariant is represented by the prefix [0 … l (excluded)] as narrow as the suffix [r(excluded) … A.length] of the string

II. *Initially*: initially the prefix and suffix are “empty” and after the first comparison include the first and last char that have just been compared

III. *Maintenance*: Thought out the recursive calls the prefix and suffix always represent the char that have been scanned, compared, and found to be equal

IV. *Termination*: The recursion terminates when either two different chars are found (s.charAt(l) != s.charAt(r)) or when l becomes bigger or equal then r (r=l if s.length is even, l>r if it’s odd). It always occurs one of these situations. So when the algorithm terminates the prefix and the suffix either represent the partial sequence of identical chars in the string (if two chars are found to not be identical) or the complete string (when l becomes bigger or equal then r).

2. Maximal Length of Ascents in Arrays

1. Implement an algorithm in Java that takes an array A of integers as input and returns the maximal length of an ascent in A

Pseudo-code:

```
int maxAscentLength(int[] A)
```

```
maxAscentLength := 1
currAscent := 1
i := 1
while (i<A.lenght-1)
    if A[i] <= A[i+1]
        currAscent++
        maxAscentLength := (currAscent>maxAscentLength)? currAscent :
            maxAscentLength
    else
        currAscent := 1
return maxAscentLength
```

2. Explain why your algorithm is correct:

- ***a. Which information does the algorithm keep while it is running? Does this information represent a partial solution? Is there additional information it maintains?***
- ***b. How is the output related to the information maintained during the execution? Why does this ensure the correctness of the answer?***

a) The algorithm has a *loop invariant which is maxAscentLength*. During the whole execution of the algorithm, it represents the maximum length of ascendant elements in the portion of the array scanned so far. Then maxAscentLength represents the partial solution for the segment of the input A[1 … i] and the remaining of the input represent the remaining problem (A[i+1 … A.length).

- Initially: it represents what has been defined because i starts as 0 and the invariant is initialized at 1 (A[0 … 0])
- Maintenance: then the invariant is increased if the value of the current ascent count is greater than its previous value.
- Termination: when the loop ends then the whole array has been scanned so the loop invariant contains the max length in A[0 … (A.length+1)-1]

The *currAscent* is an additional variant which holds the length in A[x … i] (with x that goes from 0 to i) of the ascent sequence in which is present the element A[i].

b) The loop stops when i reaches the same value as the length of A so when the whole array has been scanned, this ensures termination. Since we have defined maxAscentLength as the partial solution of A[1 .. i] and when the loop terminates we have scanned the whole array and the partial solution applies to the whole length of the array.

3. What is the worst-case running time of your algorithm with respect to the array size n? Can you characterize the running time asymptotically as Θ(f(n)) for some function f(n) of n? Explain your answer!

The *running time* for this algorithm is linear because it executes n-1 iteration of the loop for an input of size n. For each iteration of the loop we can have two options: if A[i] <= A[i+1] it operates two assignments and one comparison otherwise only one assignment.

This leads us to define the worst and best case.

In the *worst case* all the elements are in ascendent order, so the algorithm solves the comparison (for loop + if condition) and the two assignment (if -> true). We define the running time to be, for an input of length n, n-1 iteration of the loop * (2 assignment + 1 comparison). Then we define $T^{w}_{A}(n) \in \Theta(f(n))$ such that $f(n)=l*(n-1)*(a*2 + c*1)$.

For some:

- l = running time of the iteration of the loop
- a=running time of the assignment
- c=running time of the comparison

We define the <u>*best case*</u> as the case in which all elements in the input are in descent order, so the algorithm solves the comparison (for loop + if condition) and the only assignment (if -> false). So we define $T^{b}_{A}(n) \in \Theta(f(n))$ for some f(n)=l*(n-1)*(a*1+c*1).

3. Maximum Missed Gain

1. Write pseudocode for a brute-force algorithm that immediately implements the definition.

<u>Pseudo-code</u>:

```
int maxMissedGainBF(int[] A)
    buy := 0; //buying price
    sell := A.length-1 //selling price
    maxGain = A[sell]-A[buy]
    for i=A.length-1 down-to i=0 //hypothetical sell
        for j=i-1 down-to i<0 //hypothetical buy
            if A[i]>=A[j] && A[i]-A[j] > maxGain //hypothetical maxGain
                buy := j
                sell := i
                maxGain := A[i]-A[j]
    return (maxGain>=0)?maxGain:0;
```

2. How many comparisons would it make for an array of length n?(It is enough to give a rough estimate in terms of 4. Explain how many comparisons your algorithm makes in the worst case!

The outer for-loop (i) is iterated n times but within each iteration of this outer loop the inner for-loop (j) is iterated n*(n+1)/2 times ((n-1)+(n-2)+…+2+1). Within the inner loop, we then iterate the comparison and the three assignments. So overall the number of comparisons is [(iteration of i)*(iteration of j)] so $f(n)=n^2*(n-1)/2$ this can be simplified and we can say that $f(n)= \theta(n^2)$.

3. Develop an efficient algorithm and implement it in Java (method maxMissedGain of class Assignment2).

See .java file.

4. Explain how many comparisons your algorithm makes in the worst case!

We consider the worst case to be when all elements are sorted in ascending order (maxMissedGain = 0). The for-loop is iterated n-2 times (from 1 to n-1) and within each iteration, two comparisons (if) are evaluated. This sums up into an overall running time f(n)=2(n-2) that we write as $f(n) = \theta(n)$.

5. Give an explanation why your algorithm is correct. Use the same approach as for the previous question:

- ***a. Does the algorithm maintain a partial solution? If so, for which problem exactly? Does it maintains additional information?***
- ***b. How is the output related to the information maintained during the execution? Why does this ensure the correctness of the answer? Θ(f(n)) for some function f(n) of the input size n.) Explain your answer.***

a) In this algorithm two values are necessary for the loop iteration to calculate the output. The <u>*maxGain*</u> represents a partial solution by representing the max missed gain in A[0 … i-1]:

- Initially: it' is initially set as A[0] - A[0] which represents the biggest possible gain in A[0 … 0] which is 0 (buy and sell at the same price)
- Maintenance: its value is updated if the value of the scanned element A[i] - minSoFar is created that its previous value
- Termination: when the loop terminates the whole array has been scanned so the partial solution now represents the final solution

The algorithm also has another loop invariant which is the *minSoFar*, it represents the smallest element in A[0 … i-1]:

- Initially: it initially holds the first element of the array (A[0]) so the only element scanned so far
- Maintenance: it's updated every time the scanned element (A[i]) is smallest than its previous value
- Termination: when the loop terminates the whole array has been scanned so the partial solution now represents the final solution

b) The output is maxGain and its value changes until it becomes the complete solution. Even if minSoFar is not returned in the output it is necessary to evaluate the correct maxGain.

4. Array of Averages

1. Develop an efficient algorithm for this task and implement it in Java

See .java file.

2. How many assignments will your algorithm perform for an input of size n?(It is enough to give a rough estimate in terms of Θ(f(n)) for some function f(n) of the input size n.) Explain your answer!

The algorithm includes two for-loops, the outer loop (i) is iterated n times (n-1 -> 0) and within each one, the algorithm does 2 assignment and then proceed with the inner loop; it is iterated n times (0 -> n-1) and for each one of them at least one at max 3 assignment are operated.

So in the *best case* (i>=j) the algorithm executes

$f(n) = (n*2)*(1*n) = 2*n^2$ assignment

while in the *worst case* (i<j) it iterates

$g(n)=(n*2)*(2*n) = 4*n^2 = 2*f(n)$ assignment

The two functions are within the same class of function so we can say that $\theta(f)= \theta(g)= \theta(n^2)$.