Assignment 2

Julien Corman, Tiziano Dalmonte
Werner Nutt

Recursion and Algorithms for Arrays

Instructions: The coursework consists of 4 questions. Each question requires a piece of Java code as an answer and an additional conceptual answer. The conceptual answers must be submitted in PDF format via OLE. The modalities for code submission are identical to those of Coursework 1, namely:

- All exercises must be implemented as static methods. Method stubs are available on Codeboard. Your implementation must respect the methods' signatures (but of course, you are free to create and call additional methods if needed).
- The code must be submitted via Codeboard, and a copy of it (.java file) via OLE.
- When writing your code, follow the guidelines for good coding style on OLE. In particular, add comments that clarify the intention behind variables and the purpose of loops.

The conceptual answers must be submitted as a PDF file via OLE.
Note that each question contributes separately to your final mark.

1. Recursion

A palindrome is a word or phrase that reads the same forwards and backwards (examples: 'racecar', 'radar', 'noon').
By extension, we call a palindrome any string that reads the same from left to right and from right to left. For this exercise, we require the forward and backward strings to be identical. So 'rats live on no evil star' is considered as a palindrome, but 'Racecar' or 'never odd or even' are not.
Develop a recursive algorithm that takes as input a string and decides whether the string is a palindrome.

1. Implement your algorithm in Java (as a method `isPalindrome` of the class `Assignment2`).
2. Explain why your algorithm is correct. To do so, try to answer the following questions:
   - In which way do you reduce the problem with each recursive call?
   - 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.

(Weight: 15% of this CW)

2. Maximal Length of Ascents in Arrays

Consider a non-empty array $A[1..n]$ of integers. The segment $A[k..l]$ (where $k \geq 1$ and $l \leq n$) is an *ascent* if $A[j] \leq A[j+1]$ for all j such that $k \leq j < l$. In other words, an ascent is a nondecreasing segment of A.
We want to compute the maximal length of an ascent in A. For instance, for the array $A = [3, 1, 6, 9, 4, 2, 4, 4, 5]$, the ascents that cannot be extended are $A[1..1] = [3]$, $A[2..4] = [1, 6, 9]$, $A[5..5] = [4]$, and $A[6..9] = [2, 4, 4, 5]$. They have length 1, 3, 1, and 4, respectively. The maximal length of an ascent in A is therefore 4.

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 (method `maxAscentLength` of class `Assignment2`).
2. Explain why your algorithm is correct:
   - Which information does the algorithm keep while it is running? Does this information represent a partial solution? Is there additional information it maintains?
   - How is the output related to the information maintained during the execution? Why does this ensure the correctness of the answer?
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 $\Theta(f(n))$ for some function $f(n)$ of n? Explain your answer!

(Weight: 20% of this CW)

3. Maximum Missed Gain

We consider a time series with the daily quotes of a stock on the stock exchange, stored in an array like the following one:

$$A = [3, 12, 4, 2, 13, 7, 16, 1, 7].$$

We are interested in the maximum gain one could possibly have made by buying the stock and later selling it.[1]
In this example the maximum missed gain would be 14, which one would have made when buying for 2 and selling for 16. Note that $15 = 16 - 1$ is not a missed gain, since one cannot buy a stock *after* having sold it.
We want to develop an algorithm that computes the maximum missed gain for a series of daily quotes. Formally, consider a non-empty array $A[1..n]$ of integers. For any two indices $1 \leq i \leq j \leq n$ let $g(i, j) := A[j] - A[i]$ be the gain that one makes when buying on day i and selling on day j.
The *maximum missed gain* in A is the maximum of all such $g(i, j)$, that is, the number

$$\max\{\, A[j] - A[i] \mid 1 \leq i \leq j \leq n \,\}.$$

Note that the maximum missed gain is always greater or equal to 0, since we could buy and immediately sell, without gaining or losing anything.

[1]Of course, it is now too late for the gain, since the time series reports the past. This is why this is a missed gain.

Your task is to develop a method with the signature

```
int maxMissedGain(int[] A).
```

1. Write pseudocode for a brute-force algorithm that immediately implements the definition.
2. How many comparisons would it make for an array of length n?[2] Explain your answer.

Next we are interested in an efficient algorithm.

Hint: Try to compute the maximum missed gain for the segment $A[1..(i+1)]$ if you know the maximum missed gain for $A[1..i]$. Can you compute it if you have some additional information about $A[1..i]$? Can you keep and update this information for each move from i to $i+1$?

3. Develop an efficient algorithm and implement it in Java (method `maxMissedGain` of class `Assignment2`).
4. Explain how many comparisons your algorithm makes in the worst case!
5. Give an explanation why your algorithm is correct. Use the same approach as for the previous question:
    - Does the algorithm maintain a partial solution? If so, for which problem exactly? Does it maintains additional information?
    - How is the output related to the information maintained during the execution? Why does this ensure the correctness of the answer?

(Weight: 30% of this CW)

4. Array of Averages

Design an *efficient* algorithm that accomplishes the following task. Given an array $A[1..n]$ of floating point numbers, return a two-dimensional array, say M, of size $n \times n$, in which the entry $M[i][j]$ for $i \leq j$ contains the *average* of the array entries $A[i]$ through $A[j]$ (both included). That is, if $i \leq j$, then

$$\begin{aligned} M[i][j] &= \frac{A[i] + \cdots + A[j]}{j-(i-1)} \\ &= \frac{\sum_{k=i}^{j} A[k]}{j-i+1}, \end{aligned}$$

whereas for $i > j$ we have that $M[i][j] = 0$.

[2]It is enough to give a rough estimate in terms of $\Theta(f(n))$ for some function $f(n)$ of the input size n.

For instance, for the array

$$A = [2, 3]$$

the output matrix (2 dimensional array) must be

$$M = \begin{pmatrix} 2 & 2.5 \\ 0 & 3 \end{pmatrix}$$

The explanation is the following:

$$\begin{aligned} M[1][1] &= A[1]/1 = 2 \\ M[1][2] &= (A[1] + A[2])/2 = 2.5 \\ M[2][1] &= 0 \text{ (because } 1 > 0) \\ M[2][2] &= A[2]/1 = 3 \end{aligned}$$

1. Develop an efficient algorithm for this task and implement it in Java (as a method `matrixOfAverages` of the class `Assignment2`).
2. How many assignments will your algorithm perform for an input of size n?[3] Explain your answer!

(Weight: 35% of this CW)

Deliverables. Submit two copies of your code:

- one via Codeboard (instructions are available here),
- one via OLE (together with the other deliverables).

The other questions must be answered in a PDF document.
Combine all deliverables into one zip file, which you submit via the OLE submission page for this assignment. Please include name, student ID and email address in your submission.

Submission until

Wednesday, 2 November 2022, 23:55,

to Codeboard and OLE.

[3]It is enough to give a rough estimate in terms of $\Theta(f(n))$ for some function $f(n)$ of the input size n.