CS608 - Fall 2020 - Assignment #1

Assigned: September 3rd, 2020
Due: September 12th, 2020
(Extra credit: December 1st, 2020)
NO LATE SUBMISSIONS.
Non-coding answers may be resubmitted once, after receiving feedback on the first attempt.

Collaboration policy: The goal of the assignment is to give you practice in mastering the course material. Consequently, you are encouraged to collaborate with others. In fact, students who form study groups generally do better on exams than do students who work alone. If you do work in a study group, however, you owe it to yourself and your group to be prepared for your study-group meeting. Specifically, you should spend at least 30–45 minutes trying to solve each problem beforehand. If your study group is unable to solve a problem, it is your responsibility to get help from the instructor before the assignment is due.

For this assignment, you can form a team of up to three members. Each team must write up each problem solution and/or code any programming assignment without external assistance, even if you collaborate with others outside your team for discussions. You are asked to identify your collaborators outside your team. **If you did not work with anyone outside your team, you must write "Collaborators: none."** If you obtain a solution through research (e.g., on the web), acknowledge your source, but write up the solution in your own words. **It is a violation of this policy to submit a problem solution that any member of the team cannot orally explain to the instructor.** No other student or team may use your solutions; this includes your writing, code, tests, documentation, etc. It is a violation of this policy to permit anyone other than the instructor and yourself read-access to the location where you keep your solutions.

Submission Guidelines: Your team has to submit your work on Blackboard (no email) by the due date. Only one submission per team is necessary. For each class in the programming assignments you must use the header template provided in Blackboard. Make sure that you identify your team members in the header, and any collaborators outside your team, if none, write "none". Your code must follow the code formatting standards as posted in Blackboard. Format will also be part of your grade. To complete the submission, you have to upload two files to Blackboard: your source file and your class file. Your answers to questions that do not require coding must be included in the remarks section of the header. **The submission will not be accepted in any other format.**

Style and Correctness: Keep in mind that your goal is to communicate. Full credit will be given only to the correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Assignment #1 Grading Rubric

Coding:

Program characteristic	Program feature	Credit possible		
Design 30%	Algorithm	30%		
Functionality 30%	Program runs without errors	20%		
	Correct result given	10%		
Input 15%	User friendly, typos, spacing	10%		
	Values read in correctly	5%		
Output 15%	Output provided	10%		
	Proper spelling, spacing, user friendly	5%		
Format 10%	Comments, name	5%		
	Indentation	5%		
	TOTAL	100%		

Non-coding:
Embedded in questions.

1(60)	2(20)	3(20)	**TOTAL**(100)	EC

Assignment:

The Maximum Subarray Problem is the task of finding the contiguous subarray, within an array of numbers, that has the largest sum. For example, for the sequence of values $(-2, 1, -3, 4, -1, 2, 1, -5, 4)$ the contiguous subsequence with the largest sum is $(4, -1, 2, 1)$, with sum 6.

For an arbitrary input array of length n, two algorithms that compute the sum of the maximum subarray were discussed in class: (a) a brute-force algorithm that solves the problem in $O(n^2)$ steps, and (b) a divide-and-conquer algorithm that achieves $O(n \log n)$ running time.

1. (60 points) Implement in Java (or the language agreed with the instructor) the algorithms attached below as Algorithms 1, and 2. Your program must prompt the user to enter the size of the vector n, and output the time taken by each of the three algorithms. To measure the running time you can use the snippet of code attached below. Choose at random the numbers in the array (including the sign).

2. (20 points) Test the algorithms with different values of n and fill the following table with the running times measured (put the table in the code header).

	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^7$
Brute force					
Divide and conquer					

   You may run into problems, such as running out of memory or the program taking too much time. If that is the case, adjust the values of n accordingly.

3. (20 points) Based on the running times observed, draw conclusions about the running times obtained in the analysis. Do they match or not? It is not enough to simply say: yes, they match. You have to justify your claim based on the running times measured (the table). Provide your answers in the remarks section of the code header.

4. (*Extra credit*) There exists a dynamic-programming algorithm due to Kadane that runs in linear time, which is optimal because you need at least to read each number in the input. For extra credit, implement this dynamic programming algorithm as well and test it along the other three.

Example code to measure time:

```
// store the time now
long startTime = System.nanoTime();

// here goes the fragment of code
// whose execution time you want to measure

// display the time elapsed
System.out.println("t= "+(System.nanoTime() - startTime)+" nanosecs.");
```

Algorithm 1: Brute force: $O(n^2)$.

input : a vector A of n numbers.
output: maximum subarray sum.

$S_{\max} \leftarrow -\infty$
for $i = 1$ **to** n **do**
  $sum \leftarrow 0$
  for $j = i$ **to** n **do**
    $sum \leftarrow sum + A[j]$
    $S_{\max} \leftarrow \max\{S_{\max}, sum\}$
  end
end
return $S_{\max}$

Algorithm 2: Divide and conquer: $O(n \log n)$.

input : a vector A of n numbers.
output: maximum subarray sum.

```
MAX-SUBARRAY(A,low,high)
    if high = low then
        return A[low]
    else
        mid ← ⌊(low + high)/2⌋
        leftsum ← MAX-SUBARRAY(A,low,mid)
        rightsum ← MAX-SUBARRAY(A,mid + 1,high)
        crosssum ← MAX-CROSSING-SUBARRAY(A,low,mid,high)
        return max{leftsum, crosssum, rightsum}
    end
end

MAX-CROSSING-SUBARRAY(A,low,mid,high)
    leftsum ← −∞
    sum ← 0
    for i = mid to low do
        sum ← sum + A[i]
        if sum > leftsum then
            leftsum ← sum
        end
    end
    rightsum ← −∞
    sum ← 0
    for j = mid + 1 to high do
        sum ← sum + A[j]
        if sum > rightsum then
            rightsum ← sum
        end
    end
    return leftsum + rightsum
end
```