COMP 3270 PROGRAMMING HOMEWORK DUE BY 11:59 pm Friday July 21, 2023
Late submissions will not be accepted even if you have a proper excuse because you have 32 days
Not submitting this HW will result in the F grade

Empirical analysis of algorithms involves implementing, running and then analyzing the run-time data collected against theoretical predictions. This homework asks you to implement, and theoretically and empirically compute the complexities of algorithms with **different orders of complexity** to solve the problem of Maximum Sum Contiguous Subvector defined as follows:

Maximum Sum Contiguous Subvector (MSCS) Problem: Compute the sum of the subsequence of numbers (a subsequence is a consecutive set of one or more numbers) in an array of numbers (for this HW, we will use integers) that sum to the largest value possible; but if this value is negative, MSCS is defined to be zero. E.g., in an array of all positive (or all negative) numbers, the Maximum Contiguous Subvector consists of all numbers in the array (or zero). There may be multiple Maximum Subsequences in an array, all of which sum to the same largest sum. E.g., in an array of all zeroes, MSCS =0 and any 0 in the array is a Maximum Subsequence. Remember that the input array may contain zeroes, negative and/or positive integers. For example, if the input is [1,2,-4,3,-5,2,0] then MSCS =3 and there are two Maximum Subsequences that sum to the same MSCS: A[1]+A[2]=1+2=3 or A[4]=3.

Requirements:

1. **Implement** the provided algorithms of different complexity orders to solve this problem. The implementation should be faithful to the algorithms provided in this homework; **use the same variable names**; if you turn in code that implements different algorithms, perhaps you find on the web, then you will get a zero.
2. Calculate **T(n) and the order of complexity** of each algorithm in the big-Oh or Theta notation using any of the methods we discussed in class. (a) Show your calculations in the provided tables, then (b) determine and state the polynomial T(n) and (c) the complexity order of the algorithm. Incomplete answers to this part will get zero, not a partial grade.
3. You must program in C, Java or C++. With prior permission, other languages may be acceptable – talk to the TA and obtain his permission.
4. Write a main program that carries out the following tasks, one after the other:
5. First, the main program should read from a file named "**phw_input.txt**" containing 10 comma-delimited integers in the first line**,** create an array containing these 10 integers, and run each of the algorithms on that input array, and print out the answer produced by **each** on the **console** as follows**:** "algorithm-1: <answer>; algorithm-2:<answer>; algorithm-3:<answer>; algorithm-4:<answer> where <answer> is the MSCS as determined by each of the algorithms.
6. Next, create 19 integer sequences of length 10,15,20,25,……90,95,100, containing **randomly generated negative, zero and positive integers**, and store these in 19 arrays of size 10,15,…,95,100: A_1-A_{19}.
7. Then use the system clock to measure time t1, run one of the four algorithms on array A_i (starting with i=1) N times (where N is at least 100, but if your system clock does not have a good resolution you may need N to be larger, like 500 or 1000 in order to get meaningful running times), then measure time t2, and compute average time needed by that algorithm to solve the problem with input size = size of A_i. Do this for **each** of the algorithms executing on **each** of the 19 input arrays to fill the first four columns of a 19X8 matrix of integers with average execution times. Each row of this matrix corresponds to one input size, from 10-100.
8. Fill the last four columns of this matrix with values ceiling($T_1(n)$), ceiling($T_2(n)$), ceiling($T_3(n)$), and ceiling($T_4(n)$) where n = each input size and T(n) are the polynomials representing the theoretically calculated complexity of the three algorithms that you determined in step 2 part (b). So, column 1 will have measured running times of your first algorithm and column 5 will have the calculated complexity for the same algorithm; similarly for columns 2 & 6, 3 & 7, and 4 & 8. You may need to scale the complexity values (or use an appropriate time unit such as nano/micro/milli seconds for the measured running times) in order to bring all data into similar ranges.
9. Your main program should write one text line of algorithm and complexity order titles separated by commas (e.g., "algorithm-1,algorithm-2,algorithm-3,algorithm-4,$T_1(n)$,$T_2(n)$,$T_3(n)$, $T_4(n)$"), followed by the above matrix also in comma-delimited format (19 lines of 8 integers separated by commas) to a file called "**yourname_phw_output.txt**".
10. Open **yourname_phw_output.txt** with a spreadsheet and produce a labeled graph with 10-100 on the x-axis and 8 curves showing the actual time taken and predicted time (the complexity order) for each algorithm. Label the curves appropriately.

SUBMISSION INSTRUCTIONS

If you have any doubts/questions, ask the instructor or the TA before you code. Our expectation is that you know how to program by now. So, the instructor or TA **will not debug your source code** for you. We can help you with logic and algorithms.

Your submission **must** include:

1. Source code that is properly commented (if you reuse code fragments from another source such as the text, web site, etc., clearly identify the fragments and their source in comments) including all procedures, functions, classes, etc. and the main program - in short **everything we need to compile and run it**.
2. A doc or pdf file with:
    a. Details of the complexity order calculation as shown by filled tables from p.4-5 of this homework.
    b. The graph and an explanation of what it shows: Did the actual time taken match predicted complexity of each algorithm as the input size increased from 10 to 100? Why or why not?

Submit **via Canvas**, following the instructions below (you may loose points otherwise), all required items before the deadline on the due date. Late submissions will not be graded. Do not send any executables.

- Include all files as a single zipped attachment.
- Include in your submission a README file that should contain your **name**, the **names of all files** in the zipped archive, an **explanation** of how you compiled your program (what IDE or command line compilation command for the compiler you used), **and** the following **certification** statement (**verbatim**): "I certify that I wrote the code I am submitting. I did not copy whole or parts of it from another student or have another person write the code for me. Any code I am reusing in my program is clearly marked as such with its source clearly identified in comments." **Without this certification, your grade will be zero**.

If the TA has trouble with your attachments you will get an email. It is up to you to check your Canvas course account and sort out any such problem cooperatively with the TA. If this is not done in a timely manner, your homework will not be graded.

Caution: Your work should follow the academic integrity guidelines stated in the syllabus. Do your own work; do not copy that of another. If we suspect copying, we will compare suspected submissions using plagiarism detection tools. If we find evidence of copying, besides giving you a zero in the assignment, we will refer your case to appropriate authorities and your certification will be used in the proceedings.

GRADING POLICY

We will test your program with **our own** phw_input.txt file and generate a graph from the yourname_phw_output.txt that your program produces when we run it. Your grade will depend on how your program works when we test it. We will not try to debug your program; it is your responsibility to send a correct program that will compile, run, will not crash, and produce the correct output. Programs that are not well-organized, not commented properly, and not written following the directions are likely to lose points. If you make any additional assumptions about the input, beyond the ones stated in this document, your program may not run correctly during our test.

The homework will be graded out of a maximum of 100 points. The minimum requirement is to turn in everything asked for. If you do not meet this requirement, you will get 0 points. If you do, your grade will be determined as follows:

8 points for correct complexity order calculation of each algorithm: total 32 points max.
10 points for the graph and 8 points for its explanation: total 18 points max.
If your source does not compile, you will get 0 points out of the remaining 50 points.
If the source compiles correctly but the executable does not run or crashes, you will get 5 points.
If the executable runs, but produces no output file or completely wrong output, you will get 10 points.
If the output is partially correct, you will get a partial grade higher than 10.
If the output is fully correct, you will get an additional 50 points.

Algorithm-1(X : array[P..Q] of integer)

```
1       maxSoFar = 0
2       for L = P to Q
3           for U = L to Q
4               sum =0
5               for I = L to U
6                   sum = sum + X[I]
                    /* sum now contains the sum of X[L..U]  */
7               maxSoFar = max (maxSoFar, sum)
8       return maxSoFar
```

Algorithm-2(X : array[P..Q] of integer)

```
1       maxSoFar = 0
2       for L = P to Q
3           sum =0
4           for U = L to Q
5               sum = sum + X[U]
                /* sum now contains the sum of X[L..U] */
6               maxSoFar = max(maxSoFar,sum)
7        return maxSoFar
```

Algorithm-3

recursive function MaxSum(X[L..U]: array of integer, L, U: integer)

```
1       if L > U then
            return 0   /* zero- element vector */
2       if L=U then
            return max(0,X[L])  /* one-element vector */
3       M = (L+U)/2  /* A is X[L..M], B is X[M+1..U] */
        /* Find max crossing to left */
4               sum = 0; maxToLeft =0
5               for I = M downto L do
6                   sum =sum+X[I]
7                   maxToLeft = max(maxToLeft,sum)
        /* Find max crossing to right */
8               sum=0; maxToRight=0
9               for I = M+1 to U
10                  sum=sum+X[I]
11                  maxToRight = max(maxToRight, sum)
12      maxCrossing = maxToLeft + maxToRight

13      maxInA = maxSum(X, L, M)
14      maxInB = maxSum(X,M+1,U)
15      return max(maxCrossing, maxInA, maxInB)
```

Algorithm-4(X : array[P..Q] of integer)

```
1       maxSoFar = 0
2       maxEndingHere= 0
3       for I = P to Q
4           maxEndingHere = max(0, maxEndingHere + X[I])
5           maxSoFar = max(maxSoFar, maxEndingHere)
6       return maxSoFar
```

Algorithm-1

Step	Cost of each execution	Total # of times executed
1		
2		
3		
4		
5		
6		
7		
8		

Multiply col.1 with col.2, add across rows and simplify
$T_1(n)$ =

Algorithm-2

Step	Cost of each execution	Total # of times executed
1		
2		
3		
4		
5		
6		
7		

Multiply col.1 with col.2, add across rows and simplify
$T_2(n)$ =

Algorithm-3

Step	Cost of each execution	Total # of times executed in any single recursive call
1		
2		
Steps executed when the input is a base case:		
First recurrence relation: T(n=1 or n=0) =		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		(cost excluding the recursive call)
14		(cost excluding the recursive call)
15		
Steps executed when input is NOT a base case:		
Second recurrence relation: T(n>1) =		
Simplified second recurrence relation (ignore the constant term): T(n>1) =		

Solve the two recurrence relations using any method (recommended method is the Recursion Tree). Show your work below:

$T_3(n)$ =

Algorithm-4

Step	Cost of each execution	Total # of times executed
1		
2		
3		
4		
5		
6		

Multiply col.1 with col.2, add across rows and simplify

$T_4(n)$ =