CSC211—Problem Solving, Algorithms and Programming
Growth functions—compare the growth of various solutions of the maximum contiguous subvector problem.
Term 1 Practical 2 16/18 February 2026

Submit today before: 17h20, 16/18 February 2026

Please follow the instructions carefully. You will receive marks for everything done correctly that is submitted before you leave the practical *today*.

---

1. Remember that you are using **git** that should already have been initialized last week using **git init** inside your name file that was created to hold all your pracs. The practical for this week must be done inside the directory **12practical** which must lie inside your name directory where you initialised **git** and parallel to last week's **11practical** directory. Your final submission from a Windows system is always a zipped file of the entire name directory, *or* if you use Linux a tarball created from the directory containing your name directory using **tar –zcvf surname,firstname.tgz ∗**. Do a
git commit –m "Explanation of additions" regularly, i.e., at least every thirty minutes or after adding about ten to twenty lines of code. You will be penalized for not doing "enough" commits.

2. The next practical will be done under the directory **13practical**, and the one in the fourth week under **14practical**, etc.

3. Your work in the form of a **.zip** or **.tgz** file must be uploaded to iKhamva

4. We will mark your work as soon as possible when the demis start working, or when we get the automatic marking software going, and give you feedback. We are still attempting to get the automatic marking software in place. When this works you will get feedback in real time, despite always uploading completed work to iKamva *today* before leavin the laboratory..

5. **Overview** the jist of today's practical is to collect and plot useful data for the running times of four of algorithms for the *maximum contiguous subsequnce* problem coded in java, that have assymptotic complexities of $O(n^3)$, two that are $O(n^2)$, and $O(n)$. We will leave algorithms for $O(n \log n)$ and $O(k^n)$ for another practical. What the code does is to determine the sum of a contiguous string of numbers that yields the highest value. Note that there are always some negegative numbers in **X**.

   The python code for these algorithms is given here.

```
1  from random import random, randint
2
3  def mcsOn3(X):
4      n = len(X)
5      maxsofar = 0
6      for low in range(n): # [0,n)
7          for high in range(low, n): # [low,n)
8              sum = 0
9              for r in range(low, high): # [low,high)
10                 sum += X[r]
11                 if (sum > maxsofar):
```

```
                maxsofar = sum
    return maxsofar
```

```
def mcsOn2A(X):
    n = len(X)
    maxsofar = 0
    for low in range(n): # [0,n)
        sum = 0
        for r in range(low, n): # [low,n)
            sum += X[r]
            if (sum > maxsofar):
                maxsofar = sum
    return maxsofar
```

```
def mcsOn2B(X):
    n = len(X)
    sumTo = [0]*(n+1)
    for i in range(n): # [0,n)
        sumTo[i] = sumTo[i-1] + X[i]
    maxsofar = 0
    for low in range(n): # [0,n)
        for high in range(low, n): # [low,n)
            sum = sumTo[high] - sumTo[low-1]
            # sum of all elements in X[low..high)
            if (sum > maxsofar):
                maxsofar = sum
    return maxsofar
```

```
def mcsOn(X):
    N = len(X)
    maxSoFar = 0.0
    maxToHere = 0.0
    for i in range(N): # [1,N)
        maxToHere = max(maxToHere +X[i], 0.0)
        maxSoFar = max(maxSoFar, maxToHere)
    return maxSoFar
```

The following code creates an array **X** with one third of the numbers negative.

```
from random import rantint
def main():
        n = 20
        X = [randint(1,n)*(-1)**randint(2,4) for k in range(n)]
        countP = 0
        countM = 0
        for x in X:
                if x < 0:
                        countM +=1
                else:
                        countP +=1
        print(f"{countM = }  {countP = }")
```

You are required to write these methods on Java in a single class and do some timing experiments.

6. Time your java code by inserting appropriate **count** variables in your code that will measure the number of times the core of the code is repeated. Your counts should reflect the cube of n, two different squares of n and a linear n.

7. Your code should be run on various lengths n of the array **X** which contains the data for which the MCS is calculated. The n^3 and n^2 algorithms are quite simple and easy to understand. The $O(n)$ may need more reflection, and is explained in the class notes.

8. The output for this practical must produce is a table of the counts laid out as follows.

n	$O(n^3)$	$O(n^2)$	$O(n^2)$	$O(n)$
10^2	...	...	...	...
10^3	...	...	...	...
10^4	...	...	...	...
10^5	...	...	...	...
10^6	...	...	...	...