# Group Programming Assignment 1

Prof. Xiaoli Fern

Due: Thursday, January 14 at 2PM on Canvas

You are required to work in groups of two or three students. To formally form your group on canvas, I have created empty groups for this assignment that you can join. Coordinate with your team member so that you will join the same group. Please see `http://guides.instructure.com/m/4212/l/64913-how-do-i-join-a-group-as-a-student` for instruction on how to join a group.

Each group will need to submit one copy of the work to Canvas. This should include the **project report as a pdf** and the code that implements the algorithms. The report should have **all member's names included**. Your implementation can be in Java, Python, C/C++, Matlab or R. Any language outside this list will need advance approval from the instructor. See the course syllabus for instructions/policies on using old work for students retaking CS325.

For this project, you will design, implement and analyze (both experimentally and mathematically) *three* algorithms for the maximum subarray problem:

> Given array of small integers $a[1, \ldots, n]$ (that contains at least one positive integer), compute
>
> $$\max_{i \le j} \sum_{k=i}^{j} a[k]$$
>
> For example, MaxSubarray$([31, -41, \mathbf{59}, \mathbf{26}, \mathbf{-53}, \mathbf{58}, \mathbf{97}, -93, -23, 84]) = 187$

## Description of the algorithms

Your three algorithms are to be based on these ideas:

**Algorithm 1: Enumeration** Loop over each pair of indices $i \le j$ and compute the sum, $\sum_{k=i}^{j} a[k]$. Keep the best sum you have found so far.

**Algorithm 2: Better Enumeration** Notice that in the previous algorithm, the same sum is computed many times. In particular, notice that $\sum_{k=i}^{j} a[k]$ can be computed from $\sum_{k=i}^{j-1} a[k]$ in $O(1)$ time, rather than starting from scratch. Write a new version of the first algorithm that takes advantage of this observation.

**Algorithm 3: Dynamic Programming** Your dynamic programming algorithm should be based on the following idea:

- The maximum subarray either uses the last element in the input array, or it doesn't.

Describe the solution to the maximum subarray problem recursively and mathematically based on the above idea.

See pages 8-13 of `https://web.engr.oregonstate.edu/~glencora/wiki/uploads/max-subarray.pdf` for more details on these algorithms.

**Testing for correctness** You can test the correctness of your implementations using the test sets provided on canvas. The file has one test case per line (10 cases each with 100 entries). A line corresponding to the example above would be:

```
[31, -41, 59, 26, -53, 58, 97, -93, -23, 84], 187
```

with the input array followed by the sum of the maximum subarray.

## Project report

Your typeset report **must** include:

**Pseudocode** Give **pseudocode** for each of the algorithms. Your pseudocode should make clear how many times your algorithm will

- add *two* numbers together
- take the max of *two* numbers

Note that the pseudocode in the above-linked pdf does not do this.

**Run-time analysis** For each algorithm, express the number of + and max (between two numbers) operations that it makes for an input array of $n$ numbers *as a sum* (e.g. $\sum_{i=1}^{n}\sum_{j=i}^{n-1} i^2$). Give asymptotic bounds for each. (That is, you should give three sums and three asymptotic bounds, one for each algorithm.)

**Experimental run-time analysis** For the experimental analysis you will plot running times as a function of input size. Every programming language should provide access to a clock (not necessarily in seconds). Run each of your algorithms on input arrays of size $100, 200, 300, \ldots, 900$ and $1000, 2000, 3000, \ldots, 9000$ (that is, you should have 18 data points for each algorithm). The first enumerative algorithm may be frustratingly slow, so you may compute running times for sizes $100, 200, 300, \ldots, 900$.

To do this, generate random instances using a random number generator as provided by your programming language. Remember to include both positive and negative numbers! For each data point, you should take the average of a small number (say, 10) runs to smooth out any noise. For example, for the first data point, you will do something like:

```
for i = 1:10
    A = random array with 100 entries
    start clock
    maxsubarray(A)
    pause clock
return elapsed time divided by 10
```

Note that you should not include the time to generate the instance.

Plot the running times as a function of input size for each algorithm in a *single plot*. Label your plot (axes, title, etc.). Include an *additional plot* of the running times on a log-log axis. See here for a tutorial on log-log plots: `http://www.eecs.orst.edu/~glencora/cs325/videos/loglogplots.mp4` (and accompanying file: `https://web.engr.oregonstate.edu/~glencora/wiki/uploads/loglogplots.m`).

Note that if the slope of a line in a log-log plot is $m$, then the line is of the form $\Theta(x^m)$ on a linear plot. Determine the slope of the lines in your log-log plot and from these slopes, infer the experimental running time for each algorithm. Discuss any discrepancies between the experimental and theoretical running times.