Project 1: Performance Measurement

In your textbook, several methods for solving the *Maximum Subsequence Sum* problem are discussed in Section 2.4.3. Now let us extend this problem to 2-dimensional.

MAXIMUM SUBMATRIX SUM PROBLEM:

Given an N×N integer matrix $(a_{ij})_{N\times N}$ *, find the maximum value of* $\sum_{k=i}^{m}\sum_{l=j}^{n} a_{kl}$ *for all* $1 \le i \le m \le N$ *and* $1 \le j \le n \le N$. *For convenience, the maximum submatrix sum is 0 if all the integers are negative.*

Example: For matrix $\begin{bmatrix} 0 & -2 & -7 & 0 \\ 9 & 2 & -6 & 2 \\ -4 & 1 & -4 & 1 \\ -1 & 8 & 0 & -2 \end{bmatrix}$, *the maximum submatrix is* $\begin{bmatrix} 9 & 2 \\ -4 & 1 \\ -1 & 8 \end{bmatrix}$ *and has the sum of 15.*

The simplest method is to compute every possible submatrix sum and find the maximum number. An algorithm similar to *Algorithm 1* given in Section 2.4.3 will run in $O(N^6)$, and the one similar to *Algorithm 2* in $O(N^4)$.

Your tasks are:

(1) Implement the $O(N^6)$ and the $O(N^4)$ versions of algorithms for finding the maximum submatrix sum;

(2) Analyze the time and space complexities of the above two versions of algorithms;

(3) Measure and compare the performances of the above two functions for N = 5, 10, 30, 50, 80, 100.

(4) ***Bonus:*** Give a better algorithm. Analyze and prove that your algorithm is indeed *better* than the above two simple algorithms.

To measure the performance of a function, we may use C's standard library **time.h** as the following:

```
#include <time.h>
clock_t   start, stop; /* clock_t is a built-in type for processor time (ticks) */
double    duration;    /* records the run time (seconds) of a function */

int main ( )
{     … …
      /* clock() returns the amount of processor time (ticks) that has elapsed
         since the program began running */
      start = clock();    /* records the ticks at the beginning of the function call */
      function();         /* run your function here */
      stop = clock();     /* records the ticks at the end of the function call */
      duration = ((double)(stop - start))/CLK_TCK;
      /* CLK_TCK is a built-in constant = ticks per second */
      … …
      return 1;
}
```

Note: If a function runs so quickly that it takes less than a tick to finish, we may repeat the function calls for K times to obtain a total run time, and then divide the total time by K to obtain a more accurate duration for a single run of the function. The repetition factor must be large enough so that the number of elapsed ticks is at least 10 if we want an accuracy of at least 10%.

The test results must be listed in the following table:

	N	5	10	30	50	80	100
$O(N^6)$ version	Iterations (K)						
	Ticks						
	Total Time (sec)						
	Duration (sec)						
$O(N^4)$ version	Iterations (K)						
	Ticks						
	Total Time (sec)						
	Duration (sec)						

The performances of the two functions must be **plotted** in the **same** N–run_time coordinate system for illustration.

Grading Policy:

This assignment is due Monday, September 28th, 2015 at 10:00pm.

- **Programmer:** Implement the two functions **(30 pts.)** and a testing program **(20 pts.) with sufficient comments**.
- **Tester:** Decide the iteration number *K* for each test case and fill in the table of results **(8 pts.)**. Plot the run times of the functions **(12 pts.)**. Write analysis and comments **(10 pts.)**.
- **Report Writer:** Write Chapter 1 **(6 pts.)**, Chapter 2 **(12 pts.)**, and finally a complete report **(2 pts. for overall style of documentation)**.

Note: Anyone who does excellent job on solving the Bonus problem will gain an extra 5% of his/her points.