CS 293 Lab 2

31st July, 2017

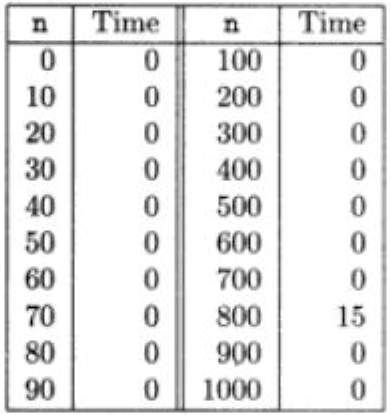
Run Time of a Program

- Asymptotic complexity
- Memory access time
- CPU speed
- Function calls/ Recursion
- Programming language
- Parallelism

Asymptotic Complexity (Insertion Sort)

```
int main()
{
   int a[1000], step = 10;
   double clocksPerMillis = double(CLOCKS_PER_SEC) / 1000;
                        // clock ticks per millisecond

   cout << "The worst-case time, in milliseconds, are" << endl;
   cout << "n \t Time" << endl;

   // times for n = 0, 10, 20, ..., 100, 200, 300, ..., 1000
   for (int n = 0; n <= 1000; n += step)
   {
      // initialize with worst-case data
      for (int i = 0; i < n; i++)
         a[i] = n - i;

      clock_t startTime = clock( );
      insertionSort(a, n);
      double elapsedMillis = (clock( ) - startTime) / clocksPerMillis;

      cout << n << '\t' << elapsedMillis << endl;

      if (n == 100) step = 100;
   }
   return 0;
}
```

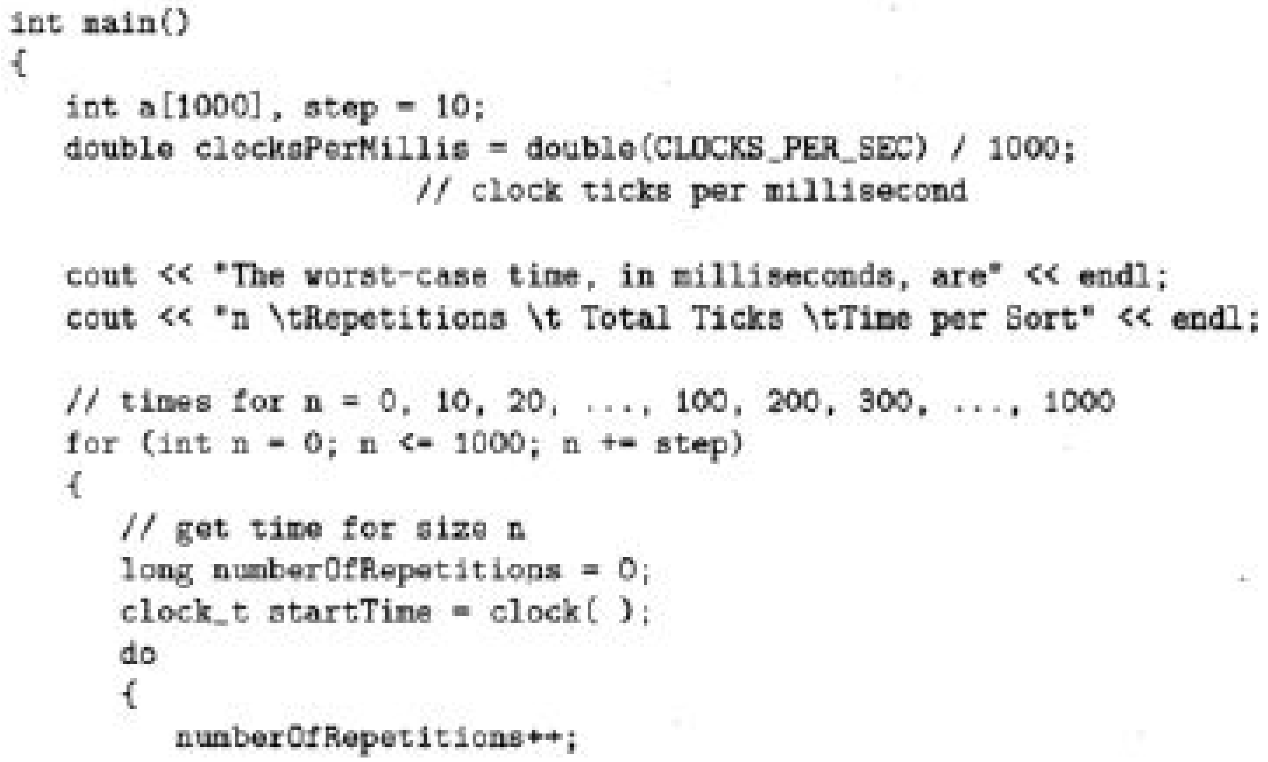
Running Time



n	Time	n	Time
0	0	100	0
10	0	200	0
20	0	300	0
30	0	400	0
40	0	500	0
50	0	600	0
60	0	700	0
70	0	800	15
80	0	900	0
90	0	1000	0

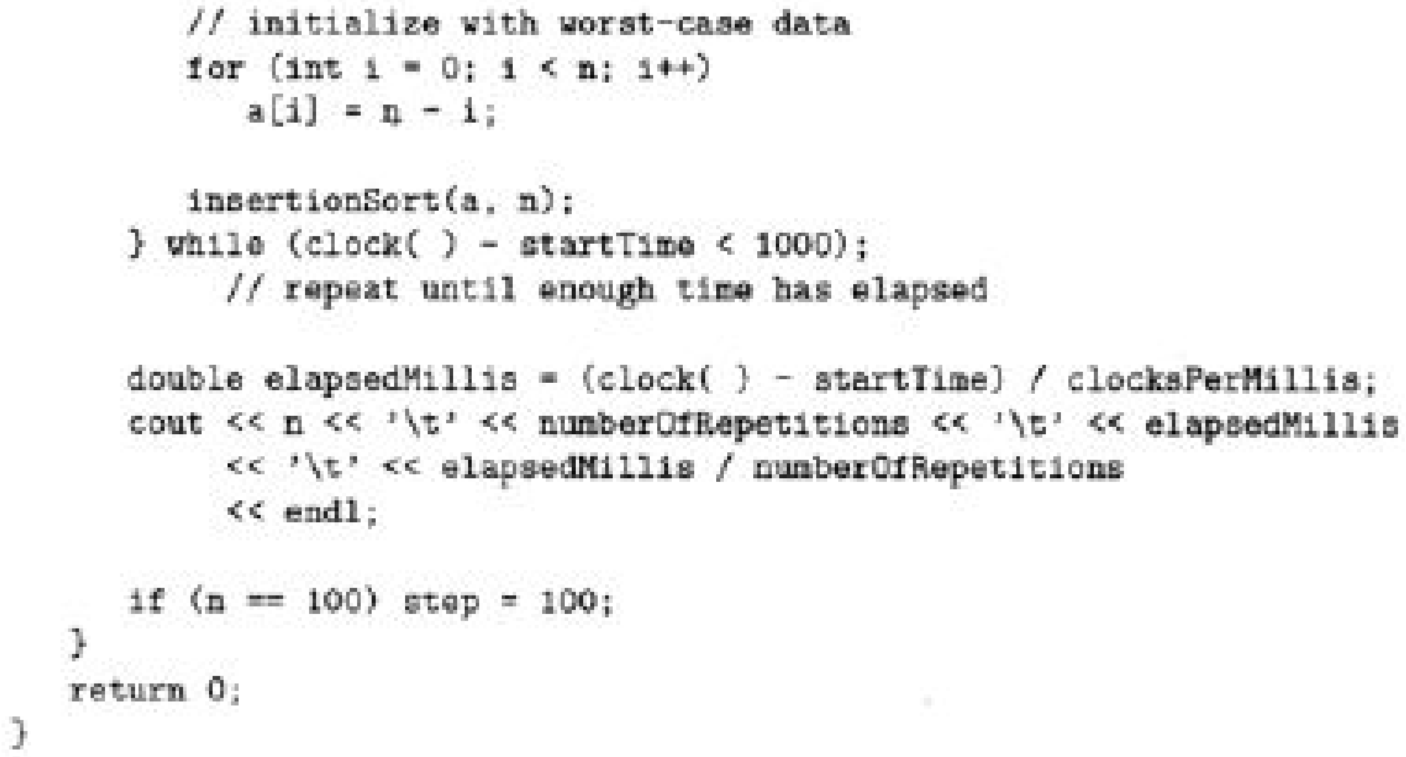
Times are in milliseconds

Modified Program



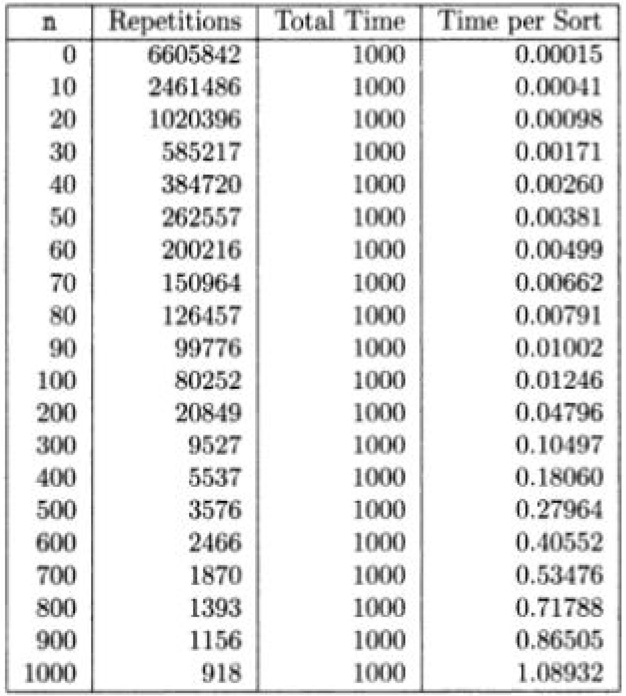
```
int main()
{
   int a[1000], step = 10;
   double clocksPerMillis = double(CLOCKS_PER_SEC) / 1000;
                            // clock ticks per millisecond

   cout << "The worst-case time, in milliseconds, are" << endl;
   cout << "n \tRepetitions \t Total Ticks \tTime per Sort" << endl;

   // times for n = 0, 10, 20, ..., 100, 200, 300, ..., 1000
   for (int n = 0; n <= 1000; n += step)
   {
      // get time for size n
      long numberOfRepetitions = 0;
      clock_t startTime = clock( );
      do
      {
         numberOfRepetitions++;
```

Modified Program



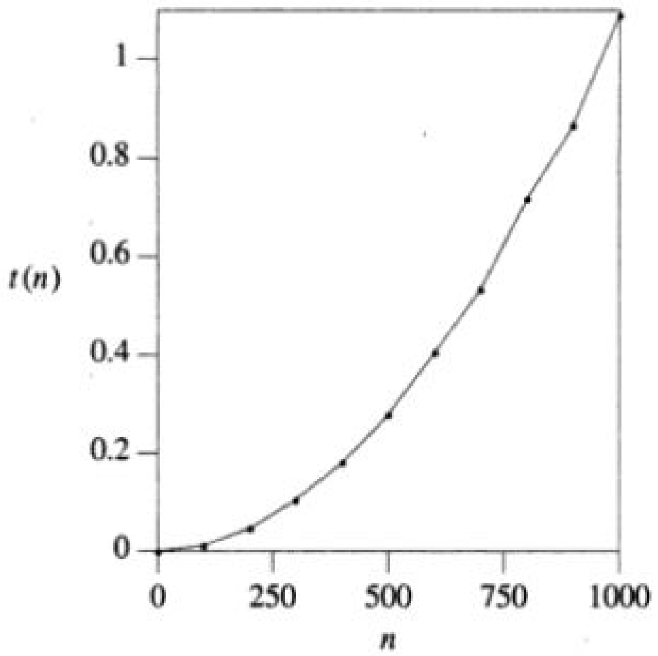
```
         // initialize with worst-case data
         for (int i = 0; i < n; i++)
            a[i] = n - i;

         insertionSort(a, n);
      } while (clock( ) - startTime < 1000);
           // repeat until enough time has elapsed

      double elapsedMillis = (clock( ) - startTime) / clocksPerMillis;
      cout << n << '\t' << numberOfRepetitions << '\t' << elapsedMillis
           << '\t' << elapsedMillis / numberOfRepetitions
           << endl;

      if (n == 100) step = 100;
   }
   return 0;
}
```

Running Times (in ms)



n	Repetitions	Total Time	Time per Sort
0	6605842	1000	0.00015
10	2461486	1000	0.00041
20	1020396	1000	0.00098
30	585217	1000	0.00171
40	384720	1000	0.00260
50	262557	1000	0.00381
60	200216	1000	0.00499
70	150964	1000	0.00662
80	126457	1000	0.00791
90	99776	1000	0.01002
100	80252	1000	0.01246
200	20849	1000	0.04796
300	9527	1000	0.10497
400	5537	1000	0.18060
500	3576	1000	0.27964
600	2466	1000	0.40552
700	1870	1000	0.53476
800	1393	1000	0.71788
900	1156	1000	0.86505
1000	918	1000	1.08932

Running Time Plot



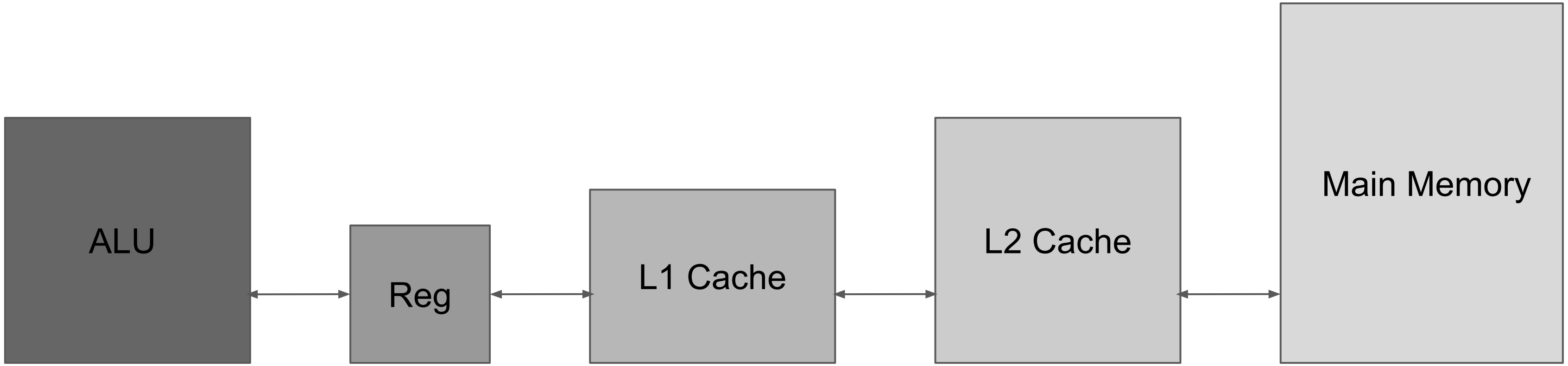
Problem 1: Insertion Sort

Do the following steps for n = 10, 20, 30, 40, 50, 60,70, 80, 90, 100, 200, 300, 400, 500, 500, 600, 700, 800, 900, 1000.

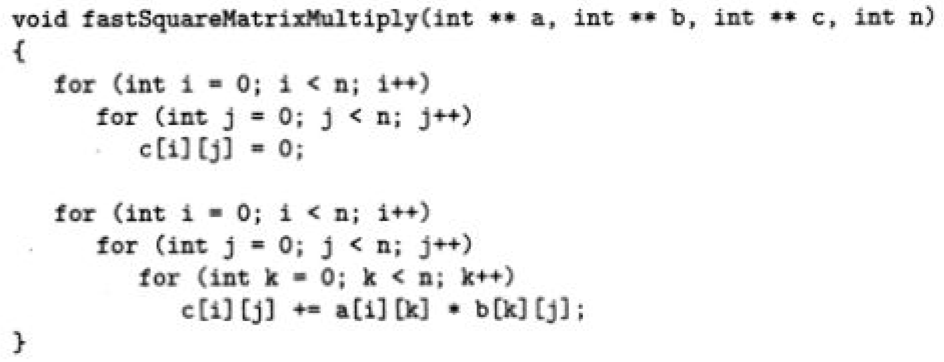
1. Using appropriate permutation of numbers from 0 to n-1 (both inclusive), find the best and worst case run time of insertion sort.
2. For average run time of insertion sort:
    a. Sort the random permutation of the numbers 1 to n on each iteration of the while loop.
    b. Set the while loop so that at least 20 random permutations are sorted.
    c. Estimate the average sort time by dividing the elapsed time by the number of permutations sorted.

Create table entry for each value of n and its best, average and worst case run time.

Memory Access



Matrix Multiplication



```
void fastSquareMatrixMultiply(int ** a, int ** b, int ** c, int n)
{
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
         c[i][j] = 0;

   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
         for (int k = 0; k < n; k++)
            c[i][j] += a[i][k] * b[k][j];
}
```

Running Times (in ms)

n	**ijk order**	**ikj order**
500	2.6	0.9
1000	26.5	6.7
2000	844.6	54.2

Program 2: Matrix Multiplication

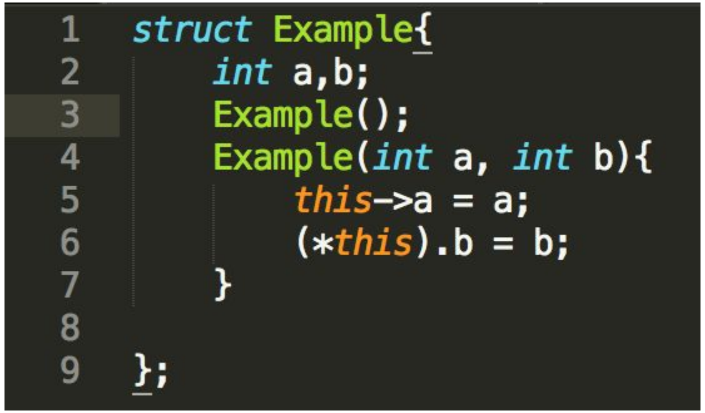
Write a program for multiplying two matrices. Now change the order of the for loops in the program. Compare the execution time of the two programs for the given inputs. Repeat the same for matrix addition.

Pointers

Basics:

- Variables that store addresses
- What we accomplish using reference parameters can also be done using pointers.

- Example: int *i
    - ‘i’ stores the address of an integer variable
    - *i - To get the integer stored at address i
    - int m; i = &m; --- & used to get the address corresponding to integer ‘m’
- *this* pointer and use of operator *->:*



```
struct Example{
    int a,b;
    Example();
    Example(int a, int b){
        this->a = a;
        (*this).b = b;
    }

};
```

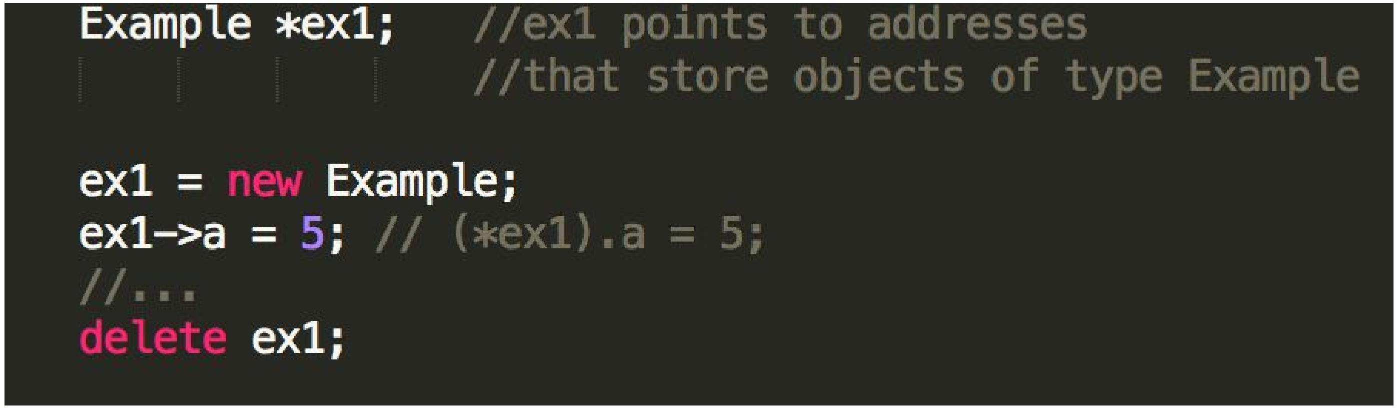
Dynamic Memory Allocation

- See reference 1 in problem statement for help

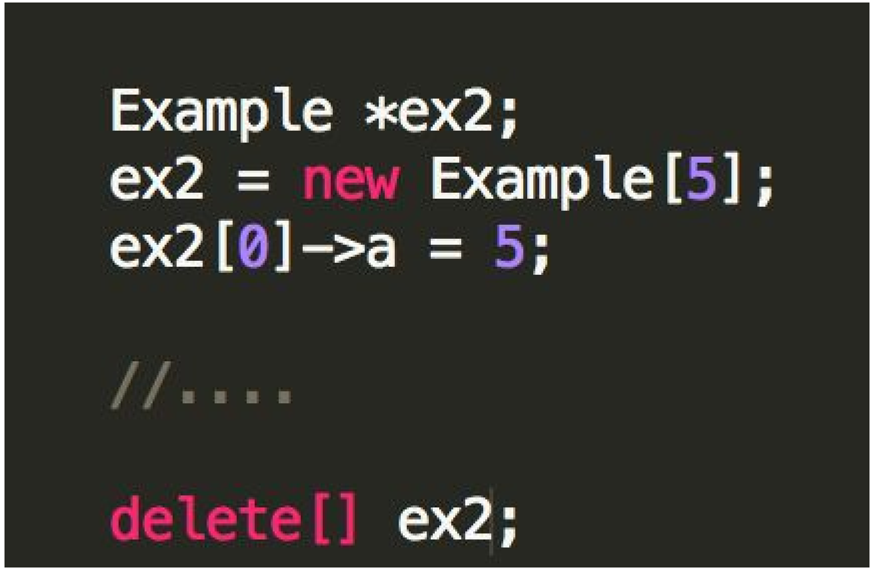
Two types of memory:

- **Stack memory** - Store all local and global variables defined in code. Automatically destroyed when variable goes out of scope.
- **Heap memory** - Not managed automatically. Allocate and deallocate yourself. Useful when size of memory not known while writing code (may be user input).

new and *delete*



```cpp
Example *ex1;    //ex1 points to addresses
                 //that store objects of type Example

ex1 = new Example;
ex1->a = 5; // (*ex1).a = 5;
//...
delete ex1;
```



```
Example *ex2;
ex2 = new Example[5];
ex2[0]->a = 5;

//....

delete[] ex2;
```

Common Errors

- Dangling reference - Using memory that has already been deallocated
  - Common when 2 pointers pointing to same location on heap and we execute delete on one of them

- Memory Leak - No pointer to some allocated memory

Strategy : Each variable on the heap is pointed to by exactly one pointer at a time. Before that pointer goes out of scope - use delete

Problem

- Implement a String class using pointers.
- The file String.h has been provided.
- For more details see lab2.txt

Thank You