CS2134 Homework 1
Spring 2015
Due 11:59 p.m. Tues, February 3, 2015

January 25, 2015

Your first assignment includes a programming portion and a written portion. The programming portion should consist of a single file called hw01.cpp, and the written portion should consist of a single file called hw01written in a standard format (.txt, .doc, .htm., or .pdf). Be sure to include your name at the beginning of each file! You must hand in both files via NYU Classes.

Programming Part:

1. Write a function (or 3 separate functions) to determine if a vector contains a duplicate item when the vector contains:
   (a) chars
   (b) ints
   (c) strings
2. Using the linked list class

```
class Node
{
   public: // These member variables are made public to simplify your coding.
           // Of course these member variables should be private!
      Node( char ch, Node * ptr = nullptr):data(ch),next(ptr){}

      char data;
      Node * next;
};
```

   and a pointer to the class `Node * nodePtr;` write the code to perform the following:

   - create a linked list of three nodes containing the items *'C', 'B', 'D'*.
   - add a node containing *'A'* to the front of your linked list
   - print out the memory locations of *'A', 'B', 'C' 'D'* and *nodePtr* (and hand in this information with the written part).

- delete the node containing *'A'*, (*nodePtr* now points the the nodes containing *'B', 'C' 'D'*)

3. For an array, `int * intPtr = new int[5];`, using only pointers (no array indexing) write the code to perform the following:

   - insert items *2, 3,4, 5* into the first four positions (positions 0,1,2,3).
   - add item *1* to the front of your array; (you need to move the other items)
   - Print out the memory locations of *1, 2, 3, 4, 5* and *intPtr* (and hand in this information with the written part).
   - delete the array pointed to by *intPtr*

4. You will compare the time it takes to solve the Max Contiguous Subsequence sum problem, using the three algorithms presented in the book in chapter 6 (the code for all three algorithms are in the file called `MaxSubsequenceSum and Timer Class`.) You will run all three algorithms with the following values of n: $2^7 = 128, 2^8 = 256, 2^9 = 512, 2^{10} = 1024, 2^{11} = 2048$, and $2^{12} = 4096$.

   What you will need to do:

   (a) Look at the code for the `timer` class in the file `MaxSubsequenceSum and Timer Class`, and then apply it in your own code.

   (b) Fill a `vector` with `n` random integers in the range from -1000 to 1000. There is a function from the standard library called `rand()` that returns a value from `0` to `RAND_MAX`; (`RAND_MAX` is at least 32767.) The code `x = (rand() % 2001) - 1000` puts a random number in the range `[-1000, 1000]` into the variable `x`.

   (c) Time how long it takes the function `maxSubsequenceSum1` to find the maximum subsequence sum. The code might look like the following

   ```
   timer t;
   double nuClicks;
   // other code to fill in the vector with n items, etc.
   t.reset();
   maxSubsequenceSum1( items, ss, se);
   nuClicks = t.elapsed();
   ```

   (d) Time how long it takes the function `maxSubsequenceSum2` to find the maximum subsequence sum, using the same n elements.

   (e) Time how long it takes the function `maxSubsequenceSum4` to find the maximum subsequence sum, again using the same n elements. (Yes, this is function number 4; I am using the textbook's name for the functions.)

   Note: Here are three suggestions to help you:

   First, after debugging your code, turn off your debugger. Do not run any other program while solving this problem. Your program should take no more than 20 min. (My code took much less.)

   Second, make sure when you print the running times you are printing enough significant digits. Here are two choices for how to do this:

```
cout.setf( ios::fixed, ios::floatfield );
cout.precision( 6 );

or

cout.precision(numeric_limits<double>::digits10 + 1);
```

Third, we are using the clock function (you need to add the header `ctime` or `time.h`) that allows us to get an estimate of the CPU time spent on the program. The problem is, different systems keep track of the time differently. My computer's clock function returns returns a value in units of (approx) $1,000,000$ of a second. Other computers have a clock that returns a value of (approx) 1000 of a second. Therefore, some short time intervals are distinguishable from zero on my computer, but not on other computers.

To determine the resolution used on your computer type:

```
cout << CLOCKS_PER_SEC << endl;
```

For the written homework problem 7, if you get a running time of 0 when you run the linear time algorithm, then run that particular function with the same input multiple times and then divide by the number of times you ran the program.

Be aware that the measurements you are getting are ”rough” because other factors will influence the time. If you wanted a more accurate time estimate you would run each algorithm on for a fixed value of n several times and average the results.

Written Part:

1. Write each of the following functions in Big-Oh notation:

   (a) $T(n) = 14n + 24$

   (b) $T(n) = 3n^2 + 2n + 120$

   (c) $T(n) = 22 + 20n + n \cdot \log(n)$

   (d) $T(n) = 10 \cdot n \cdot \log^5(n) + 0.002n^3 + 82$

   (e) $T(n) = (2n^2 + 4 \cdot n) \cdot 11n$

   (f) $T(n) = 2\log(n^{12}) + 2n$

   (g) $T(n) = \frac{4\log(n)(3n \cdot \log(n) + 2n^2)}{n}$

   (h) $T(n) = \binom{n}{2}$ (This means n choose 2)

   (i) $T(n) = \binom{n}{3}$ (This means n choose 3)

2. For each of the following code fragments[1], determine the worst case running time using Big-Oh notation as a function of n.

   (a)
   ```
   int sum = 0;
   for (int i = 0; i < n; i++)
       ++sum;
   ```

   (b)
   ```
   int sum = 0;
   for(int i = 0; i < n; i++)
       for(int j = 0; j < n; ++j)
           ++sum;
   ```

   (c)
   ```
   int sum = 0;
   for(int i = 0; i < n; i++)
       for(int j = 0; j < i; ++j)
           ++sum;
   ```

[1] Some of these problems are from Weiss' book.

(d)
```
int sum = 0;
for(int i = 0; i < n; i++)
    for(int j = 0; j < n; ++j)
        for(int k = 0; k < n; ++j)
            ++sum;
```

(e)
```
int sum = 0;
for (int i = 0; i < n; i++)
    sum += i;
for (int j = 0; j < n; j++)
    sum += j;
for (int k = 0; k < n; k++)
    sum += k;
```

(f)
```
int sum = 0;
for (int i = 0; i < n; i++)
{
    sum += i;
    for (int j = 0; j < n; j++)
        sum += j;
}
```

(g)
```
int sum = 0;
for (int i = 0; i < n; i++)
{
    sum += i;
    for (int j = 0; j < n; j++)
    {
        sum += j;
        for (int k = 0; k < n; k++)
            sum += k;
    }
}
```

(h)
```
int sum = 0;
for (int i = 1; i < n; i*=2)
{
    sum += i;
}
```

(i)
```
int sum = 0;
for (int i = 1; i < n; i*=2)
{
    sum += i;
    for (int j = 0; j < n; j++)
    {
        sum += j;
     }
}
```

(j)
```
int sum = 0;
while (n>0)
{
     cout<< n%2;
     sum += n%2;
     n/=2;
}
```

3. Using Big-Oh notation, what is the running time of the code you wrote in programming question 1a?
4. Suppose a program takes 0.05 seconds to run on input size of 2048. Estimate how long it would run for an input size of 2^{13} if:
   (a) the program had an $O(n)$ running time.

   (b) the program had an $O(n^2)$ running time.

   (c) the program had an $O(n^4)$ running time.

5. Suppose you had a very complicated code that was difficult to analyze. To get a quick idea of your algorithm's running time you ran your program on different sized inputs. Suppose the following are the timing results for your algorithm. Using the timing results below, indicate the most likely running time in big-Oh notation; choose one from the following list. $O(1), O(n), O(n^2), O(n^3), O(n^4)$.

```
n        time

2^7      0.002094
2^8      0.00834
2^9      0.033412
2^10     0.133054
2^11     0.532501
2^12     2.12835
```

6. Using the definition of *Big-Oh*, show that $2n^2 + 4n\log(n) + 5n + 12 = O(n^2)$.

7. Look at the output from the programming part 4 of this assignment. Create a chart of your answers including times for all three algorithms and for all the specified input sizes, e.g.

```
Actual Times:
   n | maxSubsequenceSum1 O(n^3) | maxSubsequenceSum2 O(n^2) | maxSubsequenceSum4 O(n)
---- | ------------------------- | ------------------------- | -------------------------
 128 |                   0.00228 |                   6.2e-05 |                   3e-06
 256 |                   0.01676 |                  0.000237 |                   2e-06
   .
   .
   .
```

8. Create another chart that estimates the running time for each of the algorithms using the method presented in class, and using the time your computer took for the algorithms when $n = 2^7$. (If you did not receive valid answers, use the run times from question 7.) Display your answer in a chart, e.g.

```
Predicted Times:
  n  |  Exp. maxSubseq. 1 O(n^3) | Exp. maxSubseq. 2 O(n^2)  | Exp. maxSubseq. 4 O(n)
---  | ------------------------- | ------------------------- | -------------------------
256  |                   0.01824 |                  0.000248 |                 6e-06
512  |                   0.14592 |                  0.000992 |               1.2e-05
1024 |                   ...
2048 |
4096 |
```

9. Using the method given in class, predict how long each algorithm would take if $n = 2^{18}$. Show how you made your prediction by providing the formula (presented in class), and then evaluate your formula.

10. Using your answer from the previous question, rewrite your answer cumulatively in seconds, minutes, hours, days, and weeks. (To clarify: you are not going to re-express that time in terms of weeks and then again in terms of days and then again in terms of hours. You should only express it once, using weeks, days, hours, minutes, and seconds.)