EC330 Applied Algorithms and Data Structures for Engineers
Fall 2018

Homework 3

Out: September 27, 2018
Due: October 5, 2018

This homework has a written part and a programming part. Both are due at 8:59 am on October 5. You should submit both parts on Blackboard. For the written part, you should submit a single PDF file containing either typeset answers or scanned copies of hand-written answers. Make sure you write your answers clearly. For the programming part, your code should be easy to read and understand, and demonstrate good code design and style. Your program must compile and run on the lab computers.

1. Recurrence Relations [40 pt]

a) Suppose you have to choose among the follow three algorithms:
- Algorithm A solves the problem by dividing it into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.
- Algorithm B solves the problem of size n by recursively solving two subproblems of size $n - 1$ and then combining the solutions in constant time.
- Algorithm C solves problems of size n by dividing them into seven problems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.

What are the running times of each of these algorithms (in big-O notation). Which would you choose? **[10 pt]**

b) Give the *tightest asymptotic upper bound* (in $O(\cdot)$) you can obtain for the following recurrences. Justify your answer. **[5 pt each]**
- $T(n) = 7T(n/3) + n^3$
- $T(n) = 2T(n/4) + 5\sqrt{n}$
- $T(n) = 20T(n/15) + n \log n$
- $T(n) = T(n-1) + 5 \log n,\ T(1) = 1$
- $T(n) = (\log n)T(n/2) + 1$
- $T(n) = n(T(n/2))^2$

2. Time Complexity [20 pt]

a) Identify the time complexity of the following piece of code, where n is given as the input. Write your answer using the $\Theta(\cdot)$ notation. Justify your answer. *Hint: Assume* $n = 2^m$. **[10 pt]**

```
int i = n;
while (i > 1) {
```

```
        int j = i;
        while (j < n) {
              int k = 0;
              while (k < n) {
                    k = k + 2;
              }
              j = j * 2;
        }
        i = i / 2;
}
```

b) Write down the recurrence relation of the following sorting algorithm. What is the time complexity of this method (in $O(\cdot)$ notation)? Justify your answer. **[10 pt]**

```
void StrangeSort(int a[], int min, int max) {
      if (min >= max)
            return;
      if  (a[min] > a[max])
            swap(a[min], a[max]);  // constant-time operation
      int one_third = (max - min + 1) / 3;
      if (one_third >= 1) {
            StrangeSort(a[], min, max - one_third);
            StrangeSort(a[], min + one_third, max);
            StrangeSort(a[], min, max - one_third);
      }
}
```

3. Programming [40 pt]

Make sure to write your name and BU ID in a comment at the top of the program, along with your collaborator's name and BU ID, if any. To use the compiler on the lab computers, run "*module load gcc*" first. **Submit only *ListNode.h*.**

a) **[20 pt]** In this problem, you will implement *insertion sort for singly linked list*. You are given two files: *ListNode.h* and *Problem3a.cpp*. Your job is to implement the *insertionSortList* method in *ListNode.h.* The *Problem3a.cpp* file is given as a sample test program for your implementation. You can compile with the test program using:
> g++ -std=c++11 (-o myProgram) Problem3a.cpp

b) Recall *Floyd's cycle-detection algorithm* that we went over in class.

- **[10 pt]** Implement this algorithm in the *detectCycle* method in the same *ListNode.h* file. Similar to part a), you are given a sample test program Problem3b.cpp to test your code. You can compile with the test program using:
  > g++ -std=c++11 (-o myProgram) Problem3b.cpp
- **[10 pt]** It turns out that you can modify Floyd's algorithm to find the node where the cycle starts as well (if one exists). Implement the *findCycleStart* method in *ListNode.h*. You can also test this using *Problem3b.cpp*.
  Hint: Think about the distance from the cycle start node to the node where the two points meet (say this is d). Think about the distances (both non-loopy and loopy parts) the two pointers have traversed respectively until they meet and their relationship to d.