EC330 Applied Algorithms and Data Structures for Engineers
Spring 2020

Homework 3

Out: February 18, 2020
Due: February 28, 2020

This homework has a written part and a programming part. Both are due at 11:59 pm on February 28. You should submit both parts on Gradescope.
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, you should make sure your code compile and run on Gradescope. You should also try to create your own test cases to test and debug your code. Make sure you write your name and BU ID in a comment at the top of the program, and use clear comments to explain the key steps in your code.

1. Recurrence Relations [35 pt]

a) Suppose you have to choose from the following three algorithms:
   - Algorithm A solves the problem by dividing it into four subproblems each of half the size of the original problem, 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 nine 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-$\boldsymbol{O}$ notation). Which would you choose (especially for large *n*)? **[10 pt]**

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

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 $\boldsymbol{\Theta}(\cdot)$ 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 + 10;
            }
            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 $\boldsymbol{O}(\cdot)$ notation)? Justify your answer. Does this algorithm sort properly? Give an explanation if you think it does. Give a counterexample showing that it fails to sort if you think it doesn't. **[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 [45 pt]

a) Implement the function *zigzagSort* in *Problem3a.cpp*. The function sorts a vector of integers *nums* in-place such that $nums[0] \leq nums[1] \geq nums[2] \leq nums[3]$ ... Below are some example inputs and the expected outputs. Note that the integers are not necessarily all distinct. Your implementation should run in $\boldsymbol{O}(n)$ time where n is the size of the input vector. **[10 pt]**

Example:
Input: [3, 1, 4, 5, 3, 7, 2]
A possible output: [1, 4, 3, 5, 3, 7, 2] because $1 \leq 4 \geq 3 \leq 5 \geq 3 \leq 7 \geq 2$
Another possible output: [1, 5, 3, 7, 3, 4, 2] because $1 \leq 5 \geq 3 \leq 7 \geq 3 \leq 4 \geq 2$

b) In this problem, you will implement *insertion sort for singly linked lists*. The `ListNode` struct is given as follows (see *ListNode.h*).

```
struct ListNode {
      int val;
      ListNode *next;
```

```
        ListNode(int x) : val(x), next(NULL) {}
};
```

Implement the *insertionSortList* method in *Problem3b.cpp.* You can assume the linked list does not contain cycles. **[15 pt]**

Example:
Input: → 2 → 4 → 2 → 1
Output: → 4 → 2 → 2 → 1 (sort the list in decreasing order)

c) Implement the method *findCycleStart* in *Problem3c.cpp* to find the node where the cycle (if one exists) starts for a singly linked list. You can assume the same `ListNode` struct as in part b). Output NULL if there is no cycle. Your algorithm should use only ***O***(1) space. **[20 pt]**