EC330 Applied Algorithms and Data Structures for Engineers
Spring 2022

Homework 3

Out: February 19, 2022
Due: February 28, 2022 (Written Part)
March 4, 2022 (Coding Part)

This homework has a written part and a programming part. The written part is due at 11:59 pm on February 28. The coding part is due at 11:59 pm on March 4. You should submit both parts on Gradescope.

The written part should be completed individually, whereas the coding part can be done in pairs. See course syllabus for policy on collaboration.

1. Recurrence [20 pt]

a) Suppose you have to choose among the follow three algorithms:
- Algorithm A solves the problem by dividing it into four 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 five problems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $\boldsymbol{O}(n^2)$ time.

What is the time complexity of each of these algorithms (in big-$\boldsymbol{O}$ notation). Which algorithm should we choose if we want to solve the problem efficiently? **[10 pt]**

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

2. Sort [20 pt]

a) What is the worst-case time complexity of this method (the tightest bound in $\boldsymbol{O}(\cdot)$ notation)? What would be the best-case input for this algorithm and its time complexity? Justify your answers. **[10 pt]**

```
void StrangeSort(int A[], int n) { // n is the size of A[]
    int ind = 0;
    while (ind < n) {
        if (ind == 0 || A[ind-1] <= A[ind]) {
            ind++;
        } else {
```

```
            swap(A[ind], A[ind-1]) // constant-time operation
            ind--;
        }
    return;
}
```

b) Write down the recurrence relation of the following sorting algorithm. What would be the best-case input for this algorithm? For that case, is it faster or slower than Bubble Sort? Just your answers. **[10 pt]**

```
void MysterySort(int A[], int min, int max) {
      if (min >= max)
            return;
      int mid = floor((min + max)/2);
      MysterySort(A[], min, mid);
      MysterySort(A[], mid+1, max);
      if (A[max] < A[mid])
            swap(A[max], A[mid])  // constant-time operation
      MysterySort(A[], min, max-1);
}
```

3. Programming [60 pt]

Note that this is a pair-programming exercise in the sense that you are allowed to collaborate with at most one other student in the class. If student A collaborates with student B, then both students can submit the same code subject to the following conditions:

- *Both students must clearly write at the top of their submission (.cpp file) who their collaborator is (as a comment).*
- *The collaboration is for both part a) and b), i.e. if A collaborates with B for a), then neither A or B can collaborate with another student for b).*

a) Recall the example given in lecture where Alice and Bob want to measure the similarity between their respective ranked lists of songs (e.g. their rankings of the same 10 songs). One way to measure this similarity is to consider the degree to which the songs in one list that are *out of order (OOO)* with respect to the other list. For example, suppose Bob's list is "Easy on Me by Adele (E)" > "Ghost by Justin Bieber (G)" > "Shivers by Ed Sheeran (S)" and Alice's list is S > E > G (with > indicating the direction of preference). The degree of "out-of-orderness" from Alice's perspective is 2, since in Bob's list S > E and S > G.

Formally, for arrays *A* and *B* which contain the same set of elements, a pair of elements (*p*, *q*) is considered *OOO* if *p* appears before *q* in A but *p* appears after *q* in *B*. The *OOO* degree is then the total number of distinct *OOO* pairs. If the orders of elements in A and in B are the same, then the *OOO* degree is 0.

Implement the function *countOOO* in *Problem3a.cpp*. Your algorithm must run in time $\boldsymbol{O}(n \log n)$ where n is the size of the input vectors. You can assume vector A and vector B contain the same *set* of elements. **[30 pt]**

Hint: Count in a merge-sort way.

b) Implement the following variant of bubble sort *twowayBubble* in *Problem3b.cpp* where we operate in both forward and backward directions, given a *doubly-linked list* of integers as input. In the first forward (rightward) pass, we will move the largest element to the end of the list. In the subsequent backward (leftward) pass, we will move the smallest element to the beginning of the list. After *i* complete passes (each complete pass consists of a forward pass and a backward pass), the invariant that we have is that the first *i* elements and the last *i* elements will be sorted (i.e. in their correct positions). **[30 pt]**

Example:
Input: $3 \leftrightarrow 5 \leftrightarrow 2 \leftrightarrow 1$
After the 1st forward pass: $3 \leftrightarrow 2 \leftrightarrow 1 \leftrightarrow 5$
After the 1st backward pass: $1 \leftrightarrow 3 \leftrightarrow 2 \leftrightarrow 5$
We can see that 1 and 5 are now in their correct positions.

Note that you algorithm should only use $\boldsymbol{O}(1)$ space.