EE 367 Spring 2018 Midterm Exam 2

Instructions:

You are allowed to use *your own*

- textbooks
- homeworks
- homework solutions
- lecture notes
- lab assignments and solutions from this course or from previous courses, e.g., your discrete math course.

You may ask the instructor questions verbally or by email. Once you get a copy of this exam, you are ***not*** allowed to use anything else including

- discussions with others,
- the Internet,
- other books or references,
- course materials from others,
- or materials from other courses.

By taking this exam, you agree to abide by these instructions.

Turn in a hard copy at the beginning of class on friday April 6, 2018, except for Problems 3, 4, and 5, which should be uploaded into laulima.

Problem 1 [20 pts]. Consider the following bubble sort algorithm for an array A[1..n] of length n. The array has the values 1, 2, ..., n, that are randomly permuted so that each permutation is equally likely.

```
BubbleSort(A)   // Recall Bubblesort was in Homework 3, Problem 2-2, page 40 in the textbook.
for  i = 1  to  n
      for j = n  downto   i+1
            if  A[j] < A[j-1]
                  swap A[j] with A[j-1]
```

BubbleSort will sort the array in increasing order.

What is the expected number of swaps in the algorithm, and explain your derivation? Note that your expected number is an exact formula dependent on n, i.e., it's not in Big-O form. (*Hint: You can use indicator functions to determine the expected number of swaps, similar to how they were used to analyze Quicksort*.)

Problem 2. For this problem, you have an array A[0], A[1], ..., A[n-1] of numbers (integers or floating point) to be sorted, where n is the length of the array. The array has already been partially pre-sorted in the following way: every t consecutive elements are already sorted, i.e., the following subarrays are sorted:

- (A[0], A[1], ..., A[t-1])
- (A[t], A[t+1], ..., A[2t-1])
- (A[2t], A[2t+1],..., A[3t-1])
- etc

The following is an example of array of values assuming t = 4:

3, 7, 13, 15, 0, 4, 9, 11, 5, 6, 10, 14, 1, 2, 8, 12

Assume that t divides n; and that t is known, i.e., it's an input parameter. Also assume that t may grow with n, e.g., t may be approximately log n or the square root of n.

(a) [10 pt] Write down the pseudocode for an algorithm that sorts such an array. The input to the algorithm is the array A[ ], the length n, and parameter t. Analyze its running time and explain your analysis. For full credit, its time complexity must be the best asymptotic running time possible.
(*Hint: Modify a sorting algorithm that you already know*.)

(b) [10 pts] Prove a lower bound for sorting such a presorted array using a comparison-based sorting algorithm. It should match the running time of the algorithm in part (a). You must explain how you derived your lower bound.

(*Hint: You can use similar arguments used to show that comparison sorting takes $\Omega(n \log n)$ time – this was in the lecture notes on Sorting-Part3. You must count the number of ways to permute the numbers 0, 1, ..., n-1 that leave you with a pre-sorted arrangement as discussed above. Also, you may find it useful to use Stirling's approximation, and in particular $log(n!) \approx n \log n$ and $log(t!) \approx t \log t$.*)

Problem 3 [20 pts]. Attached is the file list2bst.c, which is a program that will create a binary search tree from a sorted list of nodes, and the tree will be of minimum height. The program first creates the sorted list of nodes, 'node_list'. For simplicity, the keys of the nodes are 0, 1, 2, ... n-1, where n is the number of nodes. Then the program calls the function build_tree(node_list) which builds the minimum-height binary-search-tree. (Note: that you may have to write new functions that can be called by build_tree.)

Currently, build_tree currently doesn't work. Implement it correctly so that it has time complexity O(n). Explain why your algorithm has time complexity O(n). Upload your code into laulima.

(*Note: There are other functions, 'height' and 'in_order', that are used to check if your algorithm is correct*.)

Problem 4 [20 pts]. Attached is the file subseq.c, which is a program that will find the longest increasing subsequence of an array of integers val[ ]. The program first creates an integer array val[ ] of length n, where val[0] = 0, val[1] = 1, ..., val[n-1] = n-1. Then it randomly permutes the values in val[ ] so that each permutation is equally likely. Then it calls function 'subseq(val, n)' which will return the length of the longest increasing subsequence and display the subsequence itself.

For example, if the randomly permuted array val[ ] was 0, 8, 4, 12, 2, 10, 6, 14, 1, 9, then subseq will return 4, which is the length of the longest increasing sequence, and display 0, 4, 6, 14 (Note that there are other longest increasing subsequences, and any of them could be displayed).

Currently, function 'subseq( )' doesn't work. You must implement it so it works correctly, so that it's time complexity is $O(n^2)$. Explain why it's time complexity is $O(n^2)$, and explain why it's correct. For partial credit, have your algorithm display the length of the longest increasing subsequence. Upload your code into laulima.

Problem 5. Given a string, your task is to count how many palindrome subsequences in this string. A palindrome subsequence is one that is a palindrome. Note that a subsequence doesn't have to be contiguous.

Example 1:
Input: "abc"
Output: 3
Explanation: Three palindrome subsequences: "a", "b", "c".

Example 2:
Input: "aaba"
Output: 10
Explanation: Ten palindrome subsequences: "a", "a", "a", "b", "aa", "aa", "aa", "aaa", "aba", "aba".

Assume that the input string length won't exceed 1000.

(a) [10 pts]. Attached is a file palin-top.c which has a program that will count the number of palindrome sequences. To run the program, enter

./a.out <file name>

where <file name> is a file with the input string of characters. An example file is 'string.dat' which is attached.

The program reads in the string of characters, and then calls function 'palinsub(string)' which will return the number of palindrome subsequences.

The current function 'palinsub' doesn't work. You must implement it correctly using the dynamic programming, top-down approach with memoization. You may add functions. The time complexity of your algorithm should be $O(n^2)$, where n is the length of the string. Explain why your algorithm is correct, and why it has time complexity $O(n^2)$. Upload your program palin-top.c into laulima.

(b) [10 pts]. This is the same as part (a) except that the program to modify is palin-bottom.c, and palinsub should be implemented using dynamic programming, bottom-up approach.