Midterm
CS-435 Corazza
Solutions Exam B

I. TRUE/FALSE. (14points) Below are 7 true/false questions. Mark either T or F in the space provided.

__T__ 1. When QuickSelect runs on input array S = [1, 3, 5, 2, 1, 7, 2, 1, 4, 1], with k = 4, the return value is 1.

__T__ 2. All known comparison-based sorting algorithms, when run on an array of 6 distinct integers, require, in the worst case, 10 or more comparisons to sort the integers. Worst case # comparisons >= ceil(log(6!)) = ceil(log 720) = log 1024 = 10.

.

__F__ 3. Suppose the running time of an algorithm is given by the recurrence

$$T(1) = d \qquad (d > 0)$$
$$T(n) = 2T(2n/3) + 2n.$$

Then $T(n)$ belongs to $\Omega(n^2)$. a =2, b = 3/2, c = 2, k = 1, d = d, $a>b^k$
So T(n) is Theta($n^{\log(3/2)2}$). But $(3/2)^2 > 2$, so $\log_{(3/2)}2 < 2$.

__T__ 4. Although InsertionSort is an inversion-bound sorting algorithm, the generalized version of InsertionSort which we have named LibrarySort in the class is *not* inversion-bound..

__F__ 5. Suppose f(n) > g(n) for all n. Then f(n) is *not* O(g(n)).
Consider f(n) = 2n+1, g(n) = n.

__T__ 6. n! is o((n+1)!).

__F__ 7. Suppose $\log(f(n))$ is $O(\log(g(n)))$. Then $f(n)$ is $O(g(n))$. **No – $f(n) = n^3$, $g(n) = n^2$.**

II. PROBLEMS. Solve the problems below. The more you explain your work, the more partial credit you will receive (in case your answer is incorrect).

1. [12] Use RadixSort, with two bucket arrays and radix = 11, to sort the following array:

[63, 1, 48, 53, 24, 10, 12, 30, 100, 115, 17].

Show all steps of the sorting procedure. Then explain why the running time is O(*n*).

Solution:

r[] =

	100 12 1	24		48	115	17		30 63	53	10
0	1	2	3	4	5	6	7	8	9	10

q[]=

10 1	17 12	30 24		53 48	63				100	115
0	1	2	3	4	5	6	7	8	9	10

Return: [1, 10, 12, 17, 24, 30, 48, 53, 63, 100, 115]

Running Time: O(n):

- Initializing each of the bucket arrays costs O(n)
- Populating r[] and q[] costs O(n)
- Reading output from q[] is O(n)

2. [10] There are two parts of this problem – you must do ONLY ONE of the two parts. CIRCLE the letter that labels the problem you will do. Part (A) must be done using the non-limit definition of big-oh. In Part (B) you may use the limit definition of little-oh.

   A. Show that $n^2 + 2$ is *not* O(n)

We must show that for every $c > 0$ and $n_0 \geq 1$ there is an $n \geq n_0$ such that $n^2 + 2 > cn$. Given c, n_0, we must find an n that works.

Scratch: If $n^2 > cn$, then certainly $n^2 > cn - 2$ and $n^2 + 2 > cn$. But $n^2 > cn$ iff $n > c$. We therefore pick n so that it is bigger than c and $\geq n_0$.

Proof that it works. Given $c > 0, n_0$, we let $n > \max\{c, n_0\}$. Then since $n > c$, $n^2 > cn > cn - 2$, and so $n^2 + 2 > cn$. Therefore, n is a number $\geq n_0$ that satisfies the inequality.

   B. Show that $n^{1/2} + \log n$ is o(n)

Problem 1B. Show that $n^{1/2} + \log n$ is $o(n)$.

$$\begin{aligned}\lim_{n\to\infty} \frac{n^{\frac{1}{2}} + \log n}{n} &= \lim_{n\to\infty} \frac{\frac{1}{2}n^{-\frac{1}{2}} + \frac{1}{n}\log e}{n} \\ &= \lim_{n\to\infty} \frac{1}{2n^{\frac{1}{2}}} + \frac{\log e}{n^2} \\ &= 0.\end{aligned}$$

3. [15] A *twisty table* is a kind of data structure that stores Strings and that uses an array in the background (assume it is already initialized and has as many available slots as necessary without resizing) and that has three primary operations:
   - AddOne(x) – adds String x to the background array in the next available slot
   - AddThree(x,y,z) – adds Strings x, y, and z to the background array by placing them in the next three available slots.
   - ClearAll() – removes all Strings currently in the array by setting them to null.

   Show that the amortized running time of ClearAll is O(1). Do the steps required by filling in the following table. Hint: For your amortized cost function, try charging 2 cyberdollars for AddOne, 6 cyberdollars for AddThree and 0 cyberdollars for ClearAll.

The cost function c:

```
c(AddOne)   = 1
c(AddThree) = 3
c(ClearAll) = k, where k is number of non-null Strings in the
              array
```

Your amortized cost function ĉ:

$$\begin{aligned} \hat{c}(\text{AddOne}) &= 2 \\ \hat{c}(\text{AddThree}) &= 6 \\ \hat{c}(\text{ClearAll}) &= 0 \end{aligned}$$

Show that your amortized cost function never results in negative amortized profit:

If AddOne is executed k times and AddThree is executed m times, the profit will be k + 3m, which is enough to pay for a ClearAll operation, which would cost k + 3m.

Provide a bound for the amortized cost of running n of these operations in succession.

$\sum_{i=1}^{n} \hat{c}(s_i) \leq \sum_{i=1}^{n} 6 = 6n$ which is $O(n)$.

Explain why ClearAll has amortized running time O(1):

Amortized running time is O(n)/n = O(1)

4. [16 points] The *T- Numbers* are defined as follows:

$$T_0 = 0, T_1 = 2, \; T_n = T_{n-1} + 3T_{n-2} + 1.$$

The following algorithm `Tnum` is a recursive algorithm that computes the R-Numbers:

Algorithm `Tnum` (n)
Input: A non-negative integer n
Output: The T-number T_n

if(n = 0) **then**
return 0
else if (n = 1) **then**
return 2
return `Tnum` (n - 1) + 3 * `Tnum` (n - 2) + 1

a. [4] Show that `Tnum` is correct. (You must show that the recursion is valid and that the base case and recursive steps return correct values.)

- Valid recursion since there is a base case and self-calls lead to the base case
- Base case returns correct values
- Assuming Tnum(m) returns a correct value for m < n, we compute the return value of Tnum(n):
  Tnum(n) = Tnum(n-1) + 3 * Tnum(n-2) + 1 = $T_{n-1} + 3 * T_{n-2} + 1 = T_n$.

b. [4] Show that `Tnum` has an exponential running time.

- **T(n) >= S(n) (number of self-calls performed by Tnum on input n)**
- **Claim: For n > 1, S(n) >= F_n (the nth Fibonacci number)**
  Proof: Base case: When n = 2, then S(2) =2 > F_1
  Assuming the result for m < n, we have
  S(n) = 2 + S(n-1) + S(n-2) > S(n-1) + S(n-2) >= $F_{n-1} + F_{n-2} = F_n$
- **$F_n > (4/3)^n$ for n > 4.**
- **It follows that T(n) has exponential running time**

The following is an iterative algorithm that computes the T-Numbers.

Algorithm `ItTnum` (n)
Input: A non-negative integer n
Output: The T-number T_n

```
storage <- new array(n + 1)
if n = 0 then return 0
if n = 1 then return 2
storage[0] <- 0
storage[1] <- 2
for i <- 2 to n do
   storage[i] = storage[i - 1] + 3 *storage[i - 2] + 1
return storage[n]
```

[4] Prove `ItTnum` is correct (you must formulate a loop invariant, prove that it holds after each iteration of the loop, and show that the final value obtained in the loop is the correct output for the algorithm).

Loop invariant: I(i) : value in storage[i] is T_i
Valid Loop Invariant (by induction):
Base Case: At end of i = 2 loop we have:
storage[2] = storage[1] + 3*storage[0] + 1 = 2 + 3*0 + 1 = 3 = T_2
Assuming true for i pass. Then
storage[i+1] = storage [i] + 3 * storage[i-1] + 1 = T_i + 3* T_{i-1} +1 = T_{i+1}
Finally, since storage[n] stores T_n (as we just showed) the algorithm returns the correct value.

c. [4] Give an asymptotic bound for the running time of `ItTnum`.

Just one for loop – so O(n)

5. [12] (Interview Question) Two "interview" questions are given below. You must do ONLY ONE of these. To indicate which question you will be doing, circle the letter that labels that question. (If you attempt both questions and do not indicate which one should be graded, you will receive NO credit for this problem.)

A. An array A of length n -1 contains n -1 distinct integers in the range 0 .. n -1. Note that this means that at least one integer x in the range 0 .. n -1 is not in A. Describe an $O(n)$ algorithm to determine the integer x. You are allowed to use only $O(1)$ additional space (this implies you can't use an additional array or hashtable). Give an argument to show that your algorithm runs in $O(n)$ time.

Solution: Notice that the sum of the numbers 0..n-1 is s = (1/2)(n(n-1)). Let r be the number in the range 0 .. n-1 that is not in A. Then the sum of the numbers in A will be s – r. Therefore, to find r, the algorithm is: Let q = sum of elements of A and return s – q.

Running time: It takes O(n) to add up the numbers in A; the rest is constant time.

B. Give an O(*n*log *n*) algorithm for sorting integers in a linked list. Explain why your algorithm has this running time.

Solution: Given a linked list L of size n, create an array A of size n and copy elements of L into A; perform MergeSort on A; then copy the elements of A back to L.

Running time:

- O(n) to create A
- O(n) to copy L to A
- O(n log n) to sort A
- O(n) to copy A to L

Total Running time: O(n log n)

III. SCI [3 points] Elaborate upon a parallel between points and topics in Algorithms and one or more SCI principles.

The Master Formula

For recurrences that arise from Divide-And-Conquer algorithms (like Binary Search), there is a general formula that can be used.

Theorem. Suppose $T(n)$ satisfies

$$T(n) = \begin{cases} d & \text{if } n = 1 \\ aT(\lceil \frac{n}{b} \rceil) + cn^k & \text{otherwise} \end{cases}$$

where k is a nonnegative integer and a, b, c, d are constants with $a > 0, b > 1, c > 0, d \geq 0$. Then

$$T(n) = \begin{cases} \Theta(n^k) & \text{if } a < b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^{\log_b a}) & \text{if } a > b^k \end{cases}$$