

Midterm
CS-435 Corazza
Solutions Exam A

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

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

__F__ 2. There is a comparison-based sorting algorithm integers which, when run on an array of 6 distinct, requires only 9 comparisons to sort the integers, even in the worst case. **Worst case # comparisons $\geq \lceil \log(6!) \rceil = \lceil \log 720 \rceil = \log 1024 = 10$.**

__T__ 3. Suppose the running time of an algorithm is given by the recurrence
$$T(1) = d \quad (d > 0)$$
$$T(n) = 10T(n/3) + 2n^2.$$
Then $T(n)$ belongs to $\Omega(n^2)$. **$T(n) = \Theta(n^k)$ where $k = \log_3 10 > 2$.**

__F__ 4. Although none of the currently known inversion-bound sorting algorithms run faster than $\Theta(n^2)$ on average, it is possible that, one day, someone will develop an inversion-bound sorting algorithm that runs in $\Theta(n \log n)$ on average.

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

__F__ 6. $(1/2)(n+1)!$ is $O(n!)$. **$n!/(n+1)! \rightarrow 0$.**

__F__ 7. Suppose $f(n)$ is $o(g(n))$. Then $\log(f(n))$ is $o(\log(g(n)))$. **Consider $f(n) = n^2$ and $g(n) = n^3$.**

II. PROBLEMS. Each of the following requires a short explanation.

1. [12] Use RadixSort, with two bucket arrays and radix = 8, to sort the following array:

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

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

$r[] =$

24 48	1	10		12	53	30	63
0	1	2	3	4	5	6	7

$q[] =$

1 10	12 10		30 24			53 48	63
0	1	2	3	4	5	6	7

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

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 $n^2 + 2n$ is $O(n^3)$

Scratch: Notice $2n \leq n^2$ for $n \geq 2$. So we find c such that $2n^2 \leq cn^3$, which holds iff $2 \leq cn$ iff $c \geq 2/n$. When $n_0 = 2$, we can let $c = 1$.

Proof that it works. Let $c = 1, n_0 = 2$. Then for any $n \geq 2$, $n^2 + 2n \leq 2n^2$ and since $n \geq 2$, $n^3 \geq 2n^2$. Therefore, $n^2 + 2n \leq cn^3$.

B. Show that $n^2 + 2n$ is $o(2^n)$

$$\lim (n^2/2^n) = \lim(2n/c2^n) = \lim(2/c^22^n) = 0$$

3. [15] A *flexible 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

- `AddTwo(x,y)` – adds Strings `x` and `y` to the background array by placing them in the next two 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`, 4 cyberdollars for `AddTwo` and 0 cyberdollars for `ClearAll`.

The cost function c :

$c(\text{AddOne}) = 1$
 $c(\text{AddTwo}) = 2$
 $c(\text{ClearAll}) = k$, where k is number of non-null Strings in the array

Your amortized cost function \hat{c} :

$\hat{c}(\text{AddOne}) = 2$
 $\hat{c}(\text{AddTwo}) = 4$
 $\hat{c}(\text{ClearAll}) = 0$

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

If `AddOne` is executed k times and `AddTwo` is executed m times, the profit will be $k + 2m$, which is enough to pay for a `ClearAll` operation, which would cost $k + 2m$.

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

$$\sum_{i=1}^n \hat{c}(s_i) \leq \sum_{i=1}^n 4 = 4n \text{ which is } O(n).$$

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

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

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

$$R_0 = 1, R_1 = 2, R_n = R_{n-1} * R_{n-2} + 1.$$

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

Algorithm `Rnum` (*n*)

Input: A non-negative integer *n*

Output: The R-number R_n

if(*n* = 0 || *n* = 1) **then**

return *n* + 1

return `Rnum` (*n* - 1) * `Rnum` (*n* - 2) + 1

a. [4] Show that `Rnum` 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 `Rnum`(*m*) returns a correct value for *m* < *n*, we compute the return value of `Rnum`(*n*):
 $R_{\text{num}}(n) = R_{\text{num}}(n-1) * R_{\text{num}}(n-2) + 1 = R_{n-1} * R_{n-2} + 1 = R_n.$

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

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

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

Algorithm `ItRnum` (*n*)

Input: A non-negative integer *n*

Output: The R-number R_n

storage <- new array(*n*+1)

if *n* = 0 or *n* = 1 then return *n* + 1

storage[0] <- 1

storage[1] <- 2

for *i* <- 2 to *n* do

 storage[*i*] = storage[*i*-1]*storage[*i*-2] + 1

return storage[n]

- c. [4] Prove ItRnum 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 R_i

This is true at the end of $i = 2$ loop. Assuming true for i pass. Then $\text{storage}[i+1] = \text{storage}[i] * \text{storage}[i-1] + 1 = R_i * R_{i-1} + 1 = R_{i+1}$

Finally, since storage[n] stores R_n (as we just showed) the algorithm returns the correct value.

- d. [4] Give an asymptotic bound for the running time of ItRnum.

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. Describe an $O(n)$ algorithm (you can use pseudo-code, Java, or a sufficiently detailed English description) that does the following: Given an input array of n integers lying in the range $0 \dots 3n - 1$, the algorithm outputs the first integer that occurs in the array only once. (You may assume that each input array contains at least one number that has no duplicates in the array.) Explain why your algorithm has an $O(n)$ running time.

Example: If the input array is [1, 2, 4, 9, 3, 2, 1, 4, 5], then the return value is 9 since 9 is the first integer that occurs in the array only once.

The following solution – which finds some non-repeated integer, is worth 10 points:

Almost Solution. Set up a tracking T array with indices $0 \dots 3n-1$, with values initialized to 0. Scan the input array $A = a_0, a_1, \dots, a_{n-1}$. When a_j is read, increment $T[a_j]$. Then scan the tracking array T and return the first index a_j found for which $T[a_j] = 1$.

Below are two correct solutions that deserve full credit:

Solution #1 Let A denote the initial input array of integers. We use an auxiliary array T as a tracking array

```
T ← new array(3n) //initialized with 0s
for i ← 0 to n - 1 do //scan A, increment T at A[i] each i
    x ← A[i]
    T[x] ← T[x] + 1
for i ← 0 to n-1 do
    x ← A[i]
    if T[x] = 1 then
        return x
```

Running Time: Two $O(n)$ loops \rightarrow total running time is $O(n)$

Solution #2 Let A denote the initial input array of integers. Use two tracking arrays S , T . Use S to keep track of the number of occurrences of an integer in A as A is scanned. [For example, when integer j is read, increment $S[j]$; i.e. $S[j] \leftarrow S[j] + 1$.] Use T to keep track of the position of the first occurrence of each integer. [For example, if j occurs for the first time as the 6th integer in A , set $T[j] \leftarrow 6$.]

After the scan of A is complete, read out into a *results* array the indices of S whose corresponding values are precisely 1. (These are the integers in A that occur only once.) Then perform a scan of the *results* array doing the following:

```
min ← results[0]
for i ← 1 to results.length - 1 do
    if results[i] < min then
        min ← results[i]
return min
```

The returned value is the first-occurring non-repeated integer in A .

Running Time: The initial scan of A and the subsequent scan of *results* each require $O(n)$, so total running time is $O(n)$.

Example Array: 1 2 4 8 3 2 1 4 5

S:	0	1	2	3	4	5	6	7	8	9
T:	0	1	2	3	4	5	6	7	8	9

result: 3, 5, 9
min: 9

- B. An *anagram* of a string of characters is a rearrangement of the characters. For instance, the following are anagrams of the string "beets":

beets, stbee, setbe, beste, beset

(Note that *both e's* must appear in any anagram of the word "beets".)

Describe an algorithm (you can use pseudo-code, Java, or a sufficiently detailed English description) that arranges an input array of strings so that all anagrams of each string in the array occur together. For instance, if the following is input to the algorithm:

[ham, friend, mah, tree, tooth, rete, ahm, hammer]

then the output would be

[ham, mah, ahm, friend, tree, rete, tooth, hammer]

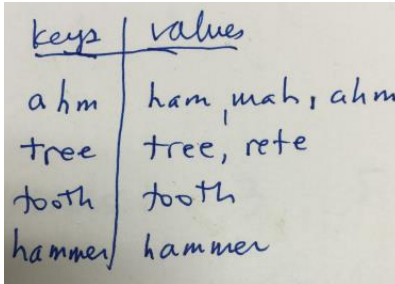
(Note that ham, mah, ahm are anagrams; tree, rete are also anagrams.)

If the max length of any word in an input array is m and the array contains n strings, what is the asymptotic running time of your algorithm?

Solution: Set up a hashtable H having Strings as keys and Lists of Strings as values. As the input array is scanned, for each String s read, sort s (producing a String t). The plan is to put t in as a key and add s to the list associated with t . To do this there are two cases: If t is already a key, add s to the list associated with t ; if t is not already a key, then create a new list L , add s to L ,

and do `H.put(t, L)`.

Finally, to obtain final arrangement: For each key `t` in `H`, output the list obtained by `H.get(t)`. All anagrams of `t` will be in this list.



keys	values
ahm	ham, mah, ahm
tree	tree, rete
tooth	tooth
hammer	hammer

Running time:

- Sorting each String of length m , for n Strings, costs $O(n(m \log m))$.
- Comparing two sorted Strings of length m requires m , and this is done n times, so $O(mn)$.
- Printing out the lists in the hashtable requires $O(n)$

Total running time: $O(nm \log m)$.

IV. 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

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