CS161 Midterm Exam

Once you turn this page, your exam has officially started!
You have three (3) hours for this exam.

Instructions: This is a **timed, closed-book** take-home exam:

- You must complete this exam within **180 minutes** of opening it.
- You may use one two-sided sheet of notes that you have prepared yourself. **You may not use any other notes, books, or online resources. You may not collaborate with others.**
- **If you have a question about the exam:** Try to figure out the answer the best you can, and clearly indicate on your exam that you had a question, and what you assumed the answer was.

You may cite any result we have seen in lecture or in the textbook without proof, unless otherwise stated. There is one blank page at the end for extra work. **Please write your name at the top of all pages.**

Advice: If you get stuck on a problem, move on to the next one. Pay attention to how many points each problem is worth. Read the problems carefully.

The following is a statement of the Stanford University Honor Code:

1. *The Honor Code is an undertaking of the students, individually and collectively:*
   (1) *that they will not give or receive aid in examinations; that they will not give or receive unpermitted aid in class work, in the preparation of reports, or in any other work that is to be used by the instructor as the basis of grading;*
   (2) *that they will do their share and take an active part in seeing to it that others as well as themselves uphold the spirit and letter of the Honor Code.*
2. *The faculty on its part manifests its confidence in the honor of its students by refraining from proctoring examinations and from taking unusual and unreasonable precautions to prevent the forms of dishonesty mentioned above. The faculty will also avoid, as far as practicable, academic procedures that create temptations to violate the Honor Code.*
3. *While the faculty alone has the right and obligation to set academic requirements, the students and faculty will work together to establish optimal conditions for honorable academic work.*

By signing your name below, you acknowledge that you have abided by the Stanford Honor Code while taking this exam.

Signature: ______________________

Name: SOLUTIONS

SUNetID: ______________________

1 Multiple Choice (36 pts)

No explanation is required for multiple-choice questions. Please clearly mark your answers; if you must change an answer, either erase thoroughly or else make it **very** clear which answer(s) you have chosen. **Ambiguous answers will be marked incorrect.**

1.1. **(1 pt.)** Suppose that $f(n) = O(n)$ and $g(n) = O(n \log n)$. True or false: $f(n) + g(n) = O(n \log n)$.

[✓ True] False

1.2. **(2 pt.)** Which of the following expressions correctly describe $T(n) = n^2 \log n$? Circle all that apply.

(A) $O(n)$ (B) $O(n^2)$ (C) $\Theta(n^2)$ [✓ (D) $\Omega(n^2)$] (E) $\Theta(n^3)$

1.3. **(15 pt.)** For each recurrence relation, choose the expression (a)-(e) below which most accurately describes it and **write your choice (a)-(e)** in the blank.

Note that **not all options need to be used,** and some options may be used more than once. If it helps, $\log_2(3) \approx 1.6$, and $\sum_{i=0}^{t} i^2 = \Theta(t^3)$. Assume that $T(1) = O(1)$ and don't worry about floors and ceilings.

(a) $\Theta(\log n)$ (b) $\Theta(n)$ (c) $\Theta(n \log n)$ (d) $\Theta\left(n^{\log_2 3}\right)$ (e) $\Theta(n^3)$

1.3.1. **(2 pt.)** d: $T(n) = 3T(n/2) + n^{3/2}$
1.3.2. **(2 pt.)** b: $T(n) = T(n/2) + n$
1.3.3. **(3 pt.)** e: $T(n) = T(n-1) + n^2$
1.3.4. **(4 pt.)** b: $T(n) = T(n/2) + T(n/10) + T(n/20) + n$
1.3.5. **(4 pt.)** a: $T(n) = T(n^{1/2}) + \log n$

1.4. **(10 pt.)** For each quantity, choose the expression (A)-(E) below which most accurately describes it and **write your choice (A)-(E)** in the blank.

Note that **not all options need to be used,** and some options may be used more than once. Unless specified otherwise, all inputs are arrays of length n, and all running times refer to worst-case running times of the best possible algorithms.

(A) $\Theta(n)$ (B) $\Theta(n^2)$ (C) $\Theta(n \log n)$ (D) $\Theta(\log n)$ (E) $\Theta(1)$

1.4.1. **(2 pt.)** C: Expected running time of QuickSort
1.4.2. **(2 pt.)** B: Running time of InsertionSort
1.4.3. **(2 pt.)** D: Time to search for an item in a Red-Black tree with n items
1.4.4. **(2 pt.)** C: Running time of RadixSort with base $r = 10$, assuming all items are integers between 0 and n^{100}
1.4.5. **(2 pt.)** A: Time to find the median element of an unsorted array.

1.5. **(8 pt.)** Consider the followed randomized algorithm, which prints out some number of smiley faces (`:)`) and some number of asterisks ($*$):

```
def randomizedSmiles(A):
    n = len(A)
    if n <= 1:
        print ":)"
        return
    for i in {0,...,n-1}:
        print "*"
    Choose a uniformly random integer p in {1,....,n-1}
    randomizedSmiles(A[:p])
    randomizedSmiles(A[p:])
```

For the following parts, **circle the tightest big-Oh bound that applies.**

Both of these answers are (c) because this prints one ":)" for each element in the array, no matter what p's are chosen.

1.5.1. **(2 pt.)** What is the expected number of `:)`'s that `randomizedSmiles` prints out?

(a) $O(1)$ (b) $O(\log n)$ [✓ (c) $O(n)$] (d) $O(n \log n)$ (e) $O(n^2)$

1.5.2. **(2 pt.)** What is the maximum number of `:)`'s that `randomizedSmiles` can print out, if you got to choose how p is selected?

(a) $O(1)$ (b) $O(\log n)$ [✓ (c) $O(n)$] (d) $O(n \log n)$ (e) $O(n^2)$

1.5.3. **(2 pt.)** What is the expected number of $*$'s that `randomizedSmiles` prints out?

(a) $O(1)$ (b) $O(\log n)$ (c) $O(n)$ [✓ (d) $O(n \log n)$] (e) $O(n^2)$

1.5.4. **(2 pt.)** What is the maximum number of $*$'s that `randomizedSmiles` can print out, if you got to choose how p is selected?

(a) $O(1)$ (b) $O(\log n)$ (c) $O(n)$ (d) $O(n \log n)$ [✓ (e) $O(n^2)$]

The number of "$*$"s printed by this algorithm is basically the same as the number of comparisons done by QuickSort.

To be more precise, if you were to throw in an extra "PARTITION" step in the algorithm above, you would basically get QuickSort, plus some print statements. Moreover, that modified alg would print out exactly the same number of "$*$"'s as the version without PARTITION (printing $*$ doesn't depend on the contents of A), and the number of $*$'s printed out would be roughly the same as the number of comparisons made by PARTITION.

So the expected number of $*$'s printed out is $O(n \log n)$, the same as the expected number of comparisons in QuickSort. The worst-case number of $*$'s printed out is $O(n^2)$, same as the worst-case number of comparisons for QuickSort.

2 Can it be done? (Short answers) (24 pts)

For each of the following tasks, either **explain briefly and clearly how you would accomplish it**, or else **explain why it cannot be done** in the worst case. If you explain how to accomplish it, you do not need to prove that your algorithm works. You may cite any result or algorithm we have seen in class, but don't make any assumptions that are not explicitly stated. Unless stated otherwise, all running times refer to deterministic worst-case running time.

The first two have been done for you to give an idea of the level of detail we are expecting.

2.1. **(0 pt.)** Find the maximum of an unsorted array of length n in time $O(n \log n)$.

I would use MergeSort to sort the array, and then return the last element of the sorted array.

2.2. **(0 pt.)** Find the maximum of an unsorted array of length n in time $O(1)$.

This cannot be done, because since the maximum could be anywhere, we need to at least look at every element in the array, which takes time $\Omega(n)$.

2.3. **(6 pt.)** Suppose you have access to a magic box which, given an array A, does the following in time $O(n)$:

- With probability $1/2$, the box outputs a sorted version of A.
- With probability $1/2$, the box outputs an array with the same elements as A, which may or may not be sorted.

Sort an array A of n elements using a comparison-based *randomized* algorithm with additional access to this magic box, which succeeds with probability $1 - 1/2^{10}$, in time $O(n)$.

This can be done by calling the magic box 10 times and checking if the result is sorted:

```
for $i = 1, \ldots, 10$:
    $A' \leftarrow$ magic_box($A$)
    if $A'$ is sorted:    // can check this in time $O(n)$ by stepping through the array
        return $A'$
return "couldn't sort A"
```

2.4. **(6 pt.)** Suppose you have access to a magic box that can circularly shift any contiguous sub-array by 1, in-place, in time $O(1)$. (For example, given the array $A = [0, 1, 2, 3, 4, 5]$, after an $O(1)$-time call `magicbox(A[1:5])`, we have $A = [0, 2, 3, 4, 1, 5]$). Sort an array A of n elements using a comparison-based algorithm with additional access to this magic box, in time $O(n)$.

This cannot be done. The magic box just manipulates the array, it doesn't do comparisons. Even with the magic box, we still need to make $\Omega(n \log n)$ comparisons, as we saw in the decision tree argument in class.

2.5. **(6 pt.)** You are given an array A of length n, where n is a perfect square. Suppose you have access to a magic box which, given any array B of length $k \leq \sqrt{n}$, returns the index of a minimum element of B in time $O(1)$. Sort A using a comparison-based algorithm with additional access to this magic box, in time $O(n)$.

This can be done: divide A into $\sqrt{n}$ lists of length $\sqrt{n}$. Then use the magic box to sort each one in time $O(\sqrt{n})$; this can be done by iteratively peeling off the minimum. Finally, we can merge the $\sqrt{n}$ sorted lists in time $O(n)$: the algorithm is the same as the MERGE alg from class, except we have $\sqrt{n}$ lists and use the magic box to identify the min of all $\sqrt{n}$ pointers and increment the appropriate one.

2.6. **(6 pt.)** Suppose that $k \leq n$. Given an array A containing n distinct elements, use a comparison-based algorithm to return all of the smallest k elements of A, in any order, in time $O(n)$. (Notice that the running time should not depend on k).

This can be done. We find the k^{th} smallest element of A in $O(n)$ time using the SELECT algorithm from class. Then we use an algorithm like PARTITION to go through the n elements of A and return the ones smaller than or equal to the k^{th}.

3 Algorithm Analysis / Proving Stuff (12 pts)

3.1. **(12 pt.)** Suppose that A is an array of n elements. Each element of A has some flavor; you cannot tell the flavors apart, but you know that one flavor is a strict majority. That is, there are strictly more than $n/2$ elements that have this one flavor. You have access to a genie called isMajority so that isMajority(A, x) returns **True** if x is an element of A with the majority flavor, and **False** otherwise.

You develop the following algorithm to return an element of the majority flavor, when n is a power of 2:

```
def findMajority(A):
    n = len(A) // assume that n is a power of 2.
    if n == 1:
        return A[0]
    a = findMajority(A[:n/2])
    b = findMajority(A[n/2:])
    if isMajority(A,a):
        return a
    else:
        return b
```

3.1.1. **(10 pt.)** Prove by induction that `findMajority` correctly returns an element of the majority flavor.

[**We are expecting:** *A formal proof by induction. Make sure to clearly label your inductive hypothesis, base case, inductive step, and conclusion.*]

Inductive Hypothesis: For any array A of length n, if A has a majority elt, then findMajority returns it.

Base Case: If $n=1$, then the majority element is A[0], the only element. Since this is what findMajority returns, the I.H. holds for $n=1$.

Inductive step: Let $k \geq 1$ and suppose that the I.H. holds for all $n \leq k-1$.
We want to show it holds for $n=k$.
Let A be an array of length n, and suppose that A has a majority elt, x.
Then x must be a majority elt of either A[:n/2] or A[n/2:]: otherwise, the total number of copies
of x in A would be at most $\frac{n}{4}+\frac{n}{4}=\frac{n}{2}$, so it would not be a strict majority.
Then by the I.H., either $a=x$ or $b=x$, where $a=$ findMajority(A[:n/2]) and $b=$ findMajority(A[n/2:]) as in the pseudocode.
If $a=x$, then isMajority(A, a) will be TRUE and findMajority will return a.
On the other hand if $a \neq x$, then $b=x$, and findMajority will return b.
Either way, the inductive hypothesis is established for $n=k$.

Conclusion. By induction, for all $n \geq 1$, findMajority returns a majority element of an array A of length n.

More space on next page!

More space for 3.1.1.

3.1.2. **(2 pt.)** Suppose that isMajority runs in time $O(\sqrt{n})$ on an array of length n. What is the big-Oh running time of `findMajority`?

[**We are expecting:** *Your answer and an explanation. You should give the best big-Oh bound you can.*]

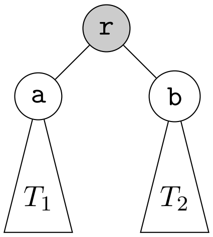
In this case, the running time $T(n)$ of isMajority satisfies the recurrence relation $T(n) = 2T(n/2) + O(\sqrt{n})$.

By the Master Theorem, $T(n) = O(n)$.

4 Algorithm Design (28 pts)

4.1. **(13 pt.)**

Suppose that there is a data structure `ModifiedRedBlackTree` which is just like a Red-Black tree, except that additionally, each node N has a method `sizeOfSubtree()` which returns the size of the subtree under N in time $O(1)$. For example, in the tree to the right, `a.sizeOfSubtree()` would return the number of nodes in T_1, **including** `a`, and not including `NIL` nodes.



4.1.1. **(10 pt.)** Give an algorithm that will find the median element in a `ModifiedRedBlackTree` containing n elements in time $O(\log n)$. The algorithm should take as input a pointer to the root of the `ModifiedRedBlackTree`. You can use any methods that you could use on a regular Red-Black tree. You may assume that n is odd.

[**We are expecting:** *Pseudocode and a clear English description of what your algorithm is doing. You do not need to prove that your algorithm is correct.*]

To implement MEDIAN, we will implement a more general k-SELECT algorithm on modified RB Trees. The basic idea is like the linear-time k-SELECT alg. from class: if there are < k nodes in the subtree on the left, we recurse on the left. Otherwise, we look for the k-(size of left subtree)-1 smallest thing on the right.

```
def MEDIAN(r):
    return SELECT(r, (r.sizeOfSubtree()-1)/2 )        //assuming n is odd

def SELECT(r, k):
    leftSize = r.left.sizeOfSubtree()
    if leftSize = k-1:                                //then r is the kth smallest
        return r
    if leftSize > k-1:                                //the the kth smallest lies to the left
        return SELECT(r.left, k)
    else:                                             // then the kth smallest lies to the right
        n = r.sizeOfSubtree()                         and is the k-leftSize-1 smallest there.
        return SELECT(r.right, k-leftSize-1)
```

More space and another part on next page!

4.1.2. **(3 pt.)** Explain why your algorithm runs in time $O(\log n)$. You should explicitly use the properties of Red-Black trees in your answer.

[**We are expecting:** *A short explanation which explicitly invokes the properties of Red-Black trees.*]

This algorithm traverses the unique path from the root of the tree to the median. Since RBTrees have depth $O(\log n)$, this path has length at most $O(\log n)$, and this algorithm runs in time $O(\log n)$.

4.2. **(15 pt.)** For this problem, suppose that A is an array containing n distinct items. There is some order on the items, but you **do not have direct access to a way to compare them.** Instead you have access to a genie named MIDDLE. When you call the genie on three distinct items, it will tell you which of the three is in the middle. For example, if $a < c < b$, then MIDDLE(a, b, c) will return c. If you call the genie on the same item multiple times, for example, MIDDLE(a, a, b), the genie will return the repeated element a.

If it helps, you can alternatively assume that MIDDLE outputs 1, 2, or 3 to indicate if the 1st, 2nd, or 3rd parameter, respectively, was the middle element: in the example where $a < c < b$, MIDDLE(a, b, c) could return 3; and MIDDLE(a, a, b) would return either 1 or 2 arbitrarily.

You may do standard array operations like accessing entries, swapping entries and so on. You may copy items, and you can tell whether or not two items are equal.

4.2.1. **(6 pt.)** Suppose that A has size n. (If it helps, you may assume that n is a power of two). Give an algorithm `getMinAndMax` which takes A as input, uses $O(n)$ calls to MIDDLE, and returns two elements a and b in A, so that one of a, b is the minimum of A, and the other is the maximum of A. Your algorithm does not need to say which is the minimum and which is the maximum.

[**We are expecting:** *Pseudocode and a clear English description of what your algorithm is doing. You* ***do not*** *need to justify the running time or the correctness.*]

NOTE. The problem was not well-defined if $n \leq 1$, so it is OK if your solution doesn't handle the $n \leq 1$ case.

Here are two solutions (only one is required for credit):

① Divide + Conquer Solution:

```
def getMinAndMax4(A):     //assume A is an array of size 4
    a = MIDDLE(A[:3])
    b = MIDDLE(all the items in A except for a)
    return the two items in A that are neither a nor b.

def getMinAndMax(A):     //assume len(A) is a power of 2
    if len(A) = 2:
        return A
    if len(A) = 4:
        return getMinAndMax4(A)
    a,b = getMinAndMax(A[:n/2])
    c,d = getMinAndMax(A[n/2:])
    return getMinAndMax4([a, b, c, d])
```

This algorithm first implements the case for $n = 4$ in getMinMax4. Then for a general array of length n, it finds the min and max in the left half, and the min and max in the right half, then uses the size 4 case again to return the min and max of those 4.

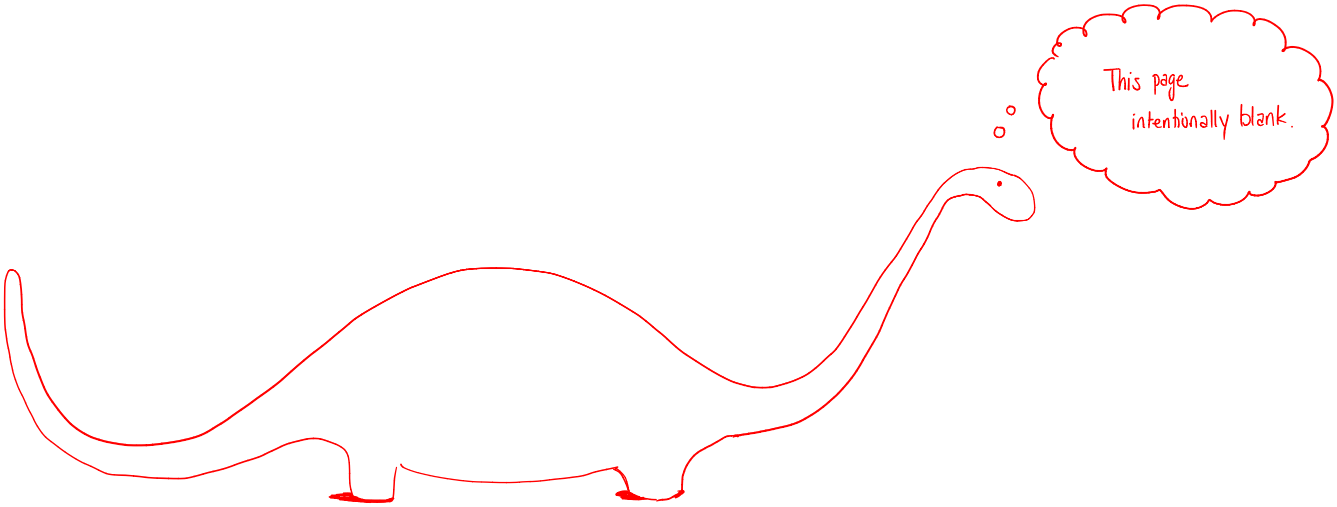
② Iterative Solution:

```
def getMinAndMax(A):
    ret = {A[0], A[1]}     \\These are our current candidates
    for i=2,3,...,n-1 :
        a = MIDDLE( ret[0], ret[1], A[i] )
        ret ← (ret ∪ {A[i]}) \ {a}     \\Update ret to be the two elements of ret ∪ {A[i]} that are not the middle element a.
    return ret.
```

This algorithm steps through the array, keeping a set ret of candidates. Every time it finds an element a that is between two other elements, it removes it from consideration, since a could not have been the max or min of A.

More space on next page!

Blank space for 4.2.1.



Note: dinosaur drawing not required for credit.

Another part on next page!

4.2.2. **(9 pt.)** Describe how to use part 4.2.1. to implement an algorithm `SortOfSort` which takes A as input, and returns a version of A which is *either* sorted from smallest to largest *or* sorted from largest to smallest. It is okay if your algorithm does not know which direction the output is sorted in.

Your algorithm should use $O(n \log n)$ calls to the MIDDLE genie. Partial credit will be given for an algorithm that makes $O(n^2)$ calls to the the MIDDLE genie.

You may use the algorithm described in part 4.2.1., even if you did not complete that part.

[**Hint:** *Can you use part 4.2.1. and* MIDDLE *to mimic a comparison-based algorithm?*]

[**We are expecting:** *A clear English description of what your algorithm would do. It is okay to refer to an algorithm that we have seen in class.* ***You do not need to write pseudocode*** *to receive credit for this part, although you may if that makes it easier to be clear. You* ***do not*** *need to justify the running time or the correctness.*]

The idea is to use getMaxOrMin and MIDDLE to implement COMPARE, and then to use a standard sorting algorithm.

The pseudo-pseudo code is:

```
a, b = getMaxAndMin(A)        // O(n) calls to MIDDLE

def COMPARE(x, y):
    z = MIDDLE(a, x, y)
    if z = x:
        return "x ≤ y"
    else:
        return "y ≤ x"
```

this is correct if a is the actual min. Otherwise, if a is the max, then COMPARE is backwards

Use MERGESORT with this version of COMPARE. We saw in class that MERGESORT makes $O(n \log n)$ calls to compare, hence this makes $O(n \log n)$ calls to MIDDLE.

If a was the min, this sorts A.
If a was the max, this sorts A in reverse order.

5 Harder problem (5 pts)

Note: This problem may be trickier and it is only worth 5 points. You might want to try the rest of the exam first.

5.1. **(5 pt.)** *(May be more difficult)*

Design an algorithm that does the following. Given an input array A of length n containing distinct positive integers, output an array B so that $B[i]$ contains the number of values $j \in \{0, ..., n-1\}$ so that $j > i$ and $3 \cdot A[j] > A[i]$.

For example, if the input were $A = [6, 2, 4, 1, 7]$, then the output would be $B = [2, 3, 1, 1, 0]$.

Your algorithm should run in time $O(n \log n)$. If it helps, you may assume that n is a power of 2.

[**We are expecting:** *Pseudocode, and a clear English description of what your algorithm is doing. You* ***do not*** *need to justify the running time or the correctness.*]

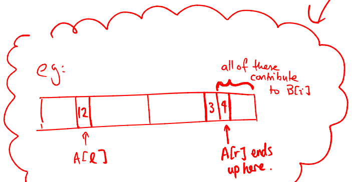
We will modify MERGESORT to build B. Before we MERGE, we update B to account for all the pairs $i < \frac{n}{2} \leq j$. We do a linear scan through the (now sorted) $A[:\frac{n}{2}]$ and $A[\frac{n}{2}:]$ to find, for each i, the last j s.t. $A[i] < 3A[j]$. Then we update the corresponding elt of B by the number of things in $A[n/2:]$ larger than that j.

```
def ACTUALANSWER(A):
    n = len(A)   \\ assume n is a power of 2
    A = [ (A[i], i) for i in range(n) ]   \\ Replace the elements of A to tag them w/ their orig. value.
                  (When we say "A[i] > A[j]", we now are using lexicographic order).
    B = [0 for i in range(n)]
    PROBLEMFIVE(A, B)
    return B

def PROBLEMFIVE(A, B):
    n = len(A)   \\ assume n is a power of 2
    PROBLEMFIVE( A[n/2 :], B[n/2 :] )   } \\ This recursively sorts both halves,
    PROBLEMFIVE( A[: n/2], B[: n/2] )   }    and fills in B to account for all the pairs
                                              i < j < n/2 and n/2 ≤ i < j
    \\ update B to account for the pairs i < n/2 ≤ j
    Initialize 2 pointers l = 0 and r = n/2.
    while l < n/2:
        while A[l] ≥ 3·A[r], r++   \\ increment r until the first place that A[l] < 3A[r]
        B[A[l][1]] += n − r   \\ account for the pairs (l, r), (l, r+1), ..., (l, n−1)
    \\ now sort A in-place
    MERGE( A[: n/2], A[n/2 :] )
```

eg

This is the end!



ANOTHER SOLN on NEXT PAGE!

This page intentionally blank for extra space for any question.
Please indicate in the relevant problem if you have work here that you want graded, and label your work clearly.

Alt. Soln for 5.1

Alternative Soln using a ModifiedRBTree (we accepted this even though we haven't seen how to do the modification).

```
B = [ ]
T = MRBTree from problem 4.1
for i in 0,...,n-1:
    T.insert(A[i])   // O(log n) per item
for i in 0,...,n-1:
    m ← getNumBiggerThan($\frac{A[i]}{3}$, T.root)   // Modify our soln from 4.1 to do this:
    B.append(m)
    T.remove(A[i])   // O(log(n))
```

This first puts all the items in an RBTree w/ the 4.1 modification.

Then for each i=0,...,n-1, it uses the RBTree to see how many elts are larger than A[i]/3, and that's what goes in B[i]; it then removes A[i] from T, so that A[i] won't show up in the count for any larger indices i' > i.

```
def numBiggerThan(x, r):
    If r == NIL:
        return 0
    If x > r:
        return numBiggerThan(x, r.right)
    If x == r:
        return r.right.sizeOfSubtree()
    If x < r:
        return r.right.sizeOfSubTree() + numBiggerThan(x, r.left)
```

This runs in time O(log n)