

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: _____

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) $\Theta(n^{\log_2 3})$ (e) $\Theta(n^3)$

1.3.1. (2 pt.) _____: $T(n) = 3T(n/2) + n^{3/2}$

1.3.2. (2 pt.) _____: $T(n) = T(n/2) + n$

1.3.3. (3 pt.) _____: $T(n) = T(n-1) + n^2$

1.3.4. (4 pt.) _____: $T(n) = T(n/2) + T(n/10) + T(n/20) + n$

1.3.5. (4 pt.) _____: $T(n) = T(n^{1/2}) + \log n$

- 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.) _____: Expected running time of QuickSort

1.4.2. (2 pt.) _____: Running time of InsertionSort

1.4.3. (2 pt.) _____: Time to search for an item in a Red-Black tree with n items

1.4.4. (2 pt.) _____: Running time of RadixSort with base $r = 10$, assuming all items are integers between 0 and n^{100}

1.4.5. (2 pt.) _____: 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.**

- 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) $O(n^2)$

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 $O(n)$.

-
- 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 $O(n)$.
- 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 $O(n)$.
- 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).

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.*]

More space for 3.1.1.

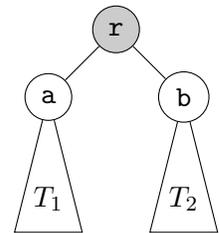
3.1.2. (2 pt.) Suppose that ISMAJORITY runs in time $O(\sqrt{n})$ on an array of length n . What is the big-Oh running time of `findMajority`?

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

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.]

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.*]

- 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 $\text{MIDDLE}(a, b, c)$ will return c . If you call the genie on the same item multiple times, for example, $\text{MIDDLE}(a, a, b)$, 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 $a < c < b$, $\text{MIDDLE}(a, b, c)$ could return 3; and $\text{MIDDLE}(a, a, b)$ 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.*]

Blank space for 4.2.1.

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.*]

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 $j \in \{0, \dots, n-1\}$ 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.*]

This is the end!

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.