CS161 Midterm Exam

Do not turn this page until you are instructed to do so!

Instructions: Solve all questions to the best of your abilities. You may cite any result we have seen in class or CLRS without proof. You have **80 minutes** to complete this exam. 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. 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: _____

Section	1 (mult. ch.)	2 (short answer)	3 (alg. analysis)	4 (alg. design)	5 (challenge)	Total
Score						
Maximum	32	28	15	25	5	105

1 Multiple Choice (32 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 you intend. **Ambiguous answers will be marked incorrect.**

1.1. **(5 pt.)** Which of the following expressions correctly describe $T(n) = n\log^2(n)$? Circle all that apply.

(A) $O(n)$ (B) $\Omega(n)$ (C) $\Theta(n^2)$ (D) $O(n^2)$ (E) $\Omega(n^2)$

1.2. **(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 $\sum_{i=0}^{t} i = \Theta(t^2)$. Assume that $T(1) = O(1)$ and don't worry about floors and ceilings.

(A) $\Theta(n)$ (B) $\Theta(n^2)$ (C) $\Theta(n\log n)$ (D) $\Theta(\log^2(n))$ (E) $\Theta\left(n^{\log_2 3}\right)$

1.2.1. **(2 pt.)** B: $T(n) = 3T(n/2) + n^2$

1.2.2. **(2 pt.)** C: $T(n) = 2T(n/2) + n$

1.2.3. **(3 pt.)** A: $T(n) = T(n-1) + 1$

1.2.4. **(4 pt.)** D: $T(n) = T(n/2) + \log(n)$

1.2.5. **(4 pt.)** A: $T(n) = T(n/2) + T(n/10) + n$

1.3. **(12 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.

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

1.3.1. **(2 pt.)** B: Worst-case running time of QuickSort

1.3.2. **(2 pt.)** C: Running time of MergeSort

1.3.3. **(2 pt.)** A: Running time of RadixSort with base $r = n$, assuming all items are integers between 0 and n^{100}

1.3.4. **(2 pt.)** B: Running time of RadixSort with base $r = n$, assuming all items are integers between 0 and n^n

1.3.5. **(2 pt.)** D: Time to insert an item into a red-black tree with n items

1.3.6. **(2 pt.)** E: Expected time to search for an item in a hash table with n items and n buckets, implemented with a universal hash family

2 Can it be done? (Short answers) (28 pts)

For each of the following tasks, either **explain briefly how you would accomplish it**, or else **explain why it cannot be done**. If you explain how to do it you do not need to justify why your answer is correct. You may cite any result we have seen in class.

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. **(7 pt.)** Give a deterministic algorithm which sorts an array A of length n containing arbitrary comparable elements in time $O(n)$.

This cannot be done. We saw in class that it takes time $\Omega(n \log(n))$ to sort arbitrary comparable items.

2.4. **(7 pt.)** Given an undirected, unweighted graph $G = (V, E)$ with n vertices and m edges and given $v \in V$, find a vertex $u \in V$ so that the number of edges on the shortest path from v to u is as large as possible, in time $O(n + m)$.

This can be done. Run BFS starting at v and return any of the vertices in the lowest layer of the BFS tree.

2.5. **(7 pt.)** Design a deterministic data structure which stores n arbitrary comparable items and supports the operations INSERT, DELETE, SEARCH and FINDMAX each in time $O(\log(n))$. Here, FINDMAX should return the item with the largest key. If you say that you can do this, make sure you explain how to implement FINDMAX.

This can be done. Use a red-black tree (which implements INSERT, SEARCH, DELETE), and to implement FINDMAX, choose the rightmost leaf in the tree.

2.6. **(7 pt.)** Design a deterministic data structure which stores n arbitrary comparable items and supports the operations INSERT, DELETE, SEARCH and FINDMAX each in time $O(1)$. Here, FINDMAX should return the item with the largest key. If you say that you can do this, make sure you explain how to implement FINDMAX.

This cannot be done. Suppose it could, and call the datastructure D. Then we could sort n arbitrary comparable items by inserting each item into D, and then calling FINDMAX and REMOVE n times. So then we could sort in time $O(n)$, a contradiction.

3 Algorithm Analysis (15 pts)

Consider the algorithm `findMaj` below, which runs on an array A of length n containing integers, where n is a power of 2. If A contains strictly more than $n/2$ copies of some integer x, then `findMaj(A)` returns x. If A does not contain strictly more than $n/2$ copies of any integer, then `findMaj(A)` can return whatever it wants.

In the pseudocode below, `findMaj` calls a special function called `isMaj(A,x)`. You many assume that `isMaj(A,x)` returns `True` if there are strictly more than $n/2$ copies of x in A, and `False` otherwise.

Algorithm 1: `findMaj`(A)

Input: An array A of length n, where n is a power of 2
Output: An integer x so that x appears strictly more than $n/2$ times in A, if such an element exists. Otherwise, the output can be anything.

```
if n = 1 then
    return A[0]
a = findMaj(A[: n/2])
b = findMaj(A[n/2 :])
for x ∈ [a, b] do
    if isMaj(A, x) then
        return x
return A[0]
```

3.1. **(5 pt.)** Suppose that `isMaj` runs in time $O(\sqrt{n})$. (It's okay if that seems impossible; suppose that `isMaj` works by asking a genie who takes time $O(\sqrt{n})$). What is the running time of `findMaj`?

[**We are expecting:** *A recurrence relation that captures the running time of* ***findMaj****, the best statement you can make of the form "The running time of* ***findMaj****(A) is $O(\_\_\_)$," and a formal proof that your recurrence relation implies your $O(\_\_\_)$ bound. You may use the Master Theorem if it applies.*]

The running time of findMaj is $O(n)$.
This is because the running time satisfies

$$T(n) = 2T(n/2) + O(\sqrt{n})$$

So $T(n) = O(n)$ by the Master Thm. More precisely, we have $a=2$, $b=2$, $d=1/2$, so $b^d = \sqrt{2} < 2 = a$, so the running time is $T(n) = O(n^{\log_b(a)}) = O(n)$.

[Another part on next page]

3.2. **(10 pt.)** Prove formally, by induction, that `findMaj` is correct. (That is, that it correctly finds a majority element in an array A whose length is a power of 2, if such an element exists). You may assume that `isMaj` is correct.

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

INDUCTIVE HYPOTHESIS If there is a majority element in an array of length 2^t, then findMaj returns it.

BASE CASE If $t=0$, then any array A of length $2^0=1$ has a majority element, $A[0]$, which is what findMaj returns.

INDUCTIVE STEP Suppose that I.H. holds for $t=k-1$. We will show that it holds for $t=k$.

Let A be an array of length 2^k, and suppose x is majority elt.

Let $L=A[:n/2]$, $R=A[n/2:]$.

Then x must be a majority in at least one of L or R.

Otherwise, the number of copies of x in A is at most $\frac{\text{len}(L)}{2}+\frac{\text{len}(R)}{2}=\frac{n}{4}+\frac{n}{4}=\frac{n}{2}$, which contradicts x being a majority element.

Then by induction, at least one of a or b in the pseudocode is equal to x.

Since isMaj works correctly, it will identify the correct item and return it.

CONCLUSION The I.H. holds for all t. In particular, findMaj correctly finds a majority elt (if it exists) in any array whose length is a power of 2.

4 Algorithm Design (25 pts)

In this section you will design an algorithm to sort an array A of length n that contains arbitrary comparable elements with distinct keys. However, the only way you can interact with an input array A is by the following operations:

- `length(A)`: returns the length of A. **This operation costs one dollar.**
- `smallerThan(A,i,j)` (for $i \neq j$): returns `True` if $A[i] < A[j]$ and `False` if $A[i] > A[j]$. Since all elements in A have distinct keys these are the only two choices. **This operation costs one dollar.**
- `flip(A,i,j)`: flips the sub-array $A[i:j]$ in-place (assuming $i < j$; otherwise it does nothing). For example:

$$A = [a_0, \underbrace{a_1, a_2, a_3, a_4}_{\text{will get flipped}}, a_5] \xrightarrow{\texttt{flip(A,1,5)}} A = [a_0, \underbrace{a_4, a_3, a_2, a_1}_{\text{got flipped}}, a_5]$$

  (Notice that we are using Python notation, so A is zero-indexed, and $A[i:j]$ includes the elements $A[i], A[i+1], \ldots, A[j-1]$). **This operation costs $|i-j|$ dollars.**

Other than that, you have all the space you need to store auxiliary information which you can manipulate for free, but you cannot copy elements of A or interact with A in any other way. (You may pass slices of A like $A[i:j]$ into subroutines if it is convenient.)

[Questions start on next page]

4.1. **(15 pt.)** Design a version of the `partition` function which works **in the model described on page 7** and costs $O(n \log(n))$ dollars. More precisely, `partition` should be a deterministic algorithm which runs on an array A of length n and an index $i < n$ so that:

- After `partition(A,i)` has been run, A should be of the form $L \circ A[i] \circ R$, where L contains all of the elements of A smaller than $A[i]$ and R contains all of the elements of A larger than $A[i]$. Above, $\circ$ denotes concatenation.
- `partition(A,i)` returns the new index of $A[i]$ in the array $L \circ A[i] \circ R$.

For example, `partition(A,2)` could do:

$$A = [4, 7, 5, 1, 0, 6] \xrightarrow{\texttt{partition(A,2)}} A = [\underbrace{0, 4, 1}_{L}, 5, \underbrace{7, 6}_{R}]$$

$A[2] = 5$ — Now 5 is in position 3, so return 3

(Hint: Try divide-and-conquer.)

[**We are expecting:** *Pseudocode **AND** a clear English description. (You may draw pictures as part of your description if it helps). You do not need to justify the cost or the correctness of your algorithm.*]

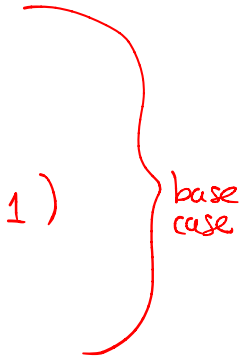
(on next page)

[More space on next page]

[More space for Problem 4.1.]

```
def partition(A,i):
    if len(A) = 1:
        return 0
    if len(A) = 2:
        if not isSmallerThan(A, 0, 1)
            flip(A, 0, 1)
            return 1-i
        return i
    p = ⌊len(A)/2⌋
    if i<p, flip(A,i,p+1)
    else, flip(A, p, i+1)
    l = partition(A[:p+1], p)
    flip(A, l, p+1)
    r = partition(A[p:], 0)
    flip(A, p, p+r+1)
    flip(A, l, p+r+1)
    return l+r
```

Picture



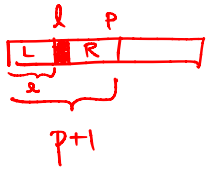
English description

If len(A) = 1, then we are done

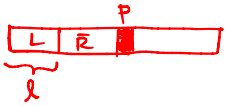
If len(A) = 2, then sort A using flip, and then return where i ends up.

← Start w/ pivot in i^{th} pos.

← first flip so that A[i] is in the middle



← then recurse on the left half



← then flip stuff so that the pivot is back in the middle

← then recurse on the right half

← and flip the pivot back to the middle.

← Now flip once more to put the smaller things on the left.

NOTE: There are many correct ways to do this problem!

4.2. **(10 pt.)** Design a *randomized* algorithm which sorts A from smallest to largest **in the model described on page 7** using $O(n \log^2(n))$ dollars in expectation. You may use the `partition` function from problem 4.1., even if you did not complete that problem.

(Hint: Think about QuickSort.)

[**We are expecting:** *Pseudocode* ***AND*** *a clear English description. You do not need to justify the cost or the correctness of your algorithm.*]

```
def sort(A):
    n = len(A)
    if n = 1:
        return
    i = random element in {0, ..., n-1}
    j = partition(A, i)
    sort(A[:j])
    sort(A[j+1:])
    return
```

English description: this is just like QuickSort. We partition around a random pivot and then recursively sort both halves.

5 Harder problem

Note: This problem is probably the hardest problem on the exam, and it is only worth 5 points. You might want to do the rest of the exam first.

5.1. **(5 pt.)** *(May be more difficult)* Design a randomized data structure which operates on items x from a universe $\mathcal{U}$ and which supports operations `insert`(x) and `isFrequent`(x) which satisfy the following properties. Suppose that `insert` is called exactly t times. Then, for $x \in \mathcal{U}$,

- if `insert`(x) has been called more than $9t/10$ times, then `isFrequent`(x) returns `True` with probability 1.
- if `insert`(x) has been called fewer than $t/10$ times, then `isFrequent`(x) returns `False` with probability at least 4/5.

The data structure should use $O(\log(|\mathcal{U}|)+\log(t))$ bits of space. Both `insert` and `isFrequent` should run in worst-case time $O(1)$.

[**We are expecting:** *A description of your data structure and how you would implement* ***insert*** *and* ***isFrequent****. You do not need to prove your data structure has the desired properties.*]

Initialize:
- $t = 0$
- An array B of 100 integers to 0
- A hash function h chosen from a universal hash family, $h: \mathcal{U} \to \{0, \dots, 99\}$, of size $|\mathcal{H}| = |\mathcal{U}|^2$

B will store integers up to size t, so that's $O(\log(t))$ bits
h takes $O(\log|\mathcal{U}|)$ bits to store, for a total of $O(\log|\mathcal{U}| + \log t)$.

INSERT(x):
  $B[h(x)]$++
  t++

ISFREQUENT(x):
  If $B[h(x)] \geq \frac{9t}{10}$:
    return TRUE
  Else
    return FALSE

You don't have to prove correctness for this problem, but here's the idea:
- If x was inserted $\geq 9t/10$ times, then $B[h(x)] \geq 9t/10$, so that's fine.
- If x was inserted $\leq t/10$ times,

$$\mathbb{E}[B[h(x)]] = \mathbb{E}\left[f(x) + \sum_{y \neq x} \mathbb{I}\{h(x)=h(y)\} \cdot f(y)\right] \quad \text{where } f(y) = \#\text{times } y \text{ was inserted.}$$
$$= f(x) + \sum_{y \neq x} \mathbb{P}\{h(x)=h(y)\} \cdot f(y)$$
$$\leq f(x) + \frac{1}{100}\sum_{y \neq x} f(y) \leq \frac{t}{10} + \frac{t}{100}$$

Then by Markov's inequality, **This is the end!**

$$\mathbb{P}\left\{B[h(x)] > \frac{9t}{10}\right\} \leq \frac{t/10 + t/100}{9t/10} = \frac{1+1/10}{9} < \frac{1}{5}$$

So $\mathbb{P}\{B[h(x)] \leq 9t/10\} \geq 4/5$, as desired.

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.