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. Please pay attention to the instructions at the beginning of each section, which provide guidance about the sort of answer we are expecting. Make sure to look at all pages. You may cite any result we have seen in class or CLRS (or in any resource linked from the course website, except Piazza) 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 that you may tear off as scratch paper, and one blank page for extra work. Please write your name at the top of all pages.

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: _____

Question	#1	#2	#3	#4	#5	Total
Score						
Maximum	20	20	10	20	30	100

1 True or False (20 points)

Please answer all questions in this section True or False. No explanation is required.

1.1. (2 pts) A function can be both $O(n)$ and $O(n^2)$.

True **False**

In fact, anything that is $O(n)$ is also $O(n^2)$.

1.2. (2 pts) RadixSort is a comparison-based sorting algorithm.

True **False**

RadixSort needs to use more than just comparisons to sort its input.

1.3. (2 pts) Computing the median of n elements requires $\Omega(n \log(n))$ time for any comparison-based algorithm.

True **False**

The SELECT algorithm that we saw in class will do it in time $O(n)$.

1.4. (2 pts) Consider the following task: given n bits $b_1, \ldots, b_n \in \{0, 1\}$, decide if the number of b_i so that $b_i = 1$ is even or odd. This task requires time $\Omega(n)$.

True **False**

We need to look at every bit b_i in order to compute the parity; just looking at the input takes $\Omega(n)$.

1.5. (2 pts) It takes time $O(1)$ to sort an array of 100 elements.

True **False**

For example, MERGESORT runs in time $O(100 \log(100)) = O(1)$.

1.6. (2 pts) The Master Theorem applies to the recurrence $T(n) = T(n/2) + T(n/10) + 7$.

True **False**

The Master Thm only applies to equal-sized sub-problems.

1.7. (2 pts) Consider the recurrence relation $T(n) = T(n/2) + T(n/20) + O(1)$, where $T(m) = 1$ for all $m \leq 20$. Then $T(n) = \Omega(n^2)$.

True **False**

The actual recurrence might be tricky to solve, but we can see $T(n) \leq 2T(n/2) + O(1) = O(n)$ by the master thm, so it can't be $\Omega(n^2)$.

1.8. (2 pts) Consider the recurrence relation $T(n) = 2T(n/2) + \sqrt{n}$, where $T(m) = 1$ for all $m \leq 2$. Then $T(n) = \Theta(n)$.

True **False**

The upper bound $T(n) = O(n)$ follows from the Master Thm. The lower bound $T(n) = \Omega(n)$ is because the recursion tree has n leaves, so $T(n) \geq n \cdot T(1) = \Omega(n)$.

1.9. (2 pts) A Red-Black Tree on n nodes always has depth $O(\log(n))$.

True **False**

of length n (written on board during exam).

We proved this in class.

1.10. (2 pts) For a given input array A, there is a nonzero probability that the running time of QuickSort with randomly chosen pivots will be $\Omega(n^2)$. However, there is **no** input array A on which QuickSort with randomly chosen pivots will always (with probability 1) take time $\Omega(n^2)$.

True **False**

For any fixed choice of pivots, there is some worst-case array that will make the running time $\Omega(n^2)$. This implies the first sentence. However, which array that is depends on the choice of the pivots. If the array is picked first, before the pivots, then all input arrays have expected runtime $O(n \log(n))$. This implies the 2nd sentence.

2 Short answers (20 points)

For all of the problems in this section, please answer with **at most a few sentences.**

2.1. (5 pts) **(QuickSort v. MergeSort)** QuickSort is very efficient in practice, and has an expected running time $O(n \log(n))$, the same as that of MergeSort. Why would anyone ever use MergeSort over QuickSort?

One reason is that MergeSort is deterministic, so it will ALWAYS run in time $O(n \log(n))$. On the other hand, there's some small probability that Quicksort will take time $\Omega(n^2)$.

2.2. (5 pts) **(MergeSort v. RadixSort)** RadixSort runs in time $O(n)$, while MergeSort requires time $\Omega(n \log(n))$. Why would anyone ever use MergeSort over RadixSort?

Note: RadixSort does NOT necessarily use a huge amount of space. We choose $r = \Theta(n)$, so the space is always $\Theta(n)$. The thing that might get worse is the running time.

- One reason is that MergeSort is a comparison-based algorithm, while RadixSort requires some assumptions about the input. More precisely, RadixSort assumes that the inputs have a canonical base-r representation (for example, they are integers).
- Another reason is that RadixSort is only $O(n)$ time if it contains elts that can be written in length $O(1)$ base-n: for example, it would be slower on n integers chosen in $\{1, \ldots, 2^n\}$.

- Yet another correct solution: MergeSort can be done in-place, but it's not clear how to do RadixSort in-place.

2.3. (5 pts) **(BFS and DFS)** Give one application of Breadth-First Search and one different application of Depth-First Search.

BFS can be used for finding shortest paths in unweighted graphs, and can also be used for testing bipartiteness.

DFS can be used for topological sorting and finding strongly connected components.

2.4. (5 pts) **(Hash families)** Let $h : \{1, \ldots, 10\} \to \{0, 1\}$ be the function $h(x) = x \bmod 2$. Your friend claims that $\mathcal{H} = \{h\}$ is a one-element universal hash family, which hashes a 10-element universe into $n = 2$ buckets. They give the following reasoning. Suppose that x and y are drawn independently and uniformly at random from $\{1, \ldots, 10\}$. Then the probability that $h(x) = h(y)$ is $1/2$. Thus, $\Pr\{h(x) = h(y)\} \leq 1/n$, which is the definition of a universal hash family. Your friend has made a big conceptual error: what is it?

Your friend thinks that the probability in "$\Pr\{h(x) = h(y)\} \leq \frac{1}{n}$" is over the random choice of x and y. In fact, it's over the choice of the hash function $h \in \mathcal{H}$.

3 Fun with big-O (10 points)

3.1. (5 points) Prove formally, using the definition of asymptotic notation that we saw in class, that if $f(n) = n$ and $g(n) = n^2$, then $f(n) = O(g(n))$.

Let $c = 1$ and $n_0 = 1$.
Then for all $n \geq 1$, $n^2 \geq n \geq 0$, aka
for all $n \geq n_0$, $0 \leq f(n) \leq c \cdot g(n)$,
which is the definition of $f(n) = O(g(n))$.

3.2. (5 points) Consider the following claim:

If $f(n) = \Omega(g(n))$, then $2^{f(n)} = \Omega(2^{g(n)})$.

This claim is **false**. Give a counter-example to show that it is false. You do not need to formally prove that your counter-example is a counter-example.

Choose $f(n) = \log(n)$, $g(n) = 2\log(n)$.
Then $f(n) = \Omega(g(n))$, since $\log(n) = \Omega(2\log(n))$.
But $2^{f(n)} = n$, and $2^{g(n)} = 2^{2\log(n)} = n^2$,
and $n \neq \Omega(n^2)$, so $2^{f(n)}$ is NOT $\Omega(2^{g(n)})$.

4 Algorithm Design (20 points)

Suppose that A and B are two **sorted** arrays of comparable items of length n. A has distinct elements, and B has distinct elements, but there may be elements that are in both A and B. Give a deterministic algorithm with worst-case runtime $O(n)$ that finds the *intersection* of A and B — that is, your algorithm should return all of the elements that are in both A and B.

You should write pseudocode and a (short, high-level) English description of what your algorithm does. **You do not need to prove that it is correct, or analyze the running time.**

Here are two other algorithms that are NOT correct (for this problem):

```
ret = ∅
Let C be A concatenated w/ B
RadixSort(C)
for i=1,...,2n:
    if C[i]=C[i+1]:
        ret.append(i)
return ret
```

- and -

```
ret = ∅
H = hashTable()
for a ∈ A:
    H.insert(a)
for b ∈ B:
    if b ∈ H:
        ret.append(b)
return ret
```

The first algorithm assumes something about the inputs to do RadixSort; but the problem specifies comparable items only. The second alg. is either not deterministic or not time $O(n)$, depending on how H is implemented.

```
ret = ∅
i = 1
j = 1
while i ≤ n and j ≤ n:
    if A[i] == B[j]:
        ret.append(A[i])
        i++
        j++
    elseif A[i] < B[j]:
        i++
    elseif A[i] > B[j]:
        j++
return ret
```

EXAMPLE

A = [1 2 3 4] B = [2 4 6 8]
i = 1, j = 2

i++

A = [1 2 3 4] B = [2 4 6 8]
i = 2, j = 2

ret = [2], i++, j++

A = [1 2 3 4] B = [2 4 6 8]
i = 3, j = 3

i++

A = [1 2 3 4] B = [2 4 6 8]
i = 4, j = 3

ret = [2,4], i++, j++
(terminate since i > n.)

This algorithm walks two pointers down A and B together, and increments whichever pointer is behind to catch up. If the 2 ptrs ever point to the same item, add it to the intersection.

5 Algorithm Analysis (30 points)

Suppose that A is a **sorted** array of **distinct integers.** A *fixed point* of an array is an index $i \in \{1, \ldots, n\}$ so that $A[i] = i$. For example, in the array $A =$`[-2,-1,3,5,7]`, $A[3] = 3$, so 3 is a fixed point. In the array $B =$`[2,3,4,5,6]`, there are no fixed points.

The goal of the following algorithm is to return `True` if there is a fixed point, and to return `False` otherwise.

```
isThereAFixedPoint(A):
    return isThereAFixedPoint_helper(A, 1, n)

isThereAFixedPoint_helper(A, lower, upper):
    mid = (lower + upper)/2
    if A[mid] == mid:
        return True
    if lower == upper:
        return False
    if A[mid] > mid:
        return isThereAFixedPoint_helper(A, lower, mid)
    if A[mid] < mid:
        return isThereAFixedPoint_helper(A, mid+1, upper)
```

Above, all arithmetic is integer arithmetic: that is, `3/2` rounds down to `1`.

5.1. (3 pts) Suppose the pseudocode above runs on the input array `[-2,-1,2,4,7]`. What are all of the calls to `isThereAFixedPoint_helper`?

1. Call w/ lower = 1, upper = 5

mid = $\frac{5+1}{2} = 3$

A[mid] = 2 < mid, so

2. Call w/ lower = 4, upper = 5

mid = $\frac{4+5}{2} = 4$

A[mid] = 4 = mid, so

RETURN TRUE.

So there are 2 calls:

isThereAFixedPoint_helper(A, 1, 5)

and

isThereAFixedPoint_helper(A, 4, 5)

[More parts on next page]

Algorithm analysis continued.

5.2. (10 pts) Analyze the worst-case runtime of `isThereAFixedPoint`. We are looking for a statement of the form "the worst-case running time of `isThereAFixedPoint` on an input of size n is $O(\_\_\_\_)$," along with justification for why this is correct.

Your answer should be the strongest statement you can make (that is, a bound of $O(2^{2^n})$ may be true but will not receive credit), although you do not have to prove this.

The algorithm satisfies the recurrence relation

$$T(n) = T(n/2) + O(1),$$

because in each call to isThereAFixedPoint_helper, we recurse on a set of size about $n/2$, and then there is $O(1)$ overhead to increment the pointers. By the Master Theorem, this means

$$T(n) = O(\log(n)).$$

as in class, it is fine on this problem to ignore floors and ceilings in the recurrence relation.

see page 10 for a different correct way to set this up

5.3. (10 pts) Suppose you were to argue by induction that `isThereAFixedPoint` is correct: that is, it returns `True` if and only if there is some $i \in \{1, \ldots, n\}$ so that $A[i] = i$. Lay out the high level overview of the argument: what inductive hypothesis would you use, what is the base case, what needs to be shown for the inductive step, and what is the conclusion? **You should prove the base case and the "conclusion/termination" step. You do not need to prove the inductive step in this part.**

We will do induction on the difference (upper-lower), with the following inductive hypothesis:

Inductive Hypothesis: If upper - lower $\leq t$, then isThereAFixedPt_helper(A, lower, upper) returns True iff there is a fixed point in A[lower .. upper] (inclusive).

Base Case: When upper - lower = 0, then upper = lower, so

$$\text{mid} = \frac{\text{upper} + \text{lower}}{2} = \frac{2 \cdot \text{upper}}{2} = \text{upper}.$$

Thus, the pseudocode reads:

```
If A[mid] == mid:
    return True
return False
```

Aka, this returns True iff there is a fixed pt btwn lower & upper.

[More space on next page]

Algorithm analysis continued.

(Extra space for Question 5.3 if needed).

For the inductive step, we need to show that if the inductive hypothesis holds for t, then it holds for $t+1$.

In more detail, we need to show that, assuming the recursive calls work (that is, return TRUE iff A[lower .. mid] has a fixed pt or A[mid+1..upper] has a fixed pt, respectively) then isThereAFixedPt_helper will return TRUE iff there is a fixed pt in A[lower .. upper].

Conclusion

When $t = n-1$ the inductive hypothesis reads:

"If upper - lower $\leq n-1$, then isThereAFixedPt_helper(A, lower, upper) returns True iff there is a fixed point in A[lower .. upper]."

In particular, we may apply this to the special case when lower = 1, upper = n, and we see:

isThereAFixedPt_helper(A, 1, n) returns TRUE iff there is a fixed pt in A[1 .. n] = A.

aka the algorithm is correct.

[Another part on next page]

Algorithm analysis continued.

5.4. (7 pts) Prove the inductive step in the outline you laid out above. If your proof breaks into two cases where the proof is basically the same, you may prove the result in only one case and write "the other case is basically the same."

Suppose that, if upper − lower ≤ t, then iTAFP_helper will return TRUE if and only if there is a fixed point in A[lower..upper].

introducing an abbreviation for isThereAFixedPoint_helper

We want to show the same is true for t+1. Suppose upper − lower = t+1.

Now, suppose A[lower..upper] has NO fixed point. Then iTAFP_helper will not return TRUE in line 4, and there are no fixed pts in A[lower..mid] or A[mid+1,..upper], so by induction neither of the recursive calls will return TRUE either. Thus this call will return FALSE.

On the other hand, suppose there is a fixed pt in A[lower..upper]. Then one of 3 things can happen:

CASE 1 A[mid] = mid. Then iTAFP_helper returns TRUE in line 4.

CASE 2 A[mid] < mid. Then we call iTAFP_helper(A, mid+1, upper).

CLAIM There is no fixed point in A[lower ... mid].

You did not need to prove this CLAIM formally to receive full credit: if you went with this proof strategy, it was enough to make the claim and then argue for it informally but convincingly.

This is true because the array holds distinct integers. Formally, for any $p \in \{lower, \ldots, mid\}$, we have $A[p] < A[p+1] < \cdots < A[mid]$, so

$$A[mid] - A[p] = (A[mid] - A[mid-1]) + (A[mid-1] - A[mid-2]) + \cdots + (A[p+1] - A[p]) \quad \text{(This is } \sum_{j=p+1}^{mid} (A[j] - A[j-1])\text{)}$$
$$\geq 1 + 1 + \cdots + 1$$
$$= mid - p, \quad \text{since that's how many terms there are.}$$

So then $A[mid] - A[p] \geq mid - p$.

$mid - A[p] > mid - p$ since $A[mid] < mid$

$p > A[p]$

So p is not a fixed point.

This is the end!

But since there is a fixed pt in A[lower..upper], and it's not in A[lower..mid], then it must be in A[mid+1..upper]. So by induction, iTAFP_helper(A, mid+1, upper) will return TRUE.

CASE 3 A[mid] < mid is basically the same.

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.

Here is another correct solution to 5.3:

Inductive hypothesis:

At level t in the recursion tree,

$A[1..n]$ has a fixed pt $\iff$ $A[lower..upper]$ has a fixed pt.

(aka, we maintain the recursive invariant that the fixed point is in our current range, if there is a fixed point.)

Base Case

At the top level of the recursion tree, $A[lower..upper] = A[1..n]$, so this is a tautology.

Inductive Step We need to show that if the fixed point is in $A[lower..upper]$ at the beginning of a iTAFP_helper call, then it will still be in $A[lower..upper]$ in the next recursive call.

(If there was no fixed pt, obviously it won't be in the next recursive call).

Termination We conclude by induction that when lower = upper,

A has a fixed point $\iff$ $A[lower..upper]$ has a fixed pt

$\iff$ $A[mid] = mid$ (since lower = upper = mid)

$\iff$ We return TRUE (from the pseudocode)

Thus, the algorithm is correct.

The proof of this inductive step is similar to the proof of the alternative one on the previous page.

This page is for **scratch space**. You may tear off this page. Nothing on this page will be graded.

This page is for **scratch space**. You may tear off this page. Nothing on this page will be graded.