CS161 Practice Midterm

One you turn this page, your (practice) 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.

  Note: for the practice exam, of course feel free to ask; but for the real exam you should follow the guidance above.

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 (30 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. (9 pt.) Below, assume that all functions \( f(n) \) map positive integers to positive integers. Which of the following functions \( f(n) \) are \( O(n^2) \)? Circle all that apply.

(A) \( f(n) = 50n \)
(B) \( f(n) = n \log n + n^2 \)
(C) Any \( f(n) \) that satisfies \( f(n) = \Omega(n^2 \log n) \)
(D) Any \( f(n) \) that satisfies the recurrence \( f(n) \leq 2 \cdot f([n/4]) + n^2 \) for \( n \geq 4 \).
(E) Any \( f(n) \) that satisfies the recurrence \( f(n) \leq f(n-1) + n^2 \) for \( n \geq 2 \).
(F) Any \( f(n) \) that satisfies the recurrence \( f(n) \leq f([n/2]) + f([n/10]) + n^2 \) for \( n \geq 10 \).

1.2. (7 pt.) Which of the following functions \( f(n) \) are \( O(n \log n) \)? Circle all that apply.

(A) Any \( f(n) \) that satisfies \( f(n) = O(n) \)
(B) Any \( f(n) \) that satisfies \( f(n) = \Omega(n) \)
(C) Any \( f(n) \) that satisfies the recurrence \( f(n) \leq 5 \cdot f(n/5) + n \) for \( n \geq 5 \).
(D) Any \( f(n) \) that satisfies the recurrence \( f(n) \leq 6 \cdot f(n/3) + n \log n \) for \( n \geq 5 \).
(E) The worst case runtime of Quicksort on an array of length \( n \).
(F) The worst case runtime of Mergesort on an array of length \( n \).
(G) The expected runtime of Quicksort on an array of length \( n \).

1.3. (5 pt.) Which of the following correctly describes \( T(n) = 3T(n/2) + n \)? Circle all that apply.

(A) \( \Omega(n) \)
(B) \( \Theta(n) \)
(C) \( O(n \log \log 2) \)
(D) \( O(n^2) \)
(E) \( O(n \log n) \)
1.4. (4 pt.) Draw a Red-Black tree containing the elements 1, 2, 3, 4, 5, 6. Use square boxes to denote the red nodes and circles to denote the black nodes. *Note: technically not multiple choice, but still good practice...*

![Red-Black tree diagram]

1.5. (5 pt.) Which of the following can you sort in time \( O(n) \) using RadixSort? Circle all that apply.

1.5.1. \( n \) infinite precision real numbers between 0 and 1.
1.5.2. \( n \) integers between 1 and 10.
1.5.3. \( n \) integers between 1 and \( n^{100} \).
1.5.4. \( n \) integers between 1 and \( 2^n \).
1.5.5. \( n \) items that you can only access via comparisons.

2. Can it be done? (Short answers) (25 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.

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. (5 pt.) Find the \( \lceil \sqrt{n} \rceil \)’th smallest element of an \( n \)-element array in time \( O(n) \).

Use the SELECT algorithm we saw in class with \( \frac{p}{r} = \lceil \sqrt{n} \rceil \).

2.4. (5 pt.) Say that an array \( A \) of odd length \( n = 2r + 1 \) is oscillating if \( A[0] \leq A[1] \geq A[2] \leq \cdots \leq A[2r - 1] \geq A[2r] \). Given an unsorted array \( B \) containing \( n = 2r + 1 \) distinct comparable items, output an oscillating array \( A \) that has all the same elements as \( B \), in time \( O(n) \).

\[
\begin{align*}
\text{Let } & p = \text{SELECT}(A, r+1) & \text{// } p \text{ is the location of the median} \\
& L, R = \text{PARTITION}(A, p) & \text{// } L \text{ contains the } r \text{ things } < A[p], \ R \text{ contains the } r \text{ things } > A[p]. \\
\text{ret} = [] \\
\text{for } i = 1, \ldots, r: \\
& \text{ret} = \text{ret} + (L[i]) \\
& \text{ret} = \text{ret} + (R[i]) \\
& \text{return ret}
\end{align*}
\]

2.5. (5 pt.) Give an algorithm which finds the maximum element in any BST \( T \) holding \( n \) elements in time \( O(\log n) \).

This cannot be done. In the worst case, the BST might look like this:

and it would take time \( n \) to follow the pointers down to the maximum.
2.6. (5 pt.) Given an array $A$ of length $n$ which contains only integers between 1 and $n$, sort $A$ in time $O(n)$.

Use Radix Sort.

2.7. (5 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.

Use a Red Black tree. To implement FindMax, find the rightmost element in the tree.

2.8. (5 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. If it could, then we could use this data structure to sort arbitrary comparable items in time $O(n)$:

```
def SORT(array A of length n):
    for x in A:
        INSERT(x)
    ret = []
    for i = 1, ..., n:
        x = FINDMAX()
        ret.append(x)
        REMOVE(x)
    ret.reverse()
    return ret
```

However, we saw in class that this is impossible.
3 Algorithm Analysis / Proving Stuff (20 pts)

3.1. (6 pt.) Consider the statement below:

**Statement:** Suppose that \( f \) and \( g \) are increasing functions defined on the integers. If \( f(n) = O(g(n)) \), then \( 2^{f(n)} = O(2^{g(n)}) \).

The statement is **FALSE**. In the following parts, you will prove this.

(a) (2 pt.) Give functions \( f \) and \( g \) that are a counter-example to the statement.

Let \( f(n) = n \), \( g(n) = 2n \).

(b) (2 pt.) Prove, using the definition of big-Oh, that \( f(n) = O(g(n)) \) for your example.

Choose \( n_0 = 1 \), \( c = 2 \).

Then for all \( n \geq n_0 \), we have

\[
2n = 2 \cdot n \leq 2 \cdot 2^n
\]

so in particular \( f(n) \leq 2 \cdot g(n) \).

(c) (2 pt.) Prove, using the definition of big-Oh, that \( 2^{f(n)} \) is not \( O(2^{g(n)}) \) for your example.

Suppose that \( 2^{f(n)} = O(2^{g(n)}) \), aka \( 2^{2n} = O(2^n) \).

Then there is some \( c, n_0 \) so that for all \( n \geq n_0 \),

\[
2^{2n} \leq c \cdot 2^n
\]

\[
2n \leq \log(c) + n
\]

\[
n \leq \log(c)
\]

But consider \( n^* = \log(c) + n_0 + 1 \). Then \( n^* \geq n_0 \)

but \( n^* > \log(c) \), a contradiction.

Thus \( 2^{f(n)} \) is **NOT** \( O(2^{g(n)}) \).
3.2. (9 pt.) Consider the following recursive algorithm, which recursively finds the minimum element in the array of length \( n \) which is a power of 2:

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

3.2.1. (3 pt.) What is the running time of \( \text{findMin} \)? As part of your answer, write down a recurrence relation that the running time of \( \text{findMin} \) satisfies.

[We are expecting: Your answer, along with a recurrence relation.]

The running time of \( \text{findMin} \) satisfies

\[
T(n) \leq 2 \cdot T(n/2) + 1.
\]

By the Master theorem, \( T(n) = O(n) \).

3.2.2. (6 pt.) Prove by induction that \( \text{findMin} \) correctly returns the minimum of an array.

[We are expecting: A formal proof by induction.]

**Inductive Hypothesis**: \( \text{findMin} \) returns the min of an array of length \( 2^k \)

**Base case**: When \( k=0 \), \( \text{findMin} \) has an array of length 1, and it returns the only elt, which is the minimum.

**Inductive Step**: Suppose that the IH holds for \( i=k-1 \).

Then \( a = \min(A[::n/2]) \) and \( b = \min(A[n/2:]) \).

Since the min of \( A \) occurs either in \( A[::n/2] \) or \( A[n/2:] \), this means that the min of \( A \) is equal to \( \min\{a, b\} \). This is what \( \text{findMin} \) returns, so the IH holds for \( i=k \).

**Conclusion**: The IH holds for all \( k \geq 0 \), which means that \( \text{findMin} \) correctly returns the min of any array whose length is a power of 2.
3.3. (5 pt.) Consider the following randomized algorithm for finding the minimum of an array.

```python
def findMinRandomized(A):
    while True:
        i = random.randint(0, len(A) - 1)
        candidate = A[i]
        # check to see if the candidate is actually the minimum:
        isMin = True
        for j in range(len(A)):
            if candidate > A[j]:
                isMin = False
                break
        if isMin:
            return candidate
```

3.3.1. (3 pt.) What is the expected running time of `findMinRandomized` on an array of size `n`?

[We are expecting: Your answer, along with a justification. Show any work.]

The probability that candidate is the true min is \( \frac{1}{n} \), so the expected number of iterations of the while loop is \( n \). In each iteration, there is \( O(n) \) work to check the min, so the running time is \( O(n^2) \) in expectation.

3.3.2. (2 pt.) What is the worst-case running time of `findMinRandomized` on an array of size `n`?

[We are expecting: Your answer, along with a justification. Show any work.]

The worst-case running time is infinite, since it could be that the algorithm never chooses the minimum.
4 Algorithm Design (30 pts)

4.1. (15 pt.) Suppose that $A$ and $B$ are sorted arrays that each have $n$ integers. Design an algorithm to find the median of the $2n$ integers in $O(\log n)$ time. Note that since the total number of integers is even, your algorithm can return either the $n$th or $(n+1)$th smallest integer in the array.

For example, if $A = [2, 6, 8, 9, 12]$ and $B = [1, 2, 4, 10, 15]$, then your algorithm could return either 6 or 8.

[We are expecting: Pseudocode and a clear English explanation of what your algorithm is doing.]

The basic idea is as follows:

First, we compute the $A_\left\lfloor \frac{n}{2} \right\rfloor$ to $B_\left\lfloor \frac{n}{2} \right\rfloor$. If $A_\left\lfloor \frac{n}{2} \right\rfloor < B_\left\lfloor \frac{n}{2} \right\rfloor$, then the median of $A, B$ is equal to the median of $A_\left\lceil \frac{n}{2} \right\rceil, B_\left\lceil \frac{n}{2} \right\rceil$, so we recurse there, and similarly if $A_\left\lfloor \frac{n}{2} \right\rfloor > B_\left\lfloor \frac{n}{2} \right\rfloor$ we recurse on $A_\left\lfloor \frac{n}{2} \right\rfloor, B_\left\lfloor \frac{n}{2} \right\rfloor$.

```python
def findMedian(A, B):
    n = len(A)
    if n == 1:
        return A[0]  # doesn't matter
    if n is odd:
        a = A[n//2]
        b = B[n//2]
        if a < b:
            return findMedian(A[:n//2], B[:n//2])
        else:
            return findMedian(A[n//2:], B[n//2:])
    else if n is even:
        b = (B[n//2] + B[n//2-1])/2
        if a < b:
            return findMedian(A[:n//2], B[:n//2])
        else:
            return findMedian(A[n//2:], B[n//2:])
```

EXAMPLE

$A = [2, 6, 8, 9, 12]$  
$B = [1, 2, 4, 10, 15]$  

So the median is between 4 and 8. We get rid of things smaller than 4 and larger than 10, and recurse on:

$A = [2, 6, 8]$  
$B = [4, 10, 15]$  

Now we know that the median is between 6 and 10, and we recurse on:

$A = [6, 8]$  
$B = [9, 10]$  

The median of both of them is 9, so we recurse on [6] and [10] and return either.
4.2. **(15 pt.)** Suppose that \( n = 2^k \) is a power of 2. You are given an \( n \times n \) grid with one square removed. Your goal is to tile the grid with \( L \)-shaped pieces that each take up three squares. The set-up looks like this:

- **Input:** \( n \times n \) grid with a missing square.
- **An \( L \)-shaped piece.**
- **Output:** a tiled grid.

Develop an algorithm to tile the grid with \( L \)-shaped pieces. Your algorithm should run in time \( O(n^2) \). You are welcome to use pictures if it helps explain what your algorithm is doing.

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

The basic idea is to divide the grid into 4 quadrants, and place an \( L \)-shaped piece in the middle so that it touches the 3 quadrants that don’t have the hole.

Now the four quadrants each have a piece missing, so we recursively tile them.

```python
def tileBoard(A):
    # If A is 2x2, this is easy:
    if A is 2x2:
        # Place a tile in the center so that it touches the 3 quadrants w/o the hole
        tileBoard(A[:n/2, :n/2])
        tileBoard(A[:n/2, n/2:])
        tileBoard(A[n/2, :n/2])
        tileBoard(A[n/2, n/2:])
```

- `tileBoard(A[:n/2, :n/2])`
- `tileBoard(A[:n/2, n/2:])`
- `tileBoard(A[n/2, :n/2])`
- `tileBoard(A[n/2, n/2:])`
5 Harder problem

Note: This problem may be trickier and it is only worth 5 points. You might want to do the rest of the exam first, and don’t worry if you can’t get it.

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

Do there exist two non-decreasing functions \( f(n) \) and \( g(n) \), mapping positive integers to positive integers, so that \( f(n) \) is not \( O(g(n)) \) and \( g(n) \) is not \( O(f(n)) \)?

Either give an example of such a pair of functions or prove that they do not exist.

[We are expecting: Your example, or a proof that no such functions exist.]

They exist. For example, define \( f \) and \( g \) recursively by:

\[
\begin{align*}
  f(1) &= g(1) = 1, \\
  f(n) &= \begin{cases} 
    f(n-1) & \text{if } n \text{ is odd} \\
    2^n \cdot g(n-1) & \text{if } n \text{ is even}
  \end{cases}, \\
  g(n) &= \begin{cases} 
    2^n \cdot f(n-1) & \text{if } n \text{ is odd} \\
    g(n-1) & \text{if } n \text{ is even}
  \end{cases}.
\end{align*}
\]

\begin{tabular}{|c|c|c|c|c|c|}
\hline
\( n \) & 1 & 2 & 3 & 4 & 5 \\
\hline
\( f(n) \) & 1 & 4 & 4 & 422 & 422 \\
\( g(n) \) & 1 & 32 & 32 & \ldots & \\
\hline
\end{tabular}

Then \( f \) and \( g \) are increasing.

Further, \( f(n) \) is not \( O(g(n)) \). To see this assume that it were. Then \( \exists n_0, c > 0 \) s.t. \( \forall n \geq n_0, \ f(n) \leq c \cdot g(n) \). Choose \( n \geq n_0 \) so that \( n \) is even, and \( 2^n > c \). Then \( f(n) = 2^n \cdot g(n-1) > c \cdot g(n-1) = c g(n) \), a contradiction.

Similarly, \( g(n) \) is not \( O(f(n)) \).

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.