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 (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.

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. We have included guidelines about how difficult we think each problem is, although it is different for everyone.

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 (multiple choice)	2 (short answer)	3 (alg. design)	4 (proofs)	Total
Score					
Maximum	25	40	20	15	100

1 Multiple Choice (25 pts)

No explanation is required for the questions in Section 1. 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. **(4 pt.)** Which of the following expressions correctly describe $T(n) = n\log(n)$? Circle all that apply.

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

1.2. **(4 pt.)** Which of the following expressions correctly describe $T(n) = 2^n$? Circle all the apply.

(A) $O(2^{100 \cdot n})$ (B) $O(n^{100})$ (C) $O(n^{\log(n)})$ (D) $O((\log(n))^n)$

1.3. **(5 pt.)** For each recurrence relation on the left, draw a line between it and all the expressions below which accurately describe it. Assume that $T(1) = O(1)$. Note that not all items on the right need to be used, and some items may be used more than once.

$T(n) = 3T(n/2) + n^2$	$\Theta(n)$
$T(n) = 3T(n/2) + n$	$\Theta(n^2)$
$T(n) = 3T(n/2) + n^{\log_2(3)}$	$\Theta(n^{\log_2(3)})$
$T(n) = 3T(n/2) + \sqrt{n}$	$\Theta(n^{\log_2(3)}\log(n))$
$T(n) = T(n/2) + T(n/10) + n$	$\Theta(n\log(n))$

1.4. **(5 pt.)** Which of the following expressions correctly describes the number of :)'s outputted by the algorithm SMILEYS below? Circle all that apply.

Algorithm 1: SMILEYS(n)

```
if n == 0 then
    print ":)"
else
    for t = 1, ..., 2^n do
        SMILEYS(n-1)
```

Let $S(n)$ = number of :)'s printed on input n.
$S(n) = 2^n \cdot S(n-1) = 2^n \cdot 2^{n-1} \cdot S(n-2) = \cdots$
$\cdots = 2^{\sum_{j=0}^{n} j} \cdot S(0) = 2^{\sum_{j=0}^{n} j} = 2^{n(n+1)/2}$

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

too small — too big — too small

$n! \le n^n = 2^{n\log(n)}$ is also too small

1.5. **(7 pt.)** For each of the quantities on the left, fill in a single expression from the choices below that describes it. Unless stated otherwise, all values are arbitrary comparable objects.

Quantity	**Expression that describes it.** (From the list below)
The depth of a red-black tree with n vertices	(D) $\Theta(\log(n))$
The expected running time of QuickSort on an array of length n	(B) $\Theta(n\log(n))$
The worst-case running time of QuickSort on an array of length n	(C) $\Theta(n^2)$
The worst-case running time of MergeSort on an array of length n	(B) $\Theta(n\log(n))$
The time it takes to sort n values in the set $\{0, \ldots, 9\}$	(A) $\Theta(n)$
The time it takes to find the $(n/3)$'rd largest value in an array of length n	(A) $\Theta(n)$
The time it takes to apply a hash function from a universal hash family which hashes a universe $\mathcal{U}$ into n buckets.	(F) Need more info.

Choices:

(A) $\Theta(n)$
(B) $\Theta(n\log(n))$
(C) $\Theta(n^2)$
(D) $\Theta(\log(n))$
(E) $O(1)$
(F) Need more information

2 Can it be done? (Short answers) (40 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.

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.)** Given an array A of length n which contains only integers between 1 and n, sort A in time $O(n)$.

BucketSort with n buckets.

2.4. **(5 pt.)** Given an array A of length n which contains arbitrary comparable elements, sort A in time $O(n)$.

This cannot be done, it violates the decision-tree lower-bound of $\Omega(n \log(n))$.

2.5. **(5 pt.)** Given an array A of length n which contains arbitrary comparable elements, *and* given access to a genie who can reverse the order of any array in-place in time $O(1)$, sort A in time $O(n)$.

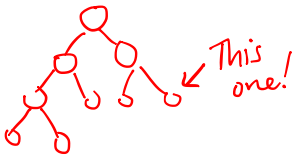
This also cannot be done. The decision tree lower bound says that the number of comparisons must be $\Omega(n\log(n))$, and this genie does not let you reduce the number of comparisons.

2.6. **(5 pt.)** Design a *randomized* data structure which stores n items and supports the operations INSERT, DELETE, SEARCH which each run in *expected* time $O(1)$ and *worst-case* time $O(\log(n))$.

Use a hash table, where you implement each bucket as a Red-Black Tree.

2.7. **(5 pt.)** Design a *deterministic* data structure which stores n 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.

A red-black tree will do this. To implement FINDMAX, return the right-most node in the tree.



2.8. **(5 pt.)** Design a *deterministic* data structure which stores n 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.

This cannot be done. If it could, this would allow us to sort in time $O(n)$, by doing:

1. INSERT all the items into the tree.
2. Repeatedly FINDMAX / DELETE to remove them in sorted order.

NOTE: The following two problems may be (more) difficult. You may wish to skip them and come back to them later.

2.9. **(5 pt.)** *(May be more difficult)* Given an array A of length n which contains distinct (but arbitrarily large) integers, and an integer $k \leq n$, find the k elements of A that are closest to the median of A, in time $O(n)$.

Here, "closest" means in absolute value: you should return the elements $A[i]$ that give the k smallest values of $|A[i] - m|$, where m is the median of A. You may assume that there are no ties.

1. Use SELECT to find the median m of A in time $O(n)$
2. In time $O(n)$, iterate through the array A to produce the array
$$B = [\ |A[0]-m|,\ |A[1]-m|,\ \ldots,\ |A[n-1]-m|\ ]$$
3. Use SELECT to find j so that $B[j]$ is the k^{th} smallest (time $O(n)$)
4. Iterate through B to find the set of indices i so that $B[i] \leq B[j]$ (time $O(n)$)
5. Return $A[i]$ for each i you found in 4. (time $O(n)$).

2.10. **(5 pt.)** *(May be more difficult)* Design a *randomized* data structure which stores n items from a universe $\mathcal{U}$ and supports the operations INSERT and QUERY. Here, QUERY(x) should return **True** if x has been inserted into the data structure and **False** otherwise. The data structure should store $O(n + \log(|\mathcal{U}|))$ bits, have INSERT and QUERY time $O(1)$ (in the worst case), and each query should be correct with probability at least 9/10.

(Note: This is very similar to HW4 #3(a)).

Let $\mathcal{H}$ be a universal hash family, containing functions $h: \mathcal{U} \to \{1, \ldots, 10 \cdot n\}$.
Choose $h \in \mathcal{H}$ uniformly at random. We can store h using $O(\log|\mathcal{U}|)$ bits if we use the universal hash family from class.

Store an array A of $10n$ bits. Then:

INSERT(x): $A[x] = 1$

QUERY(x): return $\begin{cases} \text{TRUE} & A[x]=1 \\ \text{FALSE} & A[x]=0 \end{cases}$

$\forall x$, The probability that QUERY(x) is incorrect is

$$\mathbb{P}\{\text{QUERY(x) incorrect}\} \leq \mathbb{P}\{\exists y \text{ that was inserted}; h(x)=h(y)\} \overset{\text{union bd}}{\leq} n \cdot \max_y \mathbb{P}\{h(x)=h(y)\} \overset{\text{def of universal hash family}}{\leq} \frac{n}{10n} = 1/10.$$

3 Algorithm Design (20 pts)

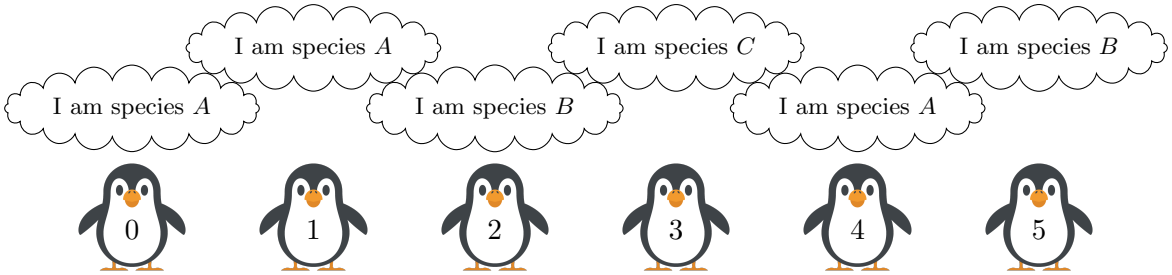
You arrive on an island with n penguins, made up of *many* different species of penguins. The differences between species are very subtle, so without help you can't tell the penguins apart at all. Fortunately, you have an expert with you, and she can tell whether or not two penguins belong to the same species. More precisely, she can answer queries of the form:

$$\texttt{isTheSame(penguin1, penguin2)} = \begin{cases} \textbf{True} \text{ if } \texttt{penguin1} \text{ and } \texttt{penguin2} \text{ belong to the same species} \\ \textbf{False} \text{ if } \texttt{penguin1} \text{ and } \texttt{penguin2} \text{ belong to different species} \end{cases}$$

The only way you can get any information about the penguins is by running `isTheSame`. You cannot hash them, ask for which species they are, or compare them in any other way.

We define a species X to be a *1/3-plurality* species if there are **strictly more** than $n/3$ penguins of that species on the island.

For example, suppose that the situation were this:



Then A is a 1/3-plurality species, since $n/3 = 2$, and there are $3 > 2$ penguins of species A.

Your goal is to return a single member of a 1/3-plurality species. (Notice that there may be more than one 1/3-plurality species. If that is the case, you could return any member of any of those species). In the example above, you can return any one of Penguins $0, 1$ or 4.

On the next two pages, you will do this in two different ways.

[Questions about penguins on next two pages]

3.1. **(10 pt.)** Assume that there is at least one 1/3-plurality species. Design a deterministic algorithm that uses $O(n \log(n))$ calls to `isTheSame` and returns a penguin belonging to a 1/3-plurality species. You may assume that $n = 3 \cdot 2^k$ for some integer k if it is helpful.

[We are expecting: Pseudocode and an English description of the idea of your algorithm. You do not need to prove that it is correct or justify the running time.]

IDEA: Divide + Conquer.
We repeatedly separate the population in half.
A species that appears more than $n/3$ times must appear at least $(n/2)/3$ times in at least one of the halves, so at least one of the two recursive calls will be successful. We return one sample of each plurality species in each recursive call.

NOTE There was a BUG in an earlier edition of these solutions!! In order for the recursion to be correct we need to return a penguin from EVERY plurality species (there might be two of them). This is because otherwise the recursion might fail, for example on [11122 11133]. Thanks Gergely for catching this! When grading, we gave full credit for anything "as correct" as our originally posted solution.

#This function uses O(n) calls to isTheSame and decides if a candidate penguin x is a member of a 1/3-plurality species.

```
def isPlurality(population P, of size n, penguin x):
    count = 0
    for y ∈ P:
        if isTheSame(y, x):
            count += 1
    if count > n/3:
        return TRUE
    return FALSE
```

#This function recursively finds a penguin from every plurality species.

```
def pluralityPenguin(population P of size n):
    if n = 3:
        If P[0] = P[1] or P[0] = P[2]: return P[0]
        If P[1] = P[2]: return P[1]
        Else return []
    X = pluralityPenguin(P[:n/2])
    Y = pluralityPenguin(P[n/2:])
        #X and Y are lists of plurality penguins,
        of length {0,1,2}.
    Z = union(X, Y)   // X and Y have size at most 2, so this takes O(1) calls to isTheSame
    ret = []
    for z ∈ Z:
        if isPlurality(z):
            ret.append(z)
    return ret
```

#helper fn for pluralityPenguin

```
def union(X, Y):
    ret = []
    for x ∈ X+Y:   //concatenate X and Y
        alreadyThere = False
        for z ∈ ret:
            if isTheSame(x, z):
                alreadyThere = True
                break
        if not alreadyThere:
            ret.append(x)
    return ret
```

[Another Penguin question on next page]

3.2. **(10 pt.)** Assume that there is at least one 1/3-plurality species. Design a *randomized* algorithm that uses $O(n)$ calls to `isTheSame` in expectation, and returns a penguin belonging to a 1/3-plurality species.

Restriction! If you got an algorithm that uses $O(n)$ calls to `isTheSame` on Problem 3.1., you must give a *different* algorithm for this part to receive credit.

[We are expecting: Pseudocode and an English description of the idea of your algorithm. You do not need to prove that it is correct or justify the running time.]

IDEA: Choose a random penguin. With probability 1/3 it belongs to a Plurality Species.

```
def findPluralPenguin(population P of size n):
    while TRUE:
        x = random penguin in P
        if isPlurality(x):        <-- This is the isPlurality function from problem 3.1.
            return x
```

The expected number of times isPlurality is run is 3, and each time it uses $O(n)$ calls to isTheSame, so the expected number of calls to isTheSame is $O(n)$.

4 Proving Stuff (15 pts)

4.1. **(10 pt.)** Consider the statement below:

Statement 1. Suppose that f and g are increasing functions defined on the integers. If $f(n) = O(g(n))$, then $2^{f(n)} = O\left(2^{g(n)}\right)$.

Statement 1 is **FALSE**. In the following parts, you will prove this.

(a) **(4 pt.)** Give functions f and g that are a counter-example to **Statement 1.**

Let $f(n) = 2\log(n)$, $g(n) = \log(n)$

(b) **(3 pt.)** Prove, using the definition of big-Oh, that $f(n) = O(g(n))$ for your example.

For all $n \geq 1$, $2\log(n) \leq 2 \cdot \log(n)$, hence

$\forall n \geq 1$, $f(n) \leq 2g(n)$.

(this is my "n_0" — this is my "c")

(c) **(3 pt.)** Prove, using the definition of big-Oh, that $2^{f(n)}$ is *not* $O(2^{g(n)})$ for your example.

We have $2^{f(n)} = 2^{2\log(n)} = n^2$ and $2^{g(n)} = 2^{\log(n)} = n$,

so we need to prove that n^2 is not $O(n)$.

To see this, suppose that n^2 WAS $O(n)$. Then $\exists\, n_0, c$ s.t.

$$n^2 \leq c \cdot n \quad \forall n \geq n_0$$

$$\Rightarrow \quad n \leq c \quad \forall n \geq n_0 .$$

But this is not true: for example, we may take $n = \max\{c, n_0\} + 1$.

Thus, n^2 is not $O(n)$.

4.2. **(5 pt.)** *(May be more difficult)* Consider the statement below:

Statement 2. Suppose that f and g are increasing functions defined on the integers so that $f(n), g(n) > 1$ for all $n \geq 0$. If $f(n) = O(g(n))$, then $\log(f(n)) = O(\log(g(n))$.

Statement 2 is **TRUE**. Prove, using the definition of big-Oh, that it is true.

Suppose that $f(n) = O(g(n))$, so $\exists n_0, c$ s.t. $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. Then, for all $n \geq n_0$, we have

$$0 \leq \log(f(n)) \leq \log(c \cdot g(n))$$

because $f(n) \geq 1 \ \forall n \geq 0$

$$= \log(c) + \log(g(n))$$

$$= \left(\frac{\log(c)}{\log(g(n))} + 1\right) \cdot \log(g(n))$$

Because g is increasing, we have $\log(g(1)) \leq \log(g(n))$

$$\leq \left(\frac{\log(c)}{\log(g(1))} + 1\right) \cdot \log(g(n))$$

$\log(g(1)) > 0$ since $g(1) > 1$, so it's OK to put it in the denominator.

$$=: \tilde{c} \cdot \log(g(n)),$$ where we define $\tilde{c} = \left(\frac{\log(c)}{\log(g(1))} + 1\right)$

Then $\log(f(n)) = O(\log(g(n))$ by the definition of $O(\cdot)$.

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.

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.

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.