Instructions (for the real exam)

- **DO NOT OPEN THE EXAM UNTIL YOU ARE INSTRUCTED TO.**
- Answer all of the questions as well as you can. You have three hours.
- The exam is **non-collaborative**; you must complete it on your own. If you have any clarification questions, please ask the course staff (we are outside the exam room). We cannot provide any hints or help.
- This exam is **closed-book**, except for **up to two double-sided sheets of paper** that you have prepared ahead of time. You can have anything you want written on these sheets of paper.
- **Please DO NOT separate pages of your exam**. The course staff is not responsible for finding lost pages, and you may not get credit for a problem if it goes missing.
- There are a few pages of extra paper at the back of the exam in case you run out of room on any problem. If you use them, please clearly indicate on the relevant problem page that you have used them, and please clearly label any work on the extra pages.
- Please do not discuss the exam before solutions are posted. *[Obviously it's okay to discuss this* ***practice*** *exam :) But these are the instructions on the main exam.]*

General Advice

- If you get stuck on a question or a part, move on and come back to it later. The questions on this exam have a wide range of difficulty, and you can do well on the exam even if you don't get a few questions.
- Pay attention to the point values. Don't spend too much time on questions that are not worth a lot of points.
- There are **105** + **5** (bonus) total points on this exam. There are **six problems** split across **14 pages**.

Name and SUNet ID (please print clearly):

________Solutions!____

This page intentionally blank. Please do not write anything you want graded here.

Honor Code

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.*

For my part, I believe that we have upheld our end of the agreement in Item 2. I don't think we are taking unusual or unreasonable precautions that would indicate a lack of confidence in the honor of students, and I believe that the in-person setting avoids temptations ot violate the honor code to the extent practicable.

[signed, Mary Wootters]

Please acknowledge that you have held up your end of the agreement in Item 1:

I have abided by the Honor Code, and in particular the policies listed above, both in letter and in spirit, while taking this exam.

signed, _______________ [your name here] _______________________

Good Luck!

This page intentionally blank. Please do not write anything you want graded here.

1. **(20 pt.)[Multiple Choice]** For each of the following, select **ALL answers that apply.**

   [**We are expecting:** *For all parts of this problem, just circle all of the answers that apply. No justification is required or will be considered for grading.*]

   (i) Which of the following algorithms runs in (worst-case) $O(n^2)$ time? Select all that apply.

   (A) Karatsuba Multiplication, where n is the number of digits in each number.

   (B) MergeSort, where n is the number of elements we're sorting.

   (C) QuickSort, where n is the number of elements we're sorting, and the chosen pivot is random.

   (D) k-SELECT, where n is the number of elements in the input array, and the chosen pivot is random.

   (E) None of the above.

SOLUTION: Answer: A, B, C, D

Explanation: Karatsuba Multiplication runs in $\Theta(n^{1.6})$. Mergesort runs in $\Theta(nlogn)$. Quicksort with a random pivot runs in worst case $O(n^2)$. k-SELECT with a random pivot runs in worst case $O(n^2)$. Note that big-Oh is an upper bound, so all of these are $O(n^2)$.

   (ii) Recall that the formal definition of big-O is:

   "$f(n)$ is $O(g(n))$ if there exist some values $c, n_0 > 0$ such that $f(n) \leq c(g(n))$ for all $n \geq n_0$."

   Which of the following are true about this definition? Select all that apply.

   (A) c and n_0 **cannot** have the same value.

   (B) c and n_0 **must** be integers.

   (C) If $f(n) = g(n)$, then $f(n) = O(g(n))$.

   (D) $g(n)$ **cannot** be $O(1)$.

   (E) None of the above.

SOLUTION: Answer: C

Explanation: Note that nothing in the definition restricts c or n_0 from having the same value. Likewise, nothing in the definition restricts c or n_0 from being fractions or irrational numbers. Thus, A and B are false. C can be proven true by setting c equal to any value greater than one, and then setting n_0 to any value greater than zero. Finally, D is false because the definition imposes no restrictions to what the function $g(n)$ can be.

(iii) Let T be a binary search tree storing n elements. Which of the following statements are necessarily true?

(A) The number of nodes in T is n.

(B) The number of leaves in T is $\Theta(n)$.

(C) The height of T is $O(\log n)$

(D) None of the above.

SOLUTION: **Answer:** A

Explanation: We are told that T is storing n items, so it has n nodes; so A is true. B is false: for example, if T is a tall skinny tree, it might have only one leaf. C is also false: for example, if T is tall and skinny, it might have depth n, not $O(\log n)$.

(iv) Let $\mathcal{A}$ be a comparison-based sorting algorithm for sorting an array of length n, and let T be the corresponding decision tree (as per our proof in Lecture 6). Which of the following are necessarily true about T?

(A) T has at least $n!$ leaves

(B) The depth of T is $O(\log n)$

(C) The depth of T is $\Omega(n \log n)$

(D) Running $\mathcal{A}$ corresponds to a path from the root of T to a leaf

(E) Running $\mathcal{A}$ corresponds to an internal node of T

(F) None of the above.

SOLUTION: Answer: A, C, D

Explanation: A is true because there are $n!$ possible orderings (aka outputs), and each possible output corresponds to a leaf. The depth of the tree is at least $\log(n!) = \Theta(n \log n)$, so B is false and C is true. D is true by the definition of this correspondence (and E is false for the same reason).

(v) Imagine we want to sort a list of numbers in **descending** order using RadixSort. Which of the following modifications (made *individually*) should we make to the algorithm shown in class? Select all that apply.

(Note: "MSD" stands for “Most Significant Digit” and “LSD” stands for “Least Significant Digit”.)

(A) We bucket the numbers in MSD-to-LSD order instead of LSD-to-MSD.

(B) We no longer need to pad the numbers with leading zeros.

(C) We “unload” each bucket in last-in-first-out order, instead of first-in-first-out.

(D) In each iteration of CountingSort, we traverse the buckets in descending order, starting with the n^{th} bucket and ending with the 0^{th} bucket.

(E) None of the above.

SOLUTION: Answer: D

Explanation: The simplest way to visualize the way RadixSort can be used to order values in descending order is to simply look at a single iteration of CountingSort, where all the numbers are single-digits. Note that to use CountingSort to order numbers in descending order, we simply have to traverse the buckets in descending order, rather than ascending order. Extrapolating this concept and applying it to multiple iterations, we can conclude that D is the correct answer.

Note that bucketing in MSD-to-LSD order leads us to sorting the values from right-to-left, but does not lead us to sorting the values in descending order. Thus, A is false. Further, we must pad all values to be the same length no matter what we're sorting (numbers, usernames, etc). Thus, B is false. Additionally, at each iteration of RadixSort, the buckets must always be unloaded first-in-first-out to preserve the ordering of the previous iteration(s). Thus, C is false.

(vi) Let T be a red-black tree that contains n items. Which must be true about T?

(A) T has $O(\log n)$ levels.

(B) T has $\Omega(\log n)$ levels.

(C) T contains $O(\log n)$ black nodes.

(D) Inserting an item into T takes time $O(\log n)$.

(E) None of the above.

SOLUTION: **Answer:** A,B,D

Explanation: A RB-tree has $\Theta(\log n)$ levels, so both A and B are true.

C is false: For example, the red-black tree could be all black, in which case there would be n black nodes.

D is true since the time it takes to insert an item into an RB-tree is big-Oh of the depth of the tree, which as above is $O(\log n)$.

2. **(10 pt.) [True or false?]** For each of the parts below, say whether the statement is true or false, and explain why.

   [**We are expecting:** *For each, either* TRUE *or* FALSE. *If it is false, give a counter-example. If it is true, give a short explanation why (a sentence or two). You don't need to give a formal proof, and you do not need to appeal to the formal definition of big-Oh.*]

   (a) If $f(n) = O(g(n))$, then $f(n) = \Omega(g(n))$.

   (b) If $f(n) = 4^{\log_2(n)}$, then $f(n) = \Omega(n)$.

   (c) If $f(n) = 2^{\log_2(\log_2(\log_2(n)))}$, then $f(n) = \Omega(n)$.

   (d) If $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$.

SOLUTION:

(a) This is false. For example, if $f(n) = 1$ and $g(n) = n$, then $f(n) = O(g(n))$, but $f(n)$ is not $\Omega(g(n))$.

(b) This is true: $f(n) = n^2$, which is $\Omega(n)$.

(c) This is false: $f(n) = \log_2(\log_2(n))$, which grows *much* more slowly than n.

(d) This is false: for example, if $g(n) = \log(n)$ and $f(n) = 2\log n$, then $f(n) = O(g(n))$, but $2^{f(n)} = n^2$ and $2^{g(n)} = n$, so $2^{f(n)}$ is *not* $O(2^{g(n)})$.

3. **(20 pt.) [Recurrence Relations!]** For each of the following recurrence relations for $T(n)$, give the best big-Oh bound that you can. Simplify your expressions so that they are of the form $O(n^{\text{something}})$. You may ignore floors and ceilings in your answers.

   [**We are expecting:** *For each part, a statement of the form $T(n) = O(n^{something})$. No justification is required if your answer is correct, but a short informal justification may be considered for partial credit.*]

   (a) $T(n) = 5T(\lfloor n/3 \rfloor) + O(n^2)$ for $n > 0$, with $T(0) = 1$.

   (b) $T(n) = 10T(\lfloor n/3 \rfloor) + O(n^2)$ for $n > 0$, with $T(0) = 1$.

   (c) $T(n) = T(\lfloor n/5 \rfloor) + T(\lfloor n/10 \rfloor) + n^2$ for $n > 10$ with $T(n) = 1$ for $0 \leq n \leq 10$.

   (d) $T(n) = T(\lfloor \log_2(n) \rfloor) + n^{10}$ for $n > 2$ with $T(n) = 1$ for $0 \leq n \leq 2$.

SOLUTION: Note: No explanation (and certainly full proofs) are not required, but we'll give some explanations and/or proofs here to help you study!

(a) $O(n^2)$. This can be seen by the Master theorem, with $a = 5, b = 3, d = 2$. In this case, $b^d = 9 > 5 = a$, so we are in the setting where the work at the top of the tree (which is $O(n^2)$) dominates.

(b) $O(10^{\log_3 n}) = O(n^{\log_3(10)})$. (It is okay if you left your answer in this form, since $\log_3(10)$ is not something we expect you to be able to compute on an exam!) This follows from the master theorem, with $a = 10, b = 3, d = 2$. In this case, $b^d = 9 < 10$, so we are in the setting where the work is dominated by the number of leaves, which is $10^{\log_3 n}$.

(c) $O(n^2)$. There are (at least) two ways to see this:

- **EXPLANATION 1:** Consider $S(n) = 2S(n/5) + n^2$. Clearly $S(n)$ will be larger than $T(n)$ (since we are essentially replacing the second $n/10$ term with an $n/5$ term, which is bigger). We can apply the master theorem to $S(n)$ to get $O(n^2)$. On the other hand, $T(n) = \Omega(n^2)$ because there is an n^2 in the expression itself. So $T(n) = \Theta(n^2)$ is the correct asymptotic answer.
- **EXPLANATION 2:** We can guess that it is $O(n^2)$ since we see the n^2 sitting there at the end, so that's clearly a lower bound (also, since we saw $T(n) = T(n/5) + T(7n/10) + n$ in our analysis of SELECT in class, and that one ended up being the "n" at the end, we might guess that this would be similar). If we tried to verify our guess formally by induction, with the inductive hypothesis $T(n) \leq c \cdot n^2$ (for some c TBD), it would look like:

  $$T(n) = T(n/5) + T(n/10) + n^2 \leq c(n/5)^2 + c(n/10)^2 + n^2 \leq cn^2,$$

  which is true as long as $n^2(c/25 + c/100) \leq n^2(c-1)$ aka $c/20 \leq c-1$, aka

  $$1 \leq \frac{19c}{20}.$$

  So we can choose c to be at least $20/19$ and the proof will work.
  NOTE: the above would *not* get full credit as a "formal proof by induction," since, e.g., we didn't deal with the base case. But it would count as an "informal justification," which is all that's (optionally) asked for, and also it accurately represents the thought process that one might go through when trying to solve this problem. It's a great exercise to go through and turn this into a formal proof! (And since we've already done the work of guessing "c", now you don't need to :)).

(d) $O(n^{10})$. Intuitively, this is because $\log_2(n)$ is *so* much smaller than n that surely we must be in the setting where the work at the top of the recursion tree will dominate, and that's n^{10}.

One way to arrive at this intuition is by unrolling the recurrence relation. We get (ignoring floors and ceilings, since this is just an intuitive explanation...)

$$\begin{aligned}
T(n) &= T(\log(n)) + n^{10} \\
&= T(\log\log(n)) + (\log(n))^1 0 + n^{10} \\
&= T(\log\log\log(n)) + (\log\log(n))^{10} + (\log n)^{10} + n^{10} \\
&\vdots \\
&= n^{10} + (\log n)^{10} + (\log\log n)^{10} + (\log\log\log n)^{10} + \cdots + 1.
\end{aligned}$$

Now, this is a pretty gross sum, that I certainly haven't seen before. (Yikes!) However, we know that $\log n$ is *way* less than n, and $\log\log n$ is *even more way* less than n, etc. So, looking at this sum, we should guess that it's dominated by its largest term, which is n^{10}.

Formally, one way we could see this is to replace $\log(n)$ with $n/2$, since $n/2$ is way bigger than $\log(n)$. (And then we'd replace $\log\log n$ with $n/4$, $\log\log\log n$ with $n/8$, and so on). Then we can bound this by:

$$\begin{aligned} T(n) &\leq \sum_{j=0}^{\infty} \left(\frac{n}{2^j}\right)^{10} \\ &= n^{10} \sum_{j=0}^{\infty} \frac{1}{2^j} \\ &= 2n^{10}. \end{aligned}$$

4. **(15 pt.) [Can it be done?]**

For each of the following tasks, either explain briefly how to do it, or prove that it cannot be done. In all of the cases, running times are to be interpreted using worst-case analysis. We have done the first two for you to give you a sense of what we are looking for.

[**We are expecting:** *For each part, either a description of an algorithm, or a short proof of impossibility. You may appeal to any result/algorithm from class, but you must clearly state how you are using it.*]

(a) **(0 pt.) (Example)** Find the maximum of a (not necessarily sorted) array A in time $O(n \log n)$.

Use `mergeSort` on the array A; then return the last element of what `mergeSort` returns.

(b) **(0 pt.) (Example)** Find the maximum of a (not necessarily sorted) array A in time $O(1)$.

This cannot be done. Since any of the n entries of the array could be the maximum, we need to at least look at every element in the array, which takes time $\Omega(n)$.

(c) Given an array A of length n which contains d-digit integers (base 10) for $d = 3\log_{10}(n)$, sort A in time $O(n)$.

SOLUTION: This can be done: use RadixSort with base $r = n$. Then the maximum value M of the n integers is at most $10^d = n^3$, and the total amount of time it takes RadixSort is $O(n \cdot (\log_n(M) + 1)) = O(n)$. Here we used the fact that $\log_n(n^3)) = 3$.

(d) Given an array A of length n which contains arbitrary comparable elements, sort A in time $O(n)$. (Here, "arbitrary comparable elements" means that you cannot interact with the values of the elements other than by comparing them to each other.)

SOLUTION: This cannot be done: we saw in class that any comparison-based sorting algorithm requires time $\Omega(n \log n)$.

(e) Design a data structure that holds n elements and that supports INSERT/DELETE/SEARCH as well as MAX/MIN, each in time $O(\log n)$.

SOLUTION: This can be done. Just use a RB-tree. We can get MAX by traversing all the way to the right; and MIN by traversing all the way to the left.

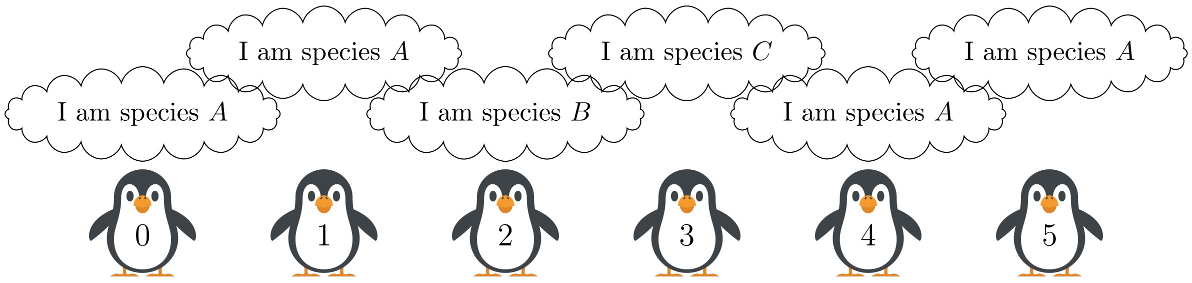
5. **(20 pt.) [Algorithm Design.]**

On an island, there are n penguins of many different species. The differences between the 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 you 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 ask them what species they are, or compare them in any other way.

The expert assures you that one species of penguin is in the majority. That is, there are *strictly greater* than $n/2$ penguins of that species. Your goal is to return a single member of that majority species. For example, if the population looked like this:



then species A is in the majority, and your algorithm should return any one of Penguins 0, 1, 4, or 5.

If there is no species with a strict majority, your algorithm may return whatever it wants.

Questions start on next page!

(a) **(10 pt.)**[1] Design a deterministic *divide-and-conquer* algorithm that uses $O(n \log(n))$ calls to `isTheSame` and returns a penguin belonging to the majority species. You may assume that n is a power of 2 if it is helpful.

[**We are expecting:** *Pseudocode (which calls `isTheSame`)* ***AND*** *a clear English description of what your algorithm is doing.*]

More parts on next page!

[1]Note: On the actual exam, we will not expect you to do creative algorithm design that requires "aha!" moments in a timed setting (except maybe for bonus points). There will be algorithm design question(s), but they will be variants of things you have seen before in class or on HW, so that if you have studied enough, you won't need (what we consider to be) an unreasonable "aha!" moment. This problem requires an "aha!" moment, so it wouldn't appear on the actual exam — but it is a good chance to get more practice with algorithm design! (Which will help on the real exam, since it will help you remember and recreate the aha! moments from HW and lecture.)

(b) **(5 pt.)** Explain why your algorithm calls `isTheSame` $O(n\log(n))$ times.
[**We are expecting:** *A short justification of the number of calls to* `isTheSame`*. You may invoke the Master Theorem if it applies.*]

(c) **(0 pt.)** [This is not required, but since you're taking a practice test, at this point it would be good practice to formally prove that your algorithm is correct!]

More parts on next page!

(d) **(5 pt.)** Give a *randomized* algorithm that *always* (with probability 1) returns a majority penguin, so that, for any input, the *expected* number of calls to `isTheSame` $O(n)$.

[**We are expecting:** *Pseudocode AND a clear English description, as well as an informal explanation (a few sentences) of why the expected number of calls to* `isTheSame` *is* $O(n)$.]

(e) **(NOT REQUIRED. 5 BONUS pt.)** Can you come up with a deterministic algorithm that is asymptotically better than $O(n\log(n))$? Either give a deterministic algorithm with asymptotically fewer calls to `isTheSame`, or else prove that no such algorithm exists. (Or, if you think that your algorithm from part (a) already does better, just write that :)).

[**We are expecting:** *Nothing. This part is not required. To get the bonus point, you should either: (a) give pseudocode AND a clear English description of what it does, along with a short informal justification of why it uses asymtotically fewer calls to* `isTheSame`*; or (b) a rigorous proof of why you cannot do better.*]

SOLUTION:

(a) We'll use a divide-and-conquer algorithm as follows. First, we'll break the set of n penguins into two sets of size $n/2$. Then we'll find a majority penguin in each half. Now (as we show below when we prove correctness), at least one of the two penguins returned will be a majority penguin; to test which one, we will compare each of these two penguins against all n penguins. Then we'll return one for which the test comes up positive. (And if neither do, then we return any old penguin). The pseudocode is as follows:

```
def majorityPenguin( penguin population P of size n ):
     n = len(P)
     if n == 1:
         return P[0]  # base case:
                      # a single penguin is in the majority.
     P1 = P[:n/2]
     P2 = P[n/2:]
     p1 = majorityPenguin(P1)
     p2 = majorityPenguin(P2)
     for p in [p1,p2]:
         count = 0
         for q in P:
             if isTheSame(p,q):
                 count += 1
         if count > n/2-1:
             return p
     return P[0] # there was no majority penguin;
                 # return an arbitrary penguin.
```

(b) Let $T(n)$ be the number of calls to `isTheSame` on an input of size n. (As allowed, we assume n is a power of 2). Then $T(n)$ obeys the recurrence relation

$$T(n) \leq 2T(n/2) + 2n,$$

since we call `majorityPenguin` twice on lists of size $n/2$, and then within `majorityPenguin` we call `isTheSame` at most $2n$ times. By the Master Theorem (with $a = 2, b = 2, d = 1$), the running time is $O(n \log(n))$.

(c) We give a formal proof by induction. We are going to structure this assuming that n is a power of 2. (We'd have to do it slightly differently if n was not a power of 2). Our inductive hypothesis is that the lower calls to `majorityPenguin` are correct:

- **Inductive hypothesis (t).** If A is a list of length $n = 2^t$ in which there is a strict majority, then `majorityPenguin`(A) returns a majority penguin in A.
- **Base case.** When $t = 0$, (so $n = 1$) `majorityPenguin` returns the only penguin in the list, which is a majority penguin by definition.
- **Inductive step.** Let $n = 2^t$ and suppose that the inductive hypothesis holds for $t - 1$, so `majorityPenguin` returns a majority penguin on a list of length $n/2$, if such a penguin exists. Now consider the lists `P1` and `P2` from the

pseudocode in part (a). Suppose that the whole population `P` has a majority species X.

We claim that X is also the majority in at least one of `P1` or `P2`. Indeed, suppose that X were a majority in neither. Then there would be at most $n/4$ members of X in both `P1` and `P2`, for a total of at most $n/2$ members of X in the whole population. But this violates the definition of a majority species, which should be *strictly* greater than $n/2$ in number.

Thus, by the inductive hypothesis, at least one of `p1` or `p2` are members of X. Suppose that `p1` is in X. Then there are at least $n/2$ other penguins `q` in `P` so that `isTheSame(p,q)` returns True, in which case our algorithm will return `p1`. On the other hand, if `p1` is not in X, then there are at most $n/2-1$ other penguins `q` in `P` that are of the same species, so our algorithm will not return `p1`. The same logic holds for `p2`, and we conclude that `majorityPenguin` returns a member of the majority species X. This establishes the inductive step for the next round.

- **Conclusion.** We conclude that the inductive hypothesis holds for all n that are a power of 2. That is, `majorityPenguin` indeed returns a member of the majority population in `P`, which is what we wanted to show.

(d) Intuitively, we keep choosing a random penguin, and then check that it is a majority penguin by comparing it to all the others. In pseudocode:

```
• While true:
    – Choose a random penguin $p$.
    – Call isTheSame $n-1$ times to compare $p$ to all the other penguins.
    – If at least $(n-1)/2$ penguins are the same as $p$, then return $p$.
```

The expected number of iterations is ≤ 2, because the probability that a random penguin is in the majority is at least $1/2$. For each iteration, we use $O(n)$ calls to `isTheSame`, so the expected number of calls to `isTheSame` is $O(n)$.

(e) There is an $O(n)$-time solution:

https://en.wikipedia.org/wiki/Boyer-Moore_majority_vote_algorithm

(Of course, a link to a wikipedia page would not be allowed on the actual exam, which is closed-book! But if you came up with that algorithm (or a similar one), and gave a clear description, it would get the bonus points.)

6. **(20 pt.) [Proofs!]** Suppose that $T(n) = T(\lfloor n/2 \rfloor) + T(\lfloor n/4 \rfloor) + n$ for all $n \geq 4$, and $T(0) = T(1) = T(2) = T(3) = 1$. Prove by induction that $T(n) = O(n)$.

   [**We are expecting:** *A rigorous proof by induction. Make sure you clearly state your inductive step, base case, inductive step, and conclusion.*]

SOLUTION: Here are two different ways to do this:

SOLUTION 1:

- Inductive Hypothesis (for n): $T(n) \leq 8 \cdot n$.
- Base case: For $n = 1, 2, 3$, we have $1 = T(n) \leq 8n$, since $8n \geq$ for $n \geq 1$. So the IH holds for $n = 1, 2, 3$.
- Inductive step: Suppose that $k \geq 4$, and that the IH holds for all ℓ so that $1 \leq \ell < k$. We want to show that it holds for k. By definition, we have

  $$T(k) = T(\lfloor k/2 \rfloor) + T(\lfloor k/4 \rfloor) + k.$$

  Since the IH applies for all $1 \leq \ell < k$, in particular it applies to $\lfloor k/2 \rfloor$ and $\lfloor k/4 \rfloor$. (Indeed, both are clearly less than k, and since $k \geq 4$ we have $\lfloor k/4 \rfloor \geq 1$, and similarly $\lfloor k/2 \rfloor \geq 1$. Applying the IH to the above equation, we conclude that

  $$\begin{aligned} T(k) &\leq 8 \cdot \lfloor k/2 \rfloor + 8 \cdot \lfloor k/4 \rfloor + k \\ &\leq 8k/2 + 8k/4 + k \\ &= (8/2 + 8/4 + 1)k \\ &= (4 + 2 + 1)k \\ &= 7k \\ &\leq 8k. \end{aligned}$$

  Thus, $T(k) \leq 8k$, and we have established the inductive hypothesis for k.
- Conclusion: We conclude that $T(n) \leq 8n$ for all $n \geq 1$. By the definition of big-Oh (with $n_0 = 1$ and $c = 8$, this implies that $T(n) = O(n)$.

Aside: how did we pick the constant 8? We (aka, Mary, when she wrote the solutions) just kind of guessed at it, since she didn't feel like solving for the best possible value of "c." Why guess 8? Well, she knew from experience (e.g. with our analysis of SELECT) that she'd need to pick a c so that $(c/2 + c/4 + 1) \leq c$; this is the expression that came up in the inductive step. This looked easier to deal with if c was divisible by 2 and 4 (adding fractions is hard!), and by inspection it was true for 8, so she just picked 8. In retrospect, she could have also picked $c = 4$. But as long as you can get any *c that works, it doesn't matter (for the conclusion of showing that something is big-Oh of something else) whether you've gotten the smallest possible c!*

SOLUTION 2:

Often when the recurrence relation involves dividing by 2 or 4, it's easier to assume that n is a power of 2, say 2^r, and then do induction on r rather than on n. The problem didn't say we could assume n was a power of 2, but for big-Oh calculations it turns out we still can essentially make that assumption, and then get rid of it later! Here's how that would go: We begin by first proving the result just for powers of 2.

- Inductive Hypothesis (for r): $T(2^r) \leq 10 \cdot 2^r$.
- Base case: For $r = 0$, we have $1 = T(2^0) \leq 10 = 10 \cdot 2^0$, so the IH holds for $r = 0$. For $r = 1$, we have $1 = T(2^1) \leq 20 = 10 \cdot 2^1$, so the IH holds for $r = 1$.
- Inductive step: Suppose that $k \geq 1$, and that the IH holds for $r = k$ and for $r = k - 1$. (Notice that since our base case covers $r = 0$ and $r = 1$, it is legit to assume this for $k \geq 1$). We want to show that it holds for $r = k + 1$. By definition, we have

$$T(2^{k+1}) = T(2^k) + T(2^{k-1}) + 2^{k+1}.$$

  Since the IH applies to k and $k-1$, we have $T(2^k) \leq 10 \cdot 2^k$ and $T(2^{k-1}) \leq 10 \cdot 2^{k-1}$. Plugging in, this says that:

$$T(2^{k+1}) \leq 10 \cdot 2^k + 10 \cdot 2^{k-1} + 2^{k+1}.$$

  Simplifying, we get

$$T(2^{k+1}) \leq 2^{k-1}(20 + 10 + 4) = 34 \cdot 2^{k-1} = \frac{34}{4} 2^{k+1} \leq 10 \cdot 2^{k+1}.$$

  Thus, $T(2^{k+1}) \leq 10 \cdot 2^{k+1}$, and we have established the inductive hypothesis for $r = k + 1$.
- Conclusion: We conclude that $T(2^k) \leq 10 \cdot 2^k$ for all $k \geq 0$.

Now, we still need to prove that $T(n) = O(n)$. To see this, first we observe that $T(n) \leq T(2^{\lceil \log_2 n \rceil})$, since $n = 2^{\log_2(n)} \leq 2^{\lceil \log_2(n) \rceil}$ and $T(n)$ is increasing. By what we just proved, we have that, for all $n \geq 1$,

$$T(n) \leq T(2^{\lceil \log_2(n) \rceil}) \leq 10 \cdot 2^{\lceil \log_2(n) \rceil} \leq 10 \cdot 2^{\log_2(n)+1} = 20 \cdot 2^{\log_2(n)} = 20n.$$

Thus, by the definition of big-Oh (with $n_0 = 1, c = 20$), we have $T(n) = O(n)$.

This is the end of the exam! You can use this page for extra work on any problem. **Keep this page attached** to the exam packet (whether or not you use it), and if you want extra work on this page to be graded, clearly label which question your extra work is for.

This page is for extra work on any problem. **Keep this page attached** to the exam packet (whether or not you use it), and if you want extra work on this page to be graded, clearly label which question your extra work is for.

This page is for extra work on any problem. **Keep this page attached** to the exam packet (whether or not you use it), and if you want extra work on this page to be graded, clearly label which question your extra work is for.

This page is for extra work on any problem. **Keep this page attached** to the exam packet (whether or not you use it), and if you want extra work on this page to be graded, clearly label which question your extra work is for.