CS 161 **Exam 1**
Winter 2021 **Length:** 2 hours = 1.5 hours + 0.5 hours for logistics

This exam is open-book and closed-collaboration. You have 120 minutes to complete the exam and submit your answers on Gradescope. You may consult inanimate materials or resources, including course materials, but inputting any exam questions or solutions into **any software, apps, or websites (Google, Stack Overflow, etc.) is in direct violation of the honor code**. You may review the Stanford Honor Code online at the following link.

`https://communitystandards.stanford.edu/policies-and-guidance/honor-code`

You may also review the detailed rules and instructions on the Exam portion of the CS 161 webpage, linked below.

`https://stanford-cs161.github.io/winter2021/exams/`

Good luck!

Section 1. Short Answer (20 points)

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.**

Throughout this section, mark **ALL answer choices that apply.** Also, you may assume that $T(1) = 1$ for question (ii) and $T(1) = T(2) = 1$ for question (iii). Furthermore, if you are writing your solutions on another document, please **box** your choices of **(A)**, **(B)**, **(C)**, and/or **(D)** to make it **unambiguously clear** which answers you selected for each of questions (i) through (v).

(i) **(4 pt.)** Which of the following correctly describes $T(n) = n^2 + 5\log n + 10$?

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

Solution

B, D. We can see that this expression is $\Theta(n^2)$ so $\Omega(n)$ and $O(n^2)$ are the only two that apply.

(ii) **(4 pt.)** Which of the following correctly describes $T(n) = T(n-1) + n$?

(A) $O(n^3)$ **(B)** $\Omega(n)$ **(C)** $\Omega(n^2)$ **(D)** $\Theta(n\log(n))$

Solution

A, B, C. This is much the same as Homework 2, Question 2(b). Expanding the sum explicitly gives us $T(n) = n + (n-1) + \ldots + 1 = \binom{n+1}{2} = \Theta(n^2)$. Only $O(n^3)$, $\Omega(n)$, and $\Omega(n^2)$ satisfy this.

(iii) **(4 pt.)** Which of the following correctly describes $T(n) = 9T(n/3) + n^2$?

(A) $O(n^3 \log(n))$ **(B)** $O(n \log^2(n))$ **(C)** $O(n^2)$ **(D)** $\Theta(n^3)$

Solution

A. We can apply the master theorem to see that $T(n) = O(n^2 \log n)$, so only $O(n^3 \log n)$ applies.

(iv) **(4 pt.)** Let $T(n) = n^c$ for some constant $c > 1$. Which of the following describes $2^{T(n)}$?

(A) $O(2^n)$ **(B)** $\Omega(2^n)$ **(C)** $\Theta\left(2^{2\cdot T(n)}\right)$ **(D)** $\Omega(2^{100nc})$

Solution

B, D. We can see that $n^c = \Omega(n)$, but $n^c \neq O(n)$, so (A) is immediately out and (B) and (D) are immediately true. Now, we need to understand (C). Suppose, for a contradiction, that $2^{2\cdot n^c} = O(2^{n^c})$. Then, there must exist constants c', N such that $2^{2\cdot n^c} \leq c' 2^{n^c}$ for all $n \geq N$. However, consider $n = \min(\log_2(c'), N)$. Then, we can see that, because $c > 1$,

$$2^{2\cdot n^c} > 2^n \cdot 2^{n^c} \geq c' 2^{n^c}$$

This is in contradiction with $2^{2\cdot n^c} \leq c' 2^{n^c}$, so $2^{2\cdot n^c} \neq O(2^{n^c})$, meaning that (C) is false.

(v) **(4 pt.)** Which of the following correctly describes $T(n) = \left(n - 2\left\lfloor \frac{n}{2} \right\rfloor\right) n$?

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

Solution

A, C. We can see that $T(n)$ takes on the value n when n is odd and 0 otherwise. Therefore, $T(n)$ is upper bounded by the function n and lower bounded by 0 (and no greater function), so it is both $O(n^2)$ and $O(n)$ because each of these contain the upper bound of n, but it is not $\Omega(n^2)$ nor $\Omega(n)$ because $T(n)$ is always smaller than either of these functions for any even n.

Section 2. Incorrect Proof (20 points)

Lucky the Lackadaisical Lemur is studying sorting algorithms in CS 161. After learning about `MergeSort` and performing some calculations of his own, he believes that he has a new, remarkable result to share with the computer science community: `MergeSort` runs in $O(n)$ time when run on an array A of length n.

He is super excited, because he has established a better bound for `MergeSort` than the $O(n \log n)$ bound we saw in class. Before releasing his result to the academic community, he decides to run his solution by you to double-check. His proof that `MergeSort` runs in $O(n)$ time is broken down into four parts below.

In each part, either assert that his logic is correct, giving 1-2 sentences explaining how you came to that conclusion, or assert that there is a flaw and give a brief explanation about why it is wrong. **Only state that a portion is incorrect if there is a mathematical error contained within.** Note that Lucky is allowed to ignore divisibility issues (like $\frac{n}{2}$), so **ignore any of his potential divisibility "errors."**

(i) **(5 pt.)** Let $T(n)$ be the running time of `MergeSort` when run on a list of length n. we know that T satisfies the recurrence $T(n) \leq 2T\left(\frac{n}{2}\right) + cn$ for some constant c.

Solution

This is correct. Although we might have to worry about divisibility issues for the recurrence (namely, what if n is odd?) we can explicitly overlook this (and let Lucky be a little lackadaisical in this regard). We showed in class that the `Merge` step of `Mergesort` takes $O(n)$ time, so we can upper bound it by the function cn for a large enough constant c. `Mergesort` breaks into two cases (merging the left and right arrays), so the $2T\left(\frac{n}{2}\right)$ term is correct too.

(ii) **(5 pt.)** **Inductive Hypothesis:** There exists a constant c' such that $T(i) \leq c'i$.

Base Case: Let $i = 1$. $T(1)$ is a constant because a list of length 1 is always sorted. Therefore, we can pick a sufficiently large constant c' such that $T(1) \leq c'$, satisfying the inductive hypothesis for $i = 1$.

Solution

This is correct as well. Although this inductive hypothesis is a bit weird, which we will see later in part (iv), it is a valid inductive hypothesis and contains no errors. Furthermore, the base case does an effective job proving the base case for the $i = 1$ case by noticing that the base case of `Mergesort` takes constant time.

(iii) **(5 pt.)** **Inductive Step:** Let k be an integer ≥ 2. Assume that the inductive hypothesis holds for all positive integers $i \leq k-1$. We will show that the inductive hypothesis holds for $i = k$. Note that $T(k) \leq 2T\left(\frac{k}{2}\right) + ck$. Since $k/2 \leq k-1$, we can apply the inductive hypothesis for $i = k/2$ to upper bound the $2T\left(\frac{k}{2}\right)$ term. Hence, there exists a constant c' such that $2T\left(\frac{k}{2}\right) \leq 2c'\left(\frac{k}{2}\right) = c'k$. Therefore,

$$T(k) \leq 2T\left(\frac{k}{2}\right) + ck \leq c'k + ck = (c'+c)k$$

Setting $c'' = c' + c$, we see that there exists a constant c'' such that $T(k) \leq c''k$. Hence the inductive hypothesis holds for $i = k$.

Solution

This is also correct, surprisingly. All that Lucky needs to do to prove the inductive hypothesis is to show that such a constant c' exists for each value of n. That is, because of how the inductive hypothesis is framed, c' can *change* based on the value of n. So, he correctly uses the inductive hypothesis to find the $c'k/2$ bound on the $k/2$ case and then adds that constant to c to get a new constant. Even though this constant changes based off of n, that is okay because that's all he needs to show to prove the inductive hypothesis.

(iv) **(5 pt.)** **Conclusion:** It follows by induction that for any integer n, there exists a constant c' such that $T(n) \leq c'n$. Hence $T(n) = O(n)$.

Solution

This step is incorrect. Indeed, as discussed in the solution for (iii), the choice of c' changes based on the choice of n. Therefore, c' isn't so much a constant at all, as it is a function of n. Therefore, the definition of big O does not apply because it is not upper bounded by a fixed constant c times n for all choices of n.

Section 3. Algorithm Design (50 points)

After his mistakes understanding the runtime of `MergeSort`, Lucky has gotten into the habit of running his proofs by Plucky the Pedantic Penguin. Plucky is preoccupied with a lot of work, but still wants to help Lucky as much as possible.

Lucky has an estimate of how long it takes to run each proof by Plucky. So he sends an array $A = [a_1, \ldots, a_n]$ of these time lengths (measured in minutes) to Plucky. Plucky has put aside a total of L minutes to help Lucky, and wants to go over **as many proofs as possible**.

In this problem, you will design an algorithm that helps Plucky find out the maximum number of proofs that fit within L minutes. You **may assume all time lengths in A are distinct**. If it helps, **you may ignore divisibility issues** (such as floors and ceilings) throughout this problem. You may use any algorithm we have learned in class and cite its correctness and runtime without proof.

Input: Array A of n distinct positive numbers indicating the time it takes to verify each proof, and number $L \geq 0$ indicating how much time Plucky has put aside.

Output: The maximum number of proofs that Plucky can verify in $\leq L$ minutes.

Examples. You can find some sample inputs and the correct outputs below.

- For $A = [5, 1, 6, 2, 3]$ and $L = 3$, the correct output is 2.
- For $A = [20, 10, 30]$ and $L = 5$, the correct output is 0.
- For $A = [4, 1, 2, 3]$ and $L = 1000$, the correct output is 4.

By following the steps below you will design an algorithm that solves this problem and runs in time $O(n)$, where n is the length of A. If the algorithms you design for parts (i), (ii), and (iii) have bugs or do not have the correct runtime, they **might still receive partial credit**.

(i) **(10 pt.)** **Warmup.** Briefly describe or write pseudocode for an $O(n \log(n))$ time algorithm `BaseLineAlgorithm`(A, L) that solves this problem. You do not need to prove its correctness or runtime in this part.

Solution

We can observe that the maximum number of proofs that Plucky can help with is equivalent to the maximum number k such that the sum of the smallest k values of A is $\leq L$. To solve this algorithmically, we simply sort A and then do a linear walk over the array to compute k.

```
BaseLineAlgorithm($(A, L)$)
$B \leftarrow$ MergeSort($(A)$)
for $i = 1, \ldots, n$ do
    if $L < B[i]$ then
        return $i - 1$
    $L \leftarrow L - B[i]$
return $|B|$
```

(ii) **(15 pt.)** **Small or Large.** Briefly describe or write pseudocode for an $O(n)$ time algorithm `IsAnswerAtLeast`(A, L, k) which in addition to A and L receives a number k between 0 and n inclusive, and outputs whether there are **at least** k proofs that Plucky can verify in L minutes. Note that the output here is not a number; it is just True (if Plucky can verify k proofs) or False (if Plucky cannot). You do not need to prove the correctness or runtime of your algorithm in this part.

Below you can find some example inputs and outputs for `IsAnswerAtLeast`.

- For $A = [5, 1, 6, 2, 3]$ and $L = 3$ and $k = 2$, the correct output is True.
- For $A = [5, 1, 6, 2, 3]$ and $L = 3$ and $k = 3$, the correct output is False.
- For $A = [20, 10, 30]$ and $L = 5$ and $k = 0$, the correct output is True.
- For $A = [20, 10, 30]$ and $L = 5$ and $k = 3$, the correct output is False.

Solution

Jumping off of our intuition from (i), we see that we simply need to split our array A into the first smallest k elements and then see if the sum of those is less than L. We can do this using `Select` and then performing another linear walk over the first k elements of A.

```
IsAnswerAtLeast(A, L, k)
p ← Select(A, k)
for i = 1, ..., n do
    if A[i] ≤ p then
        L ← L − A[i]
if L ≥ 0 then
    return yes
else
    return no
```

(iii) **(15 pt.) Algorithm.** Briefly describe or write pseudocode for an $O(n)$ time algorithm `MaximumProofs`(A, L) that solves the original problem, that is, outputs the maximum number of proofs that Plucky can verify in L minutes. You do not need to prove its correctness or runtime in this part.

(Hint: This is the hardest problem on the exam! If you get stuck, move to other unsolved problems.)

Solution

We can once again jump off of our intuition from (ii) and (i) to see that the `Select` algorithm will be helpful for us. Let k be the largest integer such that the sum of the smallest k values of A is at most L. Then, we use `Select` to choose the median of the array and split it into two halves: the part greater than the median and the part smaller than the median.
Then, we look to see if the first half of the array's sum is $< L$. If it is, then we can definitely include all of those, and we can recurse on the right half of the array to see if there are any more we can include there. If it isn't, then we immediately know that the smallest k elements must all be smaller than our median (because we've counted too many already).

```
MaximumProofs($A, L$)
if $|A| = 0$ then
    return 0
if $|A| = 1$ then
    if $A[0] \leq L$ then
        return 1
    else
        return 0
$x \leftarrow$ Median($A$)
$A_< = [A[i] \mid A[i] < x]$
$A_\geq = [A[i] \mid A[i] \geq x]$
$S \leftarrow$ Sum($A_<$)
if $S \leq L$ then
    return $|A_<| +$ MaximumProofs($A_\geq, L - S$)
else
    return MaximumProofs($A_<, L$)
```

(iv) **(10 pt.)** Prove the runtime of your algorithm (`MaximumProofs`) from part (iii). That is, either argue using a recurrence relation or otherwise that your algorithm satisfies the time bound of $O(n)$.

Solution

This answer will change based on your solution to (iii). In particular, for our algorithm, we do $O(n)$ work (from the constructions of $A_<$, $A_\geq$, and S) before recursing once on a list of at most half the length of the original. So, in total, our algorithm follows the recurrence $T(n) = T(n/2) + O(n)$, which we can use the Master theorem to solve and see that $T(n) = O(n)$.