CS 161 **Problem Set 1**
Fall 2025 **Due:** Friday October 3 at 11:59pm on Gradescope.

Style guide and expectations: Please see the top of the "Homework" page on the course webpage for guidance on what we are looking for in homework solutions. We will grade according to these standards.

Make sure to look at the "**We are expecting**" blocks below each problem to see what we will be grading for in each problem!

Collaboration policy: You may do the HW in groups of size up to three. Please submit one HW for your whole group on Gradescope. (Note that there is an option to submit as a group). See the "Policies" section of the course website for more on the collaboration policy.

LLM policy: Check out the course webpage for best practices on how to productively use LLMs on homework, if you use them at all.

Exercises

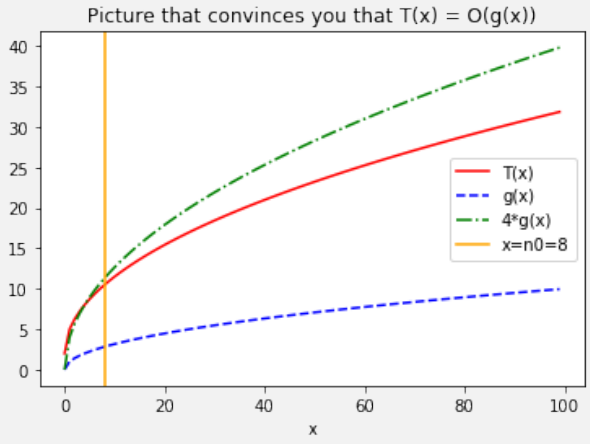
We recommend you do the exercises on your own before collaborating with your group. The point is to check your understanding.

---

1. **(1 pt.)** See the IPython notebook HW1.ipynb for Exercise 1. Modify the code to generate a plot that convinces you that $T(x) = O(g(x))$. **Note:** There are instructions for installing Jupyter notebooks in the pre-lecture exercise for Lecture 2.

   [**We are expecting:** *Your choice of c, n_0, the plot that you created after modifying the code in Exercise 1, and a short explanation of why this plot should convince a viewer that $T(x) = O(g(x))$.*]

   SOLUTION: We choose $c = 4$ and $n_0 = 8$. As we can see in the picture below, for all $n > n_0$ (that is, to the right of the yellow vertical line), we have $cg(n) \geq T(n)$, meaning that the green dashed curve lies above the solid red curve.



2. **(9 pt.)** For each row, indicate whether or not the quantity in column **A** is O, Ω, or Θ of the quantity in column **B**. Put a ✓ if your answer is "yes", and leave blank if your answer is "no." We've filled in the first two rows for you. Notice that it's possible for multiple spaces per row to have a ✓ in them. All logarithms are base 2 unless otherwise stated.

   [**We are expecting:** *Some of the blanks to be marked with a ✓ (or X, or ☺, or your favorite symbol). No explanation is required.*]

   Note: The LaTeX template for this problem set has the code for the table.

	A	**B**	O	Ω	Θ
1	$\log n$	n	✓		
2	$n/2023$	$2023 \cdot n$	✓	✓	✓
3	n^3	n^2			
4	3^n	2^n			
5	n^2	2^n			
6	$n^{0.5}$	$(0.5)^n$			
7	$\ln n$	$\log_{10} n$			
8	n^3	$8^{\log_2 n}$			
9	$n^{1/\ln n}$	1			
10	$n^{1/3}$	$(\log n)^3$			
11	$\log \log n$	$\sqrt{\log n}$			

SOLUTION:

	A	**B**	O	Ω	Θ
1	$\log n$	n	✓		
2	$n/2025$	$2025 \cdot n$	✓	✓	✓
3	n^3	n^2		✓	
4	3^n	2^n		✓	
5	n^2	2^n	✓		
6	$n^{0.5}$	$(0.5)^n$		✓	
7	$\ln n$	$\log_{10} n$	✓	✓	✓
8	n^3	$8^{\log_2 n}$	✓	✓	✓
9	$n^{1/\ln n}$	1	✓	✓	✓
10	$n^{1/3}$	$(\log n)^3$		✓	
11	$\log \log n$	$\sqrt{\log n}$	✓		

3. **(4 pt.)**

(a) **(2 pt.)** Prove that n is $O(n \log_2 n)$.
[**We are expecting:** *A formal proof, using the definition of* $O(\cdot)$ *that we saw in class.*]

(b) **(2 pt.)** Prove that n is *not* $\Omega(n \log_2 n)$.
[**We are expecting:** *A formal proof, using the definition of* $\Omega(\cdot)$ *that we saw in class.*]

SOLUTION:

(a) To show that n is $O(n \log_2 n)$, we need to find a c and an n_0 so that for all $n \geq n_0$, we have $n \leq cn \log_2 n$. We choose $c = 1$ and $n_0 = 2$. Notice that for all $n \geq n_0$, we have

$$\begin{aligned} 1 &\leq \log_2 n \\ n &\leq n \log_2 n \\ n &\leq cn \log_2 n. \end{aligned}$$

Above, we multiplied both sides of the equation by n, and then we plugged in $c = 1$. This is what we wanted to show, and we are done.

(b) To show that n is *not* $\Omega(n \log_2 n)$, we use proof by contradiction. Suppose towards a contradiction that there was some c and n_0 so that $n \geq cn \log_2 n$ for all $n \geq n_0$. That implies that $\frac{1}{c} \geq \log_2 n$ for all $n \geq n_0$, which implies that $2^{1/c} \geq n$ for all $n \geq n_0$. But now consider $n = \max\{n_0, 2^{1/c} + 1\}$. This n satisfies $n \geq n_0$, but by construction we have $2^{1/c} < n$, a contradiction.

Problems

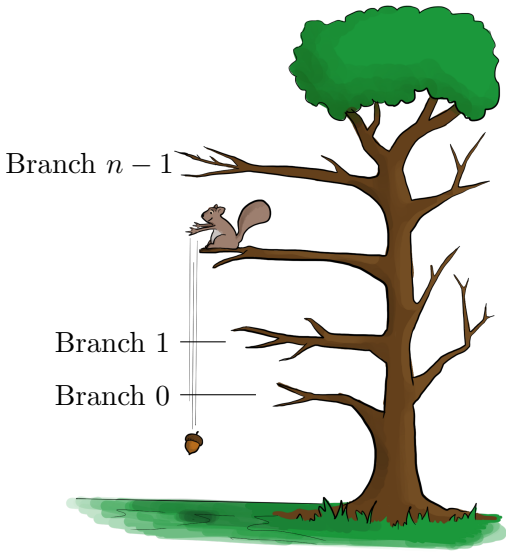
---

4. **[Nuts!] (14 pt.)**

*[**Meta problem-solving skills:** Another version of this problem could jump straight to part (d). The structure of this problem, particularly (b) and (c), is supposed to give you an example of how to solve a harder problem by building up from simpler problems. In the future, you'll have to do more and more of this on your own!]*

Socrates the Scientific Squirrel is conducting some experiments. Socrates lives in a very tall tree with n branches, and she wants to find out what is the lowest branch i so that an acorn will break open when dropped from branch i. (If an acorn breaks open when dropped from branch i, then an acorn will also break open when dropped from branch j for any $j \geq i$.)

The catch is that, once an acorn is broken open, Socrates will eat it immediately and it can't be dropped again.



(a) **(2 pt.)** Suppose that Socrates has $\lceil \log(n) \rceil + 1$ acorns. Give a procedure so that she can identify the correct branch using $O(\log(n))$ drops.

[**We are expecting:** *Very clear pseudocode **or** a short English description of your algorithm. You do not need to justify the number of drops. If it helps you may assume that n is a power of* 2.]

(b) **(2 pt.)** Suppose that Socrates has only one acorn. Give a procedure so that she can identify the correct branch using $O(n)$ drops, and explain why your $O(\log(n))$-drop solution from part (a) won't work.

[**We are expecting:** *Very clear pseudocode **or** a short English description of your algorithm, and one sentence about why your algorithm from part (a) does not apply. You do not need to justify the number of drops. If it helps you may assume that the acorn breaks when dropped from the top branch (in all parts of this problem).*]

(c) **(3 pt.)** Suppose that Socrates has two acorns. Give a procedure so that she can identify the correct branch using $O(\sqrt{n})$ drops.

[**We are expecting:** *Pseudocode **AND** a short English description of your algorithm, and a justification of the number of drops. If it helps you may assume that n is a perfect square.*]

(d) **(5 pt.)** Suppose that Socrates has $k = O(1)$ acorns. Give a procedure so that she can identify the correct branch using $O(n^{1/k})$ drops.

[**We are expecting:** *Pseudocode **AND** a short English description of your algorithm, and a justification of the number of drops. If it helps you many assume that n is of the form $n = m^k$ for some integer m.*]

(e) **(2 pt.)** What happens to the runtime of your algorithm in part (d) when $k = \lceil \log(n) \rceil + 1$? Is it $O(\log(n))$, like in part (a)? Is it $O(n^{1/k})$ when $k = \lceil \log(n) \rceil + 1$, like in part (d)?

[**We are expecting:** *A sentence of the form "the number of drops of my algorithm in part (d) when $k = \lceil \log(n) \rceil + 1$ is $O(\_\_\_)$", along with justification.* ***Also****, we are expecting two yes/no answers to the two yes/no questions (you should justify your answers but do not need to include a formal proof).*]

(f) **(NOT REQUIRED. 1 BONUS pt.)** Is $\Theta(n^{1/k})$ drops is the best that Socrates can do with k acorns, for $k = O(1)$? Either give a proof that she can't do better, or give an algorithm with asymptotically fewer drops.

[**We are expecting:** *Nothing. This part is not required.*]

SOLUTION:

(a) We will assume that n is a power of 2, since the "**We are expecting**" block said that we could. With $\lceil \log(n) \rceil + 1$ acorns, Socrates can do binary search. That is, she first drops an acorn from branch $n/2$. If it breaks, she goes to branch $n/4$, and if not she goes to branch $3n/4$. Then she recurses. The pseudocode looks like this:

```
def acornDropping(lower,upper):
    if lower == upper:
        return lower
    else:
        drop an acorn from (lower + upper)/2
        if it breaks:
            return acornDropping( lower, (lower + upper)/2  )
        else:
            return acornDropping( (lower + upper)/2 + 1, upper )
```

(b) With only one acorn, the best Socrates can do is start at the bottom of the tree and try the first branch, then the second, and so on up the tree. If the acorn breaks at branch j, then Socrates knows that branch j is the correct solution. The binary search algorithm from part (a) won't work because she might break her acorn on the first drop and then be out of luck.

(c) The problem says that we can assume that n is a perfect square, so we will do that. With two acorns, Socrates can use the first acorn to narrow down the possibilities to a window of size $\sqrt{n}$. Then she can use the second acorn to figure out which the right branch is in that window. That is, first she would drop an acorn from branch (0-indexed) $\sqrt{n} - 1$, then $2\sqrt{n} - 1$, then $3\sqrt{n} - 1$, and so on. If the acorn breaks at $j\sqrt{n} - 1$, she knows that the "correct" branch is the set $\{(j-1)\sqrt{n}, (j-1)\sqrt{n} + 1, (j-1)\sqrt{n} + 2, \ldots, j\sqrt{n} - 1\}$. So for her second acorn she starts at $(j-1)\sqrt{n}$, then $(j-1)\sqrt{n} + 1$, and so on, until the acorn breaks at $(j-1)\sqrt{n} + i$. Then Socrates knows that the correct branch is $(j-1)\sqrt{n} + i$.

```
# suppose that n is a perfect square
def acornDropping():
   for i in {1,2,...,sqrt(n)}:
      drop the first acorn from branch i*sqrt(n)-1
      if the acorn breaks:
         i1 = i-1
         break
   for j in {0,1,...,sqrt(n)-1}:
      drop the second acorn from branch i1*sqrt(n) + j
      if the acorn breaks:
         return i1*sqrt(n) + j
```

The number of drops is $O(\sqrt{n})$ because each of the two for-loops run for $O(\sqrt{n})$ iterations and each step of each contains a single drop.

(d) With k acorns, we can generalize the solution to problem (c). We use the first acorn to narrow down the possibilities to a window of size $n^{1-1/k}$, the second to get to a window of size $n^{1-2/k}$, and so on. That is, we drop the first acorn from branches

$$0, n^{1-1/k}, 2 \cdot n^{1-1/k}, 3 \cdot n^{1-1/k} \dots$$

until it breaks, say at branch $j \cdot n^{1-1/k}$. Then, starting from branch $b = (j-1) \cdot n^{1-1/k}$ (the last point we knew the acorn wouldn't break), we drop the second acorn from branches

$$b, b + n^{1-2/k}, b + 2n^{1-2/k}, b + 3n^{1-2/k}, \dots$$

until it breaks. Then, starting from the last branch we know was good, we repeat again with intervals of size $n^{1-3/k}$ and so on. At the end, we will have narrowed it down to a window of size $n^{1-k/k}$, in which case we are done.

More precisely, the pseudocode is as follows:

Algorithm 1: acornDropping(n,k):

return acornDropping_helper(n,k,1,-1)

Algorithm 2: acornDropping_helper(n,k,ℓ, highestGood)

Input: n is the number of branches, k is the number of acorns, ℓ is the level of refinement, and highestGood is the highest branch we know that an acorn will not break when dropped from.

if $\ell = k + 1$ **then**
  return *highestGood* +1

Choose an unbroken acorn, let's call it Achilles the Acorn.

for $j \in \{1, 2, ..., n^{1/k}\}$ **do**
  Drop Achilles the Acorn from branch highestGood + $j \cdot n^{1-\ell/k}$
  if *Achilles the Acorn breaks* **then**
    AchillesBreakPoint = j
    break

/* AchillesBreakPoint is now the first value so that Achilles breaks at highestGood + AchillesBreakPoint$\cdot n^{1-\ell/k}$ Notice that we already know (from the previous level) that Achilles breaks at highestGood + $n^{1/k} \cdot n^{1-\ell/k}$, so this variable does get set. */

/* now recurse to the next level, passing in the highest point from which Achilles did not break */

return *acornDropping_helper(n,k,*$\ell + 1$*, highestGood* + (*AchillesBreakPoint* − 1) $\cdot n^{1-\ell/k}$*)*

The number of drops is $O(k \cdot n^{1/k})$. At each level ℓ of the recursion, the algorithm uses at most $n^{1/k}$ drops. There are k levels of recursion (and no branching), so the total is $k \cdot n^{1/k}$. Thus when $k = O(1)$, the number of drops is $O(n^{1/k})$.

(e) As we saw above, the number of drops for this algorithm is in general $\Theta(kn^{1/k})$. When $k = \lceil \log(n) \rceil + 1$, this is $O(\log(n) \cdot n^{1/\log(n)})$. However, $n^{1/\log(n)} = 2$, so the number of drops is $\Theta(\log(n))$.

Thus, the answers to the two True/False questions are True and False respectively. That is, the number of drops for this k is $O(\log(n))$, but not $O(n^{1/k})$, since the latter is $O(1)$ when $k = \lceil \log(n) \rceil + 1$.

(f) In fact, for $k = O(1)$ the number of drops must be $\Omega(n^{1/k})$. This is actually pretty tricky to show! First, we claim that if d is the maximum number of drops allowed, then the largest number of branches n so that k acorns can find the correct branch with d drops satisfies:

$$n \leq \sum_{j=0}^{k} \binom{d}{j}.$$

To prove the claim, suppose that we have some (deterministic) dropping algorithm, that works with at most d drops. For each possible branch $i \in \{0, \ldots, n-1\}$, if i was the correct branch then that uniquely defines a "break/not-break pattern" that the algorithm will see. For example, if branch 11 was the correct branch, maybe the algorithm would say:

`break, not-break, not-break, break, break, break, not-break, break`

and then (because are assuming that the algorithm works) we should conclude that the correct branch was 11. If the sequence has length less than d (that is, we used fewer than our allotted d drops), then let's pad the sequence with "`not-break`"'s until we get a sequence of exactly d. This gives a mapping from the set $\{0, 1, \ldots, n-1\}$ into the set of sequences of the form `break, not-break, break, break, ..., break` which have:

- At most k "`break`"'s (because we have at most k acorns to work with), and
- Exactly d elements in the sequence.

Notice also that this mapping is *injective*. That is, for any two different branch numbers (like "11" or "53"), they cannot map to the same sequence. This is because, since the algorithm works, I can't simultaneously conclude that the correct branch is both "11" and "53." (Notice that the padding doesn't mess this up. Once I've concluded that the correct answer is 11, making more drops is not going to convince me that the answer is 53.)

That means that

(number of branches) $\leq$ (number of sequences that fit the above description).

How many sequences fit this description? For every $j \leq k$, there are $\binom{d}{j}$ sequences with exactly j "`break`'s." Thus, the number of such sequences is

$$\sum_{j=0}^{k} \binom{d}{j}.$$

Thus, the above says that

$$n \leq \sum_{j=0}^{k} \binom{d}{j}.$$

Once we have this formula, we have (assuming that $k \leq d/2$)

$$n \leq k\binom{d}{k} \leq (k+1)d^k,$$

where we arrived that this by estimating $\binom{d}{j} \leq d^j \leq d^k$ for each term in the sum. But now solving for d (the number of drops), we have

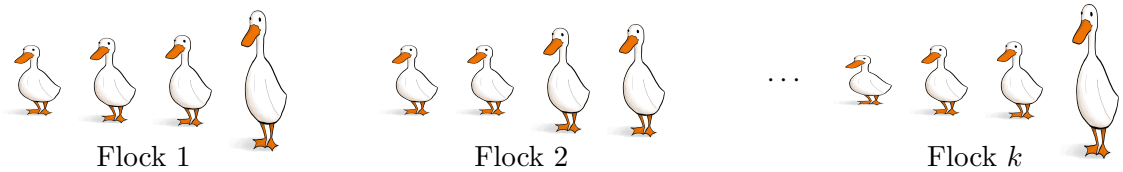
$$d \geq \left(\frac{n}{k+1}\right)^{1/k} = \Omega(n^{1/k})$$

when $k = O(1)$.

5. **[Ducks in a row.] (12 pt.)**

*[**Meta problem-solving skills:** As the hint in part (b) suggests, you might want to consider an algorithm you've already seen to solve this problem. In general, taking inspiration from algorithms you already know is a good problem-solving technique!]*

There are k flocks of ducks, and each flock has n ducks each. The k flocks are coming together for a mixer, and for a particular event, they would like to sort all kn ducks by height.[1] Each flock submits a height-ordered list of its ducks, and you (the organizer) are presented with k ordered lists, $A_1, \ldots, A_k$, each of length n.



(a) **(1 pt.)** Your assistant, having just seen Lecture 2, is excited to try MergeSort on the ducks. They suggest concatenating the lists (to get a big list of length nk), and then running MergeSort on this big list. How long will this take?

[**We are expecting:** *A single big-Oh expression. No justification is needed.*]

(b) **(7 pt.)** Suppose that k is significantly smaller than n. (For example, say that $k = O(\log n)$). Design an algorithm that is *asymptotically faster* than the algorithm suggested in part (a) [2].

[**Hint:** *Take inspiration from MergeSort.*]

[**We are expecting:** *Pseudocode **AND** a short English description of your algorithm; **AND** a clear statement of the running time; **AND** an explanation for why this is asymptotically faster than your answer in part (a) when $k = \log(n)$.*

You do not need justify the running time, but you might want to do so for your own confidence and/or for partial credit. You do not need to formally prove that your running time is asymptotically faster than part (a), but you should give a clear explanation, don't just write down two ugly expressions and assert that one is smaller than the other.]

(c) **(4 pt.)** Rigorously prove by induction that your algorithm is correct. If it's relevant, you may assume that the MERGE algorithm that we saw in class is correct. If it helps, you may assume that k is a power of 2.

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

SOLUTION:

(a) $O(nk \log(nk))$.

[1] Bonus points for coming up with the most creative description of this event. The course staff is going with an elaborate duck line dance.

[2] Here, "asymptotically faster" means that if the running time of your algorithm is $T(n)$, and if $T_0(n)$ is your running time from part (a), then $T(n) = O(T_0(n))$ but $T_0(n)$ is *not* $O(T(n))$.

(b) **English Description.** We will do a divide-and-conquer algorithm. We will divide the k flocks into two groups of size $k/2$, recursively sort each flock, and then merge them together. At the base case, we have a single flock, which is already sorted.

Pseudocode.

```
def ducksInARow(A1, ..., Ak):
    if k==1:
       Return A1
    else:
       b = floor(k/2)
       L = ducksInARow(A1, ..., Ab)
       R = ducksInARow(A(b+1), ..., Ak)
       Return Merge(L,R) // Merge is the same Merge function from class
```

Running time. The running time is $O(nk\log(k))$. The reason is that, if we think about the recursion tree, it has depth $O(\log_2(k))$ (the number of times we have to cut k in half to get down to 1). At each level, we do $O(nk)$ work, since the MERGE subroutine (across all the subproblems) touches each of the nk ducks $O(1)$ times. Together this is $O(nk\log k)$.

Comparison to part (a). When $k = O(\log n)$, say, this is $O(n\log(n)\log\log(n))$. In contrast, our answer from part (a) is

$$O(nk\log(nk)) = O(n\log n\log(n\log n)) = O(n\log^2(n)).$$

Since $\log\log n$ is asymptotically smaller than $\log n$, $n\log n\log\log n$ is asymptotically smaller than $n\log^2 n$.

(c) We prove the correctness of our algorithm by induction. As the problem says we can, we'll assume that k is a power of 2.

- **Inductive Hypothesis.** Our inductive hypothesis for $j = 0, \ldots, \log_2(k)$ is: When `ducksInARow` is called on 2^j lists $A_1, \ldots, A_{2^j}$, it correctly returns a sorted list of all $n2^j$ elements in all of those lists.
- **Base case.** When $j = 0$, the inductive hypothesis reads "When `ducksInARow` is called on one list, it correctly returns a sorted list of the n elements in A_1. This is true because in that case, our algorithm just returns A_1 itself, which by assumption is already sorted.
- **Inductive step.** Assume that the IH holds for $j \geq 0$. We want to show that it holds for $j+1$. Consider running `ducksInARow` on 2^{j+1} lists, $A_1, \ldots, A_{2^{j+1}}$. We can apply the inductive hypothesis to both recursive calls, and conclude that L and R are sorted lists, which together contain all of the elements in $A_1, \ldots, A_{2^{j+1}}$. Further, since we can assume that `Merge` is correct (since this is an algorithm we saw in class, and the problem says we can assume it is correct), this means that the output of `Merge(L,R)` is sorted. This establishes the inductive hypothesis for $j+1$.

- **Conclusion.** We conclude that the IH holds for $j = 0, 1, \ldots, \log_2(k)$. In particular, it holds for $j = \log_2(k)$. In this case, the IH reads that "when `ducksInARow` is called on k lists $A_1, \ldots, A_k$, it correctly returns a sorted list containing all of the elements in $A_1, \ldots, A_k$, which is what we wanted to show.