CS 161 **Problem Set 1**
Winter 2019 **Due:** Friday, January 18, 2019 at 3pm on Gradescope.

Style guide and expectations: Please see the "Homework" part of the "Resources" section on the 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!

Exercises

Please do the exercises on your own.

---

0. **(1 pt.)** Have you thoroughly read the course policies on the webpage?

   [**We are expecting:** *The answer "yes."*]

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 $T(x) = O(g(x))$.*]

2. **(3 pt.)** See the IPython notebook `HW1.ipynb` for Exercise 2, parts A, B and C.

   (A) What is the asymptotic runtime of the function `numOnes( lst )` given in the Python notebook? Give the smallest answer that is correct. (For example, it is true that the runtime is $O(2^n)$, but you can do better).

   [**We are expecting:** *Your answer in the form "The running time of `numOnes( lst )` on a list of size n is $O(\_\_\_)$.", and a short explanation of why this is the case.*]

   (B) Modify the code in `HW1.ipynb` to generate a picture that backs up your claim from Part (A).

   [**We are expecting:** *Your choice of c, n_0, and $g(n)$; the plot that you created after modifying the code in Exercise 2; and a short explanation of why this plot should convince a viewer that the runtime of `numOnes` is what you claimed it was.*]

   (C) How much time do you think it would take to run `numOnes` on an input of size $n = 10^{15}$?

   [**We are expecting:** *Your answer (in whichever unit of time makes the most sense) with a short explanation that references the runtime data you generated in part (B). You don't need to do any fancy statistics, just a reasonable back-of-the-envelope calculation.*]

3. **(4 pt.)** Using the definition of big-Oh, formally prove the following statements.

   (a) $3\sqrt{n} + 2 = O(\sqrt{n})$ (Note that you gave a "proof-by-picture" of this in Exercise 1).

   (b) $n^2 = \Omega(n)$.

   (c) $2^{2^{2^{100}}} = \Theta(1)$.

   (d) 4^n is **not** $O(2^n)$.

   [**We are expecting:** *A proof for each part, using the definition of $O(), \Omega()$, and $\Theta()$.*]

Problems

You may talk with your fellow CS161-ers about the problems. However:

- Try the problems on your own *before* collaborating.
- Write up your answers yourself, in your own words. You should never share your typed-up solutions with your collaborators.
- If you collaborated, list the names of the students you collaborated with at the beginning of each problem.

---

1. [**True or false?**] **(4 pt.)** In the following, suppose that $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$ are strictly increasing functions. (Recall that $\mathbb{N}$ denotes the natural numbers $\{0, 1, 2, \ldots\}$).

   True or false?

   (a) **(2 pt.)** If $f(n) = O(g(n))$, then $\log(f(n)) = O(\log(g(n)))$. (If it helps, you may assume that f and g are strictly positive).

   (b) **(2 pt.)** If $f(n) = O(g(n))$, then $2^{f(n)} = O\left(2^{g(n)}\right)$.

   [**We are expecting:** *For each part, either a proof or a counter-example (along with a proof that your counter-example is a counter-example), using the definition of* $O(\cdot)$.]

2. [**Alternative MergeSorts**] **(6 pt.)**

   In class, we saw how MergeSort works by recursively breaking up a list into two smaller lists. Consider the following version, which recursively breaks a list up into three smaller lists:

```
# Assume that A is a list of distinct integers and len(A) is a power of 3.
# This function returns a sorted version of A.
def MergeSort3(A):
     n = len(A)
     if n <= 3:
         return InsertionSort(A)
            # It takes time O(1) to InsertionSort a list of length <= 3.
     L = MergeSort3(A[0:n/3])
     M = MergeSort3(A[n/3:2n/3])
     R = MergeSort3(A[2n/3:n])
     return Merge3(L,M,R)
         # Merge3 merges three sorted lists of size n/3 into a sorted list of size n.
```

   (a) **(2 pt.)** Write pseudocode for `Merge3` that runs in time $O(n)$. Your function should take as input three sorted lists `L`, `M`, and `R` of length $n/3$ (where n is a power of 3) and return a sorted list of length n which contains all the entries of `L` and `M` and `R`.

   [**We are expecting:** *Pseudocode* ***AND*** *an English description of what it does, as well as an informal explanation of why the running time is* $O(n)$*. You may assume that* n *is a power of* 3 *and that all of the elements across* ***L,M,R*** *are distinct.*]

   (b) **(2 pt.)** Show that, with your version of `Merge3` from part (a), `MergeSort3` runs in time $O(n\log(n))$ when run on a list of length n.

   [**We are expecting:** *An informal but convincing argument. Do not use the Master Theorem, instead argue this "from scratch" like we did in Lecture 2.*]

(More parts on next page...)

(c) **(2 pt.)** Your friend has gotten pretty excited by this, and thinks they have a sorting algorithm that runs in time $O(n \log\log(n))$, even faster than MergeSort. Here is their reasoning:

i. Instead of `MergeSort3` as above, consider a version `MergeSort_k` which breaks up the list A recursively into k parts, and uses a routine `Merge_k` similar to the `Merge3` you wrote in part (a).

ii. The routine `Merge_k` takes k sorted lists of size n/k, and returns a sorted list of size n. Your approach in part (a) still applies: we've already seen that `Merge_k` runs in time $O(n)$ for $k = 2$ and $k = 3$, and it's not hard to see that the natural generalization also runs in time $O(n)$ for $k = 4, 5, 6, \ldots$.

iii. Now instantiate this algorithm `MergeSort_k` for $k = \sqrt{n}$. That is, at each step we divide a list of size n into $\sqrt{n}$ pieces of size $\sqrt{n}$, and recurse on those. (For simplicity, assume that n is of the form $n = 2^{2^t}$ for some t so that $\sqrt{n}$ and $\sqrt{\sqrt{n}}$ and so on is always an integer until $n = 2$.)

iv. Now we have a recursive algorithm with the following properties:

- A problem of size n gets broken up into $\sqrt{n}$ problems of size $\sqrt{n}$.
- The work to run `Merge_`$\sqrt{n}$ for a subproblem of size n is $O(n)$ by part (ii).

The running time is $O(n \log\log(n))$. To see this, first notice that there are $O(\log\log(n))$ levels in the recursion tree, since that's how many times you need to take the square root of n to get down to problems of size $O(1)$. At the 0'th level of the recursion tree, there is one problem of size n. At level 1, there are $\sqrt{n}$ problems of size $\sqrt{n}$. At level 2, there are $\sqrt{n} \cdot n^{1/4} = n^{3/4}$ problems of size $n^{1/4}$. In general, at the j'th level there are $n^{1-1/2^j}$ problems of size $n^{1/2^j}$. Thus, the amount of work at level j is $O(n^{1-1/2^j} \cdot n^{1/2^j}) = O(n)$. Thus, since there is $O(n)$ work per layer for each of $O(\log\log(n))$ layers, the total amount of work is $O(n \log\log(n))$.

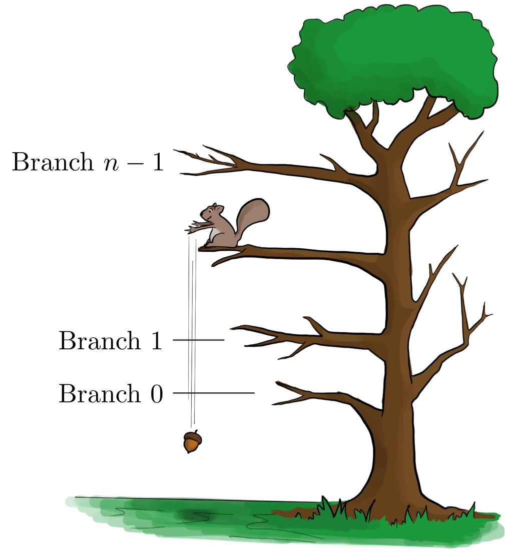
This is pretty neat! Unfortunately, it's not correct. What is the problem with your friend's argument? (Don't go looking for little bugs—there is a big conceptual error. The assumption that n is of the form 2^{2^t} is fine.)

[**We are expecting:** *An identification of which step(s) of the argument (i)-(iv) are problematic, and a clear explanation about what is wrong.*]

(More questions on next page...)

3. **[Nuts!] (8 pt.)**

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) **(1 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.*]

(c) **(2 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) **(2 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) **(1 pt.)** What happens to 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, 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.*]

Feedback

This part is not worth any points, but it is quick, painless, and anonymous, and we'd really appreciate it if you'd help us out by giving us feedback!

1. **(0 pt.)** Please fill out the following poll, which asks about your expectations for the course:

   `https://goo.gl/forms/RKAgpsDPu2IT9RWE3`