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?

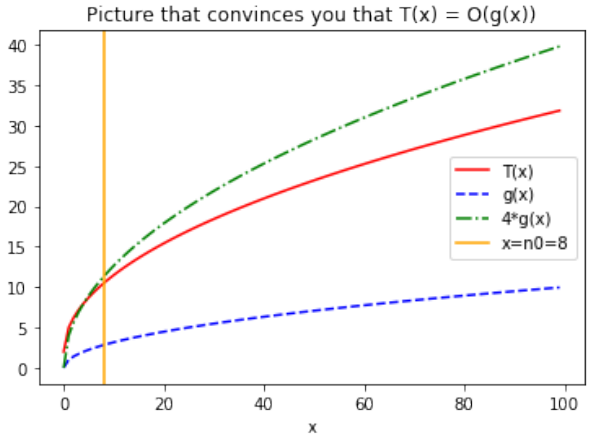
   [**We are expecting:** *The answer "yes."*]

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

   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. **(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.*]

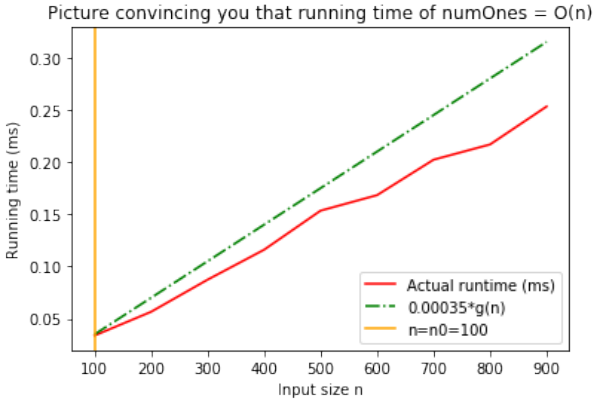
(B) Modify the code in `HW1.ipynb` to generate a picture that backs up your claim from Part (A).
[**We are expecting:** *Your choice of c, n_0, and $g(n)$; the plot that you created after modifying the code in Exercise 2; and a short explanation of why this plot should convince a viewer that the runtime 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.*]

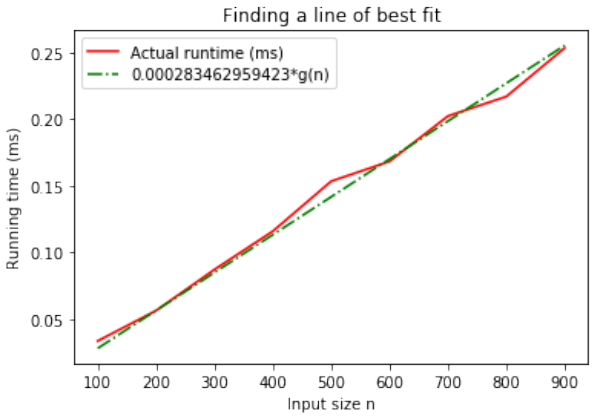
SOLUTION:

(A) The running time of `numOnes( lst )` on an input list of size n is $O(n)$. This is because the for loop goes through n iterations, and in each iteration it does a constant number of operations: it tests if $x == 1$, and then it might increment a counter. Thus, the running time is $O(n)$.

(B) See the picture below, where we have chosen $c = 0.00035$ and $n_0 = 100$. It seems in the picture that for all $n > n_0$ (that is, to the right of the yellow vertical line), the green dashed curve lies above the solid red curve, meaning that $cg(n) \geq T(n)$, where $g(n) = n$ and $T(n)$ is the running time of `numOnes` on an input of size n.



(C) Now that we've convinced ourselves through both parts (A) and (C) that the runtime is roughly linear, we can roughly model $T(n)$ (the running time of `numOnes` as $T(n) = a \cdot n$ for some slope a. In order to predict $T(10^{15})$ (without actually running it...) we should estimate a. For this problem, eyeballing it is fine for full credit. However, we chose to solve a least squares problem to find the *best* slope, which in our case turned out to be $a \approx 0.00028$. (Of course, this will be different on different computers, and even different runs on the same computer!) Projecting out, we find that the number of milliseconds that this would take is $10^{15} \cdot a$, which works out to about 283 billion milliseconds, or 3280 days, or about 9 years. Good thing we didn't try it!



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* $\Theta()$.]

   SOLUTION:

   (a) Choose $c = 5$ and $n_0 = 2$. We need to show that for all $n \geq n_0$, we have

   $$3\sqrt{n} + 2 \leq 5\sqrt{n}.$$

   Solving the above equation for n, it is equivalent to

   $$n \geq 1.$$

   This is certainly true for all $n \geq 2$, so we are done.
   Note: the solution above is written in a way that makes it clear how we got the answer. Another correct solution would be to go the other way: Choose $c = 5$ and $n_0 = 2$. Then it is true that for all $n \geq n_0$, we have

   $$\begin{aligned} n &\geq 1 \\ \sqrt{n} &\geq 1 \\ 2\sqrt{n} &\geq 2 \\ 5\sqrt{n} &\geq 3\sqrt{n} + 2 \\ c\sqrt{n} &\geq 3\sqrt{n} + 2. \end{aligned}$$

   Thus, for all $n \geq n_0$, $3\sqrt{n} + 2 \leq c\sqrt{n}$, hence $3\sqrt{n} + 2 = O(\sqrt{n})$.

   (b) Choose $n_0 = c = 1$. We need to show that for all $n \geq n_0$, we have

   $$n \leq n^2.$$

   Cancelling an n from each side (which we may do because $n > 0$), we see that this is equivalent to $n \geq 1$, which is indeed true for all $n \geq n_0$.

(c) In order to show that something is $\Theta(1)$, we need to show formally that it is $O(1)$ and $\Omega(1)$.

First, we show that $2^{2^{2^{100}}} = O(1)$. We choose $n_0 = 0$ and $c = 2^{2^{2^{100}}}$. Then it is true that for all $n \geq n_0$,

$$2^{2^{2^{100}}} \leq c \cdot 1,$$

and so

$$2^{2^{2^{100}}} = O(1).$$

Notice that the quantifier "for all $n \geq n_0$" is trivial here, since there's no n involved.

Next we show that $2^{2^{2^{100}}} = \Omega(1)$. Using the definition of $\Omega()$, we again choose $n_0 = 0$ and we choose $c = 1$. Then we have, for all $n \geq n_0$,

$$2^{2^{2^{100}}} \geq c \cdot 1 = 1,$$

and so $2^{2^{2^{100}}} = \Omega(1)$ as well. (Again, the "for all $n \geq n_0$" isn't doing anything here).

(d) Suppose toward a contradiction that 4^n is $O(2^n)$. Then there is some c, n_0 so that

$$4^n \leq c2^n \quad \forall n \geq n_0.$$

Taking logarithms of both sides,

$$n \log(4) \leq \log(c) + n \quad \forall n \geq n_0.$$

Since $\log(4) = 2$ (the log is base 2, as with all logarithms in this course), this is the same as

$$n \leq \log(c) \forall n \geq n_0.$$

Choosing $n = n_0 + \lceil \log(c) \rceil + 1$ gives a contradiction, because this is an n which is both $\geq n_0$ and also is strictly larger than $\log(c)$.

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 $O(\cdot)$.*]

   SOLUTION:

   (a) **True.** Suppose that $f(n) = O(g(n))$. Then there is some c, n_0 so that for all $n \geq n_0$, $0 \leq f(n) \leq c \cdot g(n)$. Without loss of generality, we may assume that $c > 1$; otherwise, we can replace c with $c + 1$ and the above still holds. Since f and g are strictly increasing and integer-valued, there is some n_0' so that $f(n), g(n) > 1$ for all $n \geq n_0'$. Let $n_0'' = \max\{n_0, n_0'\}$. Then for all $n \geq n_0''$,

   $$\log(f(n)) \leq \log(c \cdot g(n)) = \log(c) + \log(g(n)).$$

   Choose

   $$c' = 1 + \frac{\log(c)}{\log(g(n_0''))}.$$

   (Notice that since $g(n_0'') > 1$, $\log(g(n_0'')) > 0$, and so $c' > 0$; also we assumed above that $c > 1$ so $\log(c) > 0$). This implies that

   $$0 < \log(c) = (c' - 1)\log(g(n_0'')) \leq (c' - 1)\log(g(n))$$

   for all $n \geq n_0''$ (using the fact that $g(n)$ and hence $\log(g(n))$ is increasing.) Then we have, for all $n \geq n_0''$,

   $$\log(f(n)) \leq \log(c) + \log(g(n)) \leq (c' - 1)\log(g(n)) + \log(g(n)) = c' \log(g(n)).$$

   Because $f(n) > 1$ for all $n \geq n_0''$, we also have

   $$0 \leq \log(f(n))$$

   for all $n \geq n_0''$. Thus by the definition of $O()$, we have $\log(f(n)) = O(\log(g(n)))$.

   (b) **False.** As a counter-example, consider $f(n) = 2n$ and $g(n) = n$. Then we have $f(n) = O(g(n))$, since for all $n \geq 1$, $f(n) \leq 2g(n)$. However, $2^{f(n)} = 4^n$ and $2^{g(n)} = 2^n$, and we proved in Exercise 4(d) above that 4^n is $NOT\ O(2^n)$.

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.*]

SOLUTION:

(a) The idea is similar to the `Merge` algorithm that we saw in class. We will initialize three pointers, one at the front of each of the three lists. Then at each step we will add the smallest of the things that the pointers are pointing at to the return list `ret`, and increment that pointer. To make sure that the pointers never step too far, we add "Infinity" symbols at the end of each list.

```
def Merge3(L,M,R):
    Initialize pointers pL, pM, pR to 0.
    n = len(L) + len(M) + len(R)
    M.append(Infinity), R.append(Infinity), L.append(Infinity)
         # to make sure we don't step too far with the pointers.
    ret = []
    while len(ret) < n:
        if L[pL] is smaller than both M[pM] and R[pR]:
            ret.append(L[pL])
            pL += 1
        elif M[pM] is smaller than both L[pL] and R[pR]:
            ret.append(M[pM])
            pM += 1
        else:
```

```
            ret.append(R[pR])
            pR += 1
    return ret
```

Notice that I don't need to be too worried about whether "smaller" means "strictly smaller" or "less than or equal to" since the problem says that I can assume that all the elements are distinct.

This runs in time $O(n)$, because the `while` loop executes n times, and there is $O(1)$ work that happens inside the `while` loop.

(b) The total running time of this algorithm is $O(n \log(n))$. Mimicking the proof from class, consider the recursion tree. This tree has $O(\log_3(n))$ levels, because that's the number of times we need to divide n by 3 to get down to problems of size 1. At the j'th level of the tree, we have 3^j problems of size $n/3^j$. The total amount of work done in each problem of size $n/3^j$ is $O(n/3^j)$, so the total amount of work at level j is

$$3^j \cdot O(n/3^j) = O(n).$$

Then the total amount of work over all $O(\log_3(n))$ layers of the tree is

$$O(n \log_3(n)) = O(n \log(n)),$$

recalling that $\log_3(n) = \frac{\log(n)}{\log(3)} = O(\log(n))$.

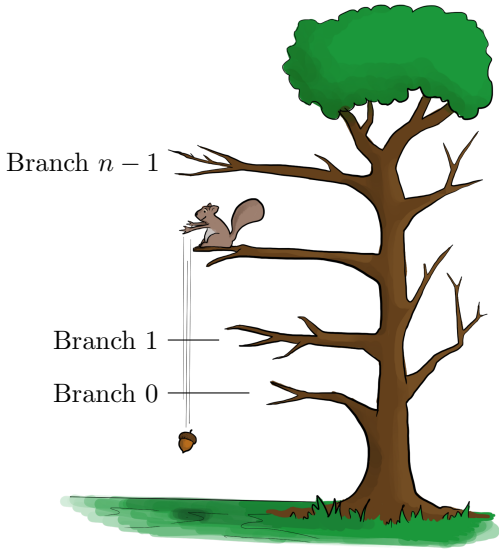
(c) Your friend's problem is in Step ii (or in the combination of Steps ii and iii). It's true that for any $k = O(1)$, the natural version of `Merge_k` runs in time $O(n)$. But for $k = \sqrt{n}$, the straightforward generalization of the pseudocode above would actually run in time $\Omega(n^{3/2})$. More precisely, if we generalize the pseudocode above, we would have k pointers for each of the k lists. At each of n steps (as we fill in the final array that we called `ret` in the code above), the algorithm has to decide which of the k pointers to advance. That is, it has to find the minimum of $k = \sqrt{n}$ things. The straightforward way to do this would be to loop through all $\sqrt{n}$ things, which takes time $\Omega(\sqrt{n})$. So we have n iterations of a loop, and each iteration takes time $\Omega(\sqrt{n})$, for a total of $\Omega(n^{3/2})$. Thus, the whole algorithm runs in time at least $\Omega(n^{3/2})$ (since this is just the work at the top `Merge_k` step), which is actually worse than the $O(n \log(n))$ bound that `MergeSort` achieves.

(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.*]

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, then $\sqrt{n}$, then $2\sqrt{n}$, then $3\sqrt{n}$, and so on. If the acorn breaks at $j\sqrt{n}$, she knows that the "correct" branch is 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)-2}:
      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

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

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

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

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·`$n^{1-\ell/k}$ `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) · $n^{1-\ell/k}$*)*

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 k drops satisfies:

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

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

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

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 $k = O(1)$.

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`