CS 161 **Problem Set 2**
Winter 2019 **Due:** Friday, January 25, 2019, at 3pm on Gradescope

Exercises

Please do the exercises on your own.

1. **(2 pt.)** In your pre-lecture Exercise for Lecture 3, you saw two different ways to solve the recurrence relation $T(n) = 2 \cdot T(n/2) + n$ with $T(1) = 1$. We saw that $T(n)$ is exactly $n(1 + \log(n))$, when n is a power of two. In this exercise, you'll do the same for a few variants.

   (a) What is the exact solution to $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$ with $T(1) = 3$, when n is a power of 2?

   (b) What is the exact solution to $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 3n$ with $T(1) = 1$, when n is a power of 2?

   [**We are expecting:** *Your answer—**no justification required.** Notice that we want the exact answer, so don't give a $O()$ statement.*]

   SOLUTION: Even though the problem says that no justification is required, we'll give two justifications here so that you can see how we did it.

   (a) The exact solution is $T(n) = (\log(n) + 3)n$. To see this, imagine drawing the same tree that we did in class, with the problem of size n at the root, 1 at the leaves, and each node has two children with problems that are half the size. This tree still has depth $(\log(n) + 1)$. For level j for $j = 0, \ldots, \log(n) - 1$, there are 2^j problems of size $n/2^j$, and the total amount of work in each problem is $n/2^j$. Thus, the total amount of work at these levels is $2^j \cdot n/2^j = n$. In the lowest level, $j = \log(n)$, there are n nodes, but the amount of work per node is 3, since $T(1) = 3$. Thus, the amount of work done on this last level is $3n$. Together, the total amount of work is

   $$T(n) = \log(n) \cdot n + 3 \cdot n = (\log(n) + 3) \cdot n.$$

   You could also do this in a different way, by "unwinding" the recurrence:

   $$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2\left(2T(n/4) + n/2\right) + n \\ &= 4T(n/4) + 2n \\ &= 4\left(2T(n/8) + n/4\right) + 2n \\ &= 8T(n/8) + 3n \\ &\vdots \\ &= 2^j T(n/2^j) + jn \end{aligned}$$

   for any $j \leq \log(n)$. Plugging in $\log(n)$ for j, we find

   $$T(n) = nT(1) + n\log(n).$$

   So since $T(1) = 3$, we get

   $$T(n) = n(\log(n) + 3).$$

(b) The exact solution is $T(n) = n \cdot (3\log(n) + 1)$. To see this, we can do exactly the same type of argument as in part (a), except the contribution of the upper layers is $2^j \cdot (3 \cdot n/2^j) = 3n$ instead of n; and the contribution of the last layer is just n. Thus, the total work is

$$T(n) = \log(n) \cdot (3n) + n = n(1 + 3\log(n)).$$

The alternative "unwinding" derivation would proceed by:

$$\begin{aligned} T(n) &= 2T(n/2) + 3n \\ &= 2\left(2T(n/4) + 3n/2\right) + 3n \\ &= 4T(n/4) + 6n \\ &= 4\left(2T(n/8) + 3n/4\right) + 6n \\ &= 8T(n/8) + 9n \\ &\vdots \\ &= 2^j T(n/2^j) + 3jn \end{aligned}$$

for any $j \le \log(n)$. Plugging in $\log(n)$ for j, we find

$$T(n) = nT(1) + 3n\log(n).$$

So since $T(1) = 1$, we get

$$T(n) = n(3\log(n) + 1).$$

2. **(4 pt.)** Use any of the methods we've seen in class so far to give big-Oh solutions to the following recurrence relations. You may treat fractions like $n/2$ as either $\lfloor n/2 \rfloor$ or $\lceil n/2 \rceil$, whichever you prefer.

   (a) $T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n}$ for $n \ge 4$, and $T(n) = 1$ for $n < 4$.

   (b) $T(n) = T(n-2) + n$ for $n \ge 2$, and $T(n) = 1$ for $n < 2$. (You may assume n is even if it helps).

   (c) $T(n) = 7T\left(\frac{n}{3}\right) + n^2$ for $n \ge 3$, and $T(n) = 1$ for $n < 3$.

   (d) $T(n) = 7T\left(\frac{n}{2}\right) + n^2$ for $n \ge 2$, and $T(n) = 1$ for $n < 2$.

   [**We are expecting:** *For each item, the best answer you can give of the form $T(n) = O(\_\_\_)$ and a justification. (That is, all of these satisfy $T(n) = O(2^n)$, but you can do better). You do not need to give a formal proof, but your justification should be convincing to the grader. You may use the Master Theorem if it applies.*]

   SOLUTION:

   (a) We apply the Master Theorem with $a = 2, b = 4, d = 1/2$. We see that $a = 2 = b^d$, so the Master Theorem says that

   $$T(n) = O(\sqrt{n}\log(n)).$$

   (b) We have

   $$T(n) = O(n^2).$$

To see this when n is even, we'd write

$$\begin{aligned} T(n) &= T(n-2) + n \\ &= T(n-4) + (n-2) + n \\ &= \cdots \\ &= T(0) + 2 + 4 + \cdots + (n-4) + (n-2) + n \\ &= 1 + \sum_{i=1}^{n/2} 2i = n/2 \cdot (n/2 + 1) + 1 \\ &= O(n^2) \end{aligned}$$

Similarly if n is odd, we have

$$T(n) = 1+3+5+\cdots+(n-2)+n = \sum_{i=0}^{(n-1)/2} (2i+1) = (n-1)/2+1+(n-1)/2\cdot((n-1)/2+1) = O(n^2).$$

(c) We apply the Master Theorem with $a = 7, b = 3, d = 2$. We have $\log_3(7) < 2$, so the Master Theorem says that

$$T(n) = O(n^2).$$

(d) We apply the Master Theorem with $a = 7, b = 2, d = 2$. We have $\log_2(7) > 2$, so the Master Theorem says that

$$T(n) = O(n^{\log_2(7)}).$$

3. **(2 pt.)** Consider the following algorithm, which takes as input an array A:

```
def printStuff(A):
     n = len(A)
     if n <= 4:
         return
     for i in range(n):
         print(A[i])
     printStuff(A[:n/4])  # recurse on first n/4 elements of A
     printStuff(A[3*n/4:])  # recurse on last n/4 elements of A
     return
```

What is the asymptotic running time of `printStuff`?

[**We are expecting:** *The best answer you can give of the form "The running time of* `printStuff` *is* $O(\_\_\_\_)$*" and a short explanation.*]

SOLUTION: The running time of `printStuff` is $O(n)$. The running time satisfies a recurrence relation of the form $T(n) = 2T(n/4) + O(n)$, since it takes $O(n)$ operations to print out all the n elements of A, and then recurses on two arrays of size $n/4$. By the Master Theorem (with $a = 2, b = 4, d = 1$), the running time is $O(n)$.

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.

---

4. **(4 pt.)** [**Substitution Method.**] Consider the function $T(n)$ defined recursively by

$$T(n) = \begin{cases} T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lceil \frac{n}{4} \rceil\right) + n & n > 4 \\ 1 & n \leq 4 \end{cases}$$

Prove using the substitution method that $T(n) = O(n)$. If it helps **you may assume that n is a power of 2.**

[**We are expecting:** *A formal proof by induction. Remember to state your inductive hypothesis, base case, inductive step, and conclusion. Notice that it's fine to prove a big-Oh bound, you don't need to come up with exact formula for $T(n)$.*]

SOLUTION: As instructed, we prove this by induction (and as permitted, we will assume that n is a power of 2, so we don't have to worry about the ceilings).

- **Inductive Hypothesis.** Our inductive hypothesis (for n) is that:
For all $1 \leq j \leq n$, $T(j) \leq 4j$.
- **Base case.** Our base case is for $n = 4$. In this case, we have for all $1 \leq j \leq 4$,

$$T(j) = 1 \leq 4 \leq 4j$$

for all $1 \leq j \leq 4$.
- **Inductive step.** Suppose that the inductive hypothesis holds for $n - 1$; that is, that $T(j) \leq 4j$ for all $1 \leq j \leq n - 1$.
Then we will show that the inductive hypthesis holds for n. (Notice that since we already know that $T(j) \leq 4j$ for all $1 \leq j \leq n - 1$, to show that $T(j) \leq 4j$ for all $1 \leq j \leq n$, we just need to show that $T(n) \leq 4n$).
By definition, we have, for all $n > 4$,

$$T(n) = T(n/2) + T(n/4) + n.$$

Now, since $n > 4$, both $n/2$ and $n/4$ are at least 1 and at most $n-1$, so they fall into the range covered by our inductive assumption. Thus, we apply the inductive hypothesis to get that

$$\begin{aligned} T(n) &\leq T(n/2) + T(n/4) + n \\ &\leq 4 \cdot n/2 + 4 \cdot n/4 + n \\ &= 2n + n + n \\ &= 4n. \end{aligned}$$

This is what we wanted to show, and it establishes the inductive hypothesis for the next round.

- **Conclusion.** Now we've shown, by induction, that for all $n \geq 4$, we have $T(n) \leq 4n$. We also have $0 \leq T(n)$ since $T(n)$ is increasing with n, and $T(4) = 1$ is positive to begin with. By the definition of $O()$ (with $c = 4$ and $n_0 = 4$), this implies that $T(n) = O(n)$.

5. **(5 pt.)** [**Fishing.**] Plucky the Pedantic Penguin is fishing. Plucky catches n fish, but plans to keep only the k largest fish and to throw the rest back. Plucky has already named all of the fish, and has measured their length and entered it into an array F of length n. For example, F might look like this:

$$F = \begin{bmatrix} \text{(Frederick the Fish, 14.2in)} \\ \text{(Fabiola the Fish, 10in)} \\ \text{(Farid the Fish, 12.35in)} \\ \vdots \\ \text{(Felix the Fish, 6.234523in)} \\ \text{(Finlay the Fish, 6.234524in)} \end{bmatrix}$$

   (a) **(2 pt.)** Give an $O(n \log(n))$-time deterministic algorithm that takes F and k as input and outputs the names of the k largest fish. You may assume that the lengths of the fish are distinct.

   Note: Your algorithm should run in time $O(n \log(n))$ even if k is a function of n. For example, if Plucky wants to keep the largest $k = n/2$ fish, your algorithm should still run in time $O(n \log(n))$.

   [**We are expecting:** *Pseudocode* ***AND*** *a short English description of your algorithm. You may (and, hint, may want to...) invoke algorithms we have seen in class. You do not need to justify why your algorithm is correct or its running time.*]

   (b) **(3 pt.)** Give an $O(n)$-time deterministic algorithm that takes F and k as input and outputs the names of the k largest fish. You may assume that the lengths of the fish are distinct. Your algorithm should also be fundamentally different than your algorithm for part (a). (That is, don't solve part (b) and then use that as your solution to part (a); part (a) should be easier).

   Note: Your algorithm should run in time $O(n)$ even if k is a function of n. For example, if Plucky wants to keep the largest $k = n/2$ fish, your algorithm should still run in time $O(n)$.

   [**We are expecting:** *Pseudocode* ***AND*** *a short English description of your algorithm. You may (and, hint, may want to...) invoke algorithms we have seen in class. You do not need to justify why your algorithm is correct or its running time.*]

   SOLUTION:

   (a) The algorithm is:

```
def findBigFish(k, F):
    run mergeSort on F, using the lengths as the keys to sort on.
    F.reverse() # put the big fish first
    keep = []
    for (fishName, fishLength) in F[:k]:
        keep.append(fishName)
    return keep
```

   That is, we just run mergeSort on the fish and take the top k.

   (b) The idea is to first run SELECT on F to find the $n - k + 1$'st smallest fish, which is the same as the k'th biggest fish. Let's call this k'th biggest fish Flounder the Fish. Then we can run through all the fish, and compare them to Flounder the Fish. If a fish is longer than Flounder, we keep it; if not, we throw it back. (And we keep Flounder the Fish too).

   SELECT runs in time $O(n)$, and it also takes time $O(n)$ to loop through all the fish, so the whole algorithm runs in time $O(n)$.

   The pseudocode is:

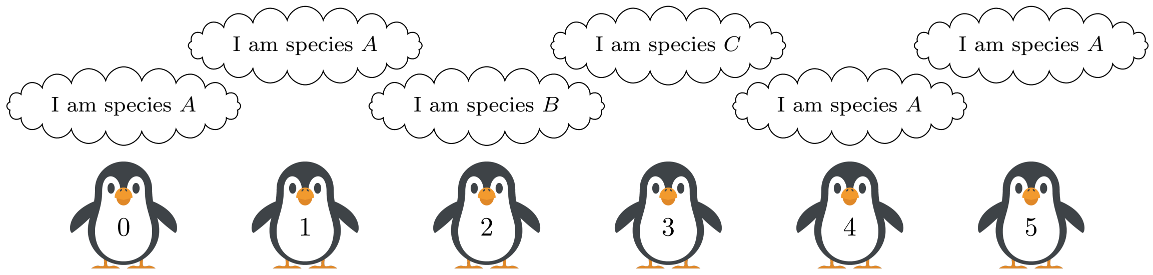
```
def findBigFish(k,F):
     keep = []
     (kName, kLength) = SELECT(F,n-k+1), using the lengths as the keys
     for (fishName, fishLength) in F:
           if fishLength >= kLength:
                keep.append(fishName)
     return keep
```

6. **(8 pt.) [Penguinology.]** 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.

(a) **(4 pt.)** Design a deterministic divide-and-conquer algorithm which 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.*]

(b) **(1 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) **(3 pt.)** Prove, using an argument by induction, that your algorithm is correct.

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

(d) **(NOT REQUIRED. 1 BONUS pt.)** Is $O(n\log(n))$ the best guarantee you can come up with? Either give an asymptotically faster algorithm which finds a majority-species penguin, or else describe prove that no such algorithm exists. (Or, if you think that your algorithm from part (a) already does better that $\Theta(n\log(n))$, just write that :)).

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

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 in part (b)), 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) There is an $O(n)$-time solution:
`https://en.wikipedia.org/wiki/Boyer-Moore_majority_vote_algorithm`

Feedback

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

1. **(0 pt.)** Please fill out the following poll, which asks about the pace of lectures so far:

   `https://goo.gl/forms/mOUlY8r2eeZGP3fB2`