CS 161
Autumn 2017

Problem Set 2
Due: Friday October 13, 2017, at 3pm on Gradescope

Exercises

Exercises should be completed **on your own.**

---

1. **(2 pt.)** In your pre-lecture Exercise for Lecture 3, you saw two different proofs that the solution to the recurrence relation $T(n) = 2 \cdot T(n/2) + n$ with $T(1) = 1$, was exactly $T(n) = n(1 + \log(n))$, when n was a power of two.

   (a) What is the exact solution to $T(n) = 2 \cdot T(n/2) + n$ with $T(1) = 2$, when n is a power of 2?

   (b) What is the exact solution to $T(n) = 2 \cdot T(n/2) + 2n$ with $T(1) = 1$, when n is a power of 2?

   [We are expecting: Your answer, with a convincing argument (it does not need to be a formal proof). Notice that we want the exact answer, so don't give a $O()$ statement.]

   SOLUTION:

   (a) The exact solution is $T(n) = (\log(n) + 2)n$. 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 is 2, since $T(1) = 2$. Thus, the amount of work done on this last level is $2n$. Together, the total amount of work is

   $$T(n) = \log(n) \cdot n + 2 \cdot n = (\log(n) + 2) \cdot n.$$

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

   $$T(n) = \log(n) \cdot (2n) + n = n(1 + 2\log(n)).$$

2. **(2 pt.)** Consider the recurrence relation $T(n) = T(n-1) + n$ with $T(1) = 1$. Your friend claims that $T(n) = O(n)$, and offers the following justification:

   Let's use the Master Theorem with $a = 1$, $b = \frac{n}{n-1}$, and $d = 1$. This applies since

   $$\frac{n}{b} = n \cdot \left(\frac{n-1}{n}\right) = n - 1.$$

   Then we have $a < b^d$, so the Master Theorem says that $T(n) = O(n^d) = O(n)$.

   What's wrong with your friend's argument, and what is the correct answer?

   [HINT: It is totally fine to apply the Master Theorem when b is a fraction; that's not the problem.]

   [We are expecting: A clear identification of the faulty logic above; your solution to this recurrence (you may use asymptotic notation[1]) and a short but convincing justification.]

[1]Unless specified otherwise, in every problem set, when we ask for an answer in asymptotic notation, we are asking either for a $\Theta(\cdot)$ result, or else the tightest $O(\cdot)$ result you can come up with.

SOLUTION: To use the master theorem, b should be constant; that is, it can't depend on n. This choice of b approaches 1 as n gets large. The actual answer is $\Theta(n^2)$. To see this, we can unroll the recurrence a bit:

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= T(n-3) + (n-2) + (n-1) + n \end{aligned}$$

and so on. We get

$$T(n) = T(n-i-1) + \sum_{j=0}^{i} (n-j)$$

for all $i < n-1$, and plugging in $i = n-2$, we get

$$T(n) = T(1) + \sum_{j=0}^{n-1} (n-j) = 1 + \sum_{j=1}^{n} j = \Theta(n^2).$$

3. **(3 pt.)** Use any of the methods we've seen in class so far to solve the following recurrence relations.[2]

   (a) $T(n) = T(n/3) + n^2$, for $n > 3$, and $T(n) = 1$ for $n \leq 3$.

   (b) $T(n) = 2T(n/2) + 10 \cdot n + 4$, for $n > 2$, and $T(n) = 1$ for $n \leq 2$.

   (c) $T(n) = T(n/2) + T(n/4) + n$ for $n > 4$, and $T(n) = 1$ for $n \leq 4$.

   [We are expecting: The answer (you may use asymptotic notation) and a justification. You do not need to give a formal proof, but your justification should be convincing to the grader.]

   SOLUTION:

   (a) We apply the Master Theorem with $a = 1$, $b = 3$ and $d = 2$. We have $a < b^d$, and so all the work is done at the root: the answer is $O(n^d) = O(n^2)$.

   (b) Notice that this can also be written as $T(n) = 2T(n/2) + O(n)$, so we can apply the Master Theorem with $a = b = 2$, $d = 1$, to conclude that $T(n) = O(n\log(n))$.

   (c) We use the substitution method, to establish that $T(n) = \Theta(n)$. We've done steps 1 and 2 (guessing the answer and trying the proof with an unspecified constant) on paper, and now we are ready to turn in the final argument:

   - **Inductive Hypothesis.** $T(n) \leq 4n$ for all $n > 0$.
   - **Base case.** For $k \leq 4$, $T(k) = 1 \leq 4n$, so the inductive hypothesis holds for $k \leq 4$.
   - **Inductive Step.** Let $n > 4$, and suppose that the inductive hypothesis holds for $n-1$. Then

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

     This establishes the inductive hypothesis for the next round.
   - **Conclusion.** We have established the inductive hypothesis for all $n > 0$; thus, $T(n) \leq 4n$ for all $n \geq 1$, and (choosing $c = 4$ and $n_0 = 1$ in the definition of big-oh) we have established that $T(n) \leq O(n)$. On the other hand, $T(n) = T(n/2) + T(n/4) + n \geq n$, so $T(n) = \Omega(n)$. Thus $T(n) = \Theta(n)$.

[2] You may either treat fractions like $n/2$ as $\lfloor n/2 \rfloor$, $\lceil n/2 \rceil$, or just as real numbers (not integers), whichever you prefer.

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. **(3 pt.)** Consider the function $T(n)$ defined recursively by

$$T(n) = \begin{cases} 2T(n/2) + \frac{n}{\log(n)} & n > 2 \\ 1 & n \leq 2 \end{cases}$$

Fill in the blank: $T(n) = \Theta(\_\_\_\_\_\_)$. **[HINT: It may be helpful that $\sum_{i=1}^{m} 1/i = \Theta(\log(m))$.]**
[We are expecting: Your answer and a convincing justification. You do not need to write a formal proof; and you may assume that n is a power of 2 if it helps.]

SOLUTION: The answer is $T(n) = \Theta(n \log\log(n))$. Here the master method does not apply. [**Note: the fancy version in the book also does not apply, since $n/\log(n)$ is neither $O(n^{1-\epsilon})$ for any constant $\epsilon > 0$, nor is it $\Theta(n)$**]. We'll unwind the recurrence. After we do this for a few steps, we see

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

Plugging in $j = \log(n)$, we get

$$\begin{aligned} T(n) &= n \cdot T(1) + \sum_{i=0}^{\log(n)-1} \frac{n}{\log(n/2^i)} \\ &= n + n \sum_{i=0}^{\log(n)-1} \frac{1}{\log(n/2^i)} \\ &= n + n \sum_{i=0}^{\log(n)-1} \frac{1}{\log(n) - i} \\ &= n + n \sum_{i=1}^{\log(n)} \frac{1}{i} \\ &= n + \Theta(n \log\log(n)) \\ &= \Theta(n \log\log(n)). \end{aligned}$$

Above, we were able to replace $\log(n) - i$ with i because we just reversed the order of the summands in the sum. Finally, we used the fact that $\sum_{i=1}^{m} 1/i = \Theta(\log(m))$.

2. **(3 pt.)** Consider the function $T(n)$ defined by

$$T(n) = \begin{cases} 2T(\lceil n/2 \rceil) + n/2 & n > 1 \\ 1 & n = 1 \end{cases}.$$

Using an argument **by induction** (not using the Master Method), prove that $T(n) = \Omega(n \log(n))$.

[We are expecting: A formal proof by induction. Make sure you explicitly state your inductive hypothesis, base case, inductive step, and conclusion.]

SOLUTION:

- **Inductive Hypothesis:** $T(k) \geq k \log(k)/4$ for all $1 \leq k \leq n$.
- **Base case:** When $n = 1$, $T(1) \geq (1/4) \cdot \log(1) = 0$ is true since $T(1) = 1 \geq 0$.
- **Inductive step:** Assume that the inductive hypothesis holds for $n - 1$. Then we will establish it for n:

$$\begin{aligned} T(n) &= 2T(\lceil n/2 \rceil) + n/2 \\ &\geq 2(\frac{1}{4} \cdot \lceil n/2 \rceil \log(n/2)) + n/2 \\ &\geq 2(\frac{1}{4} \cdot \frac{n}{2} \log(n/2)) + n/2 \\ &= \frac{n}{4}(\log(n) - 1) + n/2 \\ &= \frac{1}{4} n \log(n) + n/4 \\ &\geq \frac{n \log(n)}{4}. \end{aligned}$$

  Thus, $T(n) \geq n \log(n)/4$. Since we already were assuming that $T(k) \geq k \log(k)/4$ for all $1 \leq k < n$, this proves that $T(k) \geq k \log(k)/4$ for $1 \leq k \leq n$, establishing the inductive hypothesis for n.
- **Conclusion:** By induction, for all $n \geq 1$, there is a constant $c = 1/4$ so that $T(n) \geq Cn \log(n)$. By definition, $T(n) = \Omega(n \log(n))$.

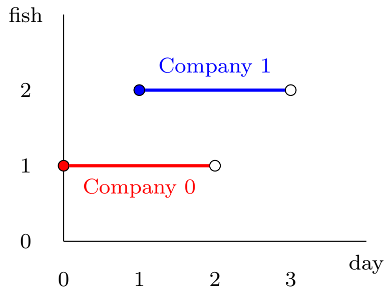
3. **(8 pt.)** Plucky the Pedantic Penguin sometimes does consulting work on the side.[3] There are n companies who are interested in Plucky's work. Plucky can work for at most one company during a given day. Each company has a range of times when they are interested in Plucky's work, and an amount they are willing to pay: between a_i and b_i (including a_i and not including b_i), Company i is willing to pay Plucky f_i fish. Here, a_i, b_i, f_i are all positive integers and $a_i \leq b_i$. Each day, Plucky chooses to work for the highest bidder. If there is no company interested in Plucky's work on a day, Plucky gets zero fish that day.

Plucky gets the bids (a_i, b_i, f_i) as inputs, and wants to make a plot of how many fish he will receive each day. To understand the format he wants the output in, see the example below.

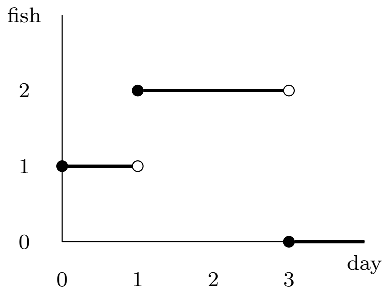
[3] He points out indexing errors for bay area start-ups.

Example: Suppose that $n = 2$. As input, Plucky would get the following data, which can be visualized as the graph below.

Company i	a_i	b_i	f_i
Company 0	0	2	1
Company 1	1	3	2



In this example, Plucky would work for Company 0 at day 0, and receive one fish. He'd work for Company 1 on days $1, 2$, and receive two fish on each of those days. On days 3 and onwards, no company was interested in Plucky's work, so he works for no company and receives zero fish. So his output plot would look like this:



To return this plot, Plucky will return a sequence $(t_0, f_0), (t_1, f_1), \ldots$, with $t_i \le t_{i+1}$, which we interpret as meaning "starting on day t_i and ending on day $t_{i+1} - 1$, Plucky makes f_i fish. In the example above, the return value would be $(t_0 = 0, f_0 = 1), (t_1 = 1, f_1 = 2), (t_2 = 3, f_2 = 0)$.

Notes:

- The last f-value will always be 0.
- In the example above, it would also be correct to return $(t_0 = 0, f_0 = 4), (t_1 = 0, f_1 = 1), (t_2 = 1, f_2 = 2), (t_3 = 3, f_3 = 0)$; that is, adding extraneous intervals of length 0 is still correct.
- In the example above, it would also be correct to return $(t_0 = 0, f_0 = 1), (t_1 = 1, f_1 = 2), (t_2 = 2, f_2 = 2), (t_3 = 3, f_3 = 0)$; that is, breaking an interval into two smaller intervals is still correct.

In this problem you'll design an algorithm for Plucky. Your algorithm should take as input a list of n bids (a_i, b_i, f_i), one for each company $i \in \{0, \ldots, n-1\}$, and return a list `fishPlot` of (t_i, f_i) pairs as described in the example above.

(a) **(2 pt.)** Describe a simple $O(n^2)$-time algorithm for Plucky.
[We are expecting: Pseudocode, and a short English description explaining the main idea of the algorithm. No justification of the correctness or running time is required.]

(b) **(6 pt.)** Design a divide-and-conquer algorithm that takes time $O(n \log(n))$.
[We are expecting: Pseudocode, and a short English description explaining the main idea of the algorithm. We are also expecting an informal justification of correctness and of the running time.]

SOLUTION:

(a) The simple algorithm is as follows:

```
def findFishPlotNaive( bids ):
    S = { all of the a_i and b_i in bids }
    fullFish = []
    for day in S:
       mostFish = 0
       for i in {0,...,n-1}:
           if a_i <= day and day < b_i:
                if f_i > mostFish:
                   mostFish = f_i
       fullFish.append( (day,  mostFish ))
    sort fullFish by day
    return fullFish
```

That is, for each day where anything interesting might happen, Plucky looks over all the companies to decide which company has offered him the most fish. Then he decides to go with that company for each day.

(b) A more complicated divide-and-conquer algorithm is as follows. We will divide up the companies into two groups, and recursively solve the problem for each group. Then we will merge the two solutions.
In order to merge two lists $[(t_0, f_0), \ldots, (t_\ell, f_\ell)]$ and $[(\tilde{t}_0, \tilde{f}_0), \ldots, (\tilde{t}_{\tilde{\ell}}, \tilde{f}_{\tilde{\ell}})]$, we will use an algorithm very similar to the MERGE algorithm that we saw in MERGESORT. The idea is that we will form a sorted list of all the t_i's and the $\tilde{t}_i$'s. At each t_i, we'll append the pair (t_i, f), where f is either f_i or the value $\tilde{f}_j$ for the j so that $t_i \in [\tilde{t}_j, \tilde{t}_{j+1}]$. And we'll do the symmetric thing for each $\tilde{t}_i$. Doing this means that we correctly merge two lists, which implies the correctness of the recursive part of the algorithm.
The pseudocode is given in Algorithm 1.
Now we analyze the runtime. We begin with one list of size n, and at level j of the recursion we have 2^j problems of size approximately $n/2^j$. In the language of the pseudocode, the amount of work done at each level is $O(\ell+\tilde{\ell})$. Here, ℓ and $\tilde{\ell}$ are then lengths of the lists returned by the recursive calls.
We claim that the length of a list returned by a problem with n firms in it is no more than $3n$. This is because at the base case, each firm contributes at most three t-values: $0, a_0, b_0$. (Every firm contributes 0; this is wasteful in practice but does not hurt the asymptotic analysis). Thus, the extra work is $O(n)$.
This means that the running time satisfies the recurrence relation

$$T(n) \leq 2T(n/2) + O(n),$$

and the solution to this (by the Master Theorem) is $T(n) = O(n \log(n))$, as desired.

Algorithm 1: findFishPlot

```
Input: A list of bids, in the form $\{(a_0, b_0, f_0), \ldots, (a_{n-1}, b_{n-1}, f_{n-1})\}$
Output: A list $\{(t_0, f_0), (t_1, f_1), \ldots, (t_{r-1}, f_{r-1})\}$ which represents Plucky's revenue as described
        above.
if the number of companies is $n = 1$ then
    /* Then there is only one company, so Plucky gets no revenue until that company
       will hire him, and then he gets $f_0$ fish per day until the company no longer
       wants to hire him.                                                            */
    return $[(a_0, f_0), (b_0, 0)]$
/* Next divide the companies into two halves and recurse on each half.              */
$p \leftarrow \lfloor m/2 \rfloor$;
$ft \leftarrow$ findFishPlot$[(a_0, b_0, f_0), \ldots, (a_{p-1}, b_{p-1}, f_{p-1})]$;
$\widetilde{ft} \leftarrow$ findFishPlot$[(a_p, b_p, f_p), \ldots, (a_{m-1}, b_{m-1}, f_{m-1})]$;
/* Finally, merge the two halves.                                                   */
Suppose that $ft = [(t_0, f_0), (t_1, f_1), \ldots, (t_\ell, f_\ell)]$;
Suppose that $\widetilde{ft} = [(\tilde{t}_0, \tilde{f}_0), \ldots, (\tilde{t}_{\tilde{\ell}}, \tilde{f}_{\tilde{\ell}})]$;
ret = [];
/* Now we'll basically do the Merge algorithm from MergeSort on the two sorted lists
   of $t$'s. The difference is that at each values $t$ (either a $t_i$ or a $\tilde{t}_i$), we'll record
   whichever of $f$ or $\tilde{f}$ is larger at that point in time.                 */
$q, \tilde{q} \leftarrow 0$;
/* $q$ will be the pointer which steps down $ft$, and $\tilde{q}$ will be the pointer for $\widetilde{ft}$     */
for $t \in \{0, \ldots, \ell + \tilde{\ell} - 1\}$ do
    /* first, a hack to deal with running over the edge of the array                */
    If $q \geq \ell$ then set $t_q \leftarrow \infty$ and if $\tilde{q} \geq \tilde{\ell}$ then set $\tilde{t}_{\tilde{q}} \leftarrow \infty$;
    if $t_q < \tilde{t}_{\tilde{q}}$ then
        /* Then add a point at $t_q$, and the value should be the max of $f_q$ (which is just
           starting to be available) and $\tilde{f}_{\tilde{q}-1}$ (which was already available).  */
        if $\tilde{q} == 0$, then add $(t_q, f_q)$ to ret; otherwise add $(t_q, \max\{f_q, \tilde{f}_{\tilde{q}-1}\})$ to ret;
        $q += 1$;
    else
        /* Then add a point at $\tilde{t}_{\tilde{q}}$, with the opposite of the above logic.   */
        /* Note that this breaks ties in $q$'s favor; it doesn't matter for the
           correctness of the algorithm.                                            */
        if $q == 0$, then add $(t_{\tilde{q}}, f_{\tilde{q}})$ to ret; otherwise add $(\tilde{t}_{\tilde{q}}, \max\{\tilde{f}_{\tilde{q}}, f_{q-1}\})$ to ret;
        $\tilde{q} += 1$;
return ret
```