CS161 Summer 2022 Problem Session 2

Problem 1: Min Cut Heuristic

In lecture, we saw that the size of the minimum cut of a graph is no larger than the smallest degree of any vertex in the graph – we can just take a smallest-degree vertex and remove all of its edges, thereby disconnecting it from the graph.

Because of this, Waverly thinks that when trying to find a min cut, it might be a good idea to look at edges for which the combined degree of the two attached vertices is small. What do you think of this idea? Can you informally argue why it always works, or describe a graph in which it would not work?

Problem 2: Decimation

Suppose that we get consistently unlucky when choosing random pivots in QuickSort, such that we always end up splitting into two lists of 90% and 10% the size of the original. (We'll handwave away what happens to the pivot itself – this is OK because we'll only make our recurrence more conservative.)

Since Quicksort does linear work when partitioning around a random pivot, let's say the recurrence here is

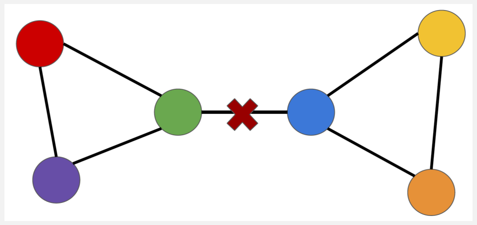
- $T(n) = 1$, for $1 \leq n \leq 10$
- $T(n) = T(\frac{9n}{10}) + T(\frac{n}{10}) + an$, for $n > 10$

where a is some constant reflecting how much work it actually takes to do the partitioning. (This is some fixed but unknown value; you do not have to set it.)

Show that this recurrence is still $O(n \log n)$. (That is, it is asymptotically the same as $T(n) = 2T(\frac{n}{2}). + \Theta(n)$, even though that divides the list beautifully in half!)

(Hints: Follow the argument in class from Lecture 3, when we solved a similar-looking recurrence, but here we just need to establish an asymptotic bound of $n \log n$, not n as in that example. Work with $\log_2$ to make the calculations more concrete. You will have to do some messing around with log terms to make this work, using log tricks described in, e.g., the first Prereq Review document. You can also make the upper bound looser when it simplifies the analysis.)

Problem 1: This seems like a good idea, but unfortunately it won't always work. Consider the "bowtie" example from class. In this case, the single edge that constitutes the min cut is actually between the two vertices of **highest** degree!



Problem 2: We try an induction argument and defer our actual choice of c until the end:

- **Claim:** $T(n) \leq cn\log_2 n$, for $n \geq 2$.
- **Base case:** $T(2) = 1$, which is $\leq c(2\log_2 2) = 2c$. (So we will need $2c \geq 1$.)
- **Inductive step:** Now suppose that $T(n) \leq cn\log_2 n$ for all $2 \leq n < k$. We will show that the claim still holds for $n = k$:

$$\begin{aligned}
T(k) &= T(\frac{9k}{10}) + T(\frac{k}{10}) + an \\
&\leq \frac{9ck}{10}\log_2\frac{9k}{10} + \frac{ck}{10}\log_2\frac{k}{10} + ak \\
&= \frac{9ck}{10}\log_2(\frac{9}{10}) + \frac{9ck}{10}\log_2(k) + \frac{ck}{10}\log_2(\frac{1}{10}) + \frac{ck}{10}\log_2(k) + ak \\
&= ck\log_2 k + \frac{9ck}{10}\log_2 9 - \frac{9ck}{10}\log_2 10 - \frac{ck}{10}\log_2 10 + ak \\
&= ck\log_2 k + \frac{9ck}{10}\log_2 9 - ck\log_2 10 + ak \\
&\leq ck\log_2 k + \frac{9ck}{10}\log_2 10 - ck\log_2 10 + ak \\
&= ck\log_2 k + ak - \frac{ck}{10}\log_2 10
\end{aligned}$$

To get this to be $\leq ck\log_2 k$, we need $\frac{ck}{10}\log_2 10 \geq ak$. We see we can divide out the k here, and then we simply need $c \geq \frac{10a}{\log_2 10}$. (We also need $2c \geq 1$ from the base case, but this is a weaker requirement – a is surely at least 1 because we have to make a pass through the whole list when partitioning.) Therefore, with that choice of c, we have established the desired bound.

Notice that this analysis would work for *any* particular fixed split – even 99.9% and 0.1%, or worse!! Also notice that the exact value of a ultimately did not end up mattering. This may help to explain why we can handwave away a term like an in a recurrences as just $\Theta(n)$.

Problem 3: RaidxSort

Suppose that the phases of RadixSort went in any order other than least significant "digit" to most significant "digit". That is, just as a concrete example, suppose that we are Radix-Sorting a list of 3-digit base 10 numbers, and we accidentally process the digits in the order: second (10s place), third (1s place), first (100s place). Give an example of a list that would be sorted incorrectly by this procedure.

Problem 4: HubrisCorp

Indy has had an idea for a new k-Select algorithm. He has the following conversation with Terry about it:

- **Terry**: But you know you have to at least look at every element of the list to be sure you have the k-th smallest, right? And that takes $\Omega(n)$ time.
- **Indy**: (to someone offscreen) Hey, back up the swamp jacuzzi into the corner there! (to Terry) Just getting the new office building ready.
- **Terry**: Did you hear what I said?
- **Indy**: Of course I did! And that's why before we do any k-Selecting, we put everything in a hash table. And yeah, I know that takes $O(n)$ time. But once we do that just once, all future calls to our new proprietary k-Select algorithm take $o(\log n)$ time each, whereas our competitors are stuck at $O(n)$ each time they want to select, like it's 2021! It even works on arbitrary elements, not just numbers, since it's totally comparison-based. Well, that's what those brainy crocs in R&D tell me.
- **Terry**: Hmm. That all sounds interesting. How does that work?
- **Indy** (grinning toothily): I said *proprietary*, Terry. Now if you'll excuse me, I have some resumes to skim.

Even without knowing the proprietary details or doing any thinking about hash tables, how can Terry argue that Indy's strategy can't possibly work?

Problem 5: Polynomial Comparison

Here's a problem that you will have already seen if you took CS9. Suppose you are given three polynomials (of very large degree) p_1, p_2, p_3, and you are asked whether $p_1p_2 = p_3$. That is, is the product of the first two polynomials the same as the third one? ("Same" here means that $(p_1p_2)(x) = p_3(x)$, for all x.)

One way to do this would be to multiply p_1 and p_2 together and then compare all of the coefficients of that with the coefficients of p_3. But this can be quite computationally cumbersome, and there is an easier way to solve the problem with a randomized algorithm. Can you come up with the idea behind that? (Don't worry too much about the details here.)

Problem 3: One such example, with just two elements, is: 998, 989. Suppose they are in that order originally. Then the orders after each phase are:

- After digit 2: 989, 998
- After digit 3: 998, 989 (here is where the problem arises!)
- After digit 1: 998, 989

That is, the "digit 1" phase trusts that all previous phases were handled correctly, and so it does not (and cannot) correct the earlier mistake.

Problem 4: If what Indy were saying were true, we could sort a list in $o(n \log n)$ time by calling k-Select once per element. The total cost would be $O(n) + n \cdot o(\log n) = o(n \log n)$. But since Indy claimed that the method was entirely comparison-based, he is subject to the $\Omega(n \log n)$ limit on sorting a list.

Problem 5: The idea is to just try plugging in a random value – choose some a and find $(p_1 p_2)a$ and $p_3(a)$. If these come out the same, i.e., $(p_1 p_2 - p_3)a = 0$ it's because either $p_1 p_2$ and p_3 really are the same, or because we just happened to pick a root of $p_1 p_2 - p_3$. But polynomials can only have so many roots, so if we try enough different random values, we will eventually either find proof that the two polynomials are different, or conclude that they are the same.

See `https://web.stanford.edu/class/cs9/slides/w9-2_problem_solving.pdf` if you want more details (since there definitely are more details!)

Problem 6: Median Heuristics

Waverly is at it again, trying to come up with ways of estimating a median that are simpler than the MedianOfMedians procedure. She decides to MergeSort the first $\lfloor \frac{n}{11} \rfloor$ elements of the list, and use the median of that sublist as the estimate of the overall median.

What is the asymptotic running time of this strategy, and how far off (in terms of *index*, not value) that the estimate can possibly be, in the worst case, on a list of size 121 (just for convenience)? The elements in the list might have any values, but you can assume here that there are no ties.

- MergeSorting the sublist of size $\lfloor \frac{n}{11} \rfloor$ takes $O(\lfloor \frac{n}{11} \rfloor \log \lfloor \frac{n}{11} \rfloor) = O(n \log n)$ time, unfortunately. (Asymptotically speaking, we might as well just sort the entire list and then take the exact median!)

- The worst case occurs when, e.g., the first $\lfloor \frac{n}{11} \rfloor$ elements of the list are the smallest in the list; with our $n = 121$ example, the list is of size 11 and we get the 6th smallest element as our median estimate. The actual median is the 61st largest, so we are 55 indices off.