CS3000: Algorithms & Data — Summer I '21 — Drew van der Poel

Homework 2
Due Friday, May 28 at 11:59pm via Gradescope

Name:
Collaborators:

- Make sure to put your name on the first page. If you are using the LaTeX template we provided, then you can make sure it appears by filling in the `yourname` command.
- This assignment is due Friday, May 28 at 11:59pm via Gradescope. No late assignments will be accepted. Make sure to submit something before the deadline.
- Solutions must be typeset in LaTeX. If you need to draw any diagrams, you may draw them by hand as long as they are embedded in the PDF. I recommend using the source file for this assignment to get started.
- I encourage you to work with your classmates on the homework problems. *If you do collaborate, you must write all solutions by yourself, in your own words.* Do not submit anything you cannot explain. Please list all your collaborators in your solution for each problem by filling in the `yourcollaborators` command.
- Finding solutions to homework problems on the web, or by asking students not enrolled in the class is strictly forbidden.

Problem 1. *Improve the MBTA (18 points)*

You have been commissioned to design a new bus system that will run along Huntington Avenue. The bus system must provide service to n stops on the eastbound route. Commuters may begin their trip at any stop i and end at any other step $j > i$. Here are some naïve ways to design the system:

1. You can have a bus run from the western-most point to the eastern-most point making all n stops. The system would be cheap because it only requires $n-1$ route segments for the entire system. However, a person traveling from stop $i = 1$ to stop $j = n$ has to wait while the bus makes $n-1$ stops.
2. You can have a special express bus from i to j for every stop i to every other stop $j > i$. No person will ever have to make more than one stop. However, this system requires $\Theta(n^2)$ route segments and will be expensive.

Using divide-and-conquer, we will find a compromise solution that uses only $\Theta(n \log n)$ route segments, but with the property that a user can get from any stop i to any stop $j > i$ making at most two stops in total. That is, it should be possible to get from any i to any $j > i$ either by taking a direct route $i \to j$ or by taking two routes $i \to m$ and $m \to j$.

(a) **[2 points]** For the base cases $n = 1, 2$, design a system using at most 1 route segment.

Solution:

- When $n = 1$, there is no segment to add, and we return an empty set of segments.
- When $n = 2$, we just add a segment from stop 1 to stop 2.

(b) **[8 points]** For $n > 2$ we will use divide-and-conquer. Assume that we already put in place routes connecting the first $n/2$ stops and routes connecting the last $n/2$ stops so that if i and j both belong to the same half, we can get from i to j in at most 2 segments. Show how to add $O(n)$ additional route segments so that if i is in the first half and j is in the second half we can get from i to j making only two stops.

Solution:

Suppose $l = 1$ and $r = n$ are the western-most and the eastern-most stops, respectively. Let $m = l + \left\lfloor \frac{r-l}{2} \right\rfloor$ be the last stop of the western half. That is, stops $l, \ldots, m$ form the western half of all the stops, and the rest form the eastern half. We add route segments to connect m to all the other stops. More specifically, for all $i \in \{l, \ldots, m-1\}$, we add (i, m) to the set of route segments, and for all $j \in \{m+1, \ldots, r\}$, we add (m, j) to the set of all route segments, unless they already exist. Therefore, the number of segments added is at most $n - 1 = \mathrm{O(n)}$, since we add at most one segment for each stop except m itself.

By adding these segments, a person could get from any i in the western half to any j in the eastern half by taking the two routes: $i \to m$ and $m \to j$.

(c) **[4 points]** Using part (b), write (in pseudocode) a divide-and-conquer algorithm that takes as input the number of stops n and outputs the list of all the route segments used by your bus system.

Solution:

The algorithm is as follows

Algorithm 1: Building Route Segments

Function AddRoutesRecursively(ℓ, r)**:**
- **If** $r - l < 1$ **:**
  - **Return** $\emptyset$
- Let $m \leftarrow \ell + \left\lfloor \frac{r-\ell}{2} \right\rfloor$
- Let $R_W \leftarrow$ AddRoutesRecursively(ℓ, m)
- Let $R_E \leftarrow$ AddRoutesRecursively($m+1, r$)
- Let $R \leftarrow R_W \cup R_E$
- **For** *each* i *from* ℓ *to* $m-1$
  - Add (i, m) to R
- **For** *each* j *from* $m+1$ *to* r
  - Add (m, j) to R
- **Return** R

Function AddRoutes(n)**:**
- **Return** AddRoutesRecursively($1, n$)

(d) **[4 points]** Write the recurrence for the number of route segments your solution uses and solve it. You may use any method for solving the recurrence that we have discussed in class.

Solution:

Let $R(n)$ be the number of route segments returned by the algorithm on a problem of size n. When there are 0 or 1 stops, the algorithm returns the empty set, so $R(0) = R(1) = 0$. Now, a call to the function recursively solves the problem on each half of the input and then merges the two results via $O(n)$ new segments. So, we get the following recurrence:

$$R(0) = 0$$
$$R(1) = 0$$
$$R(n) = 2R(n/2) + cn$$

Similarly to `Mergesort`, the number of route segments added on each level i of the recursion tree is $2^i \cdot \left(\frac{cn}{2^i}\right) = cn$. The tree has L levels, where $n = 2^L \Leftrightarrow L = \log_2(n)$. Overall, $R(n) = \Theta(n \log n)$.

Problem 2. *Stop the Arsonist! (26 points)*

Boston FD receives a letter from an arsonist. In the letter they find a list of n words and a note that the building the arsonist plans to destroy is encoded within the list. The arsonist, who thinks he is clever, also provides a hint:

"The name of the building that I plan to destroy is the longest common prefix of all words in the attached list!"

Clearly, the arsonist is not very clever because (a) it is a terrible encoding and (b) he didn't think Boston FD had you to help them!

We will let w_i be the i-th word in the list ($1 \leq i \leq n$) and $l(w_i)$ be the length of w_i ($1 <= l(w_i) <= 25000\ \forall i$). We let L be the $\max(l(w_i))$ over all i.

Your task is to determine the longest prefix of all words in the arsonist's list, before he destroys his target.

Here is an example:

The arsonist's list reads: "northeasternkhouryccis", "northeasternkhouryccisbuilding", "northeasternkhouryyyyyyyy", "northeasternkhourybuilding", "northeasternkhourynortheastern"

The longest common prefix is "northeasternkhoury".

(a) **[10 points]** You want to show off your algorithmic prowess, and you realize that you can use divide-and-conquer to solve this problem. Design a divide-and-conquer algorithm that takes as input a list of n words of maximum length L and outputs the longest common prefix of the n words in time $O(Ln)$. You should pseudocode for your algorithm, accompany this pseudocode with a written description, and provide justification for why it runs in $O(Ln)$ time. Note that you should treat L as a variable, not as a constant.

(Hint: First develop an $O(L)$ algorithm for finding the longest common prefix of two words a and b, then use this algorithm as part of your divide-and-conquer approach.)

Solution:

We first describe LCP, which finds the longest common prefix of two given words a and b of maximum length L. Essentially we will scan a and b simultaneously from front to back one letter at a time, until we find two letters that do not match. We then return the word up to this point. This runs in $O(L)$ time, because we never look at more than L letters for each of the two words. Each time we look at a pair of letters (the i-th letter in a and b), we do one comparison and one append. Thus, the total operations is at most 2L and our run time is $O(L)$.

Now we use $LCP(a, b)$ in order to find the longest common prefix of all words in our list. We follow a strategy very similar to Mergesort. If our list of words only contains two words, we use LCP to find the longest common prefix between them. Otherwise we create two smaller lists, each of size roughly $n/2$. We then recurse on these lists, finding the longest common prefix of each of the sub-lists. Once we have found these longest common prefixes on the smaller problems, we can find the longest common prefix between them. This process is captured in $LCP - main$.

The recurrence for the worst-case runtime of our algorithm is:

Algorithm 2: LCP

Function $LCP(a, b)$**:**
- L=min(len(a), len(b))
- $CP = \emptyset$
- **For** $i \in \{1, \ldots, L\}$
  - **If** *the i-th letter in* a == *the i-th letter in* b **:**
    - add the i-th letter in a to the end of CP
  - **Else**
    - break
- **Return** CP

$T(2) = C * L$

$T(n) = 2 * T(n/2) + C * L$

This is because each call to LCP is $O(L)$. We recurse into two subcases of size $n/2$, and combine these solutions with a call to LCP.

When we evaluate this recursion, we see that the total runtime is $O(N * L)$. At the i-th level of the recursion tree, we have 2^i pieces, and each piece contributes L work to the total amount. The height of our recursion tree is $log_2(n) - 1$. Thus, the total work is $\sum_{i=0}^{log_2(n)-1}(2^i L)$. We can evaluate this as $L\sum_{i=0}^{log_2(n)-1}(2^i) = LC2^{log_2(n)-1} = LCn/2 = \Theta(Ln)$. Note that we used the geometric series to simplify our equation, as our "r" in this case was $2 > 1$.

Algorithm 3: LCP-main

Function $LCP\text{-}main(\{w_1, \ldots, w_n\})$**:**
- **If** $n == 2$ **:**
  - **Return** $LCP(w_1, w_2)$
- **Else**
  - m = $\lceil n/2 \rceil$
  - LCP_1 = LCP-main($\{w_1, \ldots, w_m\}$)
  - LCP_2 = LCP-main($\{w_{m+1}, \ldots, w_n\}$)
  - **Return** $LCP(LCP_1, LCP_2)$

(b) **[10 points]** In order to better your familiarity with divide-and-conquer algorithms, we have created a hackerrank challenge (www.hackerrank.com/cs3000-summer1-2021-programming-assignment-2). Please implement your divide-and-conquer strategy and submit it to the challenge. Your grade for this part will depend on (a) how many test cases your implementation passes and (b) actually implementing a divide-and-conquer strategy (we will check!).

In order to allow for efficient grading, please write your hackerrank account below.

Solution:

(c) **[3 points]** Your friend, who has no training in algorithms, proposes the following method for solving the problem when there are at least two words in the list ($n \geq 2$): Pick an arbitrary word and find the longest common prefix between it and each of the other $n-1$ words. Then look at this new list of $n-1$ common prefixes, and whichever is the shortest one, is the solution.

What is the Big Oh runtime of your friend's approach (state in terms of n and L)? Provide a short justification of your claim. To receive full marks, your runtime should be reasonably tight (i.e. relevant to the algorithm).

Solution:

This algorithm runs in $O(Ln)$ time. We find the longest common prefix $n-1$ times, and each time we do this is an $O(L)$ operation using the LCP algorithm from part 1, making for $O(Ln)$ in total. Then we would look at the $n-1$ prefixes, and for each we would compute its length, maintaining the shortest one. Since each has length at most L, the runtime of this step would also be $O(Ln)$.

(d) **[3 points]** Is your friend's algorithm correct? If yes, provide a proof of its correctness (**Hint:** I would use a combination of direction observations and contradiction, definitely avoiding induction!). If not, provide an instance of the problem where their approach does not work (**Hint:** I would think about relatively short lists of relatively short words).

Solution:

This algorithm is correct. Let S be the returned solution. Recall that S is the longest common prefix between an aribtrary word w and another word $w*$. Thus, S is a prefix of w, and as the longest common prefix between w and all words other than $w*$ is at least as long as S, S must be a prefix of all words. In other words, S is a common prefix.

Now we will show that S is the longest common prefix. Assume not, that the actual solution S' is S plus some additional letters. Thus, the longest common prefix of any pair of words in the list must include S'. But S is the longest common prefix of w and $w*$, and S does not contain S', thus we have a contradiction. Therefore S must be the longest common prefix.

Problem 3. *A fault-tolerant OR-gate (14 points)*

Assume you are given an infinite supply of two-input, one-output gates, most of which are OR gates and some of which are AND gates. You may assume that all inputs are from the set $\{0, 1\}$. Recall that if at least one of the inputs to an OR gate is a 1, then the output is a 1, otherwise the output is a 0. If both inputs to an AND gate are 1s, then the output is a 1, otherwise the output is a 0.

The OR and AND gates have been mixed together and you can't tell them apart. For a given integer $k \geq 0$, you would like to construct a two-input, one-output combinational "k-OR" circuit from your supply of two-input, one output gates such that the following property holds: If at most k of the gates are AND gates then the circuit correctly implements OR.

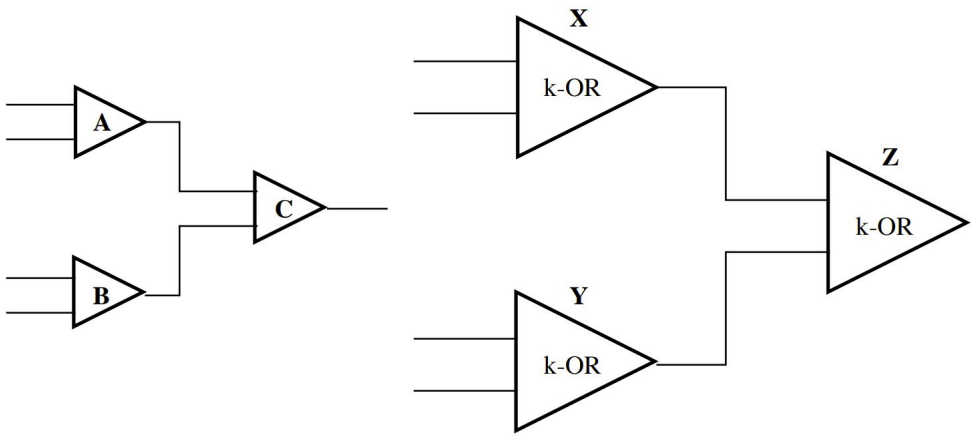
Note that the inputs to a gate can either be the two "circuit inputs" (which may be the inputs for multiple gates), or can be directed from the outputs of two other unique gates. There should be exactly one gate whose output is not directed to another gate, the output of which is returned by the circuit (is the circuit's output). A gate's output may be directed to more than one gate.

For a given integer $k \geq 0$, you would like to design a k-OR circuit that uses as few gates as possible. Assume for simplicity that k is a power of two. Note that AND and OR gates only differ in their outputs when one input is a 1 and the other is a 0. It suffices to only consider the case where the "circuit inputs" are a 1 and a 0.

(a) **[4 points]** Design a 1-OR circuit with the smallest number of gates. That is, your circuit should take two inputs (the "circuit inputs"), and always return the OR of the inputs, as long as at most one of the gates constituting the circuit is an AND gate, while the remainder are OR gates. Argue the correctness of your circuit.

Solution:

For $k = 1$, we have the circuit given in Figure (a). The circuit works correctly if at most one of the three gates is AND. If either of the gates on the first level (A or B) is AND, then the other two gates are OR, allowing the OR computed by the first OR gate to be propagated correcty to the final OR gate. If C is AND, then the outputs of A and B are both computed to be 1 becaus they are both OR gates. Then, these 1's are passed to C, and the output is correctly a 1.



(a) A 1-OR circuit (b) A 2k-OR circuit

(b) **[4 points]** Using a 1-OR circuit as a black box, design a 2-OR circuit. How many gates does your circuit have? Argue the correctness of your circuit.

Solution:

For $k = 2$, we can take 3 copies of the 1-OR circuit from part (a) and arrange them as is done in figure (b) (where $k = 1$). Note that the output from X is passed as one of the inputs to each of the gates in the first layer of Z. The same is done for the output of Y.

If the 2 AND gates are in different parts of the 2-OR circuit (X, Y, and Z), then each 1-OR circuit computes as an OR, and the OR is correctly propagated throught the circuit. Otherwise, if both AND gates are in Z (so Z behaves like an AND gate), X and Y both correclty compute the OR to be 1, and Z outputs a 1 as well. If both AND gates are in X (without loss of generality, they could equivalently be in Y), then Y correctly computes the OR and propagates it to Z, which correctly outputs a 1.

This 2-OR circuit uses 9 gates.

(c) **[6 points]** Generalizing the black box approach from part (b), design the best possible k-OR circuit you can and derive a Θ-bound (in terms of the parameter k) for the number of gates in your k-OR circuit. Show your work for deriving this bound. For full credit, the number of gates in your circuit must be a polynomial in k.

Solution:

Figure (b) illustrates the construction of a $2k$-OR circuit ($k = 2^i$ where $i \in \mathbb{N}$). It makes use of three k-OR circuits. Thus, the number of gates in an n-OR circuit is $T(n) = 3 * T(n/2)$. We saw in part (a) that $T(1) = 3$. We can solve this recurrence with a recursion tree, where on level i there would be 3^i pieces of size $n/2^i$. The height of the tree would be $\log_2(n)$, and each of the $3^{\log_2(n)} = n^{\log_2(3)}$ pieces on the bottom level would contribute 3 to the total number of gates. Thus, the number of gates in a n-OR circuit is $3 * n^{\log_2(3)} = \Theta(n^{\log_2(3)})$.

Problem 4. *Recurrences (10 points)*

Suppose we have algorithms with running times $T(n)$ given by the recurrences:

1. $T(n) = 4T(n/2) + n^2$
2. $T(n) = T(n/2) + n$
3. $T(n) = 6T(n/25) + n^{1/2}$
4. $T(n) = 8T(n/2) + n^{3.5}$
5. $T(n) = 6T(n/10) + 4$

Determine and state the asymptotic running time of each of these five algorithms, and then rank them in ascending order of their asymptotic running time. You do not need to justify your ranking.

Solution:

1. $T(n) = 4T(n/2) + n^2 \rightarrow \Theta(n^2 \log(n))$
2. $T(n) = T(n/2) + n \rightarrow \Theta(n)$
3. $T(n) = 6T(n/25) + n^{1/2} \rightarrow \Theta(n^{\log_{25}(6)} = n^{0.557})$
4. $T(n) = 8T(n/2) + n^{3.5} \rightarrow \Theta(n^{3.5})$
5. $T(n) = 6T(n/10) + 4 \rightarrow \Theta(n^{\log_{10}(6)} = n^{0.778})$

In ascending order of running time: (3), (5), (2), (1), (4)