1. (25 pts) **Dynamic Tree Paths**

You’re given a static rooted tree $T$ with $n$ nodes. Every edge of the tree is labeled *dashed* or *solid*. The labeling is *valid* if for every non-leaf node $x$ exactly one of the edges to its children is solid. (As usual, a node with no children is called a leaf.)

The tree comes with an initial valid labeling, and the labeling can change over time.

A sequence of Access($x$) requests is to be processed. The cost of Access($x$) is the number of dashed edges on the path from $x$ to the root of the tree. At any point in time the algorithm is allowed to switch to a different labeling. The cost is the number of edges that change from solid to dashed.

(a) An algorithm is called *static* if, at the very beginning, it chooses a valid labeling and changes to that one. After that it never changes the labeling of the edges. Give a static algorithm that, after the initial relabeling, incurs a cost of at most $\log_2 n$ on every access.

(b) Consider the following on-line algorithm we’ll call *greedy*: Upon receiving an Access($x$) request, greedy flips all the dashed edges on the path from $x$ to the root from dashed to solid. The cost of this operation is the number of edges flipped. The access itself is then free.

Prove that the greedy algorithm is 2-competitive.

In your proof use the following notation. $\sigma = \sigma_1, \sigma_2 \ldots \sigma_m$ is a sequence of access requests. $C_G(T, \sigma)$ is the cost of the greedy algorithm on that access sequence starting from tree $T$, and $C_B(T, \sigma)$ is the cost of some other algorithm $B$ on that sequence.

(c) Prove that 

$$C_G(T, \sigma) \leq 2n + 2m \log_2(n).$$
2. (25 pts) **Finding Split Strings**

You have a string $S$ of length $n$. The goal is to pre-process $S$ so that queries of the following form can be handled efficiently:

$$\text{find-split}(w) \quad w \text{ is a string of length } m \leq n. \text{ Determine if } w \text{ can be split into } w_1 \text{ and } w_2 \text{ (} w = w_1w_2 \text{ concatenated) and } w_1 \text{ and } w_2 \text{ occur somewhere in } S \text{ (non-overlapping) with } w_1 \text{ occurring first.}$$

Show how to preprocess $S$ in $O(n)$ time, and show how the query find-split($w$) can be handled in $O(m)$ time where $m$ is the length of $w$.

3. (25 pts) **Parallel Merge Sort**

In this problem we will design a time and processor efficient parallel merge sort algorithm. But first we will design a parallel merge algorithm.

(a) In the sequential setting, we know an easy $O(n)$ algorithm for merging two sorted arrays, each with $n$ elements. In this problem, you will design a parallel algorithm for the same task that uses $O(n/\log n)$ processors and runs in $O(\log n)$ parallel time.

Start by giving a very simple parallel algorithm that uses $2n$ processors and runs in $O(\log n)$ time. The algorithm we are thinking of does many binary searches in parallel.

Hint: you may assume the elements are distinct.

(b) Describe how to use only $O(n/\log n)$ processors to do the merge in $O(\log n)$ time. Here the new run time will be approximately twice that of your previous algorithm. The algorithm we are thinking of works with blocks of elements.

(c) Use your parallel merge algorithm to sort two list on length $n$ using $O(n/\log n)$ processors and running in time $O(\log^2 n)$ time.

(d) In the next problem the elements to be sorted may not be distinct. Explain how to modify your merge algorithm so that we get a stable merge sort algorithm without changing the asymptotic processor and time used.

4. (25 pts) **Parallelizing Suffix Array Construction**

In lecture we presented the “surrogate” algorithm for computing the suffix array in $O(n \log n)$ time.

Explain how to implement this algorithm in parallel to run in $O(\log^3 n)$ parallel time using $O(n/\log n)$ processors.

Hint: You may need to use the sorting algorithm you have designed plus parallel algorithms from lecture.

---

5. (0 pts) **InCommon: A Programming Problem**

The input to the program is a set of lower-case strings $s_1, s_2, \ldots, s_m$. The output is the longest substring common to all of the input strings. The time limit is 10 seconds. Call your program `incommon.*`.

**Input**

The first line of input is $2 \leq m \leq 1000$, the number of input strings. Each of the next $m$ lines contains one of the strings, all of which are non-empty. The characters are all lower case letters in `a–z`. The total length of all the input strings is at most $5 \times 10^5$.

**Output**

Output the longest substring common to all the input strings. In case of ties, output the lexicographically first one. Since the answer might be the empty string, enclose your answer with quotes. See the examples below.

**Examples**

The input data is on the left and the output is on the right.

```
2
abcabc
bbca
"bca"

4
ac
ca
cc
ac
"c"

3
acab
babb
bacb
"a"
```