Recap of this week’s lectures:

- Suffix Arrays
- Parallel Algorithms (perhaps)

Define $LCP(i, j), i < j$ as the length of the longest common prefix between $SA[i]$ and $SA[j]$. Consider an LCP array defined as $LCP[i] = LCP(i, i + 1)$. From lecture, we can construct such an array, given the suffix array, in linear time.

**Theorem.** For all $i < j$, $LCP(i, j) = \min_{k=i}^{j-1} LCP(k, k + 1)$.

**Solution:**

Imagine the suffix trie corresponding to the string. Note that for any two suffixes, their longest common prefix is the path from the root to the least common ancestor (LCA) of the two suffixes.

Fix $i < j$. Let $R$ be the LCA of the two suffixes $SA[i]$ and $SA[j]$, and let $T_R$ be the subtree rooted at $R$.

Clearly no term in $\min_{k=i}^{j-1} LCP(k, k + 1)$ is smaller than $LCP(i, j)$, as any pair of nodes in $T_R$ have LCA in $T_R$. It remains to find two adjacent suffixes in the range $[i, j]$ that have LCA of $R$ in the suffix tree.

Suppose $SA[i]$ and $SA[j]$ lie in different child subtrees of $R$. Denote these two child subtrees as $T_1$ and $T_2$, where $SA[i]$ is in $T_1$ and $SA[j]$ is in $T_2$. Then the largest suffix in $T_1$ and the smallest suffix in $T_2$ are adjacent in $SA$ and have longest common prefix of length equal to $LCP(i, j)$.

Otherwise, one of the suffixes is at $R$, and that suffix is a prefix the other one. This case is easy to handle after understanding the previous one.

Thus with the LCP array, we can efficiently calculate any $LCP(i, j)$ in $O(\log n)$ or $O(1)$ using segment trees and sparse tables respectively, which unfortunately are not in this semester’s material.

**Corollary.** For all valid $i$ and $j$, $LCP(i, j) \geq LCP(i, j + 1)$ and $LCP(i, j) \geq LCP(i - 1, j)$

**Solution:** trivial
We are given two strings \( S \) and \( T \). Let \( N \) be their combined length. Give an \( O(N \log N) \) algorithm to compute the longest common substring of the two strings. (In lecture we showed how to do this with a suffix tree. This time do it using a suffix array and the LCP array.)

**Solution:** Build the suffix and LCP arrays of the string \( S + $ + T \), where \( S \) and \( T \) are concatenated together and separated by a character that is not part of the alphabet. Also for each suffix, record whether it begins in \( S \) or \( T \).

The answer is now the largest value of \( \text{LCP}[i] \) where \( \text{SA}[i] \) and \( \text{SA}[i + 1] \) are from different strings. For proof, think about the facts on the previous page.
We are given a string $S$ of length $N$. Give an $O(N \log N)$ time algorithm that computes the number of distinct substrings of $S$. For instance, the string “aa” has 2 unique substrings.

**Solution:** The answer is

$$
\sum_{i=0}^{n-1} (n - 1 - \text{SA}[i] - \text{LCP}[i - 1])
$$

assuming $\text{LCP}[i - 1] = 0$ when $i = 0$.

For each $i$, this calculates the number of unique substrings that are a prefix of $\text{SA}[i]$ that have not already been counted previously in the sum (i.e. is not a prefix of any suffix $\text{SA}[i']$ where $i' < i$). Again this relies on the facts from the first page.