Suffix arrays

Suffix arrays are a clever idea that can be used to solve some tricky problems involving strings. A suffix array is a sorted list of all the suffixes of a string. For example, the suffixes of `computer` are listed below on the left. Then we sort them alphabetically, as shown on the right.

```
computer                    computer
 omputer                    er
  mputer                    mputer
   puter                    omputer
    uter                    puter
     ter                    r
      er                    ter
       r                    uter
```

Sometimes people store the indices that the suffixes start at instead of the suffixes themselves. But we'll stick to storing the suffixes themselves. Below is a function to create a suffix array from a string. There are approaches with better big O performance that should be used for real applications, but this should be fine for our purposes.

```
def suffix_array(s):
    return sorted([s[i:] for i in range(len(s))])
```

Suffix arrays are good for answering questions about substrings. Below we will look at several applications. For each of them, the key fact that makes suffix arrays useful is the fact that every substring of the main string will appear at the start of some suffix (namely the suffix that starts at the same index that the substring starts at).

Testing if a string contains a substring If we have a suffix array already created, a fast approach is to do a binary search of the suffix array. Since every substring appears at the start of some suffix, we modify the binary search algorithm so that instead of checking the entire suffix, it just checks as many characters as are in the substring. For instance, suppose we are searching for the substring `put` in `computer`. At the first step of the binary search, we would check index 3 of the suffix array, which is `omputer`. We compare the first three letters of `omputer`, which is `omp`, with `put`.

Note: This approach requires us to first create a suffix array. If we are just checking a single substring, then the time to create the suffix array would be longer than just doing a slow linear search for the substring. However, if we are doing a lot of substring searches on the same string, then this is a good approach.[1]

Counting the number of occurrences of a substring. To do this, use the binary search to find an index in the array where the substring appears at the start of a suffix. All of the other occurrences of that substring will be next to it in the array. For instance, suppose we have the string `abcaabcbcbbc`. The suffix array for it is shown below.

```
aabcbcbbc
abcaabcbcbbc
abcbcbbc
bbc
bc
bcaabcbcbbc
bcbbc
bcbcbbc
c
caabcbcbbc
cbbc
cbcbbc
```

[1] For just checking a single substring, there are a number of special string searching algorithms that are better than a linear search.

Suppose we are looking for a count of the substring `bc`. Notice that all the substrings that start with `bc` are grouped together. A binary search would find one of them, such as the one highlighted above. Then we could run one loop that goes toward the front of the string and another that goes toward the back of the string to count them all. Those loops will be quick because we can stop once the substrings stop starting with `bc`.

Finding the longest repeated substring Here we are looking at all the substrings that appear more than once, and we want the longest of those. Let's say the string is `abcabacabcabaac`. As we noted in the previous problem, each substring will show up at the start of some suffix, and all occurrences of the substring will be grouped together. The trick to finding the longest repeated substring is to look at the *prefixes* of each suffix (what they start with), and see what each one has in common with the one before it.

suffix	shared prefix with previous item
aac	–
abaac	a
abacabcabaac	aba
abcabaac	ab
abcabacabcabaac	abcaba
ac	a
acabcabaac	ac
baac	–
bacabcabaac	ba
bcabaac	b
bcabacabcabaac	bcaba
c	–
cabaac	c
cabacabcabaac	caba
cabcabaac	cab

The longest shared prefix is the answer. Above it's `abcaba`. Below is code that finds it. The code loops through the suffix array and runs another loop for each suffix to find what it shares in common with the one before it.

```
def longest_repeated_substring(s):
    array = suffix_array(s)
    shared_prefixes = []
    for i in range(1, len(array)):
        prefix = ''
        j = 0
        s = array[i-1]
        t = array[i]
        while j < len(s) and j < len(t) and s[j] == t[j]:
            prefix += s[j]
            j += 1
        shared_prefixes.append(prefix)
    return sorted(shared_prefixes, key=len)[-1]
```

Longest common substring Given two substrings, we are interested in the longest substring they both share. The trick is to apply the longest repeated substring function to the concatenation of the two strings with a special character (not present in either string) added as a separator between them. The solution is the longest shared prefix between two suffixes where one of them contains the separator and the other doesn't. This problem has practical applications to DNA sequences.

Longest palindromic substring This problem is about finding the longest substring that is a palindrome. A palindrome is a string that reads the same forwards and backwards, like `racecar` or `abcddcba`. The trick here is to use the solution for the longest common substring problem, where the two arguments are the string and its reverse.