Name: Sidhartha Sunkasari
WVUID: 800430009
Email: ss00230@mix.wvu.edu

Q1: Give a sequence T=T[1..n].

AA. Using suffix array and LCP data structures.

A. Determine all the unique substrings that repeated in the sequence (i.e.., occurred at least 2 times).

The suffix array will have the suffixes whose indexes are sorted in a lexicographic fashion.

The LCP array will have the number of repeating substrings between two consecutive suffixes in the suffix array.

LCP[i] gives the number of common substrings between SA[i] and SA[i-1].

In order to find out all the unique substrings that are repeated we have to follow the below steps.

- Construct the suffix array SA[1..n] where the indexes of the suffixes are sorted lexicographically.
- Construct the LCP array LCP[1..n] for the constructed suffix array.
- Now iterate through the given LCP and suffix array and compare the consecutive LCP's and check if there is a common LCP between the suffixes.
- An LCP value '0' denotes that there is no common prefix between two suffixes.
- An LCP value greater than 0 between two suffixes tells us the longest common prefix between two consecutive suffixes.
- The condition should be LCP[i] > 0 for SA[i] and SA[i-1].
- Maintain a hash set **S** to add all the LCPs which are greater than 1 to handle the repeated prefixes.
- This means for a value LCP[i] = r, SA[i], SA[i - 1] share r substrings which are repeated(length of each substring 1, …r). Add substrings 1…r into the sets
- One more way to handle the duplicates is to create a non repeating substring and calculate the values using the formula newLCP[i] = max(LCP[i] - LCP[i-1], 0). As the strings are already stored in a lexicographic fashion this can handle the duplicates. For this we have to

keep track of the previous LCP[i-1] value and it should be greater than LCP[i].

- As all the strings are sorted lexicographically if LCP[i] <= LCP[i-1] it means the both the suffixes in the SA have same prefix which is not possible or if the LCP value is less we might have counted the common substrings in the previous LCP[i - 1] so we don't have to include it in the LCP[i]

Maintain another array to store the new substring LCP values. We will use this values for the further operations.

B. Count all such repeated distinct substrings.
To count all the distinct repeated substrings we can add all the values in new substrings LCP values to get the count of all the repeated distinct substrings.

Count = Sum of substrings added to the hash set **Size(S)** Or ∑ newLCP[1 .. n].

C. Count the number of occurrences for each unique substring.
For each substring in the hash set compare the Suffix Array and see if the prefix matches. Keep track of the count and increment the counter if there is a match. This way we can get the number of occurrences of the substrings in the given sequence.

D. Identify the location(s) of each unique substring that is repeated.
For each substring in the set we can scan the suffix array and see if the prefix matches in the suffix array and the index associated with the suffix will help us find the location of the unique substring. So for each LCP[i] > 0 where the value is 'r', r substrings can be possible and compare these substrings for the SA[i+1] to SA[n]

BB. Analysis of Time Complexity and Space Complexity

a. Suffix Array Construction Time and Space = O(n).
b. LCP Array Construction Time and space = O(n).

Suffix array takes O(n) if Karkkainen algorithm is used and O(nlogn) if Manber-Myers is used and it takes similar time for all the best, average and worst cases.

Since we use both these arrays for finding the Unique substrings the array construction can be considered as part of pre-processing and irrespective of the string characteristics.

For all the following problems we still have to construct the suffixes and LCP so the TIme and Space will be O(n).

I. Finding all the unique substrings
Best TC: O(n) since we still have to scan the LCP array even if there are no repeating characters.
Average TC: O(n) since we still have to scan the LCP array even if there are some repeating characters.
Worst TC: O(n) since we still have to scan the LCP array even if there are a lot of repeating characters.

II. Count all such repeated distinct substrings.
Best TC: O(n) since we still have to scan the newLCP array values and add them.
Average TC: O(n) since we still have to scan the newLCP array values and add them.
Worst TC: O(n) since we still have to scan the newLCP array values and add them.

III. Count the number of occurrences for each unique substring.
Best TC: O(n) if there are all distinct characters in the LCP array.
Average and worst TC: O(n^2) if there are repeating characters and we have to find the LCP's which are greater and find the occurrences in the SA for each LCP substring.

IV. Identify the locations of such repeated distinct substrings.
Best TC: O(n) if there are all distinct characters in the LCP array and it has to scan the entire LCP array.
Average and worst TC: O(n^2) if there are repeating characters and we have to find the LCP's which are greater and find the occurrences in the SA for each LCP substring.

Q2: PageRank and Page Trust.
AA. Problems with PageRank Algorithms.
Page rank algorithm ranks the websites based on their importance by assigning a score to each page based on how many other pages link to it. A page receives a higher rank if many pages link to it. If a page does not have incoming links it will be given a lower rank. This leads to two problems.

- **Fairness:** This algorithm is not fair to the newer nodes and it may take a long time to get into the network and it may take a long time for the page to get a rank assigned to it. This leads to the rank bias problem where older pages with high incoming links are ranked higher compared to the newer pages if the content of the page is better.
- **Link Farming:** Spammer often manipulate the page rank algorithm by creating link farms where a set of spam pages link to themselves and in some cases the spam page links are pasted in the comments section of a good page which

allows the users to redirect from good page to the spam page, thus increasing the rank of the spam pages.

BB.

Addressing the Fairness (Rank-bias) Problem:

The main drawback of the page rank algorithm is it assigns the rank to pages based on the number of incoming links pointed to it from other web pages. Some spammers might also use the link farms to alter the algorithm. Due to this reason the newer pages even with the newer relevant content won't get into the page rank algorithms graph as it doesn't have any incoming or outgoing links. One more problem is as new pages are added to the graph it takes a lot of time for a new page to get into the graph network of the algorithm. To address this issue better algorithms can be used to fix this issue by applying it to the sparse matrices. When the fairness is computed for smaller chunks of the graphs, it allows the newer pages to have better probability to get a better rank. If the graph is huge the new pages often get a low rank as the established pages tend to dominate and get a higher rank. If we process a huge graph it also gets computationally expensive to apply the algorithm. So some parallel processing techniques can be used on smaller chunks of the graphs and this will definitely help in managing a big graph. Once the individual fairness is calculated from these chunks then all these chunks results can be used to make the ranking better for the newer pages. Using new spam detection AI systems can also be utilized to rank the spam pages lower.

Addressing the Link farming

Usually the page rank algorithm ranks the pages based on the number of incoming links to a page. This trait is exploited by spammers to increase their page rank.

To tackle this problem some of the following algorithms can be used

- **Trust Rank:** This method follows the analogy that good pages often don't mention any spam links on their pages. This algorithm propagates trust through outgoing links which assigns higher trust scores to pages linked from trusted websites.
- **Anti-Trust Rank:** This approach follows the reverse analogy of the trust rank method where it is unlikely for spam pages to link to good websites. This works by assigning anti-trust in backward fashion from the seed set of spam websites through the incoming links.

Along with the above two approaches, making the graph sparse also helps in maintaining the freshness of the new pages.

CC. Similarities and the differences between page rank, trust rank, antitrust rank algorithms.

Similarities and Differences:
All these approaches use graphs to represent the webpages where each page is a node and there will be incoming and outgoing links which are considered as edges. Each of these approaches ranks the pages based on a constraint.

- For PageRank it uses the number of incoming links to a web page. Due to this this algorithm can be exploited.
- For the Trust Rank, it uses something called trust. Starts by collecting the seed set of good pages and propagates the trust through outgoing links to reduce the influence of the spam pages by increasing the trust score.
- For the Anti Trust, it uses the same trust but in the reverse direction. Starts with a seed set of the spam pages and propagates the distrust backward along the incoming links which leads to demoting the spam pages with higher page ranks.

Page rank can be used to rank the overall popularity of the page and trust rank and antitrust rank offers layers to filter the spam pages.

Q3.
Main Contributions:
This paper aims to use better memory management techniques using parallelization techniques to construct the suffix array using parallelization on top of BWT. BWT requires the suffix array to reside in the physical memory which makes it inefficient in constructing the suffix array for large texts. The authors used a master slave parallel processing tecniques where the suffix arrays are build over non overlapping segments and combining them to build the suffix array. They achieved better performance compared to the FMD-index and Bwt-disk methods. They provided good performance evaluation which showed an increase in the memory efficiency compared to the older methods.

Strengths:
- This approach of using parallelization is more scalable compared to the traditional approach because parallel processing often works on the principle of running the algorithms on smaller machines and increasing the number of smaller machines which is feasible and achievable. It’s hard to increase the memory of a machine exponentially but it's easier and possible to increase the number of machines.

- ParaBWT offers configurability to users to have further control over the number of threads for different hardware configurations. This allows user who wants to try the algorithm for smaller texts to configure the ParaBWT based on their hardware availability.
- Based on the results provided even the slight increase in the performance can reduce organizations a lot of amount when it comes to infrastructure and this method proved to be better than the existing approaches and as we are dealing with big data, this approach can definitely reduce the infrastructure costs.

Weaknesses:

- Though this approach allows users to work on large texts it also contradicts itself when the text is small and it requires good enough memory. Often parallel processing approaches tend to perform not so good if the given input size is small where a slight increase in the memory the processing can be increased greatly.
- Debugging might be difficult when it comes to the FM-index representation and the progressive BWT.
- Starvation may occur in the master slave model where the algorithm does not stop till it finishes all the segments and it may have to wait when all threads are being utilized and the slave threads being idle.
- Situations like when a master thread failing is not studied.

Improvements:

- Adaptive parallelization strategy which assigns the memory based on a sample of the small segments and automatically assigning the threads and memory accordingly will definitely improve the throughput.
- Use of GPU's can greatly increase the throughput because GPUs are meant for parallelization.
- Better batching can also be used to increase the CPU and GPU efficiency.
- Optimizing the ParaBWT for short genomic sequences and other diverse datasets can offer a chance to generalize this method for other datasets.