CSE 4502/5717 Big Data Analytics. Fall 2021
Exam II Solutions

1. Do one read pass through the data to identify the C distinct elements in X. Let the distinct elements be $d_1, d_2, \ldots, d_C$ (in sorted order).

   In the main memory keep C buffers, one for each possible value $d_i, 1 \leq i \leq C$. Each buffer will be of size BD. Do one more pass. In this pass, bring BD elements from X (residing in the disks) at a time (in one parallel I/O) and distribute the keys to the buffers (based on the values of the keys). In the disks we grow C runs $R_1, R_2, \ldots, R_C$. When any buffer i is full, write these BD elements at the end of R_i and clear this buffer (for any i, $1 \leq i \leq C$). At the end of this pass, we would have fully grown the runs $R_1, R_2, \ldots, R_C$.

   Note that the first pass and the second pass can indeed be merged into one pass.

   In the second pass read and write the runs $R_1, R_2, \ldots, R_C$ into one contiguous sequence.

2. The algorithm proceeds in stages. There will be k stages. In the first stage we do one pass through the data and indentify the set Q of the BD smallest elements of X and store these BD elements in a buffer Z. This can be done by bringing BD elements at a time into the core memory and keeping the BD smallest elements seen so far. In another pass of the first stage, we scan through X and delete from X the elements in Q. Let the sequence of the remaining elements of X be X'. X' will be written to the disks as a contiguous sequence.

   for $j = 2$ **to** k **do**
       Let $X = X'$; Clear buffer Z;
       In one pass through X identify the set Q of the BD smallest elements of X and store these BD elements in Z;
       In another pass, scan through X and delete from X the elements in Q;
       Let the sequence of the remaining elements of X be X';
       Write X' to the disks as a contiguous sequence.

   The ith smallest element of the original input is in the buffer Z. we perform an appropriate selection in Z and output that element.

   The number of passes taken by the above algorithm is $2k$. Thus the total number of parallel I/O operations is $O\left(k\frac{n}{BD}\right)$.

3. Consider the following algorithm:

   Find the longest common substring R between S_1 and S_2;
   If $|R| \geq l$, then output R and stop;
   for $i = 1$ **to** n **do**
       Let the i^{th} character of S_1 be c;
       for every character $d \in \Sigma - \{c\}$ **do**

Replace the i^{th} character of S_1 with d;
Find the longest common substring R between S_1 and S_2;
If $|R| \geq l$, output R and stop;
Switch back the i^{th} character of S_1 to c;
Output: "There is no such common substring between S_1 and S_2;

Note that the longest common substring algorithm is called $O(n)$ times and each call takes $O(n)$ time. Thus the total run time of the algorithm is $O(n^2)$.

4. Here is an algorithm:

   Construct a generalized suffix tree Q on $S_1, S_2, \ldots, S_k$;
   for $i = 1$ **to** k **do**
   Traverse through Q and label a node u with i
   if the subtree rooted at u has a leaf corresponding to a suffix from S_i;
   Traverse through Q and indentify the node u that has been labelled with $1, 2, \ldots, k$ and whose string depth is the largest. Output the path label of this node u.

   Analysis: Construction of Q takes $O(M)$ time. In the **for** loop, we traverse through Q k times. Followed by this, we do one more traversal through Q. Each traversal takes $O(M)$ time.

   Thus the total run time of the algorithm is $O(kM)$.

5. Let $SA[1:m]$ be the suffix array for T. Initialize $A[1:m]$ to all zeros. This can be done in $O(1)$ time using m processors.

   (a) We will assign n processors for each entry in $SA[1:m]$.
   for $i = 1$ to m **in parallel do**
   The n processors associated with the suffix $SA[i]$ will compare the characters of P with the characters of the suffix $SA[i]$ in parallel and check if there is a match in $O(1)$ time; If there is a match, one of these processors will set $A[i]$ to 1;

   (b) In problem 5 of Homework 2, you showed that string matching can be done in $O(\log m)$ time. A key step in this algorithm was the fact that using m processors, we can compare P with any suffix $SA[i]$ and decide if there is match at $SA[i]$, P is greater than the suffix $SA[i]$, or P is less than the suffix $SA[i]$ in $O(1)$ time.

   Partition $SA[1:m]$ into $\sqrt{m}$ intervals $[1:\sqrt{m}], [\sqrt{m}+1, 2\sqrt{m}], [2\sqrt{m}+1, 3\sqrt{m}]$, etc. Assign n processors per interval. The n processors associated with any interval will decide (in $O(1)$ time) if P lies in between the two suffixes corresponding to this interval.

   At the end of the above step, we would have identified an interval within which P will lie. Assign n processors for each suffix in this interval. The n processors associated with the suffix $SA[i]$ will compare the characters of P with the characters of the suffix $SA[i]$ in parallel and check if there is a match in $O(1)$ time; If there is a match, one of these processors will set $A[i]$ to 1;