CSE 4502/5717 Big Data Analytics. Fall 2024
Exam II Solutions

1. Notice that an undirected graph on n vertices will be a tree if it has $n-1$ edges and is connected. The graph is given to us in adjacency lists form. We bring the adjacency lists to the core memory one block at a time. If G is a tree, the total size of the adjacency lists will be $2(n-1)$. If the size exceeds this number, we know that G is not a tree and we output this information and stop.

   If the total size of the adjacency lists is $2(n-1)$, then we can perform a DFS on the graph to see if it is connected. Note that the entire graph can be stored in the core memory. Or, we can run Prim's algorithm on G (assuming that each edge has a weight of 1) to compute the weight of the MST. If the weight of the MST is less than $n-1$, it will mean that G is not a tree. If it is $n-1$, we output that G is a tree.

   Clearly, the algorithm makes only $O\left(\frac{n}{B}\right)$ I/O operations.

2. We first merge all the input sequences to get a sorted sequence S. The input sequences can be merged two at a time. We can think of a binary tree T of height k where each leaf corresponds to an input sequence. At each internal node u, we merge the two sequences coming from its two children and send the merged sequence to the parent of u. The leaves will start by sending their sequences to their parents. When the root merges the two sequences of its children, we get S.

   Let u be any internal node at level i (for some i in the range $[1,k]$, the level of the leaves being 0). Let A and B be the two sorted sequences coming into u from its children. We can merge A with B in one pass through A and B. We start by bringing BD elements from A and BD elements from B into the core memory. We start merging them. We write the merged output into an output buffer of size BD (residing in the core memory). When the buffer is full, we write these BD elements across the disks in parallel and clear the buffer. When we run out of elements from any of the runs, we bring the next BD elements from that run. Clearly, we can merge A and B by bringing each element of these runs only once into the core memory. In the same way we can perform all the mergings at level i. The number of passes needed for level i is 1, for any $1 \leq i \leq k$. Thus in k passes through the data we can merge all of the input sequences.

   Followed by the above, we perform one pass through S to output $\cap_{i=1}^{n} R_i$ as follows. We bring BD elements from S into the core memory and scan through these elements to see if there are k copies of any element. If so, this element will go into an output buffer (of size BD). We bring the next BD elements and do the same, etc. When we bring the next BD elements, we keep $k-1$ elements from the previous BD elements for obvious reasons. When the output buffer if full, the elements in the buffer will be written to the disks and the buffer will be cleared.

In summary, the total number of passes taken by the above algorithm is $k+1$.

3. Note that for this problem, $B = M^{2/3}$ and $D = M^{1/3}$. We'll use the (ℓ, m) merge sort algorithm with $\ell = m = M^{1/3}$. Here are the details:

   (a) Form runs of length M each; There are $M^{1/3}$ runs that we have to merge. Let these runs be $X_1, X_2, \ldots, X_{M^{1/3}}$.

   (b) Unshuffle each run into $M^{1/3}$ parts. Let the parts of X_i be $X_i^1, X_i^2, \ldots, X_i^{M^{1/3}}$, for $1 \leq i \leq M^{1/3}$.

   (c) Recursively Merge $X_1^j, X_2^j, ..., X_{M^{1/3}}^j$ to get Y_j, for $1 \leq j \leq M^{1/3}$.

   (d) Shuffle $Y_1, Y_2, \ldots, Y_{M^{1/3}}$ to get Z.

   (e) Clean up the dirty sequence in Z.

   Analysis: Note that we have used LMM with $\ell = m = M^{1/3}$. Steps (a) and (b) take 1 pass together. Step (c) takes 1 pass.

   Assume that we have a memory of size $2M$. In this case we can clean up the dirty sequence while we are shuffling. Let Z be partitioned into blocks of size M each: $Z = Z_1, Z_2, \ldots$, where each block Z_i is of size $\ell m = M^{2/3}$. Note that the dirty sequence can only span two successive blocks. Therefore, one way of cleaning the sequence Z is to: sort and merge Z_1 and Z_2; Z_2 and Z_3; etc. If we have $2M$ memory, we can do this cleaning as well as Step (d) in a total of one pass.

   As a result, Steps (d) and (e) take 1 pass.

   In summary, the total number of passes = 3.

4. We first compute the longest common substring S between α and β. This can be done in $O(m)$ time as was shown in class. S is either a prefix or a suffix of β. Consider the case that S is a suffix of β. We now check if S is a prefix of α and the remaining suffix of α is a prefix of β. This also takes $O(m)$ time. The case that S is a prefix of β can be handled similarly.

5. We extract all the k-mers from S. There are $m-k+1$ such k-mers. This can be done in $O(m)$ time. Followed by this we sort all the k-mers using the integer sorting algorithm. Since each k-mer is of size $O(\log m)$ bits, we can sort them in $O(m)$ time. We scan through the sorted sequence to output all the distinct k-mers and their counts.

6. Note that on a common CRCW PRAM we can compute the minimum or maximum of n integers in the range $[1, n^{O(1)}]$ in $O(1)$ time using n processors.

   We partition the interval $[1, m]$ into $\sqrt{m}$ equal sized intervals: $[1, \sqrt{m}], [\sqrt{m}+1, 2\sqrt{m}], \ldots$. We then assign n processors per interval to identify which interval P belongs to. This can be done using the above minimum finding algorithm in $O(1)$ time. Let I be this interval. There are $\sqrt{m}$ suffixes in this interval. We assign n processors for each suffix to see if P matches a prefix of the suffix. This also takes $O(1)$ time. Thus the total run time is $O(1)$.