1. Partition the main memory into two equal parts $Q_1$ and $Q_2$ each of size $BD$. In one parallel I/O bring $BD$ elements into the main memory and store them in $Q_1$. Do one more parallel I/O and bring the next $BD$ elements to the main memory and store them in $Q_2$. From out of these $2BD$ elements keep the smallest $BD$ elements in $Q_1$. Perform one more parallel I/O and bring the next $BD$ elements and store them in $Q_2$. From out of the elements in $Q_1$ and $Q_2$, identify the smallest $BD$ elements and store them in $Q_1$.

Repeat the above process and in one pass through the entire input identify the smallest $BD$ elements and store them in $Q_1$. Let the largest element of $Q_1$ be $L$.

Do one more pass through the data and similar to the first pass identify the $BD$ smallest element of $X$ that are greater than $L$. From out of these elements output the largest.

2. Have an output buffer of size $\frac{BD}{C}$ for each value in the range $[1, C]$. Bring $BD$ elements at a time from the disks into the main memory. Distribute these keys to the buffers based on the key values. Repeat this process. When any buffer is full, write these $\frac{BD}{C}$ elements into the disks. One possibility is to write them in $\frac{D}{C}$ disks (a block each). In the disks, we will grow $C$ runs in separate regions. After one read pass through the data, $X$ has been sorted into $C$ runs in the disks. Note that the number of write passes is $O(C)$.

Now we have to write the runs contiguously in the disks. This can be done in one more pass through the data.

3. Construct a suffix tree $Q$ for $S$ in $O(n)$ time. Followed by this, perform an in-order traversal of $Q$ to label every internal node $u$ of $Q$ with an integer $c[u]$ such that $c[u]$ is the number of leaves in the subtree rooted at $u$.

Now, perform one more traversal through $Q$ to mark every node whose string depth is $\geq k$. In one additional traversal through $Q$ identify the node $u$ that is marked and whose $c[u]$ is the largest. Finally, output any substring of the path label of $u$ whose length is $k$.

Clearly, the total run time of the algorithm is $O(n)$.

4. We generate all possible $k$-mers from all of the input strings. The number of such $k$-mers is $\sum_{i=1}^{n}(|S_i| - k + 1) \leq M$. We sort these $k$-mers. Since $k$ is a constant, this sorting can be done in $O(M)$ time using the integer sorting algorithm. We can scan through the sorted list of $k$-mers to output all the unique $k$-mers and their frequencies. The total run time is $O(M)$.

5. Let $T$ be the text and $P$ be the pattern with $|T| = m$ and $|P| = n$. We can use binary search on the suffix array. In any iteration of binary search, we have to compare the pattern $P$ with a suffix $T_i$ of the text. This comparison involves the identification of the smallest
integer \( q \) such that \( P[q] \neq T_i[q] \). This can be done in \( O(\log n) \) time using \( \frac{n}{\log n} \) CREW PRAM processors as follows.

Consider the comparison of \( P \) with \( T_i = t_i t_{i+1} \ldots t_{i+n-1} \). We want to find the smallest \( q \) such that \( P[q] \neq t_{i+q-1} \). We can generate an array \( E[1:n] \) such that \( E[j] = \infty \) if \( P[j] = t_{i+j-1} \) and \( E[j] = j \) if \( P[j] \neq t_{i+j-1} \). \( q \) is nothing but the minimum of \( E[1], E[2], \ldots, E[n] \) and can be found using a prefix computation in \( O(\log n) \) time using \( \frac{n}{\log n} \) CREW PRAM processors.

There are \( \log m \) iterations of binary search and in each stage we spend \( O(\log n) \) time. Thus the entire binary search takes \( O(\log m \log n) \) time.