CSE 4502/5717 Big Data Analytics
Fall 2019; Homework 1 Solutions

1. Consider the following algorithm:
   for $i := 1$ **to** $\alpha\sqrt{n}\log_e n$ **do**

   Pick a random $j \in [1, n]$; If $a[j] = a[j+1]$ or $a[j] = a[j-1]$ then output: "Type II" and quit;

   Output: "Type I";

   Analysis: Note that if the array is of type I, the above algorithm will never give an incorrect answer. Thus assume that the array is of type II. We'll calculate the probability of an incorrect answer as follows.

   Probability of coming up with the correct answer in one iteration of the for loop is $\frac{\sqrt{n}}{n} = \frac{1}{\sqrt{n}}$. Thus, probability of failure in any iteration is $1 - \frac{1}{\sqrt{n}}$. As a consequence, probability of failure in q successive iterations is $\left(1 - \frac{1}{\sqrt{n}}\right)^q \leq \exp(-q/\sqrt{n})$ (using the fact that $(1 - 1/x)^x \leq 1/e$ for any $x > 0$). This probability will be $\leq n^{-\alpha}$ when $q \geq \alpha\sqrt{n}\log_e n$.

   Thus the output of this algorithm is correct with high probability.

   Note: There exists a deterministic algorithm to solve this problem in $O(\sqrt{n})$ time.

2. It was shown in class that the maximum of n elements can be found in $O(1)$ time using n^2 common CRCW PRAM processors.

   Consider the case when $\epsilon = \frac{1}{2}$. Divide the elements into groups fo size $\sqrt{n}$. Assign the first $\sqrt{n}$ elements to the first n processors and the second $\sqrt{n}$ elements to the next n processors and so on. The maximum element in each group can be found in $O(1)$ time. At this stage, we have $\sqrt{n}$ elements and $n\sqrt{n}$ processors. Hence, the maximum of these elements can be found in $O(1)$ time. Total time = $O(1)$.

   Next, consider the case when $\epsilon = \frac{1}{3}$. Here, divide the elements into groups of size $n^{1/3}$. Assign the first $n^{1/3}$ elements to the first $n^{2/3}$ processors and the second $n^{1/3}$ elements to the next $n^{2/3}$ processors and so on. The maximum element of each group can be found in $O(1)$ time and using $n^{4/3}$ prceossors the maximum of these maximum elements can be found in $O(1)$ time.

   For the general case, partition the input into groups with n^ϵ elements in each group. Find the maximum of each group assigning $n^{2\epsilon}$ processors to each group. This takes

$O(1)$ time. Now the problem reduces to finding the maximum of $n^{1-\epsilon}$ elements. Again, partition the elements with n^{ϵ} elements in each group and find the maximum of each group. There will be only $n^{1-2\epsilon}$ elements left. Proceed in a similar fashion until the number of remaining elements is $\leq \sqrt{n}$. The maximum of these can be found in $O(1)$ time. Clearly, the run time of this algorithm is $O(1/\epsilon)$. This will be a constant if ϵ is a constant.

3. The algorithm runs in phases. In each phase we eliminate a constant fraction of the input keys that cannot be the element of interest. When the number of remaining keys is $\leq \sqrt{n}$, one of the processors performs an appropriate selection and outputs the right element.

   To begin with all the keys are *alive*. In any phase of the algorithm let N stand for the number of alive keys at the beginning of the phase. At the beginning of the first phase, $N = n$.

   Consider a phase where the number of alive keys is N at the beginning of the phase. Let Y be the collection of alive keys. We employ $\sqrt{N}$ processors in this phase. Partition the N keys into $\sqrt{N}$ parts with $\sqrt{N}$ keys in each part. Each processor is assigned a part. Each processor in parallel finds the median of its keys in $O(\sqrt{N})$ time. Let $M_1, M_2, \ldots, M_{\sqrt{N}}$ be these group medians. One of the processors finds the median M of these $\sqrt{N}$ group medians. This will take $O(\sqrt{N})$ time. Now partition Y into Y_1 and Y_2, where $Y_1 = \{q \in Y | q < M\}$ and $Y_2 = \{q \in Y | q > M\}$. There are 3 cases to consider: **Case 1:** If $|Y_1| = i - 1$, M is the element of interest. In this case, we output M and quit. **Case 2:** If $|Y_1| \geq i$, Y_1 will constitute the alive keys for the next phase. **Case 3:** If the above two cases do not hold, Y_2 will constitute the collection of alive keys for the next phase. In this case we set $i := i - |Y_1| - 1$. In cases 2 and 3 we can perform the partitions using a prefix computation that can be done in $O(\sqrt{N})$ time using $\sqrt{N}$ processors.

   It is easy to see that $|Y_1| \geq \frac{N}{4}$ and $|Y_2| \geq \frac{N}{4}$. As a result, it follows that the number of alive keys at the end of this phase is $\leq \frac{3}{4}N$.

   Thus we infer that the run time of the algorithm is $O\left(\sqrt{N} + \sqrt{(3/4)N} + \sqrt{(3/4)^2 N} + \ldots\right) = O(\sqrt{N})$.

4. If we employ k-way merge where $k = cM/B$, the height of the merge tree will be $\frac{\log(N/M)}{\log(cM/B)}$. However, in the worst case we may have to do c passes through the data at each level of the tree, since we can only keep B/c keys of each run. Thus the worst case number of I/O passes needed is $1 + \frac{c \log(N/M)}{\log(cM/B)}$.

5. Dijkstra's algorithm can be described as follows:

Algorithm 1: Dijkstra(V, E, s)

Data: (V, E): a graph;
s: a source node;
let $w(u, v)$ be the weight of edge (u, v);
Result: array d where d_u is the length of the shortest path from s to u;
begin
  for u *in* V **do**
    $d_u := \infty$;
  $d_s := 0$;
  Create a priority queue Q to store pairs of the form (node, distance);
  Insert the pair $(s, 0)$ into Q;
  while Q *not empty* **do**
    $(u, r) :=$ ExtractMin(Q);
    for *every child* c *of* u **do**
      if $d_c > d_u + w(u, c)$ **then**
        $d_c := d_u + w(u, c)$;
        Insert$(Q, (c, d_c))$; // update distance if c present

We assume that we can store the priority queue in memory ($O(|V|)$). The algorithm will read the neighbors of each node at most once. Therefore, the total number of I/Os is $\sum_{u \in E} \left\lceil \frac{deg_u}{B} \right\rceil = O\left(\frac{|E|}{B} + |V| \right)$.