Homework #2

(Due: Nov 16)

Task 1. [90 Points] Average-Case Analysis of Quickselect

Consider the selection algorithm given in Figure 1.

QUICKSELECT($A[q:r],\ k$)

Input: An array of distinct elements, and an integer $k \in [1, r-q+1]$.
Output: An element x of $A[q:r]$ such that $rank(x, A[q,r]) = k$.

1. ***if*** $n < 1$ ***then return*** NIL
2. ***else***
3. $\quad x \leftarrow A[1]$
4. $\quad$ rearrange the numbers of $A[q:r]$ such that
   - $A[t] = x$ for some $t \in [q, r]$,
   - $A[i] < x$ for each $i \in [q, t-1]$,
   - $A[i] > x$ for each $i \in [t+1, r]$,
5. $\quad$ ***if*** $k = t-q+1$ ***then return*** $A[t]$
6. $\quad$ ***else if*** $k < t-q+1$ ***then return*** QUICKSELECT($A[q:t-1],\ k$)
7. $\quad\quad$ ***else return*** QUICKSELECT($A[t+1:r],\ k$)
8. ***return***

Figure 1: [Task 1] Return the k^{th} smallest number in the input array $A[q:r]$, where $k \in [1, r-q+1]$.

Given an input array A of size n, in this task we will analyze the average number of element comparisons (i.e., comparisons between two numbers of A) performed by this algorithm over all $n!$ possible permutations of the numbers in A and all n possible values of k. We will assume that the partitioning algorithm (in line 4) is *stable*, i.e., if two numbers y and z end up in the same partition and y appears before z in the input, then y must also appear before z in the resulting partition.

(*a*) **[15 Points]** Let t_n be the average number of element comparisons performed by the algorithm given in Figure 1 on an input array A of size n, where $n \geq 0$ and the average is taken over all $n!$ possible permutations of the numbers in A and all n possible values of integer k. Argue that

$$t_n = \begin{cases} 0 & \text{if } n < 1, \\ n - 1 + \frac{1}{n^2} \sum_{k=1}^{n} \left[(k-1)t_{k-1} + (n-k)t_{n-k} \right] & \text{otherwise.} \end{cases}$$

(*b*) [**30 Points**] Let $T(z)$ be a generating function for t_n:

$$T(z) = t_0 + t_1 z + t_2 z^2 + \ldots + t_n z^n + \ldots \quad \ldots$$

Show that $T''(z) = \frac{3z-1}{z(1-z)}\ T'(z) + \frac{2z+4}{(1-z)^4}$.

(*c*) [**15 Points**] Solve the differential equation from part (*b*) to show that

$$T'(z) = \frac{8-2z}{(1-z)^3} + \frac{8\ln(1-z)}{z(1-z)^2}.$$

(*d*) [**30 Points**] Use your solution from part (*c*) to show that for $n > 0$,

$$t_n = 3n + 13 - 8\left(1 + \frac{1}{n}\right) H_n = \Theta(n),$$

where, $H_n = \sum_{i=1}^{n} \frac{1}{i}$ is the n^{th} *Harmonic Number*.

Compute the numerical value of t_n for $1 \leq n \leq 10$.

Task 2. [90 Points] Get Root, Merge and Unmerge

Given a forest of $n \geq 1$ single-node trees with the i^{th} tree containing node $i \in [1, n]$, we will perform an intermixed sequence of $m \geq n$ operations on the forest, where each operation is one of the following.

- GET-ROOT(x): Given a node x, return the node at the root of the tree containing x.
- MERGE(x, y): Given the roots x and y of two different trees, merge the two trees by making the root of the larger tree the parent of the smaller (ties broken arbitrarily).
- UNMERGE( ): Undo the latest MERGE operation that is yet to be undone.

We assume that a stack S keeps track of all MERGE operations performed so far which are yet to be undone by calling UNMERGE. The entries from the top to the bottom of S correspond to the most recent to the earliest such merge operations. Each merge operation is given a unique id. Each entry of S stores three integers: the merge id and the id and size of the root involved in this merge operation which has just become a nonroot.

Each node x is associated with a stack S_x which is empty when x is a root, otherwise S_x has at least one entry. The entries in S_x point to nodes in x's current tree each of which was/is its parent at some point. If S_x is nonempty, then the entries from top to bottom point to the latest to the earliest parents of x with the topmost entry pointing to x's current parent and the bottommost entry pointing to its parent right after it was made a nonroot node by a merge operation. The bottommost entry also includes the unique id of the corresponding merge operation and the location of its entry in S when it was executed. Each of the remaining entries (if exist) also include the unique id of the most recent merge operation along with its location in S whose undoing will invalidate the pointer from x to its parent node stored in that entry. If S_x has more than one entry

then all but the bottommost entry point to nodes that became x's parent due to pointer changes by GET-ROOT(y) operations, where either $y = x$ or y was a descendant of x at the time the operation was performed.

All stacks are initially empty.

The GET-ROOT(x) operation finds the root of the tree containing x by following the current (valid) parent pointers of the nodes along the unique path P from x to the root. For every node y along P its current parent u is found from the topmost valid (more on this later) entry of S_y. The function also changes y's parent pointer by changing it to point to y's grandparent v provided y has a grandparent, otherwise the parent pointer remains unchanged. The pointer is changed without overwriting the pointer to u already stored in S_y by simply pushing the pointer to v into S_y. However, the entry of S_y containing the pointer to v must also include the unique id of the most recent merge operation whose undoing will invalidate the pointer from y to v. That merge id is nothing but the one stored as part of the top entry of S_u which is copied from there along with its location in S to the current top entry of S_y containing v.

The MERGE(x, y) operation assumes that both x and y are the roots of two different trees. It merges the two trees by making the root of the larger tree (i.e., containing more nodes) the parent of the smaller. Ties are broken arbitrarily. Suppose $u \in \{x, y\}$ becomes the parent of $v \in \{x, y\} \setminus \{u\}$. Also, let k be the unique id of this merge operation. Then it pushes into S an entry containing k, v and the size of the subtree rooted at v. It also pushes an entry into S_v containing u, k and the location of k's entry in S.

The UNMERGE( ) operation pops the top entry from stack S. Let x be the node in that entry along with size s and merge id k. Then it pops all entries from S_x except the last one which will contain the node y representing the root of the tree merged with the tree rooted at x by the merge operation identified by k which we will have to undo. The undoing is done simply by restoring the size field of node x to s, and emptying S_x so that x becomes a root again. Note that this undo operation may invalidate many pointers created by GET-ROOT operations performed after that merge operation which we will not remove at this time. We will skip over those invalid pointers if encountered by GET-ROOT operations performed after this unmerge operation. We can easily do that as follows. Suppose the top entry of S_x contains node y, merge id k, and location l in S where the entry for merge k is supposed to be stored. We will know that the pointer from x to y is no longer valid provided either (i) S currently has fewer than l entries or (ii) S has at least l entries but location l in S contains a merge id $k' \neq k$. In that case, we simply throw away the top entry of S_x and check the next entry for validity, and so on.

Now answer the following questions.

(a) [**30 Points**] Write down precise commented pseudocodes for GET-ROOT(x), MERGE(x, y) and UNMERGE(.).

(b) [**20 Points**] Prove that the data structure uses $\mathcal{O}(n \log n)$ space.

(c) [**40 Points**] Prove that given a forest of n single-node trees as input, a sequence of $m \geq n$ GET-ROOT, MERGE and UNMERGE operations can be performed on it in $\mathcal{O}\left(\frac{\log n}{\log \log n}\right)$ amortized time per operation.