(25 pts) 1. (Permutations and Union-Find.)

(a) (15 pts) Recall the tree-based union-find data structure from Lecture #7 (Section 4 of the notes). Suppose you are given a sequence of operations, where first all the makeset operations happen, then all the union operations occur, then all finds happen. There are \( n \) makesets and \( m \) unions and finds. Show that the data structure from class (no changes allowed) incurs a total cost of \( O(m + n) \) on such a request sequence.

(b) (Rest of the points) Now you’re given as input \( \pi = (\pi_1, \pi_2, ..., \pi_n) \), a permutation of \( 1 \ldots n \). Let \( G_\pi \) be an undirected graph constructed from \( \pi \) as follows: It has \( n \) vertices, and there’s an undirected edge between \( i \) and \( j \) where \( i < j \) and \( \pi_i > \pi_j \). (In other words, there’s an edge for each inversion in the permutation.) E.g., if the permutation was \( \pi = (2, 3, 1, 4) \), then \( G_\pi \) is the following:

\[
\begin{array}{cccc}
2 & 3 & 1 & 4 \\
\end{array}
\]

Your goal, should you choose to accept it, is to develop an algorithm for computing the connected components of \( G_\pi \). I.e., the output of the algorithm on the above input could be \( \{1, 2, 3\}, \{4\} \).

For full credit your algorithm should run in \( O(n) \) time. You may use the result of part (a) if it helps.⁴

**Solution:** (a) Suppose there are \( U \) unions and \( F \) minds, with \( U + F = m \). The \( n \) makesets and \( U \) unions cost \( O(1) \) each. Observe that after all the unions, we get a forest – i.e., a set of trees – such that the roots of these trees don’t change after this point.

Consider some tree \( T \) in this forest. We use the banker’s method, and we put down $2 on each node in \( T \) that is not the root, and not a child of the root. This costs us at most $2T$, and hence at most $2n$ in all.

Now consider a find at \( x \) such that \( x \) belongs to \( T \). This requires us to walk up the tree from \( x \) to the root \( r \), and to change the parent pointer of each node on this path (except

---

⁴You need not use the result of part (a), or indeed the union-find data structure if you don’t want to. We know of solutions both ways.
the root, and the child of the root) to point to \( x \). We use the \$2 at these (non-root, non-child-of-root) nodes to pay for this work. The find operation pays for the last two steps of visiting the child of the root, and then the root. The crucial observation is that all these nodes whose money is used up are now are children of the root, and will never have to pay again.

Hence the total amortized cost is \( O(n) \) for the initial money plus \( O(F) \) for the finds. All in all, \( O(n + U + F) = O(n + m) \).

(b) Imagine the numbers \( \{1, 2, \ldots, n\} \) placed on a line. Two useful observations are:

**Lemma 1:** For any \( i \), there is no edge going from the first \( i \) numbers to the last \( n - i \) numbers (i.e., from \( L_i := \{\pi_1, \pi_2, \ldots, \pi_i\} \) to \( R_i := \{\pi_{i+1}, \ldots, \pi_n\} \)) if and only if the first \( i \) numbers \( L_i \) equals \( \{1, 2, \ldots, i\} \).

**Proof:** If \( L_i \) is not equal to \( \{1, 2, \ldots, i\} \), it must contain some number greater than \( i \), and not contain some number less or equal to \( i \) — and then there would be an edge going across from \( L_i \) to \( R_i \). Conversely, if \( L_i \) equals \( \{1, 2, \ldots, i\} \), then clearly there is no edge from it to \( R_i \) since all numbers in \( R_i \) are greater than all those in \( L_i \).

**Lemma 2.** If there is an edge from \( \pi_i \) to \( \pi_j \) for \( j > i \) then all of \( \{\pi_i, \ldots, \pi_j\} \) is connected.

**Proof:** Note that \( \pi_i > \pi_j \). Look at \( \pi_k \) for \( i < k < j \). Then either \( \pi_k < \pi_i \) (and lies after it) and so there is an edge \( (i, k) \), or else \( \pi_k > \pi_i > \pi_j \) and lies before \( \pi_j \), so there is an edge \( (k, j) \). So all the numbers in the middle are connected either to \( i \) or to \( j \).

These lemmas mean the connected components of the graph are precisely the sets that lie between two consecutive “gaps” across which there is no edge. So we need to find out all the locations \( i \) for which \( L_i = \{\pi_1, \pi_2, \ldots, \pi_i\} = \{1, 2, \ldots, i\} \). This is easy: scan from left to right, and look for locations \( i \) such that the maximum element in the first \( i \) positions equals \( i \). Now we can output the sets between consecutive gaps.

(25 pts) 2. (Quick Quacks.)

(a) We want to maintain a max-stack, which is a data structure that supports the following operations.

- **push**(number \( x \)), pushes \( x \) on the stack
- **pop**, pops the top element off the stack
- **return-max**, returns the maximum number among the elements still on the stack. (Does not push or pop anything.)

How would you implement a max-stack, while maintaining constant time per operation (worst-case).

Clarifications: You are allowed to allocate infinitely large arrays, etc. On an empty stack, return NULL on a **pop** or **return-max**. (Similar clarifications apply to the following parts.)

**Solution:** Have an array with a top pointer as in the usual array-based implementation of a stack, but give each element of the array two fields: one for the item and one
for the max so far. When pushing a new item $x$, we set $A[++top].item = x$ as usual, and then update the maximum by setting $A[top].max = \max(A[top-1].max, x)$. By induction, we maintain the invariant that $A[i].max$ is the maximum out of $A[0].item$, ..., $A[i].item$ for all values of $i$ from 0 to $top$. (And this invariant is clearly maintained on a pop as well). Thus, the return-max operation can just return $A[top].max$.

(b) We want to now maintain a max-queue, which is what you’d expect given the previous definition. It supports the following operations.

- enqueue(number $x$), adds $x$ to the end of the queue
- dequeue, removes the element at the front of the queue
- return-max, returns the maximum number among the elements still in the queue. (Does not enqueue or dequeue anything.)

Show how to use two max-stacks to implement a max-queue. This max-queue should take $O(1)$ amortized time per operation. That is, a sequence of $n$ operations (consisting of some number of enqueue, dequeue and return-max operations in any order) should take total time $O(n)$.

**Solution:** we implement the queue using two stacks where elements are pushed onto one stack and popped from the other. When the “pop stack” is empty we will dump contents of the ”push stack” onto it in amortized $O(1)$ time. We use push and pop operations from part (a) to implement this dump operation. This ensures that both stacks maintain the invariant from part (a), namely $A[i].max$ is the maximum out of $A[0].item$, ..., $A[i].item$. To implement return-max we just call return-max on each of the two stacks and return the larger value.

(c) Consider the streaming setting, where you are making one pass over the stream $a_1, a_2, \ldots, a_n$, with each $a_i$ being a number. You are given a number $r \geq 0$. For each time $t$, when you see $a_t$, you want to output the maximum value among the past $r$ elements $a_{t-r+1}, a_{t-r+2}, \ldots, a_t$. E.g., if $r = 4$ and the input stream is

$$3, 17, 3, 9, 2, 0, 7, 2, 8, 9, 1, 8, 4, 5, 9$$

then the output stream should be

$$3, 17, 17, 17, 9, 9, 7, 8, 9, 9, 9, 9, 9, 8, 9$$

Give an algorithm that makes one pass over a stream of $n$ numbers to produce the desired output stream, and the total time taken is $O(n)$. You are allowed enough space to store $O(r)$ elements. (Note that taking time $O(rn)$ would be trivial by just storing the most recent $r$ elements and recomputing the max from scratch each time – you want to do better than that.)

**Solution:** We just use the queue from part (b). For the first $r$ entries, we just do an enqueue($x$) and return-max. For the rest we do enqueue($x$), dequeue() and then return-max.
3. **(Let’s Just Hash it Out.)**

We saw (in Recitation #3) that \( H \) is \( \ell \)-universal over range \( m \) if for every fixed sequence of \( \ell \) distinct keys \( \langle x_1, x_2, \ldots, x_\ell \rangle \), if we choose a hash function \( h \) at random from \( H \), the sequence \( \langle h(x_1), h(x_2), \ldots, h(x_\ell) \rangle \) is equally likely to be any of the \( m^\ell \) sequences of length \( \ell \) with elements drawn from \{0, 1, \ldots, m - 1\}. It’s easy to see that if \( H \) is 2-universal then it is universal. (**Check for yourself, or see recitation notes!**)

Consider a universe \( U \) of strings \( s = s_1, s_2, \ldots, s_n \) of length \( n \) from an alphabet of size \( k \). (Each character is an integer in \{0, 1, \ldots, k - 1\}.) Hence \( |U| = k^n \). Assume that \( m = 2^b \). E.g., if \( k = 3 \) then we’re hashing down from \( U = [3]^n \) to \([m] = [2^b] \).

An interesting universal family \( G \) (of functions from \( U \) to \{0, \ldots, m - 1\}) can be obtained as follows. First, generate a 2-dimensional table \( T \) of \( b \)-bit random numbers; recall that \( b = \lg(m) \). The first index of \( T_{i,j} \) is in the range \([1, n]\) and the second index is in the range \([0, k - 1]\). Now define the hash function \( g_T(s) \) as follows:

\[
 g_T(s) = \bigoplus_{i=1}^{n} T_{i,s_i}
\]

where “\( \bigoplus \)” represents the bitwise-xor function (recall, each \( T_{i,j} \) is a \( b \)-bit string). The output of \( g_T(s) \) is a \( b \)-bit string which is then interpreted as a number in \{0, \ldots, m - 1\}. Note that since each choice of the table \( T \) gives a hash function \( g_T \), and \( T \) is specified by \( n \cdot k \cdot b \) bits, the family \( G \) consists of \( 2^{nkb} \) functions.

(a) Prove that \( G \) is not 4-universal.

**Hint:** To show that \( G \) is not 4-universal, you should exhibit 4 distinct keys \( \langle x_1, x_2, x_3, x_4 \rangle \) such that if you were told the values of \( g_T(x_1), g_T(x_2), \) and \( g_T(x_3) \), you could infer the value of \( g_T(x_4) \) uniquely (without knowing anything else about \( T \)). This will mean that not all 4-tuples of hash-values are equally likely, since the first 3 entries in the tuple \( \langle g_T(x_1), g_T(x_2), g_T(x_3), g_T(x_4) \rangle \) determined the 4th entry. You can do this using \( n = 2 \) and \( k = 2 \).

**Solution:** For instance, consider the following 4 keys of length 2 (so \( n = 2 \)): 00, 11, 01, 10. Whatever \( T \) is, these keys have the property that \( g_T(00) \oplus g_T(11) \oplus g_T(01) \oplus g_T(10) = 0 \) since each of the four entries in \( T \) is xor’ed twice. This means that the hash of any one of the strings can be determined from the hash of the other three, so there is no way that all 4-tuples of hash-codes are equally likely.

(b) Prove that \( G \) is 3-universal.

**Solution:** To show why \( G \) is 3-universal, consider three distinct keys \( x, y, z \). We now consider two cases:

Case 1: there is some index \( i \) such that all three keys differ in that index. We now argue as follows. Consider choosing all of \( T \) except for \( T_{i,x} \). Thus, filling in this row of \( T \) will determine the hash codes for \( x, y, \) and \( z \). We first fill in \( T_{i,x} \); since each value for this
entry produces a different value for $g_T(x)$, we have that $x$ is equally likely to hash to all possible $b$-bit values. Now $g_T(x)$ is fixed. We now fill in $T_{i,y_i}$: again, since each value for this entry produces a different value for $g_T(y)$, we have that $y$ is equally likely to hash to all possible $b$-bit values, even conditioning on everything done so far. This means that $x$ and $y$’s hash values are independent. Now $g_T(x)$ and $g_T(y)$ are fixed. We now fill in $T_{i,z_i}$: again, since each value for this entry produces a different value for $g_T(z)$, we have that $z$ is equally likely to hash to all possible $b$-bit values, even given everything so far. So all triples of hash values are equally likely.

Case 2: there is no index $i$ such that all three keys differ in that index. In this case, choose some index $i$ such that $x_i \neq y_i$ (this must exist since $x \neq y$). We know $z_i$ matches one of the other two, so say without loss of generality that $z_i = y_i$. Choose some other index $j$ such that $z_j \neq y_j$. We know that $x_j$ matches one of those two values so say without loss of generality that $x_j = y_j$. We can now argue as in Case 1. First fill in all of $T$ except for $T_{i,x_i}, T_{i,y_i}, T_{j,z_j}$. Now, filling in $T_{i,x_i}$ determines $x$’s hash (all equally likely), then filling in $T_{i,y_i}$ determines $y$’s hash (all equally likely, no matter what $x$ hashed to), then filling in $T_{j,z_j}$ determines $z$’s hash value (all equally likely, no matter what $x, y$ hashed to)