1 Hashing (50 points)

We saw in class that universal hashing lets us give guarantees that hold for arbitrary (i.e. worst case) sets \( S \), in expectation over our random choice of hash function. Let’s work out some more of those guarantees.

(a) (25 points) Let \( H \) be a universal hash family from \( U \) to a table of size \( m \). Let \( S \subseteq U \) be a set of \( m \) elements which we want to hash (so we’re hashing the same number of elements as the size of the table). Prove that if we choose \( h \) from \( H \) uniformly at random, the expected number of pairs \( x, y \in S \) that collide is at most \( m - \frac{1}{2} \).

\[ E[C] = \sum_{x \in S} \sum_{y \in S} C_{xy} \]

We know (from the definition of universal hashing) that \( E[C_{xy}] \leq \frac{1}{m} \). Thus, by linearity of expectation, we have that

\[ E[C] \leq \frac{m(m - 1)}{2} \cdot \frac{1}{m} = \frac{m - 1}{2} \]

(b) (25 points) Prove that with probability at least 3/4, no bin in the table gets more than \( 2\sqrt{m} + 1 \) elements.

\[ \Pr[\text{no bin in the table gets more than } 2\sqrt{m} + 1 \text{ elements}] \geq \frac{3}{4} \]

This is equal to \( (1 - \Pr[\text{some bin in the table gets more than } 2\sqrt{m} + 1 \text{ elements}]) \).
Observe that, whenever one bin gets more than \(2\sqrt{m} + 1\) elements, there will be at least \(\binom{2\sqrt{m} + 1}{2}\) collisions (one for every pair of elements in that bin). If we let \(C\) denote the total number of collisions again, we get that \(C \geq \binom{2\sqrt{m} + 1}{2}\). It follows that:

\[
\Pr[\text{some bin in the table gets more than } 2\sqrt{m} + 1 \text{ elements}] \leq \Pr\left[C > \frac{2\sqrt{m} + 1}{2}\right]
\]

since if you think about the sample space, the event where some bin in the table gets more than \(2\sqrt{m} + 1\) elements is a subset of the event where at least \(\binom{2\sqrt{m} + 1}{2}\) collisions take place.

Now a simple calculation reveals that

\[
\left(\frac{2\sqrt{m} + 1}{2}\right) = \frac{(2\sqrt{m} + 1)2\sqrt{m}}{2} = 2m + \sqrt{m} > \frac{4m - 1}{2}.
\]

Now note that by part (a), \(\mathbb{E}[C] \leq \frac{m-1}{2}\). Putting this together and using Markov’s inequality with \(k = 4\), we get that

\[
\Pr[\text{some bin in the table gets more than } 2\sqrt{m} + 1 \text{ elements}]
\leq \Pr\left[C > \frac{2\sqrt{m} + 1}{2}\right]
\leq \Pr\left[C > \frac{4m - 1}{2}\right]
\leq \Pr[C > 4\mathbb{E}[C]]
\leq 1/4.
\]

Thus with probability at least 3/4, no bin in the table gets more than \(2\sqrt{m} + 1\) elements.

Hint: use part (a), and consider using Markov’s Inequality. To remind you: if \(X\) is a nonnegative random variable with expectation \(\mathbb{E}[X]\), then \(\Pr[X > k\mathbb{E}[X]] < 1/k\) for any \(k > 0\). For example, the probability that \(X\) is more than 100 times its expectation is less than 1/100.

2 Union-Find (50 points)

In class we proved that if we use trees to represent disjoint sets and use both union-by-rank and path compression, then the amortized cost of any operation (Make-Set, Find, or Union) is only \(O(\log^* n)\). In this question we’ll analyze what happens if we change our data structure in two ways: we do not use path compression, and we do union-by-cardinality rather than union-by-rank.

Slightly more formally, we change our data structure as follows. As before, every node has three values: an element, a parent pointer, and a value. But the value isn’t a rank, but is rather the number of nodes in the subtree rooted at it, i.e., the cardinality of the subtree. So the root of any tree stores the cardinality of the set represented by the tree. The basic operations work as follows:

- Make-Set\((x)\) simply returns a single node with the element \(x\), where the parent pointer points to itself and the initial cardinality is 1.
• Find($x$) follows the parent pointer from $x$ recursively until it ends up at the root (which it knows since only the root will have its parent pointer point to itself), and then it returns the element at this root.

• Union($x, y$) first does Find($x$) to find the root $r_x$ of the tree containing $x$ and Find($y$) to find the root $r_y$ of the tree containing $y$. If ($r_x \rightarrow \text{cardinality}$) ≥ ($r_y \rightarrow \text{cardinality}$), then we set the parent of $r_y$ to be $r_x$ and set ($r_x \rightarrow \text{cardinality}$) = ($r_x \rightarrow \text{cardinality}$) + ($r_y \rightarrow \text{cardinality}$). Similarly, if ($r_y \rightarrow \text{cardinality}$) > ($r_x \rightarrow \text{cardinality}$), then we set the parent of $r_x$ to be $r_y$ and set ($r_y \rightarrow \text{cardinality}$) = ($r_x \rightarrow \text{cardinality}$) + ($r_y \rightarrow \text{cardinality}$).

(a) (25 points) Prove that the worst-case running time of any operation is $O(\log n)$, where $n$ is the number of Make-Set operations (i.e., the number of elements). Hint: can you bound the depth/height of a tree by its cardinality?

Solution. Make-Set clearly takes $O(1)$ time. The running time of each Union is $O(1)$ plus the time to do two Find operations. Hence we need only prove that the running time of a Find is $O(\log n)$. We will first argue that the height of any tree is $O(\log n)$. Then since the running time of a Find operation is at most the height of a tree (since it is just a walk up to the root), we get that the running time of a Find is $O(\log n)$. In order to show the height of a tree is $O(\log n)$, we use induction to show that this is always true.

Slightly more formally, we claim that the height of a tree rooted at node $r$ is at most $\log c_r$, where $c_r$ is the cardinality of $r$. We will prove this by induction over time (i.e., an invariant). For the base case, this is clearly true before the first operation is called (since there are no elements or trees). For the inductive step, suppose that this is true after the $(t - 1)$st operation, and consider the $t$th operation. Clearly a Find does not change the heights or the cardinalities, so if the $t$th operation is a Find then it will still be true. If the $t$th operation is a Make-Set, then the only new tree is a single node of cardinality 1 and depth 0 = $\log 1$, so the claim is still true.

Finally, if the $t$th operation is Union($x, y$), then let $c_x$ and $c_y$ be the cardinalities of the roots of the trees containing $x$ and $y$ (call these roots $r_x$ and $r_y$ respectively). Without loss of generality, let us assume that $c_x \geq c_y$, so after the union $r_x$ becomes the parent of $r_y$ and the new cardinality of $r_x$ becomes $c_x + c_y$ (no other cardinalities change). By the inductive hypothesis, the height of the new tree is equal to

$$\max(\log c_x, \log c_y + 1) = \max(\log c_x, \log(2c_y)) \leq \max(\log c_x, \log(c_x + c_y)) = \log(c_x + c_y),$$

which is precisely what we needed to prove.

(b) (25 points) We now want to provide a matching lower bound not just on the worst-case running time, but even on the amortized running time. So suppose that we first do $n$ Make-Set operations (so there are $n$ elements). Give a sequence of $O(n)$ Union operations which cumulatively take $\Omega(n \log n)$ time (implying that the amortized cost of an operation is $\Omega(\log n)$).

Each union operation should be a union of two distinct sets, i.e., you should never call Union($x,y$) on elements $x$ and $y$ that are in the same set.
Solution. Consider a set of elements \( A = \{a_1, \ldots, a_n\} \), and let \( i \) be the largest integer such that \( 2^i \leq n/2 \). In other words, \( i = \lfloor \log(n/2) \rfloor \). Note that \( n/4 \leq 2^i \).

We first do \( i \) phases of unions. In phase 1, we perform \( \text{Union}(a_{2j}, a_{2j-1}) \) for all \( j \in \{1, 2, \ldots, 2^{i-1}\} \). We then continue this process, and in phase \( k \) we perform \( \text{Union}(a_{2^k-1(2j)}, a_{2^k-1(2j-1)}) \) for all \( j \in \{1, 2, \ldots, 2^{i-k}\} \). After these phases are complete, we then take \( \text{Union}(a_\ell, a_j) \) for all \( 2^i < j \leq n \) (where \( \ell \) is a fixed integer which we will set later).

We first show that the total number of Unions is at most \( O(n) \). In phase \( k \) we perform \( 2^{i-k} \) unions, so the total number of unions from the first \( i \) phases is \( \sum_{k=1}^{i} 2^{i-k} \leq 2^i \). And clearly the number of Unions after the first \( i \) phases is at most \( n - 2^i \). Thus the total number of unions is at most \( 2^i + n - 2^i \leq n \).

Now we want to prove that the cumulative time of these Unions is \( \Omega(n \log n) \). To see this, we first need to analyze the tree which is constructed over the course of the first \( i \) phases. We claim that after phase \( k \), there are \( 2^{i-k} \) trees, each of which has height equal to \( i \) and cardinality equal to \( 2^k \), and where the roots are the nodes \( a_{2^k-1(2j)} \) for all \( j \in \{1, 2, \ldots, 2^{i-k}\} \). We prove this by induction on \( k \). For the base case, after phase 1, there are clearly \( 2^i/2 = 2^{i-1} \) trees of height 1 and cardinality 2, and by the definition of the Union algorithm the roots will be precisely the nodes \( a_{2j} \) for \( 1 \leq k \leq 2^{i-1} \). So the base case is true.

Now assume that the inductive hypothesis is true after phase \( k - 1 \). Then by the definition of the Union algorithm, in phase \( k \) each Union will result in a tree with cardinality \( 2(2^{k-1}) = 2^k \) and height \( 1 + (k - 1) = k \), as required. And since we do \( 2^{i-k} \) unions in this phase, that is how many such trees we will have at the end of the phase. Finally, since whenever we perform a union the two trees have the same cardinality, the new root will always be the root of the first tree in the union call, which in our case are the nodes \( a_{2j} \), as claimed.

So now we know that after the first \( i \) phases, we have a tree of height \( i \geq \log(n/4) = \Omega(\log n) \). Let \( a_\ell \) be the node in this tree that is furthest from the root, i.e., a node whose path to the root has length \( i \). So then our final \( n - 2^i \geq n/2 \) Unions are to \( a_\ell \) and a node which has not yet participated in a Union. Each of these operations requires doing a Find from \( a_\ell \) which will take time at least \( i \geq \Omega(\log n) \), and hence the total running time of these operations is at least \( (n/2)i \geq \Omega(n \log n) \), as required.