Data Structures for Disjoint Sets

In this lecture, we describe some methods for maintaining a collection of disjoint sets. Each set is represented as a pointer-based data structure, with one node per element. We will refer to the elements as either ‘objects’ or ‘nodes’, depending on whether we want to emphasize the set abstraction or the actual data structure. Each set has a unique ‘leader’ element, which identifies the set. (Since the sets are always disjoint, the same object cannot be the leader of more than one set.) We want to support the following operations.

- **MAKESET**(*x*): Create a new set \{*x*\} containing the single element *x*. The object *x* must not appear in any other set in our collection. The leader of the new set is obviously *x*.

- **FIND**(*x*): Find (the leader of) the set containing *x*.

- **UNION**(*A*, *B*): Replace two sets *A* and *B* in our collection with their union *A* ∪ *B*. For example, **UNION**(*A*, **MAKESET**(*x*)) adds a new element *x* to an existing set *A*. The sets *A* and *B* are specified by arbitrary elements, so **UNION**(*x*, *y*) has exactly the same behavior as **UNION**(**FIND**(*x*), **FIND**(*y*)).

Disjoint set data structures have lots of applications. For instance, Kruskal’s minimum spanning tree algorithm relies on such a data structure to maintain the components of the intermediate spanning forest. Another application is maintaining the connected components of a graph as new vertices and edges are added. In both these applications, we can use a disjoint-set data structure, where we maintain a set for each connected component, containing that component’s vertices.

### 11.1 Reversed Trees

One of the easiest ways to store sets is using trees, in which each node represents a single element of the set. Each node points to another node, called its **parent**, except for the leader of each set, which points to itself and thus is the root of the tree. **MAKESET** is trivial. **FIND** traverses
parent pointers up to the leader. \textsc{Union} just redirects the parent pointer of one leader to the other. Unlike most tree data structures, nodes do not have pointers down to their children.

\begin{algorithmic}
\State \textbf{MakeSet}(x):
\State \hspace{1em} \textbf{parent}(x) \leftarrow x
\State \textbf{depth}(x) \leftarrow 0
\State return $x$
\end{algorithmic}

\begin{algorithmic}
\State \textbf{Find}(x):
\State \hspace{1em} while $x \neq \textbf{parent}(x)$
\State \hspace{2em} $x \leftarrow \textbf{parent}(x)$
\State \hspace{1em} return $x$
\end{algorithmic}

\begin{algorithmic}
\State \textbf{Union}(x, y):
\State \hspace{1em} $\overline{x} \leftarrow \textbf{Find}(x)$
\State \hspace{1em} $\overline{y} \leftarrow \textbf{Find}(y)$
\State \hspace{1em} if $\text{depth}(\overline{x}) > \text{depth}(\overline{y})$
\State \hspace{2em} $\text{parent}(\overline{y}) \leftarrow \overline{x}$
\State \hspace{1em} else
\State \hspace{3em} $\text{parent}(\overline{x}) \leftarrow \overline{y}$
\State \hspace{3em} if $\text{depth}(\overline{x}) = \text{depth}(\overline{y})$
\State \hspace{4em} $\text{depth}(\overline{y}) \leftarrow \text{depth}(\overline{y}) + 1$
\end{algorithmic}

Merging two sets stored as trees. Arrows point to parents. The shaded node has a new parent.

\textsc{MakeSet} clearly takes $\Theta(1)$ time, and \textsc{Union} requires only $O(1)$ time in addition to the two \textsc{Finds}. The running time of \textsc{Find}(x) is proportional to the depth of $x$ in the tree. It is not hard to come up with a sequence of operations that results in a tree that is a long chain of nodes, so that \textsc{Find} takes $\Theta(n)$ time in the worst case.

However, there is an easy change we can make to our \textsc{Union} algorithm, called \textit{union by depth}, so that the trees always have logarithmic depth. Whenever we need to merge two trees, we always make the root of the shallower tree a child of the deeper one. This requires us to also maintain the depth of each tree, but this is quite easy.

With this new rule in place, it’s not hard to prove by induction that for any set leader $\overline{x}$, the size of $\overline{x}$’s set is at least $2^{\text{depth}(\overline{x})}$, as follows. If $\text{depth}(\overline{x}) = 0$, then $\overline{x}$ is the leader of a singleton set. For any $d > 0$, when $\text{depth}(\overline{x})$ becomes $d$ for the first time, $\overline{x}$ is becoming the leader of the union of two sets, both of whose leaders had depth $d - 1$. By the inductive hypothesis, both component sets had at least $2^{d-1}$ elements, so the new set has at least $2^d$ elements. Later \textsc{Union} operations might add elements to $\overline{x}$’s set without changing its depth, but that only helps us.

Since there are only $n$ elements altogether, the maximum depth of any set is $\log n$. We conclude that if we use union by depth, both \textsc{Find} and \textsc{Union} run in $\Theta(\log n)$ time in the worst case.

### 11.2 Shallow Threaded Trees

Alternately, we could just have every object keep a pointer to the leader of its set. Thus, each set is represented by a shallow tree, where the leader is the root and all the other elements are its
children. With this representation, **MakeSet** and **Find** are completely trivial. Both operations clearly run in constant time. **Union** is a little more difficult, but not much. Our algorithm sets all the leader pointers in one set to point to the leader of the other set. To do this, we need a method to visit every element in a set; we will ‘thread’ a linked list through each set, starting at the set’s leader. The two threads are merged in the **Union** algorithm in constant time.

![Merging two sets stored as threaded trees.](image)

*Bold arrows point to leaders; lighter arrows form the threads. Shaded nodes have a new leader.*

**Code**

**MakeSet** \((x)\):

- `leader(x) ← x`
- `next(x) ← x`

**Find**:

- return `leader(x)`

**Union** \((x, y)\):

1. \(x' ← \text{Find}(x)\)
2. \(y' ← \text{Find}(y)\)
3. \(y ← y'\)
4. `leader(y) ← x'`
5. `next(y) ← next(x')`
6. `next(x') ← y`
7. return `leader(y)`

The worst-case running time of **Union** is a constant times the size of the *larger* set. Thus, if we merge a one-element set with another \(n\)-element set, the running time can be \(Θ(n)\). Generalizing this idea, it is quite easy to come up with a sequence of \(n\) **MakeSet** and \(n − 1\) **Union** operations that requires \(Θ(n^2)\) time to create the set \(\{1, 2, \ldots, n\}\) from scratch.

**WorstCaseSequence** \((n)\):

1. **MakeSet** \((1)\)
2. for \(i ← 2\) to \(n\)
   - **MakeSet** \((i)\)
   - **Union** \((1, i)\)

We are being stupid in two different ways here. One is the order of operations in **WorstCaseSequence**. Obviously, it would be more efficient to merge the sets in the other order, or to use some sort of divide and conquer approach. Unfortunately, we can’t fix this; we don’t get to decide how our data structures are used! The other is that we always update the leader pointers in the larger set. To fix this, we add a comparison inside the **Union** algorithm to determine which set is smaller. This requires us to maintain the size of each set, but that’s easy.

**Code**

**MakeWeightedSet** \((x)\):

1. `leader(x) ← x`
2. `next(x) ← x`
3. `size(x) ← 1`

**WeightedUnion** \((x, y)\):

1. \(x' ← \text{Find}(x)\)
2. \(y' ← \text{Find}(y)\)
3. if \(\text{size}(x) > \text{size}(y)\)
   - **Union** \((x', y)\)
   - \(\text{size}(x) ← \text{size}(x) + \text{size}(y)\)
4. else
   - **Union** \((y', x)\)
   - \(\text{size}(y) ← \text{size}(x) + \text{size}(y)\)
The new **WeightedUnion** algorithm still takes \( \Theta(n) \) time to merge two \( n \)-element sets. However, in an amortized sense, this algorithm is much more efficient. Intuitively, before we can merge two large sets, we have to perform a large number of **MakeWeightedSet** operations.

**Theorem 1.** A sequence of \( m \) **MakeWeightedSet** operations and \( n \) **WeightedUnion** operations takes \( O(m + n \log n) \) time in the worst case.

**Proof:** Whenever the leader of an object \( x \) is changed by a **WeightedUnion**, the size of the set containing \( x \) increases by at least a factor of two. By induction, if the leader of \( x \) has changed \( k \) times, the set containing \( x \) has at least \( 2^k \) members. After the sequence ends, the largest set contains at most \( n \) members. (Why?) Thus, the leader of any object \( x \) has changed at most \( \lfloor \log n \rfloor \) times.

Since each **WeightedUnion** reduces the number of sets by one, there are \( m - n \) sets at the end of the sequence, and at most \( n \) objects are not in singleton sets. Since each of the non-singleton objects had \( O(\log n) \) leader changes, the total amount of work done in updating the leader pointers is \( O(n \log n) \). \( \square \)

The aggregate method now implies that each **WeightedUnion** has **amortized cost** \( O(\log n) \).

### 11.3 Path Compression

Using unthreaded trees, **Find** takes logarithmic time and everything else is constant; using threaded trees, **Union** takes logarithmic amortized time and everything else is constant. A third method allows us to get both of these operations to have *almost* constant running time.

We start with the original unthreaded tree representation, where every object points to a parent. The key observation is that in any **Find** operation, once we determine the leader of an object \( x \), we can speed up future **Find**s by redirecting \( x \)'s parent pointer directly to that leader. In fact, we can change the parent pointers of all the ancestors of \( x \) all the way up to the root; this is easiest if we use recursion for the initial traversal up the tree. This modification to **Find** is called **path compression**.

![Path compression during Find(c). Shaded nodes have a new parent.](image)

```plaintext
**Find(x)**
if x ≠ parent(x)
    parent(x) ← **Find**(parent(x))
return parent(x)
```

If we use path compression, the ‘depth’ field we used earlier to keep the trees shallow is no longer correct, and correcting it would take way too long. But this information still ensures that **Find** runs in \( \Theta(\log n) \) time in the worst case, so we’ll just give it another name: **rank**. The following algorithm is usually called **union by rank**:
**Lecture 11: Disjoint Sets [Fa’13]**

### Disjoint Sets

**MAKESET**(x):
- **parent**(x) ← x
- **rank**(x) ← 0

**UNION**(x, y):
- \( \overline{x} \leftarrow \text{FIND}(x) \)
- \( \overline{y} \leftarrow \text{FIND}(y) \)
- if \( \text{rank}(\overline{x}) > \text{rank}(\overline{y}) \)
  - **parent**(\( \overline{y} \)) ← \( \overline{x} \)
- else
  - **parent**(\( \overline{x} \)) ← \( \overline{y} \)
  - if \( \text{rank}(\overline{x}) = \text{rank}(\overline{y}) \)
    - \( \text{rank}(\overline{y}) \leftarrow \text{rank}(\overline{y}) + 1 \)

**Find** still runs in \( O(\log n) \) time in the worst case; path compression increases the cost by only most a constant factor. But we have good reason to suspect that this upper bound is no longer tight. Our new algorithm memoizes the results of each **Find**, so if we are asked to **Find** the same item twice in a row, the second call returns in constant time. Splay trees used a similar strategy to achieve their optimal amortized cost, but our up-trees have fewer constraints on their structure than binary search trees, so we should get even better performance.

This intuition is exactly correct, but it takes a bit of work to define precisely how much better the performance is. As a first approximation, we will prove below that the amortized cost of a **Find** operation is bounded by the *iterated logarithm* of \( n \), denoted \( \log^* n \), which is the number of times one must take the logarithm of \( n \) before the value is less than 1:

\[
\log^* n = \begin{cases} 
1 & \text{if } n \leq 2, \\
1 + \log^*(\log n) & \text{otherwise}.
\end{cases}
\]

Our proof relies on several useful properties of ranks, which follow directly from the **UNION** and **FIND** algorithms.

- If a node \( x \) is not a set leader, then the rank of \( x \) is smaller than the rank of its parent.
- Whenever **parent**(\( x \)) changes, the new parent has larger rank than the old parent.
- Whenever the leader of \( x \)'s set changes, the new leader has larger rank than the old leader.
- The size of any set is exponential in the rank of its leader: \( \text{size}(\overline{x}) \geq 2^{\text{rank}(\overline{x})} \). (This is easy to prove by induction, hint, hint.)
- In particular, since there are only \( n \) objects, the highest possible rank is \( \lfloor \log n \rfloor \).
- For any integer \( r \), there are at most \( n/2^r \) objects of rank \( r \).

Only the last property requires a clever argument to prove. Fix your favorite integer \( r \). Observe that only set leaders can change their rank. Whenever the rank of any set leader \( \overline{x} \) changes from \( r-1 \) to \( r \), mark all the objects in \( \overline{x} \)'s set. Since leader ranks can only increase over time, each object is marked at most once. There are \( n \) objects altogether, and any object with rank \( r \) marks at least \( 2^r \) objects. It follows that there are at most \( n/2^r \) objects with rank \( r \), as claimed.

\( \text{five.oldstyle} \)
11.4 \( O(\log^* n) \) Amortized Time

The following analysis of path compression was discovered just a few years ago by Raimund Seidel and Micha Sharir.\(^1\) Previous proofs\(^2\) relied on complicated charging schemes or potential-function arguments; Seidel and Sharir’s analysis relies on a comparatively simple recursive decomposition. (Of course, simple is in the eye of the beholder.)

Seidel and Sharir phrase their analysis in terms of two more general operations on set forests. Their more general \textsc{Compress} operation compresses any directed path, not just paths that lead to the root. The new \textsc{Shatter} operation makes every node on a root-to-leaf path into its own parent.

\begin{align*}
\text{\textsc{Compress}}(x, y): & \\
& \begin{cases}
\text{\{y must be an ancestor of x\}} \\
\text{if } x \neq y \\
\text{\textsc{Compress}}(\text{parent}(x), y) \\
\text{\text{parent}(x) \leftarrow \text{parent}(y)}
\end{cases}
\end{align*}

\begin{align*}
\text{\textsc{Shatter}}(x): & \\
& \begin{cases}
\text{if parent}(x) \neq x \\
\text{\textsc{Shatter}}(\text{parent}(x)) \\
\text{\text{parent}(x) \leftarrow x}
\end{cases}
\end{align*}

Clearly, the running time of \textsc{Find}(x) operation is dominated by the running time of \textsc{Compress}(x, y), where \( y \) is the leader of the set containing \( x \). Thus, we can prove the upper bound by analyzing an arbitrary sequence of \textsc{Union} and \textsc{Compress} operations. Moreover, we can assume that the arguments of every \textsc{Union} operation are set leaders, so that each \textsc{Union} takes only constant worst-case time.

Finally, since each call to \textsc{Compress} specifies the top node in the path to be compressed, we can reorder the sequence of operations, so that every \textsc{Union} occurs before any \textsc{Compress}, without changing the number of pointer assignments.

Each \textsc{Union} requires only constant time, so we only need to analyze the amortized cost of \textsc{Compress}. The running time of \textsc{Compress} is proportional to the number of parent pointer assignments, plus \( O(1) \) overhead, so we will phrase our analysis in terms of pointer assignments. Let \( T(m, n, r) \) denote the worst case number of pointer assignments in any sequence of at most \( m \) \textsc{Compress} operations, executed on a forest of at most \( n \) nodes, in which each node has rank at most \( r \).

The following trivial upper bound will be the base case for our recursive argument.


Theorem 2. \( T(m, n, r) \leq nr \)

**Proof:** Each node can change parents at most \( r \) times, because each new parent has higher rank than the previous parent.

Fix a forest \( F \) of \( n \) nodes with maximum rank \( r \), and a sequence \( C \) of \( m \) COMPRESS operations on \( F \), and let \( T(F, C) \) denote the total number of pointer assignments executed by this sequence.

Let \( s \) be an arbitrary positive rank. Partition \( F \) into two sub-forests: a 'low' forest \( F_- \) containing all nodes with rank at most \( s \), and a 'high' forest \( F_+ \) containing all nodes with rank greater than \( s \). Since ranks increase as we follow parent pointers, every ancestor of a high node is another high node. Let \( n_- \) and \( n_+ \) denote the number of nodes in \( F_- \) and \( F_+ \), respectively. Finally, let \( m_+ \) denote the number of COMPRESS operations that involve any node in \( F_+ \), and let \( m_- = m - m_+ \).

Any sequence of COMPRESS operations on \( F \) can be decomposed into a sequence of COMPRESS operations on \( F_+ \), plus a sequence of COMPRESS and SHATTER operations on \( F_- \), with the same total cost. This requires only one small modification to the code: We forbid any low node from having a high parent. Specifically, if \( x \) is a low node and \( y \) is a high node, we replace any assignment \( \text{parent}(x) \leftarrow y \) with \( \text{parent}(x) \leftarrow x \).

This modification is equivalent to the following reduction:

\[
\text{COMPRESS}(x, y, F): \quad \langle y \text{ is an ancestor of } x \rangle \\
\text{if } \text{rank}(x) > s
\quad \text{COMPRESS}(x, y, F_+) \quad \langle \text{in } C_+ \rangle \\
\text{else if } \text{rank}(y) \leq s
\quad \text{COMPRESS}(x, y, F_-) \quad \langle \text{in } C_- \rangle \\
\text{else}
\quad z \leftarrow x \\
\quad \text{while } \text{rank}('parent_F(z)) \leq s
\quad z \leftarrow 'parent_F(z) \\
\quad \text{COMPRESS('parent_F(z)), y, F_+} \quad \langle \text{in } C_+ \rangle \\
\quad \text{SHATTER}(x, z, F_-) \\
\quad \text{parent}(z) \leftarrow z \quad \langle ? \rangle 
\]

This modification is equivalent to the following reduction:
The pointer assignment in the last line (†) looks redundant, but it is actually necessary for the analysis. Each execution of that line mirrors an assignment of the form \( \text{parent}(z) \leftarrow w \), where \( z \) is a low node, \( w \) is a high node, and the previous parent of \( z \) was also a high node. Each of these ‘redundant’ assignments happens immediately after a \text{Compress} in the top forest, so we perform at most \( m_+ \) redundant assignments.

Each node \( x \) is touched by at most one \text{Shatter} operation, so the total number of pointer reassignments in all the \text{Shatter} operations is at most \( n \).

Thus, by partitioning the forest \( F \) into \( F_+ \) and \( F_- \), we have also partitioned the sequence \( C \) of \text{Compress} operations into subsequences \( C_+ \) and \( C_- \), with respective lengths \( m_+ \) and \( m_- \), such that the following inequality holds:

\[
T(F, C) \leq T(F_+, C_+) + T(F_-, C_-) + m_+ + n
\]

Since there are only \( n/2^i \) nodes of any rank \( i \), we have \( n_+ \leq \sum_{i \geq s} n/2^i = n/2^s \). The number of different ranks in \( F_+ \) is \( r - s < r \). Thus, Theorem 2 implies the upper bound

\[
T(F_+, C_+) < rn/2^s.
\]

Let us fix \( s = \lfloor \lg r \rfloor \), so that \( T(F_+, C_+) \leq n \). We can now simplify our earlier recurrence to

\[
T(F, C) \leq T(F_-, C_-) + m_+ + 2n,
\]

or equivalently,

\[
T(F, C) - m \leq T(F_-, C_-) - m_- + 2n.
\]

Since this argument applies to any forest \( F \) and any sequence \( C \), we have just proved that

\[
T'(m, n, r) \leq T'(m, n, \lfloor \lg r \rfloor) + 2n,
\]

where \( T'(m, n, r) = T(m, n, r) - m \). The solution to this recurrence is \( T'(n, m, r) \leq 2n \lg^* r \).

\textbf{Voilá!}

\textbf{Theorem 3.} \( T(m, n, r) \leq m + 2n \lg^* r \)

\begin{itemize}
\item[\textbf{*11.5}] \textbf{Turning the Crank}
\end{itemize}

There is one place in the preceding analysis where we have significant room for improvement. Recall that we bounded the total cost of the operations on \( F_+ \) using the trivial upper bound from Theorem 2. But we just proved a better upper bound in Theorem 3! We can apply precisely the same strategy, using Theorem 3 recursively instead of Theorem 2, to improve the bound even more.

Suppose we fix \( s = \lg^* r \), so that \( n_+ = n/2^\lfloor \lg^* r \rfloor \). Theorem 3 implies that

\[
T(F_+, C_+) \leq m_+ + 2n \frac{\lg^* r}{2^\lfloor \lg^* r \rfloor} \leq m_+ + 2n.
\]

This implies the recurrence

\[
T(F, C) \leq T(F_-, C_-) + 2m_+ + 3n,
\]

which in turn implies that

\[
T''(m, n, r) \leq T''(m, n, \lg^* r) + 3n,
\]
where $T''(m, n, r) = T(m, n, r) - 2m$. The solution to this equation is $T(m, n, r) \leq 2m + 3n \lg^{**} r$, where $\lg^{**} r$ is the iterated iterated logarithm of $r$:

$$
\lg^{**} r = \begin{cases} 
1 & \text{if } r \leq 2, \\
1 + \lg^{**}(\lg^* r) & \text{otherwise}.
\end{cases}
$$

Naturally we can apply the same improvement strategy again, and again, as many times as we like, each time producing a tighter upper bound. Applying the reduction $c$ times, for any positive integer $c$, gives us $T(m, n, r) \leq cm + (c + 1)n \lg^{**} r$, where

$$
\lg^{**} r = \begin{cases} 
\lg r & \text{if } c = 0, \\
1 & \text{if } r \leq 2, \\
1 + \lg^{**}(\lg^{c-1} r) & \text{otherwise}.
\end{cases}
$$

Each time we ‘turn the crank’, the dependence on $m$ increases, while the dependence on $n$ and $r$ decreases. For sufficiently large values of $c$, the $cm$ term dominates the time bound, and further iterations only make things worse. The point of diminishing returns can be estimated by the minimum number of stars such that $\lg^{**-a} r$ is smaller than a constant:

$$
\alpha(r) = \min \left\{ c \geq 1 \mid \lg^{c} n \leq 3 \right\}.
$$

(The threshold value 3 is used here because $\lg^{c} 5 \geq 2$ for all $c$.) By setting $c = \alpha(r)$, we obtain our final upper bound.

**Theorem 4.** $T(m, n, r) \leq ma(r) + 3n(\alpha(r) + 1)$

We can assume without loss of generality that $m \geq n$ by ignoring any singleton sets, so this upper bound can be further simplified to $T(m, n, r) = O(ma(r)) = O(ma(n))$. It follows that if we use union by rank, Find with path compression runs in $O(\alpha(n))$ amortized time.

Even this upper bound is somewhat conservative if $m$ is larger than $n$. A closer estimate is given by the function

$$
\alpha(m, n) = \min \left\{ c \geq 1 \mid \lg^{c} (\lg n) \leq m/n \right\}.
$$

It’s not hard to prove that if $m = \Theta(n)$, then $\alpha(m, n) = \Theta(\alpha(n))$. On the other hand, if $m \geq n \lg^{**} n$, for any constant number of stars, then $\alpha(m, n) = O(1)$. So even if the number of Find operations is only slightly larger than the number of nodes, the amortized cost of each Find is constant.

$O(\alpha(m, n))$ is actually a tight upper bound for the amortized cost of path compression; there are no more tricks that will improve the analysis further. More surprisingly, this is the best amortized bound we obtain for any pointer-based data structure for maintaining disjoint sets; the amortized cost of every Find algorithm is at least $\Omega(\alpha(m, n))$. The proof of the matching lower bound is, unfortunately, far beyond the scope of this class.\(^3\)

11.6 The Ackermann Function and its Inverse

The iterated logarithms that fell out of our analysis of path compression are the inverses of a hierarchy of recursive functions defined by Wilhelm Ackermann in 1928.\footnote{Ackermann didn’t define his functions this way—I’m actually describing a slightly cleaner hierarchy defined 35 years later by R. Creighton Buck—but the exact details of the definition are surprisingly irrelevant! The mnemonic up-arrow notation for these functions was introduced by Don Knuth in the 1970s.}

\[ 2 \uparrow^c n := \begin{cases} 2 & \text{if } n = 1 \\ 2n & \text{if } c = 0 \\ 2 \uparrow^{c-1} (2 \uparrow^c (n-1)) & \text{otherwise} \end{cases} \]

For each fixed integer \( c \), the function \( 2 \uparrow^c n \) is monotonically increasing in \( n \), and these functions grow incredibly faster as the index \( c \) increases. \( 2 \uparrow n \) is the familiar power function \( 2^n \). \( 2 \uparrow \uparrow n \) is the tower function:

\[ 2 \uparrow \uparrow n = \underbrace{2 \uparrow 2 \uparrow \ldots \uparrow 2}_{n} = 2^{2^{2^{\ldots^{2}}}} \]

John Conway named \( 2 \uparrow \uparrow \uparrow n \) the wower function:

\[ 2 \uparrow \uparrow \uparrow n = \underbrace{2 \uparrow 2 \uparrow \ldots \uparrow 2}_{n} \]

And so on, et cetera, ad infinitum.

For any fixed \( c \), the function \( \log^c n \) is the inverse of the function \( 2 \uparrow^{c+1} n \), the \((c+1)\)th row in the Ackerman hierarchy. Thus, for any remotely reasonable values of \( n \), say \( n \leq 2^{256} \), we have \( \log^c n \leq 5, \log^{c+1} n \leq 4 \), and \( \log^{c+2} n \leq 3 \) for any \( c \geq 3 \).

The function \( \alpha(n) \) is usually called the inverse Ackerman function.\footnote{Strictly speaking, the name ‘inverse Ackerman function’ is inaccurate. One good formal definition of the true inverse Ackerman function is \( \bar{\alpha}(n) = \min \{ c \geq 1 \mid 2 \uparrow^c c \geq n \} = \min \{ c \geq 1 \mid 2 \uparrow^{c+2} c \geq n \} \}. However, it’s not hard to prove that \( \bar{\alpha}(n) \leq \alpha(n) + 1 \) for all sufficiently large \( n \), so the inaccuracy is completely forgivable. As I said in the previous footnote, the exact details of the definition are surprisingly irrelevant!} Our earlier definition is equivalent to \( \alpha(n) = \min \{ c \geq 1 \mid 2 \uparrow^{c+2} 3 \geq n \} \); in other words, \( \alpha(n) + 2 \) is the inverse of the third column in the Ackermann hierarchy. The function \( \alpha(n) \) grows much more slowly than \( \log^c n \) for any fixed \( c \); we have \( \alpha(n) \leq 3 \) for all even remotely imaginable values of \( n \). Nevertheless, the function \( \alpha(n) \) is eventually larger than any constant, so it is not \( O(1) \).

<table>
<thead>
<tr>
<th>( 2 \uparrow n )</th>
<th>( n = 1 )</th>
<th>( n = 2 )</th>
<th>( n = 3 )</th>
<th>( n = 4 )</th>
<th>( n = 5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2n )</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>( 2 \uparrow n )</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>( 2 \uparrow \uparrow n )</td>
<td>2</td>
<td>4</td>
<td>16</td>
<td>65536</td>
<td>( 2^{65536} )</td>
</tr>
<tr>
<td>( 2 \uparrow \uparrow \uparrow n )</td>
<td>2</td>
<td>4</td>
<td>65536</td>
<td>( 2^{65536} )</td>
<td>( 2^{65536} )</td>
</tr>
<tr>
<td>( 2 \uparrow \uparrow \uparrow \uparrow n )</td>
<td>2</td>
<td>4</td>
<td>( 2^{65536} )</td>
<td>( 2^{65536} )</td>
<td>( 2^{65536} ) | ( \langle \text{Yeah, right.} \rangle )</td>
</tr>
<tr>
<td>( 2 \uparrow \uparrow \uparrow \uparrow \uparrow n )</td>
<td>2</td>
<td>4</td>
<td>( 2^{65536} )</td>
<td>( 2^{65536} )</td>
<td>( 2^{65536} ) | ( \langle \text{Very funny.} \rangle ) | ( \langle \text{Argh! My eyes!} \rangle )</td>
</tr>
</tbody>
</table>

Small (!!!) values of Ackermann’s functions.
11.7 To infinity... and beyond!

Of course, one can generalize the inverse Ackermann function to functions that grow arbitrarily more slowly, starting with the \textit{iterated} inverse Ackermann function

\[
\alpha^*(n) = \begin{cases} 
1 & \text{if } n \leq 4, \\
1 + \alpha^*(\alpha(n)) & \text{otherwise},
\end{cases}
\]

then the \textit{iterated} iterated iterated inverse Ackermann function

\[
\alpha^{**}(n) = \begin{cases} 
\alpha(n) & \text{if } c = 0, \\
1 & \text{if } n \leq 4, \\
1 + \alpha^{**}(\alpha^{**}-1(n)) & \text{otherwise},
\end{cases}
\]

and then the diagonalized inverse Ackermann function

\[
\text{Head-asplode}(n) = \min\{c \geq 1 \mid \alpha^{**} n \leq 4\},
\]

and so on ad nauseam. Fortunately(?), such functions appear extremely rarely in algorithm analysis. Perhaps the only naturally-occurring example of a super-constant sub-inverse-Ackermann function is a recent result of Seth Pettie, who proved that if a splay tree is used as a double-ended queue — insertions and deletions of only smallest or largest elements — then the amortized cost of any operation is $O(\alpha^*(n))$!

\section*{Exercises}

1. Consider the following solution for the union-find problem, called \textit{union-by-weight}. Each set leader $\overline{x}$ stores the number of elements of its set in the field $\text{weight} (\overline{x})$. Whenever we \textsc{Union} two sets, the leader of the \textit{smaller} set becomes a new child of the leader of the \textit{larger} set (breaking ties arbitrarily).

   \begin{itemize}
   \item \textbf{MAKESET}(x):
   \begin{align*}
   \text{parent}(x) & \leftarrow x \\
   \text{weight}(x) & \leftarrow 1 
   \end{align*}
   \item \textbf{FIND}(x):
   \begin{align*}
   \text{while } x \neq \text{parent}(x) & \\
   & x \leftarrow \text{parent}(x) \\
   \text{return } x 
   \end{align*}
   \item \textbf{UNION}(x, y):
   \begin{align*}
   \overline{x} & \leftarrow \text{FIND}(x) \\
   \overline{y} & \leftarrow \text{FIND}(y) \\
   \text{if } \text{weight}(\overline{x}) > \text{weight}(\overline{y}) & \\
   \text{parent}(\overline{y}) & \leftarrow \overline{x} \\
   \text{weight}(\overline{x}) & \leftarrow \text{weight}(\overline{x}) + \text{weight}(\overline{y}) \\
   \text{else} & \\
   \text{parent}(\overline{x}) & \leftarrow \overline{y} \\
   \text{weight}(\overline{x}) & \leftarrow \text{weight}(\overline{x}) + \text{weight}(\overline{y})
   \end{align*}
   \end{itemize}

   Prove that if we use union-by-weight, the \textit{worst-case} running time of $\text{FIND}(x)$ is $O(\log n)$, where $n$ is the cardinality of the set containing $x$.

2. Consider a union-find data structure that uses union by depth (or equivalently union by rank) \textit{without} path compression. For all integers $m$ and $n$ such that $m \geq 2n$, prove that there is a sequence of $n$ MakeSet operations, followed by $m$ Union and Find operations, that require $\Omega(m \log n)$ time to execute.

3. Suppose you are given a collection of up-trees representing a partition of the set \( \{1, 2, \ldots, n\} \) into disjoint subsets. **You have no idea how these trees were constructed.** You are also given an array \( \text{node}[1..n] \), where \( \text{node}[i] \) is a pointer to the up-tree node containing element \( i \). Your task is to create a new array \( \text{label}[1..n] \) using the following algorithm:

```plaintext
LABEL EVERYTHING:
  for \( i \leftarrow 1 \) to \( n \)
    \( \text{label}[i] \leftarrow \text{Find}(\text{node}[i]) \)
```

(a) What is the worst-case running time of \( \text{LABEL EVERYTHING} \) if we implement \( \text{Find} \) without path compression?

(b) **Prove** that if we implement \( \text{Find} \) using path compression, \( \text{LABEL EVERYTHING} \) runs in \( O(n) \) time in the worst case.

4. Consider an arbitrary sequence of \( m \) \( \text{MakeSet} \) operations, followed by \( u \) \( \text{Union} \) operations, followed by \( f \) \( \text{Find} \) operations, and let \( n = m + u + f \). Prove that if we use union by rank and \( \text{Find} \) with path compression, all \( n \) operations are executed in \( O(n) \) time.

5. Suppose we want to maintain an array \( X[1..n] \) of bits, which are all initially zero, subject to the following operations.

- **LOOKUP**\((i)\): Given an index \( i \), return \( X[i] \).
- **BLACKEN**\((i)\): Given an index \( i < n \), set \( X[i] \leftarrow 1 \).
- **NEXTWHITE**\((i)\): Given an index \( i \), return the smallest index \( j \geq i \) such that \( X[j] = 0 \).
  (Because we never change \( X[n] \), such an index always exists.)

If we use the array \( X[1..n] \) itself as the only data structure, it is trivial to implement **LOOKUP** and **BLACKEN** in \( O(1) \) time and **NEXTWHITE** in \( O(n) \) time. But you can do better! Describe data structures that support **LOOKUP** in \( O(1) \) worst-case time and the other two operations in the following time bounds. (We want a different data structure for each set of time bounds, not one data structure that satisfies all bounds simultaneously!)

(a) The worst-case time for both **BLACKEN** and **NEXTWHITE** is \( O(\log n) \).

(b) The amortized time for both **BLACKEN** and **NEXTWHITE** is \( O(\log n) \). In addition, the worst-case time for **BLACKEN** is \( O(1) \).

(c) The amortized time for **BLACKEN** is \( O(\log n) \), and the worst-case time for **NEXTWHITE** is \( O(1) \).

(d) The worst-case time for **BLACKEN** is \( O(1) \), and the amortized time for **NEXTWHITE** is \( O(\alpha(n)) \). [Hint: There is no **WHITEN**.]

6. Suppose we want to maintain a collection of strings (sequences of characters) under the following operations:

- **NEWSTRING**\((a)\) creates a new string of length 1 containing only the character \( a \) and returns a pointer to that string.
• **CONCAT**(S, T) removes the strings S and T (given by pointers) from the data structure, adds the concatenated string ST to the data structure, and returns a pointer to the new string.

• **REVERSE**(S) removes the string S (given by a pointer) from the data structure, adds the reversal of S to the data structure, and returns a pointer to the new string.

• **LOOKUP**(S, k) returns the kth character in string S (given by a pointer), or NULL if the length of the S is less than k.

Describe and analyze a simple data structure that supports **CONCAT** in \(O(\log n)\) amortized time, supports every other operation in \(O(1)\) worst-case time, and uses \(O(n)\) space, where \(n\) is the sum of the current string lengths. Unlike the similar problem in the previous lecture note, there is no **SPLIT** operation. [Hint: Why is this problem here?]

7. (a) Describe and analyze an algorithm to compute the size of the largest connected component of black pixels in an \(n \times n\) bitmap \(B[1..n, 1..n]\).

For example, given the bitmap below as input, your algorithm should return the number 9, because the largest connected black component (marked with white dots on the right) contains nine pixels.

(b) Design and analyze an algorithm **BLACKEN**(i, j) that colors the pixel \(B[i, j]\) black and returns the size of the largest black component in the bitmap. For full credit, the amortized running time of your algorithm (starting with an all-white bitmap) must be as small as possible.

For example, at each step in the sequence below, we blacken the pixel marked with an X. The largest black component is marked with white dots; the number underneath shows the correct output of the **BLACKEN** algorithm.

(c) What is the worst-case running time of your **BLACKEN** algorithm?

*8. Consider the following game. I choose a positive integer \(n\) and keep it secret; your goal is to discover this integer. We play the game in rounds. In each round, you write a list of at most \(n\) integers on the blackboard. If you write more than \(n\) numbers in a single round, you lose. (Thus, in the first round, you must write only the number 1; do you see why?) If \(n\) is one of the numbers you wrote, you win the game; otherwise, I announce which of
the numbers you wrote is smaller or larger than $n$, and we proceed to the next round. For example:

<table>
<thead>
<tr>
<th>You</th>
<th>Me</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>It’s bigger than 1.</td>
</tr>
<tr>
<td>4, 42</td>
<td>It’s between 4 and 42.</td>
</tr>
<tr>
<td>8, 15, 16, 23, 30</td>
<td>It’s between 8 and 15.</td>
</tr>
<tr>
<td>9, 10, 11, 12, 13, 14</td>
<td>It’s 11; you win!</td>
</tr>
</tbody>
</table>

Describe a strategy that allows you to win in $O(\alpha(n))$ rounds!