Chapter 7

Union/Find data structures

The Union/Find problem is the well-known problem of maintaining disjoint sets, such that we can create new sets, take the union of two distinct sets, and return for any given element an identifier of its set. Formally:

Problem 3 (Union/Find) *Given: Disjoint sets $S_1, \ldots, S_k$ from a universe U.*
Want: Support three operations:

- *MakeSet(x): This creates a new set that contains only x. x was not previously in any set.*
- *Union(S_1, S_2): This removes S_1 and S_2 from the list of sets and adds a new set S which contains $S_1 \cup S_2$.*
- *Find(x): This returns an identifier of the set that contains x. The identifier can be anything (e.g., a number or a reference), but it must satisfy that Find(x) =Find(y) if and only if x and y belong to the same set.*

7.1 Basic implementations

There are many simple implementations of Union/Find, such as linked lists (with or without head references) and trees. CLRS, Chapter 21 reviews these well.

7.2 Union/Find in $O(\alpha(n))$ amortized time

The best data structure for Union/Find is to use trees with union-by-size heuristic and path compression. To do this, each item x stores $p(x)$, which is its parent in the tree, and it also stores an integer $size(x)$ (which we will analyze in detail later.) Here is the code:

```
MakeSet(x) {
    Create a one-node tree that stores x.
    size(x) = 1; p(x) = NULL
}
Union(x,y) {
    If size(x) > size(y) {
        p(y) := x; size(x) += size(y)
    } else {
        p(x) := y; size(y) += size(x)
    }
}
Find(x) {
    If p(x) == NULL return x
    else p(x) := Find(p(x))
}
```

It should be obvious that Union and MakeSet take $O(1)$ time. The difficulty is to show that Find takes $O(\alpha(n))$ amortized time for the very slow-growing so-called Ackerman function that we will define formally later.

7.2.1 Observations about size

Observe the following about $size(x)$, which are all quite straightforward to prove by induction.

1. $size(x)$ can only change if x is a root.
2. If x is a root of a tree, then $size(x)$ equals the number of descendants of x. Otherwise, $size(x)$ is an upper bound on the number of descendants of x.
3. If x is not a root, then $size(p(x)) \geq 2 \cdot size(x)$.
4. If x acquires a new parent, then $size(p_{new}(x)) \geq size(p_{old}(x))$ (because the new parent is an ancestor of the old parent.)

We also need one less trivial observation.

Lemma 7.1 *For any $a > 0$, the number of nodes x for which $size(x) \geq a$ is at most $\frac{n}{a}$.*

Proof: We will show that to each such node x, we can assign a other nodes such that every node is assigned at most once; this proves the claim.

Consider the first time that x had $size(x) \geq a$. So $size(x)$ just changed, which means that by property (1) above x is a root. So x has $size(x) \geq a$ descendants, and we assign a of them to x.

Presume now this double-counts a node. Say at some later time a node y that was assigned to x is again assigned to some other node z. This happens when $size(z) \geq a$ for the first time. So the size of z just increased, so z is the root of its subtree, and y is a descendant of z. Node x must be in the same tree as y, because at an earlier time y was x's descendants, and nodes never get removed from trees.) So x is a descendant of z. But then $size(z) \geq 2size(x) \geq 2a$ already, contradicting the assumption. $\square$

7.2.2 Counting parent-changes

Note that the time for operation Find is dominated by the time to do parent-changes. We will hence in the following only count how often a node acquires a new parent; all other operations are proportional to this.

Our goal is hence to show that during m Find's on n elements, there are at most $O(n + m\alpha(n))$ parent-changes. We will prove this by induction, and to be able to do so, we need to make a stronger claim.

Definition 1 *For any integers $1 \leq a < b \leq n+1$ and any integer $m \geq 0$, let $T(a,b,m)$ be the maximal number of parent-changes at a node of size $\geq a$ that could happen during m Find-operations for which the result has size in the interval $[a,b)$.*

Since $1 \leq size(x) < n+1$ for any node x, we hence want to bound $T(1, n+1, m) \in O((n+m)\alpha(n))$.

We will in the following sometimes use $m_{[\ell,r)}$ to denote the number of Find-operations where the result has size in the interval $[\ell, r)$. In particular, for $T(a,b,m)$, the "m" would more properly be written as $m_{[a,b)}$.

7.2.3 A recursive formula

We now develop a formula for $T(a,b,m)$ that is recursive (on $b-a$). The base case is $b - a \leq a$, i.e., we want to bound $T(a,b,m)$ for $b \leq 2a$. This is very easy. Since all Finds are required to return an element of size $< b$, any parent-change result in a new parent that has size $< b$. By Property (3) hence the old parent must have had size $< b/2 \leq a$. But we only count parent-changes where the node itself has size $\geq a$ (and so the old parent had size $\geq 2a$. So no such parent-change can happen, hence $T(a,b,m) = 0$ for $b \leq 2a$.

Now we develop the recursive case. Pick an arbitrary integer d with $a < d < b$ (this exists, otherwise $b = a + 1 \leq 2a$.) View d as a "separating line" for the sizes of nodes. Consider a parent-change where node x previously had parent y, and it now becomes a child of z. By definition, we have $size(x) \geq a$. Furthermore, the new parent z is the result of the Find, so $size(z) \leq b$. Since sizes increase as we go up in the tree, we have $a \leq size(x) < size(y) < size(z) < b$. The value of d must fall onto one of those four inequalities, yielding the four cases below.

- For some parent-changes, we have $size(z) < d$. Since z is the result of a Find-query, this can happen only during the Find-operations where the result had size $< d$. There are (per the above notations) $m_{[a,d)}$ such Find-operations. Therefore, all such parent-changes would be counted by $T(a, d, m_{[a,d)})$.

- For some parent-changes, we have $size(x) \geq d$. Similarly as above one argues that they would all be counted in $T(d, b, m_{[d,b)})$.
- For some parent-changes, we had $size(y) < d$, but $size(z) \geq d$. In other words, the parent of x "crosses the line" from below d to above d. Since sizes never decrease, this can happen to each node only once. And since x has $size(x) \geq a$, there are only $\frac{n}{a}$ such nodes by Lemma 7.1. Hence there are at most $\frac{n}{a}$ such parent-changes.
- For some parent-changes, we have $size(x) < d$ but $size(y) \geq d$. In other words, the path from the node that triggered the Find to the root "crosses the line" from below d to above d. This can happen only once per Find-operation, and only for such Find-operations that return a node with size in the interval $[d, b)$. So there are at most $m_[d, b)$ such parent-changes.

This mostly gives a recursion, except that we don't know what $m_{[a,d)}$ and $m_{[d,b)}$ are (other than that they sum to at most $m_{[a,b)}$.) So we take the maximum over all of them for an upper bound. On the flip-side, the bound holds for all possible values of d, so we can take the minimum over all values of d. In summary, we obtain the following recursion:

$$\begin{aligned}
T(a, b, m_{[a,b)}) &= 0 \quad \text{if } b \leq 2a \\
T(a, b, m_{[a,b)}) &\leq \min_{a<d<b} \max_{m_{[a,d)}+m_{[d,b)} \leq m_{[a,b)}} \left\{ T(a, d, m_{[a,d)}) + T(d, b, m_{[d,b)}) + \frac{n}{a} + m_{[d,b)} \right\} \\
&\qquad \text{otherwise}
\end{aligned}$$

The good news is that from now on, we can forget all about the Union/Find data structure and only need to analyze this recursion. The bad news is that this is definitely not a simple recursion to analyze.

7.2.4 Resolving the recursion

Rather than solving the recursion directly, let us make partial progress first by evaluation it for $d := b/2$. We get:

$$\begin{aligned}
& T(a, b, m_{[a,b)}) \\
= \;& \max_{m_{[a,b/2)}+m_{[b/2,b)} \leq m_{[a,b)}} T(a, b/2, m_{[a,b/2)}) + T(b/2, b, m_{[b/2,b)}) + \frac{n}{a} + m_{[b/2,b)} \\
& \text{(The second term is 0. For brevity omit the max-quantifiers.)} \\
\leq \;& T(a, b/2, m_{[a,b/2)}) + \frac{n}{a} + m_{[b/2,b)} \\
& \text{(Repeat the process on the first term, using } d := b/4.\text{)} \\
\leq \;& \left[T(a, b/4, m_{[a,b/4)} + \frac{n}{a} + m_{[b/4,b/2)} \right] + \frac{n}{a} + m_{[b/2,b)} \\
& \text{(Combine the } m\text{-terms (they count different Finds) and clean-up.)}
\end{aligned}$$

$$\begin{aligned}
&= T(a, b/4, m_{[a,b/4)}) + 2 \cdot \frac{n}{a} + m_{[b/4,b)} \\
&\quad \text{(Do it again with } d := b/8.\text{)} \\
&\leq T(a, b/8, m_{[a,b/8)}) + 3 \cdot \frac{n}{a} + m_{[b/8,b)} \\
&\quad \text{(Repeat this } \ell \text{ times until } b/2^\ell \leq 2a\text{)} \\
\ldots &\leq T(a, b/2^\ell, m_{[a,b/2^\ell)}) + \ell \cdot \frac{n}{a} + m_{[b/2^\ell,b)} \\
&\quad \text{(the first term vanishes, the third term is increased, and } \ell \leq \log(b/a).\text{)} \\
&\leq \log(b/a) \cdot \frac{n}{a} + m_{[a,b)}
\end{aligned}$$

Since we will need this later, let's write this down separately:

Lemma 7.2 $T(a, b, m_{[a,b)}) \leq \log(b/a) \cdot \frac{n}{a} + m_{[a,b)}$.

Now let's do the fun again, but use $d := d_1 := \log(b/a) \cdot a$. As before, we'll omit the max quantifiers.

$$\begin{aligned}
T(a, b, m_{[a,b)}) &\leq T(a, d_1, m_{[a,d_1)}) + T(d_1, b, m_{[d_1,b)}) + \frac{n}{a} + m_{[d_1,b)} \\
&\quad \text{(use Lemma 7.2 on the second term, but with lower value } d_1 \text{ instead of } a\text{)} \\
&\leq T(a, d_1, m_{[a,d_1)}) + \left[\log(b/d_1)(n/d_1) + m_{[d_1,b)}\right] + \frac{n}{a} + m_{[d_1,b)} \\
&\quad \text{(evaluate } d_1 = \log(b/a) \cdot a \geq a \text{ and simplify)} \\
&\leq T(a, d_1, m_{[a,d_1)}) + \log(b/a)\frac{n}{\log(b/a)a} + \frac{n}{a} + 2 \cdot m_{[d_1,b)} \\
&\quad \text{(simplify more)} \\
&\leq T(a, d_1, m_{[a,d_1)}) + 2 \cdot \frac{n}{a} + 2 \cdot m_{[d_1,b)} \\
&\quad \text{(Now use the formula that we just got, with } d_2 = \log(d_1/a) \cdot a = \log\log(b/a) \cdot a. \\
&\leq \left[T(a, d_2, m_{[a,d_2)}) + 2 \cdot \frac{n}{a} + 2 \cdot m_{[d_2,d_1)}\right] + 2 \cdot \frac{n}{a} + 2 \cdot m_{[d_1,b)} \\
&= T(a, d_2, m_{[a,d_2)}) + 4 \cdot \frac{n}{a} + 2 \cdot m_{[d_2,b)} \\
&\quad \text{(Repeat } \ell \text{ times, where } \ell \text{ is so big that } d_\ell = \underbrace{\log\log\ldots\log}_{\ell \text{ times}}(b/a) \cdot a \leq 2a.\text{)} \\
\ldots &\leq T(a, d_\ell, m_{[a,d_\ell)}) + 2\ell \cdot \frac{n}{a} + 2 \cdot m_{[d_\ell,b)} \\
&\quad \text{(the first term vanishes since } d_\ell \leq 2a.\text{ Also } \ell \leq \log^*(b/a).\text{)} \\
&\leq 2\log^*(b/a) \cdot \frac{n}{a} + 2 \cdot m_{[a,b)}
\end{aligned}$$

So we get:

Lemma 7.3 $T(a, b, m) \leq 2\log^*(b/a) \cdot \frac{n}{a} + 2m$.

Note that this gives already an $O(\log^* n)$ amortized bound for Union/Find, which already would be very good. But we can be much much smaller!

To do so, we bootstrap again. In other words, now use $d := \log^*(b/a) \cdot a$, apply the recursion using Lemma 7.3, the terms nicely cancel out and we get $T(a, b, m) \leq 3\ell \cdot \frac{n}{a} + 3m$, where ℓ is the number of times that we need to apply $\log^*$ to b/a until we get a number smaller than 2. Then bootstrap again. And again. And again. Until finally we are at a place where with the very first recursion we reach the base case. Now how often have we bootstrapped?

7.2.5 A strange function

It is time to define a strange function. Actually, a family of functions.

Definition 2 *Define $\ell_0(n) = n/2$. For all $k > 0$, define $\ell_k(n)$ to be the number of times that we have to apply ℓ_{k-1} to n until we reach a value that is less than 2. Function ℓ_k is also called the k*th iterated log.

So $\ell_1(n)$ is the number of times that we need to divide n in half until we reach a number smaller than 2. In other words, $\ell_1(n)$ is roughly $\log n$. Function $\ell_2(n)$ is the number of times we need to apply log to reach a number smaller than 2, or in other words, $\ell_2(n) \approx \log^*(n)$. This is already *incredibly* slow-growing. $\ell_3(n)$ is even slower-growing, and $\ell_4(n)$ *even* slower-growing.

Definition 3 *Define $\alpha(n)$ to be the smallest number k such that $\ell_k(n) < 2$. This function is called the* inverse Ackerman function. [1]

One can now show:

Lemma 7.4 *For all $k > 0$ with $\ell_k(b/a) \geq 1$, we have $T(a, b, m) \leq k\ell_k(b/a) \cdot \frac{n}{a} + km$.*

The proof of this is a double induction, both on k and on the parameter L that says how often we applied function ℓ_k. It is not difficult but lengthy, and we've seen all the basic ingredients already: The base case is the proof of Lemma 7.2, and the induction step is done as in the proof of Lemma 7.3, using ℓ_k in place of log. Details can be found in Section 7.2.7.

[1]This is not quite the definition of $\alpha(n)$ found in other textbooks, but the two functions can be shown to be equally slow-growing. In fact, there are quite a few functions called "Ackerman function." The function is famous for being extremely slow-growing but not constant. It is also famous for being computable but not primitive-recursive, but that's another story.

7.2.6 Putting it all together

From Lemma 7.4, we know $T(a,b,m) \le k\ell_k(b/a) \cdot \frac{n}{a} + km$. In particular therefore, $T(1,n+1,m) \le k\ell_k(n+1) \cdot n + km$. Apply this using $k = \alpha(n+1)$ and hence $\ell_k(b/a) \le 2$, we get $T(1,n+1,m) \le 2\alpha(n+1)n + \alpha(n+1)m$. Since $\alpha(n+1) \in O(\alpha(n))$, we get:

Theorem 1 *m Find-operations on n elements have run-time $O(\alpha(n)(n+m))$.*

7.2.7 The proof of Lemma 7.4.

In this section, we now give a formal proof of Lemma 7.4. In fact, we prove two things simultaenously with a double induction.

Claim 1 *For all $k \ge 1$ for which $\ell_k(b/a) \ge 1$, we have $T(a,b,m_{[a,b)}) \le k\ell_k(b/a) \cdot \frac{n}{a} + k \cdot m_{[a,b)}$.*

Claim 2 *For all $k \ge 0$ for which $\ell_k(b/a) \ge 1$, and all $L \ge 0$, we have*

$$T(a,b,m_{[a,b)}) \le T(a, d_k^L, m_{[a,d_k^L)}) + (k+1)L\frac{n}{a} + (k+1)m_{[d_k^L,b)},$$

where $d_k^L = \underbrace{\ell_k(\ell_k(\dots(\ell_k}_{L \text{ times}}(b/a))) \cdot a.$

Before proving this, note that Claim 2 expresses the "do it again and again" instruction from the proofs of Lemma 7.2 and 7.3, except that it makes it formal by using L to denote how often we have done the evaluation. So we have already proved all base cases for the claims, but to be precise, we will re-prove them and give a complete proof below.

We prove the two claims simultaneously, using an induction on k.

Induction base: $k = 0$: In the case $k = 0$ nothing is to show for Claim 1 (since it must only hold for $k \ge 1$.) To prove Claim 2, we do an induction on L. The claim for $L = 0$ is trivial since $d_k^L = d_0^0 = (b/a) \cdot a = b$, and the left hand side and right hand side are identical. So assume that $L \ge 1$. By induction we can then use Claim 2 for $L-1$ and get:

$$\begin{aligned}
T(a,b,m_{[a,b)}) &\le T(a, d_0^{L-1}, m_{[a,d_0^{L-1})}) + (L-1)\frac{n}{a} + m_{[d_0^{L-1},b)} \\
&\quad \text{(Evaluate first term using recursive formula with } d := d_0^L.\text{)} \\
&\le \left[T(a,d,m_{[a,d)}) + T(d, d_0^{L-1}, m_{[d,d_0^{L-1})}) + \frac{n}{a} + m_{[d,d_0^{L-1})}\right] + (L-1)\frac{n}{a} + m_{[d_0^{L-1},b)} \\
&\quad \text{(Observe that } d = d_0^L = \tfrac{1}{2^L}(b/a) \cdot a = b/2^L = d_0^{L-1}/2.\text{)} \\
&\quad \text{(So the second term vanishes. Also simplify.)} \\
&\le T(a,d,m_{[a,d)}) + L\frac{n}{a} + m_{[d,b)}
\end{aligned}$$

which proves Claim 2 for $k = 0$ since $d = d_0^L$.

Induction Step: $k-1 \to k$**:** We first prove Claim 1, which is really easy. Let $L = \ell_k(b/a)$, i.e., $\underbrace{\ell_{k-1}(\ell_{k-1}(\dots(\ell_{k-1}}_{L \text{ times}}(b/a))) \leq 2$ and therefore $d_{k-1}^L \leq 2a$ for this L. Apply Claim 2 for $k-1$ and this L. Then

$$\begin{aligned}
T(a,b,m_{[a,b)}) &\leq T(a, d_{k-1}^L, m_{[a,d_{k-1}^L)}) + ((k-1)+1)L\frac{n}{a} + ((k-1)+1)m_{[d_{k-1}^L,b)} \\
&\quad \text{(The first term vanishes since } d_{k-1}^L \leq 2a. \text{ Also } L = \ell_k(b/a).) \\
&\leq k\ell_k(b/a)\frac{n}{a} + m_{[a,b)}
\end{aligned}$$

as desired.

We will prove Claim 2 with an indunction on L. The case $L = 0$ is again trivial (and actually not needed); we will prove the case $L = 1$ as base case.

Inner induction base: $L = 1$**:** Use the recursive formula to evaluate $T(a,b,m)$ with $d := d_k^1 = \ell_k(b/a)$. Then we have

$$\begin{aligned}
T(a,b,m_{[a,b)}) &\leq T(a,d,m_{[a,d)}) + T(d,b,m_{[d,b)}) + \frac{n}{a} + m_{[d,b)} \\
&\quad \text{(use the just-proved Claim 1 for } k \text{ on the second term.)} \\
&\leq T(a,d,m_{[a,d)}) + \left[k\ell_k(b/d)(n/d) + k \cdot m_{[d,b)}\right] + \frac{n}{a} + m_{[d,b)} \\
&\quad \text{(evaluate } d = \ell_k(b/a) \cdot a \geq a \text{ and simplify.)} \\
&\leq T(a,d,m_{[a,d)}) + k\ell_k(b/a)\frac{n}{\ell_k(b/a)a} + \frac{n}{a} + (k+1)m_{[d,b)} \\
&\quad \text{(simplify more.)} \\
&\leq T(a,d,m_{[a,d)}) + (k+1)\frac{n}{a} + (k+1)m_{[d,b)}
\end{aligned}$$

which proves Claim 2 for arbitrary k and $L = 1$ since $d = d_k^1$.

Inner induction step: $L-1 \to L$**:** To prove this for larger L, we start using Claim 2 for the same k and for $L-1$.

$$\begin{aligned}
T(a,b,m_{[a,b)}) &\leq T(a, d_k^{L-1}, m_{[a,d_k^{L-1})}) + (k+1)(L-1)\frac{n}{a} + (k+1)m_{[d_k^{L-1},b)} \\
&\quad \text{(Evaluate the first term, using Claim 2 for the same } k \text{ and } L = 1.) \\
&\quad \text{(Since the upper end in this term is } d_k^{L-1}, \text{ define } d := \ell_k(d_k^{L-1}/a) \cdot a.) \\
&\leq \Big[T(a,d,m_{[a,d)}) + (k+1)\frac{n}{a} + (k+1)m_{[d,d_k^{L-1})}\Big] \\
&\qquad + (k+1)(L-1)\frac{n}{a} + (k+1)m_{[d_k^{L-1},b)} \\
&\quad \text{(simplify)} \\
&= T(a,d,m_{[a,d)}) + (k+1)L\frac{n}{a} + m_{[d,b)}
\end{aligned}$$

Now observe that

$$d = \ell_k(\frac{d_k^{L-1}}{a}) \cdot a = \ell_k \left(\frac{\underbrace{\ell_k(\ell_k(\dots(\ell_k}_{L-1 \text{ times}}(b/a))) \cdot a}{a} \right) \cdot a = \underbrace{\ell_k(\ell_k(\ell_k(\dots(\ell_k}_{L \text{ times}}(b/a)))) \cdot a = d_k^L$$

so this proves the inductive claim, hence Lemma 7.4 and Theorem 1.