DISJOINT SET DATA STRUCTURE OPERATIONS

- makeset (u): This creates a set with one element, u
- find(u): finds the set to which u belongs
- union(u,v): merges the sets containing u and v

Runtime of Kruskal’s:

\[ |V| \text{makeset} + 2|E| \text{find} + (|V| - 1)\text{union} + \text{sort}(|E|) \]
If we are using a data structure operation or other sub-routine, an upper bound for the total time is:

$$TotalTime \leq (\text{Number of times operation is performed}) \times (\text{Worst-case time of operation})$$

But this upper bound can be too pessimistic if the time for the operation is highly variable, and "typical" times are much less than worst-case times. (Here, we mean "typical" time through the run of the algorithm even on the worst-case input, not "for typical inputs")
Amortized analysis: Bound total time of $m$ operations, rather than worst-case time for single operation $\times m$.

Intuition: Fast operations make things worse in the future, but slow operations make things better in the future. (Merging might build up the height of the tree, but finding a deep vertex shrinks the average heights.)

Potential function: $P_t$: Some measure of how bad the situation is after $t$'th operation. $P_0=0$, $P_t$ non-negative

Today: All potential functions are in terms of ``tokens'' . We give a token a value $v$, and charge operations if they hand out tokens, but subtract ``redeemed'' tokens from amortized time.
Define "amortized time" of operation j to be

\[ AT_j = Time_j + P_j - P_{j-1} \]

\[ TotalAmortizedTime = AT_1 + AT_2 + \cdots + AT_m = Time_1 - P_0 + P_1 + Time_2 - P_1 + P_2 + Time_3 - P_2 + P_3 + \cdots + Time_m - P_{m-1} + P_m = TotalTime + P_m - P_0 \]

\[ \geq \text{Total time} \]
  
  since \( P_0 = 0, P_m \geq 0 \)
Thus, the total amortized time is an upper bound on the total time of all operations.

Then the total time for $m$ operations is at most $m \times$ Worst-case amortized time of an operation.

In other words, worst-case ``amortized” time can be a tighter bound on ``average time” for data structure operations. Averaging is over a sequence of operations, not over random inputs.
Say we write the merge algorithm, which combines two sorted lists into one as:

\[
\text{Merge}\left[ A[1..n], B[1..m] \right]
\]

\[
I = 1, \ J = 1, \ K = 1
\]

While \( I \leq n \) and \( J \leq m \) do:

\[
\]

\[
C[K] = A[I], \ I++ , \ K++
\]

If \( I > n \), copy rest of \( B \) into \( C \), else copy rest of \( A \) into \( C \)

There are two nested loops, inside has worst-case time \( O(m) \), outside loops up to \( n \) times. However, actual worst-case time is \( O(m+n) \)
\( I = 1,  J = 1,  K = 1 \). Give tokens worth \( C = \) inside while time to each \( B[J] \)

While \( I \leq n \) and \( J \leq m \) do:


\( C[K] = A[I],  I++,  K++ \)

If \( I > n \), copy rest of \( B \) into \( C \), else copy rest of \( A \) into \( C \)

Initialization: Amortized time \( O(m) \)

Each step in inside while, except last, is paid for with token.

Amortized time for inside while = \( O(1) \).

Total amortized time : \( O(m) + n O(1) = O(m+n) \).
Bills have powers of 10 denominations, 1, 10, 100, 1000, 10000

When we reach 10 of one denomination, trade in for larger.

Bills deposited one at a time.

If m bills are deposited, and the register starts empty, bound the total number of trades.
If we have $n$ consecutive denominations each with 9 bills, one deposit could cause $n$ trades.

But that situation needs to be built towards.

Let the tokens be the bills in the register, and make them worth $v$, for a value of $v$ we'll solve for later.
Each time we trade in bills, we take 9 bills and replace them by 1. Amortized time of that is 1 swap – 8 v since the number of tokens went down by v. Let’s set \( v = \frac{1}{8} \), making this amortized cost 0.

Any deposit then has an immediate amortized cost of \( v = 1/8 \) followed by a series of swaps, which each have amortized cost 0, so all deposits have amortized cost 1/8

That means, the total number of swaps \( \leq \frac{1}{8} \) total number of deposits. No matter what sequence of deposits get made.
Last time, we came up with a pretty good data structure to keep track of a partition of objects into disjoint sets, which worked well in Kruskal’s algorithm.

We showed that Find took at most $O(\log |V|)$ time, and Union constant time.

But then we had an idea for an improvement, called Path Compression. We asked: how much of a difference could Path Compression take? It doesn’t improve worst-case time for Find, but could it improve amortized time? If so, by how much?
Find(v).  \text{Time} = O(\text{depth}) \quad \text{As we find the ancestors of v, we make them point directly to the leader.}

\text{If } p(v) \text{ is not } v, \quad p(v) = \text{Find}(p(v)). \quad \text{Return } p(v)

. \quad \text{We make smaller } "\text{depth}" \quad \text{root the child of the other.}

Union(u, v). \text{Still } O(1) \text{ time}

\text{If } \text{rank}(u) > \text{rank}(v) \text{ Then: } p(v) = u; \quad \text{If } \text{rank}(u) < \text{rank}(v). \text{ Then: } p(u) = v

\text{If } \text{rank}(u) = \text{rank}(v). \text{ Then } p(v) = u, \text{ rank}(u)++

\text{Note: rank might no longer} = \text{depth, because path compression might flatten tree. Still an upper bound on depth.}
We saw how fast the exponential function grew. $2^{200}$ is bigger than the time steps in the universe's history.

The inverse of the exponential function is the log function. It goes to infinity, but relatively slowly. $\log(time\ in\ universe's\ history) \leq 200$

Can you think of any functions that are even faster growing than exponential? What does that say about their inverses?
Some functions that are faster growing than $2^n$ might be $4^n, n^n, n!,\text{ or } 2^{n^2}$. I would still call these exponential type functions, but either the base or the exponent is bigger.

How about a function that is to exponential as exponential is to polynomial?
How about exponential in an exponential quantity?

\[ F(n) = 2^{2^n} \] grows amazingly fast. \( F(200) = 2^{2^{200}} \) = a number you couldn't write the binary expansion of if every time Step in history you wrote down a bit.

The inverse function is \( \log(\log n) \). \( \log(\log(\text{number of times in history})) \leq \log 200 \leq 8 \)
We can keep defining such functions, and each one is exponentially larger than the previous one. Can any function be faster than all of these at once?
The tower function, $T(n)$ is defined by

$$T(1)=2, \ T(n+1)=2^{T(n)}.$$  In other words, $T(n)$ is a tower of exponentials $n$ high.

$T(1)=2, \ T(2) = 4, \ T(3) =16, \ T(4) = 256, \ T(5) \text{ is bigger than the universe, } T(6) \text{ is too long to write in the universe, } T(7) \text{ is too long to write in every multiverse, assuming quantum physics has split the universe into parallel universes every time step in history, } T(8) \text{ is exponential in that number,.....}$

The inverse of the tower function is called $\log^* n$. You can compute it as $\log^* n = 1 \ if \ n \leq 2, \log^* n = 1 + \log^*(\log n) otherwise.$
While in principle $\log^* n$ grows to infinity as $n$ goes to infinity, you are pretty safe assuming say $\log^* n \leq 6$.

Could such a function come up in algorithm analysis?

Yes, it happens more often than you’d think.

We’ll prove a $\log^* |V|$. Upper bound on the amortized complexity of the union find data structure with path compression.
Let $b > 1$.
We can define a base $b$ version of tower function:

$$T_b(1) = b, T_b(n + 1) = b^{T_b(n)}.$$  
While $T_b(n)$ can be much smaller than $T(n)$ for $b < 2$, the height of the tower is more important than the base, so the inverse function

$$\log^*_b n \leq \log^* n + C.$$  
For some constant $C$.

We’ll show an upper bound of $\log_\left(\frac{3}{2}\right)^n$ but that is the same order as $\log^* n$. 

To do a similar analysis for union-find, imagine giving out coupons to the elements. The larger the size field, the more coupons the element will have. Each coupon will be worth "one free pointer change" for a future operation. In other words, if the time for the find operation is $C + C'$ (depth of vertex found), we'll set $v = C'$.

We'll give rules in terms of the algorithm for handing out and using coupons, but these are just for the analysis; the algorithm itself does not change.
Make set: Give the new element 2 tokens

Merge: The root that becomes the child gives the one that becomes its parent half of its tokens, round down. If the number at the child was odd, the token protocol gives the parent one more (so the parent gets half the child’s tokens, round up)

Find: If a vertex’s pointer changes, and it has at least one token, it spends that token

\[ P_t = C \times \text{Total tokens on all vertices at time } t, \text{ where } C \text{ is time to change a pointer and look-up parent pointer.} \]
VERSION 2C: DSDS OPERATIONS WITH TOKENS

Find(v).
If p(L) is not L:
  If L has tokens, and p(L) is not the root, spend one of L’s tokens
  p(L)=Find(p(L)).
  Return p(L)
  \textbf{We make smaller depth root the child of the other.}

Union(u, v).
If rank(u) > rank(v) Then: p(v)= u; \textbf{Move half of v’s tokens to u}
If rank(u) < rank(v). Then: p(u)=v. \textbf{Move half of u’s tokens to v}
If rank(u)=rank(v). Then p(v)=u, rank(u)++ \textbf{Move 1/2 v’s tokens to u}
AMORTIZED COSTS OF OPERATIONS

Make-set: $O(1) + 2C = O(1)$

Merge: $O(1) + C$ (if round-up needed) = $O(1)$

Find: $O(1) + C^* \text{ (number of pointer changes along path)} - C^* \text{ (number of vertices whose pointers change who have at least one token)} = O(1) + \text{number of ``broke” vertices along path}$

Worst-case amortized time = max number of broke vertices along a path
Claim every leader of rank $r$ has at least $2 \left(\frac{3}{2}\right)^r$ tokens.

Proof: True at start, since start with 2 tokens = $2 \left(\frac{3}{2}\right)^0$. Leaders never spend tokens, so find doesn’t change. Union only could change if both vertices have equal rank $r$, and merger has rank $r+1$.

In this case, both merged leaders had at least $2 \times \left(\frac{3}{2}\right)^r$ tokens. New vertex gets all of one and half the other, for a total of $2 \times \left(\frac{3}{2}\right)^r + \left(\frac{3}{2}\right)^r$

$= 3 \times \left(\frac{3}{2}\right)^r = 2 \times \left(\frac{3}{2}\right)^{r+1}$ tokens.
CONSEQUENCES

Corollary: At the time it becomes a non-root, each vertex $v$ has at least $\left(\frac{3}{2}\right)^{\text{rank}(v)}$ tokens.

Lemma: $\text{rank}(p(v)) > \text{rank}(v)$ for any non-leader $v$.
True first time $v$ is not a leader. After that, $\text{rank}(p(v))$ might increase, but $\text{rank}(v)$ never changes.

Lemma: Every time $v$ spends a token, $\text{rank}(p(v))$ increases.
When $v$ spends a token, $p(v)$ changes to an ancestor of $p(v)$.

Corollary: If $v$ is broke, $\text{rank}(p(v))$ is at least $\left(\frac{3}{2}\right)^{\text{rank}(v)}$
As we go up the path, every time we encounter a broke vertex, the s field \( \text{rank}(v) \) jumps to at least \( \left( \frac{3}{2} \right)^{\text{rank}(v)} \). After we pass a second one, we’ll be doubly exponential in \( \text{rank}(v) \), and so on. . So if we have \( k \) such broke vertices, top rank is at least \( T\binom{3}{2}(k) \leq \log|V| \)
Amortized cost for Find $\leq O(1 + \text{number of broke vertices})$

Number of broke vertices is $O(\log^* n)$.

Amortized cost of other operations is constant.

So total time for all finds and unions is at most $O(|E|\log^*|V| + |V| \times 1)$

Not technically linear time, but so close.....

We still have to handle sorting.
n make set operations: $O(n)$

m merge operations: $O(n)$

m find operations: $O(m \log^* n)$

Not counting sorting, last is total time..
If we can sort in linear time, and we use this data structure, time for Kruskal’s algorithm is

\[ O((m+n) \log^* n + n) = O(m \log^* n) \]

Almost as close to linear as you could imagine!
Fifth Ackerman function: Iterate fourth Ackerman function n times.

6th: Iterate fifth n times.

Ackerman function: Take n’th Ackerman function and apply it to n.

Don’t think too much about how big these numbers get. It will hurt.
Tarjan: The inverse of the ackerman function is a tight bound on the amortized time for the union-find data structure.

So although the ackerman function doesn’t seem to have anything to do with this simple data structure, this really is the time for union-find.