# Disjoint Sets Union

Ivan Dyachenko

December 9, 2021

## Contents



## 1 Part 1

Disjoint sets union (DSU) is a data structure that supports disjoint sets on $n$ elements and allows two type of queries:

- `get(a)` - return the identifier of the set to which $a$ belongs to;
- `union(a, b)` - join two sets that contain $a$ and $b$.

For example, when we call `get(a)` and `get(b)`, we can compare whether $a$ and $b$ are in the same set.

What is the simplest way to define the identifier of the set? - As an identifier, we can choose the leader of the set.

Let us maintain an array $p$, where $p[a]$ is the identifier (the leader) of a set to which $a$ belongs to.

Let us consider the pseudo-code of two functions:

```
init():
  p = new int[n]
  for i in 1..n:
    p[i] = i

get(a):
  return p[a]
```

```
union(a, b):
  a = p[a]
  b = p[b]
  for i in 1..n:
    if p[i] == a:
      p[i] = b
```

The function `get(a)` just returns the leader of a set, and the function `union(a, b)` takes the leaders of both sets and set $b$ as a leader of elements with the leader $a$.

Unfortunately, this algorithm is too slow: `get` works in $\mathcal{O}(1)$, however, `union` works in $\mathcal{O}(n)$. Is there a way to improve the algorithm?

Let us consider the simplest idea - let us iterate not over all the elements, but over the elements with the leader $a$. For that, for each leader, we will maintain a linked list $l[a]$. When we have to unite two sets, we will just link two lists together.

```
init():
  p = new int[n]
  l = new List[n]
  for i in 1..n:
    p[i] = i
    l[i] = { i }

get(a):
  return p[a]

union(a, b):
  a = p[a]
  b = p[b]
  for x in l[a]:
    p[x] = b
  l[b].append(l[a])
```

Now, `get(a)` works in $\mathcal{O}(1)$ and `union(a, b)` works in $O(|l[a]|)$. Unfortunately, this complexity is not good enough: it is possible to find an execution such that `union` will work in $O(n)$ in amortization. Consider this execution:

- `union(1, 2)`, where $|l[1]| = 1$ and $|l[2]| = 1$,
- `union(2, 3)`, where $|l[2]| = 2$ and $|l[3]| = 1$,

- `union(3, 4)`, where $|l[3]| = 3$ and $|l[4]| = 1$,
- and so on.

All operations in total work in $1+2+3+...+(n-1) = \mathcal{O}(n^2)$, and, thus, `union` still works in $\mathcal{O}(n)$. How to improve it? Note that the main problem is that we always join the first set to the second one. But what if we join the smallest set to the largest? Then, the code of `union` becomes the following:

```
union(a, b):
  a = p[a]
  b = p[b]
  if size(l[a]) > size(l[b]):
    swap(a, b)
  for x in l[a]:
    p[x] = b
  l[b].append(l[a])
```

We compare two sets and if the set $a$ is larger than the set $b$ we swap them. Note that we can implement `size(l[a])` in $\mathcal{O}(1)$ - for that we have to store the size of the list separately. How fast does this algorithm work? `get(a)` still works in $\mathcal{O}(1)$, but did we improve `union(a, b)`? Let us calculate how many times we changed the leader of $x$, i.e., the algorithm performs line `p[x] = b`. The first time we changed the leader of $x$ is when we unite it with the larger set. This means that the size of the union is at least 2. The second time we changed the leader of $x$ is when we unite the set with the larger set of size at leaset 2. This means that the size of the union is at least 4. And so on. We change the leader of $x$ only when we unite with the larger set. Since we unite all the sets together, we perform $\mathcal{O}(\log n)$ changes per element and, thus, the total cost is $\mathcal{O}(n \log n)$. Since, there are $n-1$ `union` operations, each of them works in $\mathcal{O}(\log n)$ in amortization.

## 2 Part 2

In the previous part we explained how to implement `get(a)` in $\mathcal{O}(1)$ and `union(a, b)` in $\mathcal{O}(\log n)$ amortized. But can we reduce the complexity of `union`, while slowdown `get` a little bit? It appears to be possible, but we should treat the data structure another way. We need to store the elements another way rather than in linked lists - for example, we can store them in trees. We are already given an array $p$: let us store there a parent of an element in a tree. If $p[a]$ is equal to $a$, then $a$ is a root and a leader of

the corresponding set. Initially, each element is a root of its own set, i.e., $p[a] = a$. To implement `get`, we just simply need to follow the parent links until we find the root. To implement `union`, we need to find the leaders of both sets and link one set to another.

```
get(a):
  while a != p[a]:
    a = p[a]
  return a

union(a, b):
  a = get(a)
  b = get(b)
  p[a] = b
```

Unfortunately, such an algorithm is subject to the problem discussed in the previous part: the total time of `union` operations can reach $\mathcal{O}(n^2)$. But we already know how to solve such an issue. For that, we have to join the smaller set to the larger one. When we unite two sets, elements of the smaller set now have one more link to the root. It is not hard to show that for each element the total number of links to pass to reach the root cannot exceed $\mathcal{O}(\log n)$. Thus, we get that `get` and `union` works in $\mathcal{O}(\log n)$ (not in amortization). It is pretty simple to implement.

```
union(a, b):
  a = get(a)
  b = get(b)
  if size[a] > size[b]:
    swap(a, b)
  p[a] = b
  size[b] += size[a]
```

How to improve the algorithm further? Note, that when we call `get` we find the root. Then, it is reasonable to update $p[a]$ to point to the root, so that next `get` will work faster. Operation `get` becomes the following.

```
get(a):
  if p[a] != a:
    p[a] = get(p[a])
  return p[a]
```

We rewrote the function in a recursive manner. If $a$ is a root, then the result is $p[a]$, otherwise, we set $p[a]$ to the root. This heuristic is named **path-compression**.

It appears that if we apply both heuristics: the path-compression heuristic and the link-small-to-large heuristic, we get that `get` and `union` work in $\mathcal{O}(\alpha(m, n))$ time amortized, where $\alpha(m, n)$ is the inverse Ackermann function, $m$ is the number of performed operations `get` and $n$ is the number of elements.

To give the intuition on how slow the inverse Ackermann function rises, we look at $\mathcal{O}(\log^* n)$ that rises a little bit faster. This function means how many times we should take the binary logarithm of $n$ to get a value smaller than one. Consider an example. Suppose we take a very large number $2^{65536}$ and calculate its $\log^*$. $2^{65536} \to 65536 = 2^{16} \to 16 = 2^4 \to 4 = 2^2 \to 2 \to 1 \to 0$. In total we get that $\log^* 2^{65536} = 6$. So, we can suppose that for all reasonable $n$ this function is almost constant, while the inverse Ackermann function rises even slower.