**Disjoint Sets Union. Part 1.**

DSU is a data structure that supports disjoint sets on nn elements and allows two type of queries:

* get(aa) — return the identifier of the set to which aa belongs to;
* union(aa, bb) — join two sets that contain aa and bb.

For example, when we call get(aa) and get(bb), we can compare whether aa and bb 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 pp, where p[a]p[a] is the identifier (the leader) of a set to which aa 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(aa) just returns the leader of a set, and the function union(aa, bb) takes the leaders of both sets and set bb as a leader of elements with the leader aa.

Unfortunately, this algorithm is too slow: get works in O(1)O(1), however, union works in O(n)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 aa. For that, for each leader, we will maintain a linked list l[a]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(aa) works in O(1)O(1) and union(aa, bb) works in O(|l[a]|)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)O(n) in amortization. Consider this execution:

* union(11, 22), where |l[1]|=1|l[1]|=1 and |l[2]|=1|l[2]|=1,
* union(22, 33), where |l[2]|=2|l[2]|=2 and |l[3]|=1|l[3]|=1,
* union(33, 44), where |l[3]|=3|l[3]|=3 and |l[4]|=1|l[4]|=1,
* and so on.

All operations in total work in 1+2+3+…+(n−1)=O(n2)1+2+3+…+(n−1)=O(n2), and, thus, union still works in O(n)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 aa is larger than the set bb we swap them. Note that we can implement size(l[a]) in O(1)O(1) — for that we have to store the size of the list separately. How fast does this algorithm work? get(aa) still works in O(1)O(1), but did we improve union(aa, bb)? Let us calculate how many times we changed the leader of xx, i.e., the algorithm performs line p[x] = b. The first time we changed the leader of xx is when we unite it with the larger set. This means that the size of the union is at least 22. The second time we changed the leader of xx is when we unite the set with the larger set of size at least 22. This means that the size of the union is at least 44. And so on. We change the leader of xx only when we unite with the larger set. Since we unite all the sets together, we perform O(logn)O(log⁡n) changes per element and, thus, the total cost is O(nlogn)O(nlog⁡n). Since, there are n−1n−1 union operations, each of them works in O(logn)O(log⁡n) in amortization.

We explain how to improve the algorithm further.

**Disjoint Sets Union. Part 2.**

In the previous part we explained how to implement get(aa) in O(1)O(1) and union(aa, bb) in O(logn)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 pp: let us store there a parent of an element in a tree. If p[a]p[a] is equal to aa, then aa is a root and a leader of the corresponding set. Initially, each element is a root of its own set, i.e., p[a]=ap[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 O(n2)O(n2). 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 O(logn)O(log⁡n). Thus, we get that get and union works in O(logn)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]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 aa is a root, then the result is p[a]p[a], otherwise, we set p[a]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 O(α(m,n))O(α(m,n)) time amortized, where α(m,n)α(m,n) is the inverse Ackermann function, mm is the number of performed operations get and nn is the number of elements.

To give the intuition on how slow the inverse Ackermann function rises, we look at O(log∗n)O(log∗⁡n) that rises a little bit faster. This function means how many times we should take the binary logarithm of nn to get a value smaller than one. Consider an example. Suppose we take a very large number — 265536265536 and calculate its log∗log∗. 265536→65536=216→16=24→4=22→2→1→0265536→65536=216→16=24→4=22→2→1→0. In total we get that log∗265536=6log∗⁡265536=6. So, we can suppose that for all reasonable nn this function is almost constant, while the inverse Ackermann function rises even slower!

**Disjoint Sets Union. Thoughts.**

It appears that one can implement get in Java and C++ in one line using the ternary conditional operator.

**get**(a):  
 **return** p[a] = (p[a] == a ? a : **get**(p[a]))

Typically, the operation union is implemented using rank heuristic instead of small-to-large one: we maintain the rank rr of sets and also link the smallest to largest. The code becomes very clean without swap operation, which is not supported by Java.

**union**(a, b):  
 a = **get**(a)  
 b = **get**(b)  
 **if** r[a] == r[b]:  
 r[a]++  
 **if** r[a] > r[b]:  
 p[b] = a  
 **else**:  
 p[a] = b

Except for maintaining the sets, we can support associative and commutative functions. An operation ⊗⊗ is associative, if its result does not depend on the order of application, i.e., (a⊗b)⊗c=a⊗(b⊗c)(a⊗b)⊗c=a⊗(b⊗c). An operation ⊗⊗ is commutative, if its result does not depend on the order of the arguments, i.e., a⊗b=b⊗aa⊗b=b⊗a. For example, we can maintain sum or minimum on elements of a set. Then, the code of union becomes the following. Of course, arrays sum and min should be properly initialized.

**union**(a, b):  
 a = **get**(a)  
 b = **get**(b)  
 **if** r[a] == r[b]:  
 r[a]++  
 **if** r[a] > r[b]:  
 p[b] = a  
 sum[a] += sum[b]  
 min[a] = min(min[a], min[b])  
 **else**:  
 p[a] = b  
 sum[b] += sum[a]  
 min[b] = min(min[a], min[b])