Sergio E. Garcia Tapia
Algorithms by Sedgewick and Wayne (4th edition) [SW11]
October 6th, 2024

1.5: Case Study: Union-find

Exercise 1. Show the contents of the `id[]` array and the number of times the array is accessed for each input pair when you use quick-find for the sequence

```
9-0 3-4 5-8 7-2 2-1 5-7 0-3 4-2
```

Solution.

`id[]` (Quick-Find)

p	q	0	1	2	3	4	5	6	7	8	9	Array Accesses
9	0	0	1	2	3	4	5	6	7	8	9	
		0	1	2	3	4	5	6	7	8	0	15
3	4	0	1	2	3	4	5	6	7	8	0	
		0	1	2	4	4	5	6	7	8	0	15
5	8	0	1	2	4	4	5	6	7	8	0	
		0	1	2	4	4	8	6	7	8	0	15
7	2	0	1	2	4	4	8	6	7	8	0	
		0	1	2	4	4	8	6	2	8	0	15
2	1	0	1	2	4	4	8	6	2	8	0	
		0	1	1	4	4	8	6	1	8	0	16
5	7	0	1	1	4	4	8	6	1	8	0	
		0	1	1	4	4	1	6	1	1	0	16
0	3	0	1	1	4	4	1	6	1	1	0	
		4	1	1	4	4	1	6	1	1	4	16
4	2	4	1	1	4	4	1	6	1	1	4	
		1	1	1	1	1	1	6	1	1	1	18

Each input sequence incurs the cost of a `connected()` operation which is two calls to `find()`, and hence 2 arrays accesses. Then in calling `union()`, we have two more array accesses since we call `find()` twice again. Then, we have at least 10 array accesses as we iterate through the `id` array. Finally, we have an extra array access for each identifier matching `pID`, the identifier of the component of the first site given.

Exercise 2. Do Exercise 1.5.1, but use quick-union (page 224). In addition, draw the forest of trees represented by the `id[]` array after each input pair is processed.

Solution.

`id[]` (Quick-Union)

p	q	0	1	2	3	4	5	6	7	8	9	Array Accesses
9	0	0	1	2	3	4	5	6	7	8	9	
		0	1	2	3	4	5	6	7	8	0	3
3	4	0	1	2	3	4	5	6	7	8	0	
		0	1	2	4	4	5	6	7	8	0	3
5	8	0	1	2	4	4	5	6	7	8	0	
		0	1	2	4	4	8	6	7	8	0	3
7	2	0	1	2	4	4	8	6	7	8	0	
		0	1	2	4	4	8	6	2	8	0	3
2	1	0	1	2	4	4	8	6	2	8	0	
		0	1	1	4	4	8	6	2	8	0	3
5	7	0	1	1	4	4	8	6	2	8	0	
		0	1	1	4	4	8	6	2	1	0	9
0	3	0	1	1	4	4	8	6	2	1	0	
		4	1	1	4	4	8	6	2	1	0	5
4	2	4	1	1	4	4	8	6	2	1	9	
		4	1	1	4	1	8	6	2	1	0	5

See Figure 1 for the forest of trees representation of `id[]`.

Exercise 3. Do Exercise 1.5.1, but use weighted quick-union (page 228).

Solution.

`id[]` Weighted Quick-Union

p	q	0	1	2	3	4	5	6	7	8	9	`id[]` Array Accesses
9	0	0	1	2	3	4	5	6	7	8	9	
		9	1	2	3	4	5	6	7	8	9	3
3	4	9	1	2	3	4	5	6	7	8	9	
		9	1	2	3	3	5	6	7	8	9	3
5	8	9	1	2	3	3	5	6	7	8	9	
		9	1	2	3	3	5	6	7	5	9	3
7	2	9	1	2	3	3	5	6	7	5	9	
		9	1	7	3	3	5	6	7	5	9	3
2	1	9	1	7	3	3	5	6	7	5	9	
		9	7	7	3	3	5	6	7	5	9	5
5	7	9	7	7	3	3	5	6	7	5	9	
		9	7	7	3	3	7	6	7	5	9	3
0	3	9	7	7	3	3	7	6	7	5	9	
		9	7	7	9	3	7	6	7	5	9	5
4	2	9	7	7	9	3	7	6	7	5	9	
		9	7	7	9	3	7	6	7	5	7	7

See Figure 2.

Exercise 4. Show the contents of the `sz[]` and `id[]` arrays and the number of array accesses for each input pair corresponding to the weighted quick-union examples in the text (both the reference input and the worst-case input).

Solution. First the reference input:

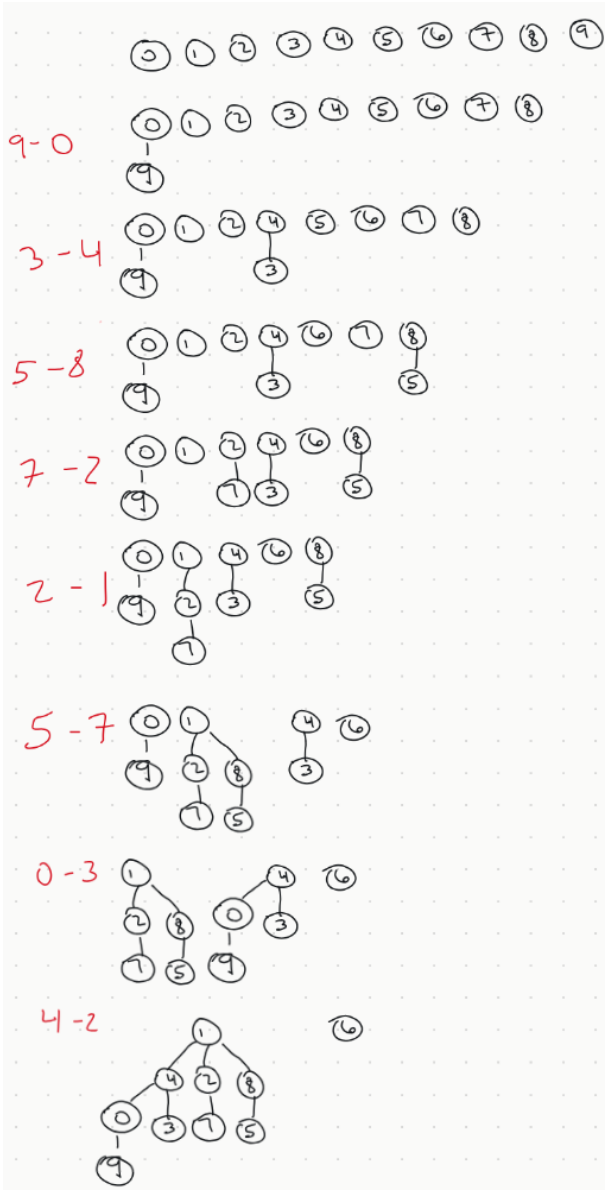

Figure 1: Forest of trees representation of `id[]` for quick-union in Exercise 2.

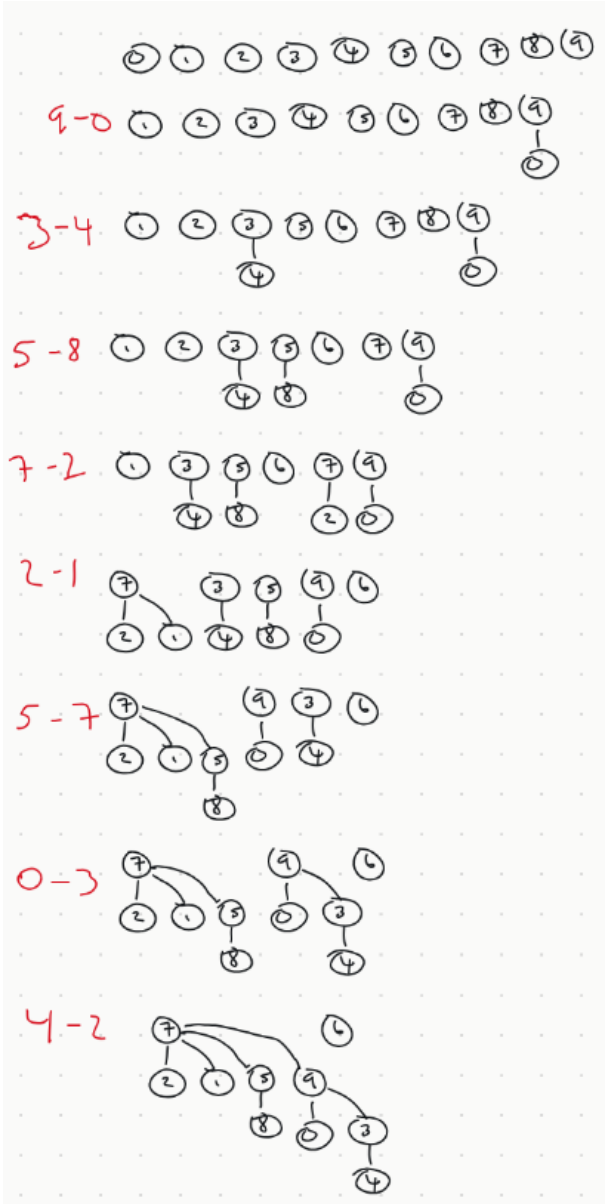

Figure 2: Forest of trees representation of `id[]` for quick-union in Exercise 2.

Weighted Quick-Union, Reference Input

		id[]											sz[]								
p	q	0	1	2	3	4	5	6	7	8	9	id[] Accesses	0	1	2	3	4	5	6	7	8	9
4	3	0	1	2	3	4	5	6	7	8	9		1	1	1	1	1	1	1	1	1	1
		0	1	2	4	4	5	6	7	8	9	3	1	1	1	1	2	1	1	1	1	1
3	8	0	1	2	4	4	5	6	7	8	9											
		0	1	2	4	4	5	6	7	4	9	5	1	1	1	1	3	1	1	1	1	1
6	5	0	1	2	4	4	5	6	7	4	9											
		0	1	2	4	4	6	6	7	4	9	3	1	1	1	1	3	1	2	1	1	1
9	4	0	1	2	4	4	6	6	7	4	9											
		0	1	2	4	4	6	6	7	4	4	3	1	1	1	1	4	1	2	1	1	1
2	1	0	1	2	4	4	6	6	7	4	4											
		0	2	2	4	4	6	6	7	4	4	3	1	1	2	1	4	1	2	1	1	1
8	9	0	2	2	4	4	6	6	7	4	4											
		0	2	2	4	4	6	6	7	4	4	6										
5	0	0	2	2	4	4	6	6	7	4	4											
		6	2	2	4	4	6	6	7	4	4	5	1	1	2	1	4	1	3	1	1	1
7	2	6	2	2	4	4	6	6	7	4	4											
		6	2	2	4	4	6	6	2	4	4	3	1	1	3	1	4	1	3	1	1	1
6	1	6	2	2	4	4	6	6	2	4	4											
		6	6	2	4	4	6	6	2	4	4	5	1	1	3	1	4	1	6	1	1	1
1	0	6	6	2	4	4	6	6	2	4	4											
		6	6	2	4	4	6	6	2	4	4	8										
6	7	6	6	2	4	4	6	6	2	4	4											
		6	6	2	4	4	6	6	2	4	4	6										

Next, the worst-case input:

Weighted Quick-Union, Worst-Case Input

		id[]									sz[]							
p	q	0	1	2	3	4	5	6	7	id[] Accesses	0	1	2	3	4	5	6	7
0	1	0	1	2	3	4	5	6	7		1	1	1	1	1	1	1	1
		0	0	2	3	4	5	6	7	3	2	1	1	1	1	1	1	1
2	3	0	0	2	3	4	5	6	7									
		0	1	2	2	4	5	6	7	3	2	1	2	1	1	1	1	1
4	5	0	0	2	2	4	5	6	7									
		0	1	2	2	4	4	6	7	3	2	1	2	1	2	1	1	1
6	7	0	0	2	2	4	4	6	7									
		0	1	2	2	4	4	6	6	3	2	1	2	1	2	1	2	1
0	2	0	0	2	2	4	4	6	6									
		0	0	0	2	4	4	6	6	3	4	1	2	1	2	1	2	1
4	6	0	0	0	2	4	4	6	6									
		0	0	0	2	4	4	4	6	3	4	1	2	1	4	1	2	1
0	4	0	0	0	2	4	4	4	6									
		0	0	0	2	0	4	4	6	3	8	1	2	1	4	1	2	1

Exercise 5. Estimate the minimum amount of time (in days) that would be required for quick-find to solve a dynamic connectivity problem with 10^9 sites and 10^6 input pairs, on

a computer capable of executing 10^9 instructions per second. Assume that each iteration of the inner `for` loop requires 10 machine instructions.

Solution. Because there are 10^6 input pairs, that means the program requires 10^6 iterations, regardless of the union-find implementation. Let $n = 10^9$. The inner loop of quick-find is a `for` loop that iterates through all $n = 10^9$ sites. By assumption, each iteration takes 10 machine instructions, so the cost of the inner for loop is 10^{10} instructions. Altogether, we need $10^6 \cdot 10^{10} = 10^{16}$ instructions in order to process all input pairs. With a computer capable of executing 10^9 instructions per second, it would take $10^{16}/10^9 = 10^7$ seconds to process all pairs. Now the total time T would be:

$$T \approx 10^7 \text{seconds} \cdot \frac{1 \text{ day}}{86400 \text{ seconds}} \approx 116 \text{ days}$$

Exercise 6. Repeat Exercise 1.5.5 for weighed quick union.

Solution. A similar analysis holds. That is, we require 10^6 iterations to process all 10^6 pairs. We still have $n = 10^9$. However, the order of growth is $\lg n \approx 23$. Assuming the rest of the information is the same:

$$\begin{aligned} T &\approx \frac{10^6 \cdot \lg(10^9) \cdot 10 \text{ instructions}}{10^9 \text{ instructions per sec}} \\ &= \frac{\lg(10^9) \text{ seconds}}{10^2} \\ &\approx 0.3 \text{ seconds} \end{aligned}$$

Exercise 7. Develop classes `QuickUnionUF` and `QuickFindUF` that implement quick-union and quick-find, respectively.

Solution. See the `com.segarciat.algs.ch1.sec5.ex07` package.

Exercise 8. Give a counterexample to show why this intuitive implementation of `union()` for quick-find is not correct:

```
public void union(int p, int q)
{
   if (id[p] == id[q]) return;

   // Rename p's component to q's name.
   for (int i = 0; i < id.length; i++)
      if (id[i] == id[p]) id[i] = id[q];
   count--;
}
```

Solution. Suppose we have three sites, numbered 0, 1, 2, and we get the following input sequence:

```
0-1
0-2
```

The end result is that all sites will be connected and there will only be one component at the end. However, the contents of the `id[]` array are as follows after the algorithm above is applied:

```
# start
0 1 2

# after 0-1
1 1 2

# after 0-2
2 1 2
```

The problem surfaces while processing input pair `0-2`. The algorithm determines that the identifiers of the components are distinct, so it proceeds to rename `p`'s component to `q`'s component. In this case, renaming `0`'s component (with identifier 1) to `2`'s component (with identifier 2). The first iteration changes `id[0]` to `2`, which is correct. In the second iteration, `id[1]` is compared against `id[0]`, and they are determined to be unequal because `id[0]` was changed in the previous iteration to `2`. Thus, `id[1]` will remain as `1`, which is incorrect.

Exercise 9. Draw the tree corresponding to the `id[]` array depicted at right. Can this be the result of running weighted quick-union? Explain why this is impossible or give a sequence of operations that results in this array.

i	0	1	2	3	4	5	6	7	8	9
id[i]	1	1	3	1	5	6	1	3	4	5

Solution. See Figure 3. The tree has 10 nodes, and its height is 4. By Proposition H in [SW11], the depth of any node in a forest built by weighted quick-union of n sites is at most $\lg n$. Letting $n = 10$, we see that $\lg n < 4$. Therefore, it's impossible for weighted quick union to have produced such an array.

Exercise 10. In the weighted quick-union algorithm, suppose that we set `id[find(p)]` to `q` instead of to `id[find(q)]`. Would the resulting algorithm be correct?

Solution. Yes because it would ensure that `p` and `q` are on the same tree and hence the same component. However, `q` may not be the root of the tree; it may be a leaf or a non-root internal node. As a result, we would not have the logarithmic performance guarantee of weighted quick union.

Exercise 11. Implemented *weighted quick-find*, where you always change the `id[]` entries of the smaller component to the identifier of the larger component. How does this affect performance?

Solution. This is not an improvement over (unweighted) quick-find. In particular, if `p` is any site in any component, then `id[p]` always evaluates to the identifier of the root

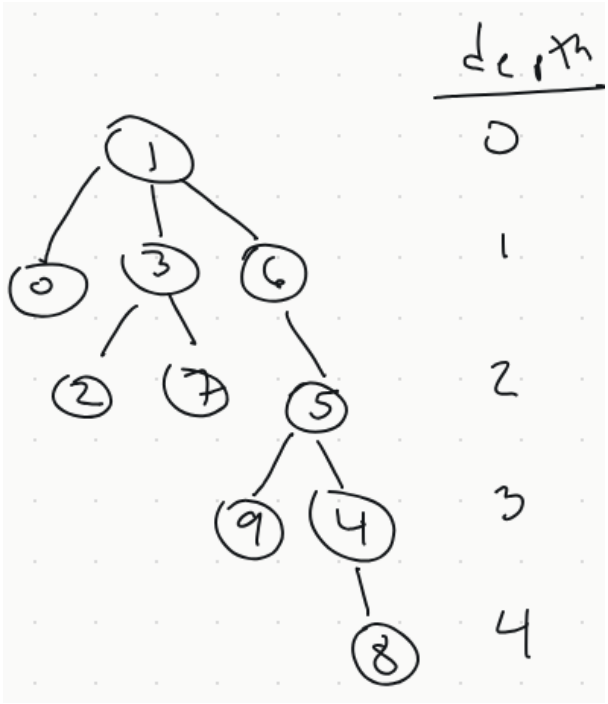

Figure 3: Tree from array in Exercise 10.

of the tree. As a result, `find()` takes constant time. However, in order to change the `id[]` entries in the smaller component to the identifier of the larger component in the `union()` implementation, we must iterate through all the sites. This is necessary because the links are unidirectional; that is, they always point towards the root. In particular, if an input pair consists of two roots, the current arrangement does not provide an easy way to find the leaves, or any of the internal non-root nodes. As a result, the performance of `union()` degrades to linear, as it was in the (unweighted) quick-find implementation. In this case, maintaining the size of the trees for each component in the `sz[]` array added complexity to our algorithm and increased our space requirement, with no payoff.

Exercise 12. *Quick-union with path compression.* Modify quick-union (page 224) to include *path compression*, by adding a loop to `find()` that links every site on the path from `p` to the root. Give a sequence of input pairs that causes this method to produce a path of length 4. *Note*: The amortized cost per operation for this algorithm is known to be logarithmic.

Solution. See the `com.segarciat.algs4.ch1.sec5.ex12.QuickUnionPC` class. The following sequence of input pairs yields a path of length 4:

```
1-0 2-3 0-2 0-4 4-5
```

In general, joining the root of a tree with another will increase the tree height. This is because if `p` is a root, then `find(p)` will only encounter `p` on the path to the root. Thus, the paths with greater depth will not be renamed. In the tree that is formed, node `3` has a depth of 4.

Exercise 13. *Weighted quick-union with path compression.* Modify weighted quick-union (Algorithm 1.5) to implement path compression, as described in Exercise 1.5.12. Give a

sequence of input pairs that causes this method to produce a tree of height 4. *Note*: The amortized cost per operation for this algorithm is known to be bounded by a function known as the *inverse Ackermann function* and is less than 5 for any conceivable value of n that arises in practice.

Solution. The following sequence of input pairs produces a tree of height 3 with `0` as the root:

```
0-1 2-3 0-2 4-5 6-7 0-4
```

The following sequence of input pairs produces another tree of height 3 with `8` as the root:

```
8-9 10-11 8-10 12-13 14-15 8-12
```

Finally, the following pair combines the trees and creates a tree of height 4 rooted at `0`:

```
0-8
```

In this tree, there are two nodes with a depth of 4: node `7` and node `15`.

Exercise 14. *Weighted quick-union by height.* Develop a `UF` implementation that uses the same basic strategy as weighted quick-union but keeps track of tree height and always links the shorter tree to the taller one. Prove a logarithmic upper bound on the height of the trees for n sites with your algorithm.

Solution. I claim when the size of every tree of n sites is at most $\lg n$. The proof is by strong induction.

Proof. If $n = 1$, then $\lg 1 = 0$, proving the base case. Suppose that $n > 1$, and whenever a tree is built out of $k < n$ sites, then its height is at most $\lg k$. If the algorithm has more than 1 component, then each tree has less than n sites, and hence, the result follows by the inductive hypothesis. Suppose, however, that there is only one component, and hence there is a single tree with n sites.

Let n_1 and n_2 be the number of sites in the two trees prior to the input pair that causes them to combine into one tree. If the two trees have different heights, then the algorithm does not increment the height of the resulting tree when the smaller one is attached to the larger one, and the result holds since in this case the height is less than $\max\{\lg(n_1), \lg(n_2)\} \leq \lg(n)$. If the trees have different heights, then the algorithm increases the height by 1. Since the height of each tree is bounded by $\lg(n_1)$ and $\lg(n_2)$, respectively, and the heights are the same, this means that both of these serve as upper bounds. Thus, if we let $N = \min\{n_1, n_2\}$, then the height of the resulting tree is bounded by $1 + \lg(N)$, and hence

$$\begin{aligned} \text{height} &\leq 1 + \lg(N) \\ &= \lg(2) + \lg(N) \\ &= \lg(2N) \\ &= \lg(N + N) \\ &\leq \lg(n_1 + n_2) \\ &= \lg(n) \end{aligned}$$

The result now holds for induction. □

Exercise 15. *Binomial trees.* Show that the number of nodes at each level in the worst-case trees for weighted quick-union are *binomial coefficients*. Compute the average depth of a node in a worst-case tree with $k = 2^n$ nodes.

Solution. Note that the level of a node is one more than the depth.

Proof. I will prove that if T is a tree of height h built by the worst-case input, then level k is $\binom{h}{k-1}$, where $0 \leq k-1 \leq h$ and h is an integer.

If there are two nodes, then the height of the tree is $h = 1$, and the tree has $\binom{h}{1-1} = \binom{1}{0} = 1$ node at level 1, and $\binom{h}{2-1} = \binom{1}{1}$ nod at level 2. Thus the base case holds.

Proceeding by induction, suppose that the algorithm has continued to process worst-case input and that the height of every tree of height $m < h$ has the property that level k has $\binom{m}{k-1}$ nodes, where $h > 1$ and $0 \leq k-1 \leq m$. Under the assumption of worst-case input, the algorithm will only combine two trees that of the same size.

Suppose T_1 and T_2 are two trees of height m that are combined into a tree T of height h. Note that $m+1 = h$ Without loss of generality, suppose that T_2 is joined to T_1. Then the root of T is the root of T_1, and the nodes in T_1 retain their level in the new tree T. However, a node at level i in T_2 moves to level $k+1$ in T when T_2 is joined to T_1. As a result, the number of nodes at level k in T is given by:

$$\#\text{ of nodes at level } k = \binom{m}{k-1} + \binom{m}{(k-1)-1}, \quad 1 \leq k-1 \leq m$$

for all levels k greater than 1 (not the root). The first term is the number of nodes at level k as a result of T_1, whose place in the resulting tree T^* is unchanged because T_2 is joined to T_1. The second term follows from the fact that every node in T_2 has moved down a level.

Pascal's triangle says that

$$\binom{h}{k-1} = \binom{m+1}{k-1} = \binom{m}{k-1} + \binom{m}{(k-1)-1}, \quad 1 \leq k-1 \leq m$$

The resulting formula can be extended to $0 \leq k-1 \leq m+1$ because at level 1 we only have the root of T_1, so we have $\binom{h}{1-1} = \binom{m+1}{0} = 1$, and at level $h+1$ we only have the single node that is at level m in tree T_2, consistent with $\binom{h}{(h+1)-1} = \binom{h}{h} = 1$. □

If a worst-case tree has $k = 2^n$ nodes, then the height of such a tree is $\lg(k) = \lg(2^n) = n$, so the average depth of any node is

$$\frac{\sum_{j=0}^{n} \binom{n}{j} \cdot j}{2^n} = \frac{n \cdot 2^{n-1}}{n} = \frac{n}{2}$$

where I have used the formula $\sum_{j=0}^{n} \binom{n}{j} \cdot j = n \cdot 2^{n-1}$; see Pedro's proof on StackExchange.

Exercise 16. *Amortized cost plots.* Instrument your implementations from Exercise 1.5.7 to make amortized cost plots like those in the text.

Solution. See the classes in the `com.segarciat.algs4.ch1.sec5.ex16` package.

Exercise 17. *Random connections.* Develop a `UF` client `ErdosRenyi` that takes an integer value `n` from the command line, generates random pairs of integers between `0` and `n-1`, calling `connected()` to determine if they are connected and then `union()` if not (as in our development client), looping until all sites are connected, and printing the number of connections generated. Package your program as a static method `count()` that takes `n` as an argument and returns the number of connections and a `main()` that takes `n` from the command line, calls `count`, and prints the returned value.

Solution. See the `com.segarciat.algs4.ch1.sec5.ex17.ErdosRenyi` class.

Exercise 18. *Random grid generator.* Write a program `RandomGrid` that takes an `int` value `n` from the command line, generates all the connections in an `n`-by-`n` grid, puts them in random order, randomly orients them (so that `p` and `q` and `q` and `p` are equally likely to occur), and prints the result to standard output. To randomly order the connections, use a `RandomBag` (see Exercise 1.3.34 on page 167). To encapsulate `p` and `q` in a single object, use the `Connection` nested class shown below. Package your program as two static methods: `generate()`, which takes `n` as argument and returns an array of connections, and `main()`, which takes `n` from the command line, calls `generate()`, and iterates through the returned array to print the connections.

Solution. See the `com.segarciat.algs4.ch1.sec5.ex18.RandomGrid` class.

Exercise 19. *Animation.* Write a `RandomGrid` client (see Exercise 1.5.18) that uses `UF` as in our development client to check connectivity and uses `StdDraw` tod raw the connections as they are processed.

Solution. See the `com.segarciat.algs4.ch1.sec5.ex19.RandomGridAnimation` class.

Exercise 20. *Dynamic growth.* Using linked lists or a resizing array, develop a weighted quick-union implementation that removes the restriction on needing the number of objects ahead of time. Add a method `newSite()` to the API, which returns an `int` identifier.

Solution. See the `com.segarciat.algs4.ch1.sec5.ex20.ResizingArrayUF` class.

Exercise 1.5.21. *Erdös-Renyi Model.* Use your client from Exercise 1.5.17 to test the hypothesis that the number of pairs generated to get one component is $\sim \frac{1}{2} n \ln n$.

Solution. See the class `com.segarciat.algs4.ch1.sec5.ex21.SingleUFComponent`.

Exercise 1.5.22. *Doubling test for Erdos-Renyi model.* Develop a performance-testing client that takes an `int` value `T` from the command line and performs `T` trials of the following experiment: Use your client from Exercise 1.5.17 to generate random connections, using `UF` to determine connectivity as in our development client, looping until all sites are connected. For each `n`, print the value of `n`, the average number of connections processed, and the ratio of the running time to the previous. Use your program to validate the hypothesis in the text that the running times for quick-find and quick-union are quadratic and weighted quick-union is near-linear.

Solution. See the class `com.segarciat.algs4.ch1.sec5.ex22.DoublingTestErdosRenyi`.

References

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms*. 4th ed. Addison-Wesley, 2011. ISBN: 9780321573513.