EXERCISES

1.5.1 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`.

1.5.2 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.

1.5.3 Do EXERCISE 1.5.1, but use weighted quick-union (page 228).

1.5.4 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).

1.5.5 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.

1.5.6 Repeat EXERCISE 1.5.5 for weighted quick-union.

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

1.5.8 Give a counterexample that shows why this intuitive implementation of `union()` for quick-find is not correct:

```
public void union(int p, int q)
{
   if (connected(p, 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--;
}
```

1.5.9 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

EXERCISES *(continued)*

1.5.10 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?

Answer: Yes, but it would increase the tree height, so the performance guarantee would be invalid.

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

CREATIVE PROBLEMS

1.5.12 *Quick-union with path compression.* Modify quick-union (page 224) to include *path compression*, by adding a loop to `union()` that links every site on the paths from `p` and `q` to the roots of their trees to the root of the new tree. 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.

1.5.13 *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 practical value of N.

1.5.14 *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.

1.5.15 *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 $N = 2^n$ nodes.

1.5.16 *Amortized costs plots.* Instrument your implementations from EXERCISE 1.5.7 to make amortized costs plots like those in the text.

1.5.17 *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 argument and returns the number of connections and a `main()` that takes `N` from the command line, calls `count()`, and prints the returned value.

1.5.18 *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 q` and `q 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,

CREATIVE PROBLEMS *(continued)*

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.

1.5.19 *Animation.* Write a `RandomGrid` client (see EXERCISE 1.5.18) that uses `UnionFind` as in our development client to check connectivity and uses `StdDraw` to draw the connections as they are processed.

1.5.20 *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.

```
private class Connection
{
   int p;
   int q;

   public Connection(int p, int q)
   {  this.p = p; this.q = q;  }
}
```

Record to encapsulate connections

EXPERIMENTS

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 ~ ½$N \ln N$.

1.5.22 *Doubling test for Erdös-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 `UnionFind` 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 hypotheses in the text that the running times for quick-find and quick-union are quadratic and weighted quick-union is near-linear.

1.5.23 *Compare quick-find with quick-union for Erdös-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. Save the connections, so that you can use both quick-union and quick-find to determine connectivity as in our development client, looping until all sites are connected. For each `N`, print the value of `N` and the ratio of the two running times.

1.5.24 *Fast algorithms for Erdös-Renyi model.* Add weighted quick-union and weighted quick-union with path compression to your tests from EXERCISE 1.5.23 . Can you discern a difference between these two algorithms?

1.5.25 *Doubling test for random grids.* Develop a performance-testing client that takes an `int` value `T` from the command line and performs `T` trials of the following experiement: Use your client from EXERCISE 1.5.18 to generate the connections in an `N`-by-`N` square grid, randomly oriented and in random order, then use `UnionFind` 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 hypotheses in the text that the running times for quick-find and quick-union are quadratic and weighted quick-union is near-linear. *Note*: As `N` doubles, the number of sites in the grid increases by a factor of 4, so expect a doubling factor of 16 for quadratic and 4 for linear.

EXPERIMENTS *(continued)*

1.5.26 *Amortized plot for Erdös-Renyi.* Develop a client that takes an `int` value `N` from the command line and does an amortized plot of the cost of all operations in the style of the plots in the text for the process of generating 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.