

## 1 Disjoint Sets, a.k.a. Union Find

In lecture, we discussed the Disjoint Sets ADT. Some authors call this the Union Find ADT. Today, we will use union find terminology so that you have seen both.

- (a) What are the last two improvements (out of four) that we made to our naive implementation of the Union Find ADT during lecture 14 (Monday's lecture)?

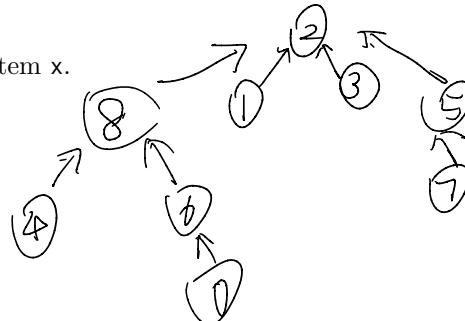
1. Improvement 1: *keeping track of sets rather than connections, tracking set membership by recording parent not set.*
2. Improvement 2: *Union by size, path compression*

- (b) Assume we have nine items, represented by integers 0 through 8. All items are initially unconnected to each other. Draw the union find tree, draw its array representation after the series of `connect()` and `find()` operations, and write down the result of `find()` operations using **WeightedQuickUnion** without path compression. Break ties by choosing the smaller integer to be the root.

Note: `find(x)` returns the root of the tree for item `x`.

```
connect(2, 3);
connect(1, 2);
connect(5, 7);
connect(8, 4);
connect(7, 2);
find(3);    2
connect(0, 6);
connect(6, 4);
connect(6, 3);
find(8);    2
find(6);    2
```

$\{6, 2, -9, 2, 8, 2, 8, 5, 2\}$



*(e) Extra:* Repeat the above part, using **WeightedQuickUnion with Path Compression**.

- (d) What is the runtime for "connect" and "isConnected" operations using our Quick Find, Quick Union, and Weighted Quick Union ADTs? Can you explain why the Weighted Quick Union has better runtimes for these operations than the regular Quick Union?

| OPERATION  | Quick Find | Quick Union | WQU         |
|------------|------------|-------------|-------------|
| Connect    | $O(N)$     | $O(N)$      | $O(\log N)$ |
| Is Connect | $O(1)$     | $O(N)$      | $O(\log N)$ |

The Weighted Quick Union has better run times because by picking the smaller tree to be the child, we can achieve shorter overall heights in our underlying tree. This means that for any child, traversing up the tree to find its root, or its set representative, is limited to this shortened tree height. For both our standard Quick Union and Weighted Quick Union, the time it takes to connect two items depends on this height, as it requires checking the roots of the current items and then changing one to be the other (if they're not already connected). Then the time it takes to find the root of current element is proportional to the time it takes to connect two items. Similarly, the time it takes to check if two items are connected relies on finding the roots of the current elements.

Not included in this chart is the WQU with path compression. While the proof for its runtime is out of scope for this class, it achieves amortized constant runtime for both of Connect and isConnected.

## 2 Asymptotics

- (a) Order the following big- $O$  runtimes from smallest to largest.

$$\begin{aligned} O(\log n), \underline{O(1)}, O(n^n), O(n^3), O(n \log n), O(n), O(n!), O(2^n), O(n^2 \log n) \\ O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2 \log n) \subset O(n^3) \subset O(2^n) \subset O(n!) \subset O(n^n) \end{aligned}$$

- (b) Are the statements in the right column true or false? If false, correct the asymptotic notation ( $\Omega(\cdot)$ ,  $\Theta(\cdot)$ ,  $O(\cdot)$ ). Be sure to give the tightest bound.  $\Omega(\cdot)$  is the opposite of  $O(\cdot)$ , i.e.  $f(n) \in \Omega(g(n)) \iff g(n) \in O(f(n))$ .

*Hint: Make sure to simplify the runtimes first.*

|                                |                           |                         |
|--------------------------------|---------------------------|-------------------------|
| i) $f(n) = 20501$              | $g(n) = 1$                | $f(n) \in O(g(n))$      |
| ii) $f(n) = n^2 + n$           | $g(n) = 0.000001n^3$      | $f(n) \in \Omega(g(n))$ |
| iii) $f(n) = 2^{2n} + 1000$    | $g(n) = 4^n + n^{100}$    | $f(n) \in O(g(n))$      |
| iv) $f(n) = \log(n^{100})$     | $g(n) = n \log n$         | $f(n) \in \Theta(g(n))$ |
| v) $f(n) = n \log n + 3^n + n$ | $g(n) = n^2 + n + \log n$ | $f(n) \in \Omega(g(n))$ |
| vi) $f(n) = n \log n + n^2$    | $g(n) = \log n + n^2$     | $f(n) \in \Theta(g(n))$ |
| vii) $f(n) = n \log n$         | $g(n) = (\log n)^2$       | $f(n) \in O(g(n))$      |

- i) True, this is an accurate tightest bound.  
 ii) True, this is an accurate tightest bound.  
 iii) True, this is an accurate tightest bound.  
 iv) True, this is an accurate tightest bound.  
 v) True, this is an accurate tightest bound.  
 vi) True, this is an accurate tightest bound.  
 vii) True, this is an accurate tightest bound.
- False, an accurate tightest bound would be  $\underline{\Omega(1)}$   
 False, an accurate tightest bound would be  $\underline{\Omega(1)}$

- (c) Give the worst case and best case runtime in terms of  $M$  and  $N$ . Assume ping is in  $\Theta(1)$  and returns an **int**.

```

1  for (int i = N; i > 0; i--) {
2      for (int j = 0; j <= M; j++) {
3          if (ping(i, j) > 64) break;
4      }
5  } 
```

Worst:  $\Theta(MN)$  Best:  $\Theta(N)$

We repeat the outer loop  $N$  times, no matter what. For the inner loop, we see the amount of times we repeat it depends on the result of ping. In the best case, it returns true immediately, such that we'll only ever look at the inner loop once and then break the inner loop. In the worst case, ping is always false and we complete the inner loop  $M$  times for every value of  $N$  in the outer loop.

- (d) Below we have a function that returns true if every int has a duplicate in the array, and false if there is any unique int in the array. Assume  $\text{sort}(\text{array})$  is in  $\Theta(N \log N)$  and returns array sorted.

```

1  public static boolean noUniques(int[] array) {
2      array = sort(array);
3      int N = array.length;
4      for (int i = 0; i < N; i += 1) {
5          boolean hasDuplicate = false;
6          for (int j = 0; j < N; j += 1) {
7              if (i != j && array[i] == array[j]) {
8                  hasDuplicate = true;
9              }
10         }
11         if (!hasDuplicate) return false;
12     }
13     return true;
14 }
```

*never forget it*

1. Give the worst case and best case runtime where  $N = \text{array.length}$ .

Its runtime is  $\Theta(N \log N + N^2) = \Theta(N^2)$  for the worst case the if statement always sets  $x$  to true. The best case is if we don't set  $x$  to be true in the very first loop, which allows us to only go through the entire array once giving us  $\Theta(N \log N + N) = \Theta(N \log N)$

2. Try to come up with a way to implement `noUniques()` that runs in  $\Theta(N \log N)$  time. Can we get any faster?

```

public static boolean noUniques(int[] array) {
    array = sort(array);
    int N = array.length;
    int curr = array[0];
    boolean unique = true;
    for (int i = 1; i < N; i += 1) {
        if (curr == array[i]) {
            unique = false;
        } else if (!unique) {
            return false;
        } else {
            unique = true;
            curr = array[i];
        }
    }
    return !unique;
}
```