CS 61B
Spring 2021

Disjoint Sets and Asymptotics

Discussion 6: February 22, 2021

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: ______________________________

2. Improvement 2: ______________________________

The naive implementation was maintaining a record of every single connection. Improvements made were:

```
- Keeping track of sets rather than connections (QuickFind)
- Tracking set membership by recording parent not set # (QuickUnion)
- Union by Size (WeightedQuickUnion)
- Path Compression (WeightedQuickUnionWithPathCompression)
```

We will focus on attention on the last two, union by size and path compression.

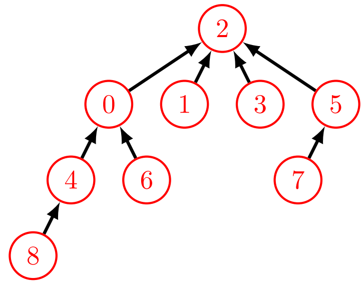
(b) Assume we have nine items, represented by integers 0 through 8. All items are initially unconnected to each other. Draw the union find tree, draw its array representation after the series of `connect()` and `find()` operations, and write down the result of `find()` operations using **WeightedQuickUnion** 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);
connect(0, 6);
connect(6, 4);
connect(6, 3);
find(8);
find(6);
```

`find()` returns 2, 2, 2 respectively.

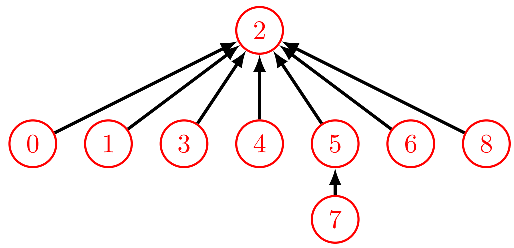
The array is [2, 2, -9, 2, 0, 2, 0, 5, 4].



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

`find()` returns 2, 2, 2 respectively.

The array is [2, 2, -9, 2, 2, 2, 2, 5, 2].



(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?

Runtime comparisons			
OPERATION	Quick Find	Quick Union	WQU
Connect	O(N)	O(N)	O(logN)
IsConnected	O(1)	O(N)	O(logN)

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 it's root, or it's 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 the 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 it's 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.

$$O(\log n), O(1), O(n^n), O(n^3), O(n \log n), O(n), O(n!), O(2^n), O(n^2 \log n)$$

$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2 \log n) \subset O(n^3) \subset O(2^n) \subset O(n!) \subset O(n^n)$

(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.*

$f(n) = 20501$	$g(n) = 1$	$f(n) \in O(g(n))$
$f(n) = n^2 + n$	$g(n) = 0.000001n^3$	$f(n) \in \Omega(g(n))$
$f(n) = 2^{2n} + 1000$	$g(n) = 4^n + n^{100}$	$f(n) \in O(g(n))$
$f(n) = \log(n^{100})$	$g(n) = n \log n$	$f(n) \in \Theta(g(n))$
$f(n) = n \log n + 3^n + n$	$g(n) = n^2 + n + \log n$	$f(n) \in \Omega(g(n))$
$f(n) = n \log n + n^2$	$g(n) = \log n + n^2$	$f(n) \in \Theta(g(n))$
$f(n) = n \log n$	$g(n) = (\log n)^2$	$f(n) \in O(g(n))$

i) False. Although this bound is technically correct, it is NOT the tightest bound. $\Theta(\cdot)$ is a better bound.
ii) False, $O(\cdot)$. Even though n^3 is strictly worse than n^2, n^2 is still in $O(n^3)$ because n^2 is always as good as or better than n^3 and can never be worse.
iii) False. Again, technically correct, but it is not a tight bound. $\Theta(\cdot)$ is a better bound.
iv) False, $O(\cdot)$.
v) True.
vi) True.
vii) False, $\Omega(\cdot)$.

(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**.

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

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 `sort(array)` is in $\Theta(N \log N)$ and returns `array` sorted.

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

1. Give the worst case and best case runtime where $N =$ `array.length`.

   Its runtime is $\Theta(NlogN + N^2) = \Theta(N^2)$ for the worst case the if statement always sets x to true. The best case is if we 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 $\Theta(NlogN + N) = \Theta(NlogN)$.

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

   We should rely on the fact that a sorted array means all duplicates will be adjacent. `curr` represents the current item we are checking, and we check the item after curr (since our array is sorted) to see if a duplicate exists. There is a possible $\Theta(N)$ solution, but that involves data structures we haven't covered yet!

```
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;
}
```