CS 61B Spring 2019 — Disjoint Sets and Asymptotics

Discussion 6: February 25, 2019

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.

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

(a) Improvement 1: **union by size**

(b) Improvement 2: **path compression**

The naive implementation was maintaining a `List<Set<Integer>>`. 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)
```

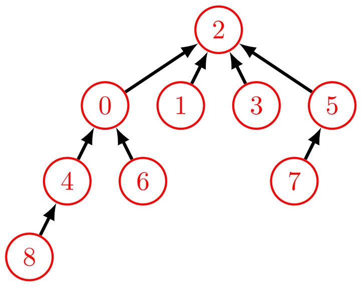
We will focus on attention on the last two, union by size and path compression.

1.2 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 `union()` and `find()` operations, and write down the result of `find()` operations using **only improvement 1**. Break ties by choosing the smaller integer to be the root.

Note: `union` is the same as the `connect` operation from lecture. `find(x)` returns the root of the tree for item `x`.

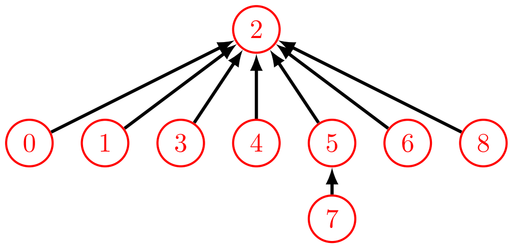
```
union(2, 3);
union(1, 2);
union(5, 7);
union(8, 4);
union(7, 2);
find(3);
union(0, 6);
union(6, 4);
union(6, 3);
find(8);
find(6);
```

`find()` returns 2, 2, 2 respectively.
The array is [2, 2, -9, 2, 0, 2, 0, 5, 4].



1.3 Repeat the above part, using **both improvement 1 and 2**.

`find()` returns 2, 2, 2 respectively.
The array is [2, 2, -9, 2, 2, 2, 2, 5, 2].



2 Asymptotics

$\boxed{2.1}$ Order the following big-O runtimes from smallest to largest.

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

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

$\boxed{2.2}$ 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. $f(n) \in \Omega(g(n)) \iff g(n) \in O(f(n))$.

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

- True, although $\Theta(\cdot)$ is a better bound.
- 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.
- True, although $\Theta(\cdot)$ is a better bound.
- False, $O(\cdot)$.
- True.
- True.
- False, $\Omega(\cdot)$.

$\boxed{2.3}$ Give the worst case and best case runtime in terms of M and N. Assume `ping` is in $\Theta(1)$ and returns an **`int`**.

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

Worst: $\Theta(M + N)$, Best: $\Theta(N)$ The trick is that `j` is initialized outside the loops!

2.4 Give the worst case and best case runtime where N = `array.length`. Assume `mrpoolsort(array)` is in $\Theta(N \log N)$ and returns `array` sorted.

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

Worst: $\Theta(N^2)$, Best: $\Theta(N \log N)$ Remember sorting in the beginning!

(a) What is `mystery()` doing?

`mystery()` returns true if every **`int`** has a duplicate in the array (ex. `{1, 2, 1, 2}`) and false if there is any unique **`int`** in the array (ex. `{1, 2, 2}`).

(b) Using an ADT, describe how to implement `mystery()` with a better runtime. Then, if we make the assumption an **`int`** can appear in the `array` at most twice, develop a solution using only constant memory.

Note: ADT's weren't covered in lecture yet at this time, so don't worry if you are having trouble remembering them!

A $\Theta(N)$ algorithm is to use a map and do $key = element$ and $value = number$ of appearances, then make sure all values are > 1. Uses $O(N)$ memory however. Can do constant space by sorting then going through, but sorting is generally in $O(n \log n)$ time.