CS 61B
Fall 2025

Asymptotics and Disjoint Sets

Discussion 05: September 29, 2025

1 xStep

Parts of this problem were inspired by course material for Princeton's introductory data structures and algorithms course, COS226.

Analyze the following loops and determine the asymptotic runtime of each using big Theta notation with respect to `N`. Assume that `System.out.println(...)` runs in constant time.

```
for (int x = 7; x < N + 14; x++) {
  System.out.println("tidal wave");
}
```

Runtime: $\Theta(\ N\ )$

```
for (int x = 1; x < N; x++) {
  for (int y = 1; y < N; y++) {
    System.out.println("amethyst");
  }
}
```

Runtime: $\Theta(\ N^2\ )$

```
for (int x = 1; x < N; x *= 2) {
  for (int y = 1; y < N; y++) {
    System.out.println("flamewall");
  }
}
```

Runtime: $\Theta(\ N \log N\ )$

```
for (int x = 2; x < N; x += 2) {
  for (int y = 1; y < 1000000; y) {
    System.out.println("anathema");
  }
}
```

Runtime: $\Theta(\ N\ )$

```
for (int x = 3; x < N * N * N; x *= 3) {
  System.out.println("nullscapes");
}
```

Runtime: $\Theta(\ \log N^3\ )$

```
for (int x = 6; x < N; x += 6) {
  for (int y = x; y < N; y += 6) {
    System.out.println("grief");
  }
}
```

Runtime: $\Theta(\ N^2\ )$

2 Disjoint Sets

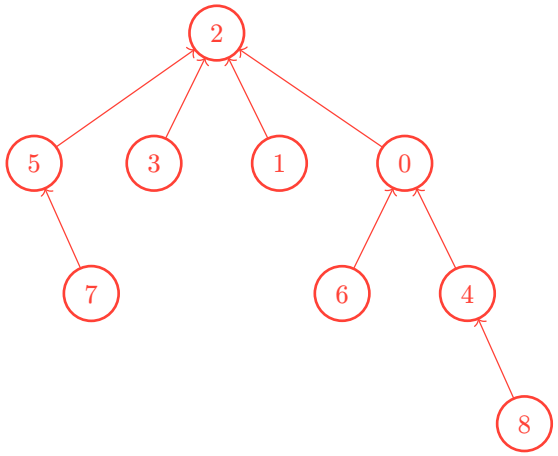
In lecture, we discussed the Disjoint Sets ADT. Some authors call this the Union Find ADT. Today, we will use union find terminology so that you have seen both.

(a) 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);
```

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



A walkthrough of how we arrive at this result can be found on the website, linked here.

Below is an implementation of the **find** function for a Disjoint Set. Given an integer **val**, **find(val)** returns the root value of the set **val** is in. The helper method **parent(int val)** returns the direct parent of **val** in the Disjoint Set representation. Assume that this implementation only uses **QuickUnion**.

```
public int find(int val) {
    int p = parent(val);
    if (p == -1) {
        return val;
    } else {
        int root = find(p);
        return root;
    }
}
```

(b) If **N** is the number of nodes in the set, what is the runtime of **find** in the worst case? Draw out the structure of the Disjoint Set representation for this worst case.

$\Theta(N)$
The worst case would occur if we have to traverse up $N-1$ nodes to find the root set representative as shown below for **find(0)**. Suppose we started out with elements 0, 1, 2, and 3. Consider the following Disjoint Set:

index	0	1	2	3
parent	1	2	3	−1

The worst case runtime of **find** is $\Theta(N)$, for **find(0)**. Since this implementation does not use WeightQuickUnion, this could potentially arise if we unioned 0 to 1, setting 1 as the root, then unioning 1 to 2, setting 2 as the root, and finally unioning 2 to 3, setting 3 as the root (try drawing this out for yourself!). WQU solves this "spindly set" problem by ensuring that the smaller set is merged into the larger one, so when we try unioning 1 to 2, 1 must be the root and not the 2.

Note that this function also does not implement path compression, making the disjoint set more susceptible to worst cases like this.

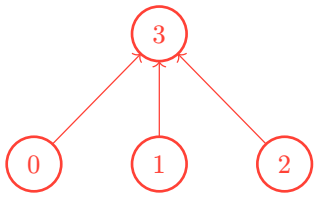
(c) Using a function **setParent(int val, int newParent)**, which updates the value of **val**'s parent to **newParent**, modify **find** to achieve a faster runtime using path compression. You may add at most one line to the provided implementation.

Solution:

```
public int find(int val) {
    int p = parent(val);
    if (p == -1) {
        return val;
    } else {
        int root = find(p);
        setParent(val, root); // sets the val's parent to be the root of the set.
        return root;
    }
}
```

Although our worst case would still be $\Theta(N)$ runtime as in the call to `find(0)` above. However, after one call to `find(0)`, the structure of the disjoint set would change so subsequent calls to `find` would be completed in amortized $O(\log^*(N))$.

Here's the structure of the set after one call to `find(0)`:

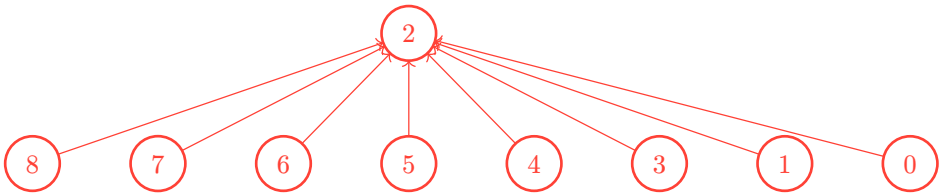


index	0	1	2	3
parent	3	3	3	−1

(d) *Extra Practice:* Draw out the tree and array representation for the following WeightedQuickUnion with path compression that has 9 elements from 0 to 8. Break ties by choosing the smaller integer to be root.

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

[2, 2, −9, 2, 2, 2, 2, 2, 2]



3 Algorithm Analysis

(a) Say we have a function **findMax** that iterates through an unsorted int array one time and returns the maximum element found in that array. Give the tightest lower and upper bounds ($\Omega(\cdot)$ and $O(\cdot)$) of **findMax** in terms of **N**, the length of the array. Is it possible to define a $\Theta(\cdot)$ bound for **findMax**?

Because the array is unsorted, we don't know where the max will be, so we have to iterate through the entire array to ensure that we find the true max. Therefore, we know that we can never go faster than linear time with respect to the length of the array. Since the function is both lower and upper bounded by N, we can say that the function is theta-bounded by N as well ($\Theta(N)$).

(b) Give the worst case and best case runtime in terms of M and N. Assume **ping** runs 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 case runtime: N	Best case runtime: N

We repeat the outer loop N times, no matter what. For the inner loop, 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.

(c) 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;
}
```

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

Worst case runtime: N	Best case runtime: N

Notice that we call `sort` at the beginning of the function, which we are told runs in $\Theta(N \log N)$.

First, we consider the best case. We notice that if `hasDuplicate` is false after the inner loop (i.e. `! hasDuplicate` has truth value `true`) we can exit the `for` loop early via the return statement on line 11. Thus, the best case is when we never set `hasDuplicate` to be `true` during the first time we run the inner loop. In this case, we can return after only looping through the array once, giving us $\Theta(N \log N + N) = \Theta(N \log N)$.

For the worst case, we notice that if `hasDuplicate` is always set to `true` by the inner loop, we never return on line 11. Thus, we consider the worst case where `hasDuplicate` is always set to `true` in every loop, forcing us to have to loop fully through both the inner and outer loop. One such input is an array of all the same integer! Since we have to fully loop through both loops, our worst-case runtime is $\Theta(N \log N + N^2) = \Theta(N^2)$.