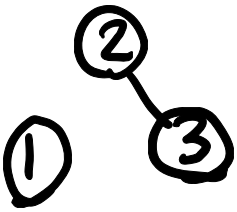
CS 61BL Summer 2021 — Disjoint Sets and Asymptotics

Recurring Section 6: Thursday July 15, 2021

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

In lab, we discussed the Union Find ADT. Today, we will use union find terminology so that you have seen both.

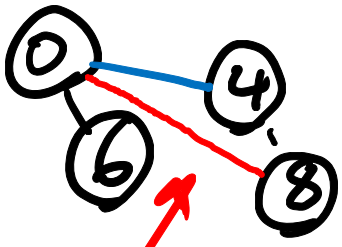
(a) Assume we have nine items, represented by integers 0 through 8. All items are initially unconnected to each other. Draw the union find tree, draw its array representation after the series of `connect()` and `find()` operations, and write down the result of `find()` operations using **WeightedQuickUnion**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

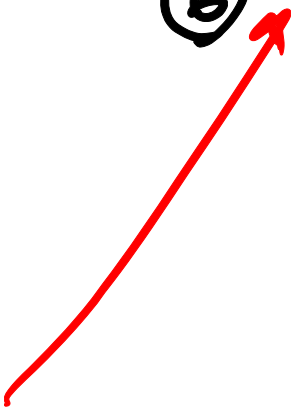
```
connect(2, 3);
connect(1, 2);
connect(5, 7);
connect(8, 4);
connect(7, 2);
find(3);
connect(0, 6);
connect(6, 4);
connect(6, 3);
find(8);
find(6);
```

0	1	2	3	4	5	6	7	8
2	2	-9	2	0	2	0	5	4



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

0	1	2	3	4	5	6	7	8
2	2	-9	2	2	2	2	2	2



(c) 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?

0 1 2 3 4 5
1 5 5 5 5 5

0 1 2 3 4 5
1 1 1 1 1 1
5 5 5 5 5 5

Quick Find:
- connect: N
- isConnected: constant

Quick Union:
- connect: O(N)
- isConnected: O(N)

Weighted Quick Union:
- connect: O(logN)
- isConnected: O(logN)

2 Asymptotics

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

2 1 9 6 4 3 8 7 5

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

(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.*

$f(n) = 20501$	$g(n) = 1$	$f(n) \in O(g(n))$	False - big theta is better
$f(n) = n^2 + n$	$g(n) = 0.000001n^3$	$f(n) \in \Omega(g(n))$	False - not true
$f(n) = 2^{2n} + 1000$	$g(n) = 4^n + n^{100}$	$f(n) \in O(g(n))$	False - big theta
$f(n) = \log(n^{100})$	$g(n) = n \log n$	$f(n) \in \Theta(g(n))$	False - not true
$f(n) = n \log n + 3^n + n$	$g(n) = n^2 + n + \log n$	$f(n) \in \Omega(g(n))$	True
$f(n) = n \log n + n^2$	$g(n) = \log n + n^2$	$f(n) \in \Theta(g(n))$	True

(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  }
```

ping could always be returning something greater than 64 i.e. 65

Worst: Big Theta(MN)
Best: Big Theta(N)

(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.

```
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 }
```

5 6 4 9 2 1 9 8 7

NlogN

1 2 4 5 6 7 8 9 9

Best Case: NlogN + N = NlogN
Worst Case: NlogN + N^2 = N^2

both in big theta

1. Give the worst case and best case runtime where $N =$ `array.length`.
2. Try to come up with a way to implement `noUniques()` that runs in $\Theta(NlogN)$ time. Can we get any faster?

Key Insight: sorted arrays are easier to check! (see example in upper right)

```
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;
}
```

3 Extra: Finish the Runtimes

Below we see the standard nested for loop, but with missing pieces!

```
for (int i = 1; i < ______; i = ______) {
    for (int j = 1; j < ______; j = ______) {
        System.out.println("We will miss you next semester Akshit :(");
    }
}
```

For each part below, **some** of the blanks will be filled in, and a desired runtime will be given. Fill in the remaining blanks to achieve the desired runtime! There may be more than one correct answer.

Hint: You may find `Math.pow` helpful.

(a) Desired runtime: $\Theta(N^2)$

```
for (int i = 1; i < N; i = i + 1) {
    for (int j = 1; j < i; j = ____) {
        System.out.println("This is one is low key hard");
    }
}
```

j+1

1+2+3+...+N → N^2 ↳ arithmetic series

(b) Desired runtime: $\Theta(log(N))$

log N

```
for (int i = 1; i < N; i = i * 2) {
    for (int j = 1; j < ____; j = j * 2) {
        System.out.println("This is one is mid key hard");
    }
}
```

2

(c) Desired runtime: $\Theta(2^N)$

```
for (int i = 1; i < N; i = i + 1) {
    for (int j = 1; j < ____; j = j + 1) {
        System.out.println("This is one is high key hard");
    }
}
```

2^i ← not java syntax

...2^N ↳ geometric series

(d) Desired runtime: $\Theta(N^3)$

```
for (int i = 1; i < ____; i = i * 2) {
    for (int j = 1; j < N * N; j = ____) {
        System.out.println("yikes");
    }
}
```

2^N

N

j+1

N^2