# Asymptotics & Disjoint Sets

## Exam-Level 05

# Announcements

| Sunday | Monday | Tuesday | Wednesday | Thursday | Friday | Saturday |
|---|---|---|---|---|---|---|
| | | 2/20<br>Lab 4 Due<br>Project 1C Due | | | | |
| | 2/26<br>Lab 5 Due<br>Homework 2 Due | | | | | |

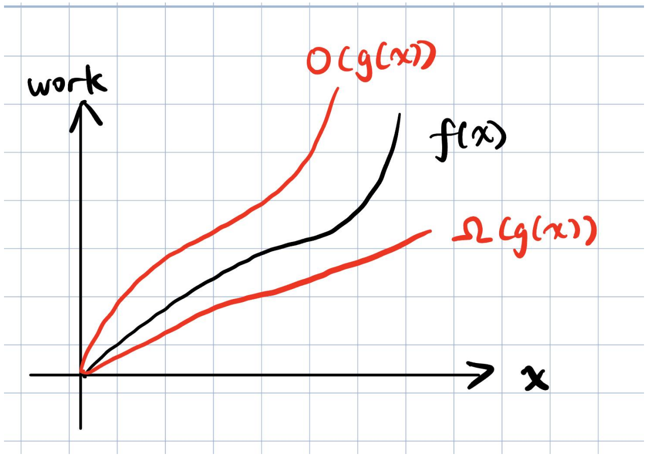
# Content Review

# Asymptotics

**Asymptotics** allow us to evaluate the performance of programs using math.

<u>We ignore all constants and only care about the total work done when the input is very large.</u>

Big O - If a function f(x) has big O in g(x), it grows at most as fast as g(x).
Big Ω - f(x) grows at least as fast as g(x),
Big Θ - When a function is both O(g(x)) and Ω(g(x)), it is Θ(g(x))




Industry note: usually Big O is used.

Big Omega is often useful for proving what you have is best.

# Common Orders of Growth

- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(c^n)$
- Alternatively: constant < logarithmic < linear < nlogn < quadratic < exponential
    - In practice, constant ~ logarithmic << linear ~ nlogn << quadratic/polynomial << exponential
- [Desmos example here]
    - Constants don't matter in the long run!
    - In industry, *sometimes* constants matter, though not always.
- Fun sums:

$$1 + 2 + 3 + \ldots + N = \Theta(N^2)$$
$$1 + 2 + 4 + 8 + \ldots + N = \Theta(N)$$

# Tightest Bound?

- Sometimes it's easier to bound the runtime than to calculate the runtime.
- When you bound, always provide the tightest bound
  - i.e, the bound that provides the most information about the runtime.
- Ex. Given $f(n) = 2n + 5$, we could say that $f(n) \in O(n^n)$, but that doesn't tell us very much
  - a lot of functions are upper bounded by $O(n^n)$ (grows really fast!)
  - A better, tighter bound would be $f(n) \in \Theta(n)$

# Best vs. Worst Case

- In best-worst case analysis, we still assume the input is very large.
- Therefore, you cannot make assumptions such as N == 1 or N <= 10 in these analyses.
  - Instead, you make other assumptions.
- **The best case is not when the input is 1.**
- **The best case is not when the input is 1.**
- **Seriously, the best case is not when the input is 1.**

# Best vs. Worst Case

- Represented with tight bound Θ because they should be consistent (always run in the same time)
- Look out for branching statements, loop conditions, breaks
- Check: What is the best/worst case runtime of the function below?
- **Remember: The best case is not when the input is 1.**

```
public static void example(int N) {
    while (N > 0) {
        if (func(N)) {
            break;
        }
        N -= 1;
    }
}
```

Best case: Θ(1), where N = some int for which func(N) is immediately true

I'm not assuming N is 1 or N is small. func(N) could be return whether N is an even number, and when N is very large but even number this function runs in constant time

Worst case: Θ(N), where N = some int for which func(N), func(N - 1), ..., func(1) are all false

# Best vs. Worst Case

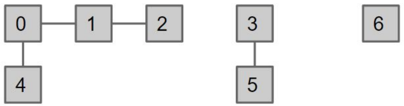
- Best/worst case vs. lower/upper bound analogy: how much does it cost to eat at a restaurant?
  - Best/worst-case: “the cheapest thing on the menu is $5 and the most expensive is $50”
  - Lower/upper bound: “every item is at least $5 and at most $50” (credit: Alex Schedel)
- Which one is more informative?
  - The first one: the best/worst-case are the tightest lower/upper-bounds you can give.

# Disjoint Sets, also known as Union Find

```
public interface DisjointSet {
     void connect (x, y); // Connects nodes x and y (you may also see union)
     boolean isConnected(x, y); // Returns true if x and y are connected
}
```

**QuickFind** uses an array of integers to track which set each element belongs to.

Before connect(2, 3) operation:

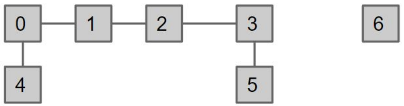



{ 0, 1, 2, 4 }, {3, 5}, {6}

int[] id

| | | | | | | |
|---|---|---|---|---|---|---|
| 4 | 4 | 4 | 5 | 4 | 5 | 6 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

After connect(2, 3) operation:




{ 0, 1, 2, 4, 3, 5}, {6}

int[] id

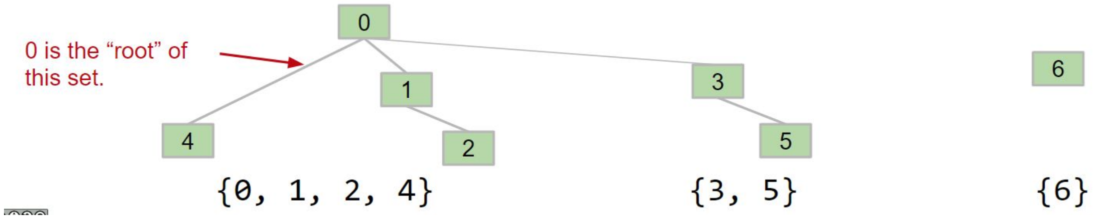
| | | | | | | |
|---|---|---|---|---|---|---|
| 5 | 5 | 5 | 5 | 5 | 5 | 6 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

# Disjoint Sets, also known as Union Find

```
public interface DisjointSet {
     void connect (x, y); // Connects nodes x and y (you may also see union)
     boolean isConnected(x, y); // Returns true if x and y are connected
}
```

**QuickUnion** stores the parent of each node rather than the set to which it belongs and merges sets by setting the parent of one root to the other.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| parent | -1 | 0 | 1 | 0 | 0 | 3 | -1 |



# Disjoint Sets, also known as Union Find

```
public interface DisjointSet {
     void connect (x, y); // Connects nodes x and y (you may also see union)
     boolean isConnected(x, y); // Returns true if x and y are connected
}
```

**WeightedQuickUnion** does the same as QuickUnion except it decides which set is merged into which by size (merge smaller into larger), reducing stringiness.

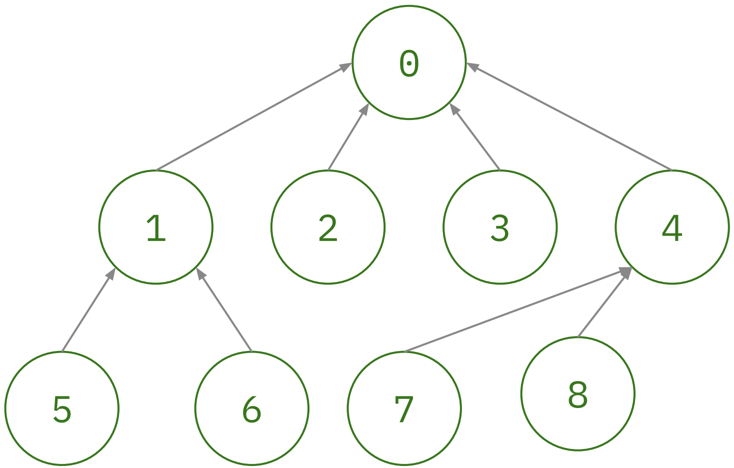
**WeightedQuickUnion with Path Compression** sets the parent of each node to the set's root whenever find() is called on it.

# Disjoint Sets Representation

- We can use a single array to represent our disjoint set when implementing `connect()` optimally (ie. `WeightedQuickUnion`)

- `arr[i]` contains the parent of element i in the set; the index of a root contains - `(# elements in set rooted at that index)`

`[-9, 0, 0, 0, 0, 1, 1, 3, 4]`



# Disjoint Sets Asymptotics

```
public interface DisjointSet {
     void connect (x, y); // Connects nodes x and y (you may also see union)
     boolean isConnected(x, y); // Returns true if x and y are connected
}
```

| Implementation | Constructor | connect() | isConnected() |
|---|---|---|---|
| QuickUnion | Θ(N) | O(N) | O(N) |
| QuickFind | Θ(N) | O(N) | O(1) |
| Weighted Quick Union | Θ(N) | O(log N) | O(log N) |
| WQU with Path Compression | Θ(N) | O(log N)<br>Θ(1)-ish amortized | O(log N)<br>Θ(1)-ish amortized |

* we don't really talk about QU/QF in application, more to show the asymptotic motivation for WQU

# Worksheet

## 1 Asymptotics Introduction

Give the runtime of the following functions in Θ notation. Your answer should be as simple as possible with no unnecessary leading constants or lower order terms.

```
private void f1(int N) {
    for (int i = 1; i < N; i++) {
        for (int j = 1; j < i; j++) {
            System.out.println("shreyas 1.0");
        }
    }
}
```

| i | 1 | 2 | ... | N |
|---|---|---|---|---|
| j | 1 | 1 | | 1 |
| | | 2 | . . . | 2 |
| | | | | ⋮ |
| | | | | N |

Θ( $N^2$ )

Work $1 + 2 + 3 + ... + N = \frac{N(N+1)}{2} \to N^2$

```
private void f2(int N) {
    for (int i = 1; i < N; i *= 2) {
        for (int j = 1; j < i; j++) {
            System.out.println("shreyas 2.0");
        }
    }
}
```

| i | 1 | 2 | 4 | ... | N |
|---|---|---|---|---|---|
| j | 1 | 1 | 1 | | 1 |
| | | 2 | 2 | | ⋮ |
| | | | 3 | | |
| | | | 4 | . . . | N |

Θ( $N$ )

Work $1 + 2 + 4 + ... + N = \frac{1(1-2^{\log_2 N + 1})}{1-2} = 2N - 1 \to N$

↑ $\log_2 N + 1$ terms

## 2 Disjoint Sets

For each of the arrays below, write whether this could be the array representation of a weighted quick union with path compression and explain your reasoning. **Break ties by choosing the smaller integer to be the root.**

Try constructing "maximal" height WQUs for intuition building

```
         i:   0   1   2   3   4   5   6   7   8   9
         ------------------------------------------
A. a[i]:      1   2   3   0   1   1   1   4   4   5
B. a[i]:      9   0   0   0   0   0   9   9   9 -10
C. a[i]:      1   2   3   4   5   6   7   8   9 -10
D. a[i]:    -10   0   0   0   0   1   1   1   6   2
E. a[i]:    -10   0   0   0   0   1   1   1   6   8
F. a[i]:     -7   0   0   1   1   3   3  -3   7   7
```

A. ✗ cycle
B. ✗ 0 wouldn't connect as child to 9 after 1-5 joined
C. ✗ height is 9 – a linked list, effectively
D. ✓
E. ✗ height is 4 – one too many → $\log_2 10$
F. ✗ height is 3 – one too many → $\log_2 7$
↑ the base is mathy

# 3 Asymptotics of Weighted Quick Unions

Note: for all big $\Omega$ and big $O$ bounds, give the *tightest* bound possible.

(a) Suppose we have a Weighted Quick Union (WQU) without path compression with N elements.

1. What is the runtime, in big $\Omega$ and big $O$, of `isConnected`?

   check root and descedant

   $\Omega$(__1____), $O$(_logN___) check bottom layer and <u>any</u> other node

   $\log N+1 \equiv 2\log N$ for runtime

2. What is the runtime, in big $\Omega$ and big $O$, of `connect`?

   connect to root / connect to bottom layer

   $\Omega$(__1____), $O$(_logN__)

(b) Suppose we add the method `addToWQU` to a WQU without path compression. The method takes in a list of `elements` and `connects` them in a random order, stopping when all elements are connected. Assume that all the `elements` are disconnected before the method call.

```
void addToWQU(int[] elements) {
        int[][] pairs = pairs(elements);
        for (int[] pair: pairs) {
            if (size() == elements.length) {
                return;
            }
            connect(pair[0], pair[1]);
        }
}
```

The `pairs` method takes in a list of `elements` and generates all possible pairs of elements in a random order. For example, `pairs([1, 2, 3])` might return `[[1, 3], [2, 3], [1, 2]]` or `[[1, 2], [1, 3], [2, 3]]`.

The `size` method calculates the size of the largest component in the WQU.

Assume that `pairs` and `size` run in constant time.

What is the runtime of `addToWQU` in big $\Omega$ and big $O$?

Best case: $(0,k)$ for $k$ $1,\ldots,N$ to directly connect everything

$\Omega$(__N___), $O$(_$N^2\log N$) Worst case: $(i,j)$ for $i \in 1,\ldots,N$, $j \in 1,\ldots,N$, $i \neq j$. Then add $i$ in at the end

Assuming combinations: $\binom{N-1}{2} = \frac{(N-1)!}{2!(N-3)!} = \frac{(N-1)(N-2)}{2}$ pairs, plus 1 for the new item

*Hint: Consider the number of calls to* `connect` *in the best case and worst case. Then, consider the best/worst case time complexity for one call to* `connect`.

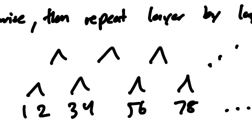
(c) Let us define a **matching size connection** as `connecting` two components in a WQU of equal size. For instance, suppose we have two trees, one with values 1 and 2, and another with the values 3 and 4. Calling `connect(1, 4)` is a matching size connection since both trees have 2 elements.

What is the **minimum** and **maximum** number of matching size connections that can occur after executing `addToWQU`. Assume `N`, i.e. `elements.length`, is a power of two. Your answers should be exact.

minimum: __1___, maximum: _N-1_

Minimum: Add everything straight to root; won't be different

Maximum: Go pairwise, then repeat layer by layer: $\frac{N}{2}$ size 1, $\frac{N}{4}$ size 2, ...




$\hookrightarrow N\left(\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots\frac{1}{N}\right)$

$= N-1$