CS 61B Spring 2026

Asymptotics I, Disjoint Sets

Discussion 06: February 23, 2026

1 The Space-Time Continuum

Consider the following runtimes...

$\log(N) + N$ $\quad \log(2^N)$ $\quad N^2 + 3N\log(N)$ $\quad (2^N * 2)$ $\quad \log(9)$ $\quad 2^{N+4}$ $\quad 2 + \frac{1}{N}$ $\quad \log(N^2)$ $\quad 5N^2$ $\quad 6\log(N)$

Theta bound, simplify, and sort each of the above runtimes in the below categories.

Recall the rules for simplifying asymptotic bounds and logarithm rules!

$\Theta(1)$	$\Theta(\log N)$	$\Theta(N)$	$\Theta(N^2)$	$\Theta(2^N)$
$2 + \frac{1}{N}$, $\log(9)$	$\log(N^2)$, $6\log(N)$	$\log(2^N)$, $\log(N) + N$	$N^2 + 3N\log(N)$, $5N^2$	2^{N+4}, $2^N * 2$

slowest-growing $\longleftrightarrow$ fastest-growing

2 Thrown for a Loop

A major component of asymptotic analysis is the analysis of loops. Let `N` be some arbitrarily large integer. Let `doWork` be a function that runs in $\Theta(1)$ (also known as constant time).

Consider the below loop.

```
for (int i = 1; i < N; i++) {
  doWork();
}
```

(a) What is the asymptotic runtime in terms of N?

Runtime: $\Theta(N)$

(b) If we were to change `i < N` to `i < N * 5`, would the asymptotic runtime change? If so, what would the new theta bound be? If not, why not?

Nope! The loop will run 5N times, but the overall runtime still depends on a linear function of N.

(c) If we were to call `doWork` N times instead of 1, would the asymptotic runtime change? If so, what would the new theta bound be? If not, why not?

The runtime would change! Since the number of `doWork` calls now depends on N, the runtime would simplify to $N * N = N^2$

Consider a nested for loop.

```
for (int i = 0; i < N; i++) {
  for (int j = 0; j < N; j++) {
    doWork();
  }
}
```

(d) What is the asymptotic runtime in terms of N?

Runtime: $\Theta(N^2)$

(e) If we were to change `j = 0` to `j = i`, would the asymptotic runtime change? If so, what would the new theta bound be? If not, why not?

The runtime would not change! This is evident if you track the number of inner loops as a function of `i` - the total runtime comes out to be an arithmetic sum.

3 xStep

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.

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

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

Runtime: $\Theta(N)$

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

Runtime: $\Theta(N^2)$

4 Disjoint Sets

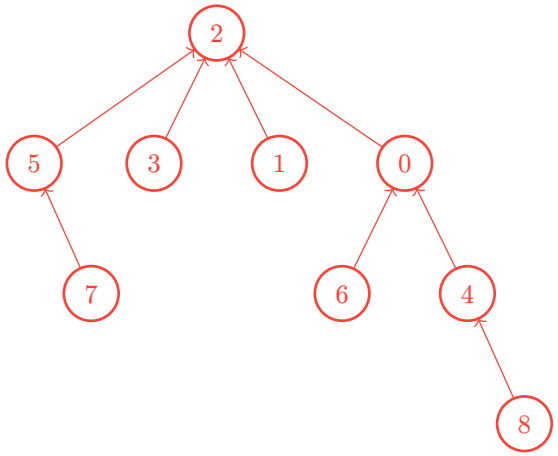
In lecture, we discussed the Disjoint Sets ADT. Some authors call this the Union Find ADT. Today, we will use union find terminology so that you have seen both.

(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.

Runtime: $\Theta(N)$

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.

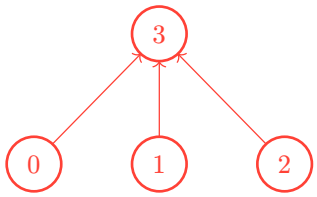
(c) Using a function **setParent(int val, int newParent)**, which updates the value of **val**'s parent to **newParent**, modify **find** to achieve a faster runtime using path compression. You may add at most one line to the provided implementation.

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