

1 The Space-Time Continuum

For each growth function on the left, Theta bound it, simplify it, and draw a line to the matching Theta bound on the right. Once you're done, go to <https://hugocode.com/61b/discs/> for an interactive visualization of these functions similar to what we saw in [lecture 13](#).

Recall the rules for simplifying asymptotic bounds and logarithm rules!

$\log(N) + N$	$\Theta(1)$
---------------	-------------

$N^2 + 3N \log(N)$

$\log(2^N)$	$\Theta(\log N)$
-------------	------------------

$2^N * 2^{-100}$

$\log(9)$	$\Theta(N)$
-----------	-------------

2^{N+4}

$\Theta(N \log N)$

$2 + \frac{1}{N}$

$\log(N^2)$	$\Theta(N^2)$
-------------	---------------

$5N^2$

$6 \log(N)$	$\Theta(2^N)$
-------------	---------------

2 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) {
    IO.println("tidal wave");
    x++;
}
```

Runtime: $\Theta(\rule{1.5cm}{0.4pt})$

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

Runtime: $\Theta(\rule{1.5cm}{0.4pt})$

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

Runtime: $\Theta(\rule{1.5cm}{0.4pt})$

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

Runtime: $\Theta(\rule{1.5cm}{0.4pt})$

3 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);
```

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: _____

- (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.