# Asymptotics and Disjoint Sets

Exam-Level Discussion 05: September 29, 2025

## 1 Finish the Runtimes

Below we see some standard nested for loops, but with missing pieces!

For each part, **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 = j + 1) {
        System.out.println("This is one is low key hard");
    }
}
```

Remember the arithmetic series $1 + 2 + 3 + 4 + ... + N = \Theta(N^2)$. We get this series by incrementing $j$ by 1 per inner loop.

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

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

Any constant would work here, 2 was chosen arbitrarily.

The outer loop already runs $\log n$ times, since $i$ doubles each time. This means the inner loop must do constant work (so any constant `j < k` would work).

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

```
for (int i = 1; i < N; i = i + 1) {
    for (int j = 1; j < Math.pow(2, i); j = j + 1) {
        System.out.println("This is one is high key hard");
    }
}
```

Remember the geometric series $1 + 2 + 4 + ... + 2^N = \Theta(2^N)$. We notice that $i$ increments by 1 each time, so in order to achieve this $2^N$ runtime, we must run the inner loop $2^i$ times per outer loop iteration.

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

```
for (int i = 1; i < Math.pow(2, N); i = i * 2) {

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

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

One way to get $N^3$ runtime is to have the outer loop run $N$ times, and the inner loop run $N^2$ times per outer loop iteration. To make the outer loop run $N$ times, we need stop after multiplying **`i = i * 2`** $N$ times, giving us the condition **`i < Math.pow(2, N)`**. To make the inner loop run $N^2$ times, we can simply increment by 1 each time.

## 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.**

There are three criteria here that invalidate a representation:

- If there is a cycle in the parent-link.
- For each parent-child link, the tree rooted at the parent is smaller than the tree rooted at the child before the link (you would have merged the other way around).
- The height of the tree is greater than $\log_2 n$, where $n$ is the number of elements.

(a)

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

**Impossible**: has a cycle 0-1, 1-2, 2-3, and 3-0 in the parent-link representation.

(b)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 9 | 0 | 0 | 0 | 0 | 9 | 9 | 9 | $-10$ |

**Impossible**: the nodes 1, 2, 3, 4, and 5 must link to 0 when 0 is a root; hence, 0 would not link to 9 because 0 is the root of the larger tree.

(c)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | $-10$ |

**Impossible**: tree rooted at 9 has height $9 > \log_2 10$.

(d)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $-10$ | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 6 | 2 |

**Possible**: 8-6, 7-1, 6-1, 5-1, 9-2, 3-0, 4-0, 2-0, 1-0.

(e)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $-10$ | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 6 | 8 |

**Impossible**: tree rooted at 0 has height $4 > \log_2 10$.

(f)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $-7$ | 0 | 0 | 1 | 1 | 3 | 3 | $-3$ | 7 | 7 |

**Impossible**: tree rooted at 0 has height $3 > \log_2 7$.

# 3 This is NOT an Interview!

Given an **int x** and a *sorted* array **A** of $N$ distinct integers, design an algorithm to find if there exists indices **i** and **j** such that **A[i] + A[j] == x**.

Let's start with the naive solution.

```
public static boolean findSum(int[] A, int x) {
    for (int i = 0; i < A.length; i++){
        for (int j = 0; j < A.length; j++) {
            if (A[i] + A[j] == x) return true;
        }
    }
    return false;
}
```

(a) How can we improve this solution? *Hint*: Does order matter here?

```
public static boolean findSumFaster(int[] A, int x){
    int left = 0;
    int right = A.length - 1;
    while (left <= right) {
        if (A[left] + A[right] == x) {
            return true;
        } else if (A[left] + A[right] < x) {
            left++;
        } else {
            right--;
        }
    }
    return false;
}
```

(b) What is the runtime of both the original and improved algorithm?

Naive: Worst $= \Theta(N^2)$, Best $= \Theta(1)$. Optimized: Worst $= \Theta(N)$, Best $= \Theta(1)$