Java 2 Algorithm Analysis

- **Algorithm Analysis** is an approach to describing certain efficiency characteristics of an algorithm in terms of certain problem characteristics.
- Typically this means describing time/work or space requirements of an algorithm in terms of input size.
- Algorithm analysis allows us to **predict** the performance.

Performance bugs

- A good understanding of algorithm analysis also allows us to avoid performance bugs in our software. A poor understanding of algorithm analysis can lead directly to software that:
  - Performs too slowly, especially as input sizes increase
  - consumes too much memory
  - Opens security vulnerabilities, like denial of service attacks
  - **(QUADRATIC TIME COMPLEXITY IS BADDDDDD)**

Analyzing

- **Empirical** - Analyze running time based on observations and experiments.
  - Use the scientific method.
- **Mathematical -** Develop a cost model that includes cost for individual operations.
  - Basically use summations.
  - cost of executing operation *i* multiplied by frequency of execution of operation *i.*
  - Treat the cost of primitive operations and simple statements as some unspecified constant.
  - Running time is a constant.
  - Focus only on “core” operations instead of counting every single operation that is performed.
  - The running time of sumB is c*N which is linear.
  - Focus only on the highest order term and ignore coefficients, constants, and low-order terms.
  - So running time is some quadratic function (N^2)
  - **Only Care About the Fastest Growing Term**

Analyzing the Binary Search Algorithm

```
public int search(int[] a, int target) {
	int left = 0, right = a.length -1;
	while(left <= right) {
		int middle = (left + right) / 2; //WORST CASE ANALYSIS//
		if (target < a[middle])
			left = middle + 1;
		else
			return middle;
	}
	return -1;
}
```

Worst case situation is if the target is not in the array.

Search Space:

```
******************************************* N
*********************                       N/2
********                                    N/4
****                                        N/8
```

.
. *after k operations...*

.
$N/2^k$
.
.
.
1

Solving for k:
- $N/2^k = 1$
- $N = 2^k$
- $k = \log_2 N$

Growth Rate:

- In here we call it **Order**
- All quadratics slow down by a factor of 4 when the size is doubled.
- We describe growth rate in **big-Oh** notation.
- **$O(N^2)$** is understand as not getting any bigger than N^2.
- We want to use the tightest growing bound.

Asymptotic Notation

- **In 3270, we learn Big-Omega and Big-Theta**
- **We often (mis)use big-Oh to mean big-Theta**

Common Orders of Growth

- 1, log N, N, N log N, N^2, N^3, 2^N, N!

Calculating big-Oh

- We will use a simple syntax-based approach to calculating worst-case big-Oh

1. All simple statement and primitive ops have constant cost.
2. The cost of a sequence of statements is the sum of the costs of each individual statement.
3. The cost of a selection statement is the cost of the most expensive branch.
4. The cost of a loop is the cost of the body multiplied by the maximum number of iterations that the loop makes.

- Constants go away.

1. $O(n^3)$
2. $O(n^2)$
3. O(1)