Building Java Programs

Chapter 13

binary search and complexity

reading: 13.1-13.2

Now Playing:
Manatee commune

Road Map

CS Concepts

- Client/Implementer
- Efficiency
- Recursion
- Regular Expressions
- Grammars
- Sorting
- Backtracking
- Hashing
- Huffman Compression

Java Language

- Exceptions
- Interfaces
- References
- Comparable
- Generics
- Inheritance/Polymorphism
- Abstract Classes

Data Structures

- Lists
- Stacks
- Queues
- Sets
- Maps
- Priority Queues

Java Collections

- Arrays
- ArrayList 🛠
- LinkedList 🛠
- Stack
- TreeSet / TreeMap
- HashSet / HashMap
- PriorityQueue

Sum this up for me

- Let’s write a method to calculate the sum from 1 to some n

```
public static int sum1(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += i;
    }
    return sum;
}
```

- Gauss also has a way of solving this

```
public static int sum2(int n) {
    return n * (n + 1) / 2;
}
```

- Which one is more efficient?

Runtime Efficiency (13.2)

- **efficiency**: measure of computing resources used by code.
  - can be relative to speed (time), memory (space), etc.
  - most commonly refers to run time
- We want to be able to compare different algorithms to see which is more efficient

Efficiency Try 1

- Let's time the methods!

```
n = 1                sum1 took    0ms, sum2 took 0ms
n = 5                sum1 took    0ms, sum2 took 0ms
n = 10               sum1 took    0ms, sum2 took 0ms
n = 100              sum1 took    0ms, sum2 took 0ms
n = 1,000            sum1 took  0/1ms, sum2 took 0ms
n = 10,000,000       sum1 took 10/18ms, sum2 took 0ms
n = 100,000,000      sum1 took 142/123ms, sum2 took 0ms
n = 2,147,483,647    sum1 took 1570/1804ms, sum2 took 0ms
```

- Downsides
  - Different computers give different run times
  - The same computer gives different results!!! D:<

Efficiency – Try 2

- Count number of "simple steps" our algorithm takes to run
- Assume the following:
  - Any single Java statement takes same amount of time to run.
    - `int x = 5;`
    - `boolean b = (5 + 1 * 2) < 15 + 3;`
    - `System.out.println("Hello");`
  - A loop's runtime, if the loop repeats N times, is N times the runtime of the statements in its body.
  - A method call's runtime is measured by the total runtime of the statements inside the method's body.

Efficiency examples

```
public static void method1(int N) {
    statement1;
    statement2;
    statement3;

    for (int i = 1; i <= N; i++) {
        statement4;
    }

    for (int i = 1; i <= N; i++) {
        statement5;
        statement6;
        statement7;
    }
}
```

statement1–statement3: 3

first loop: N

second loop: 3N

total: 4N + 3

Efficiency examples 2

```
public static void method2(int N) {
    for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= N; j++) {
            statement1;
        }
    }

    for (int i = 1; i <= N; i++) {
        statement2;
        statement3;
        statement4;
        statement5;
    }
}
```

N^2

$4N$

$N^2 + 4N$

- How many statements will execute if N = 10? If N = 1000?

≈ 140 ≈ a lot

Sum this up for me

- Let's write a method to calculate the sum from 1 to some n

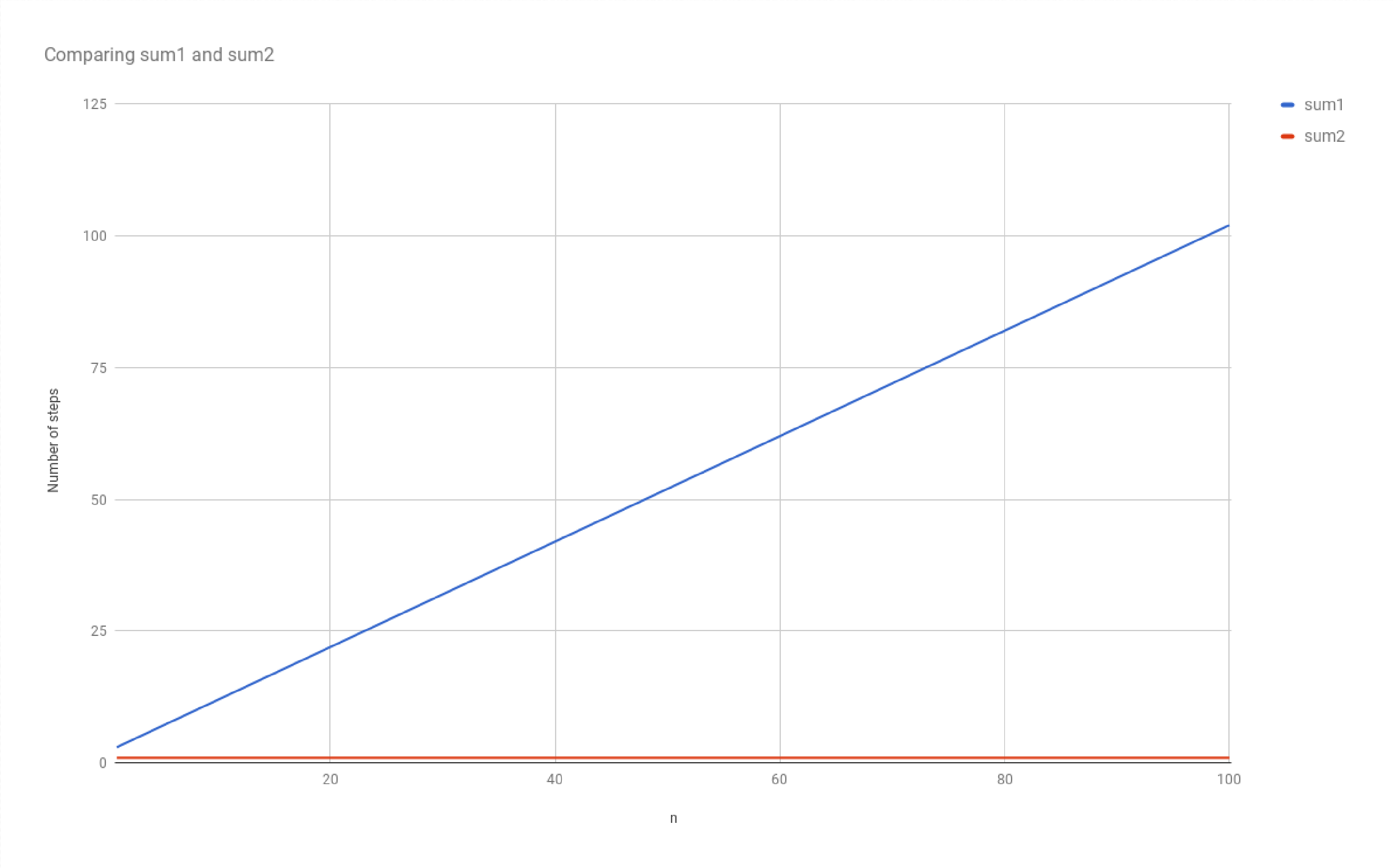
```
public static int sum1(int n) {
    int sum = 0;                      } 1
    for (int i = 1; i <= n; i++) {    }
        sum += i;                     } N
    }                                           } N + 2
    return sum;                       } 1
}
```

- Gauss also has a way of solving this

```
public static int sum2(int n) {
    return n * (n + 1) / 2;           } 1       } 1
}
```

- Which one is more efficient?

Visualizing Difference



Algorithm growth rates (13.2)

- We measure runtime in proportion to the input data size, N.
  - **growth rate**: Change in runtime as N changes.

- Say an algorithm runs ~~$\mathbf{0.4N^3 + 25N^2 + 8N + 17}$~~ statements.
  - Consider the runtime when N is *extremely large* .
  - We ignore constants like 25 because they are tiny next to N.
  - The highest-order term (N^3) dominates the overall runtime.
  - We say that this algorithm runs "on the order of" N^3.
  - or **O(N^3)** for short ("**Big-Oh** of N cubed")

If you double input, will take about 8x as long

Poll Everywhere pollev.com/cse143

- Suppose our list had the contents

```
public void method(int n) {
    int value = 0;
    for (int i = 0; i < 7; i++) {
        for (int j = 0; j < n; j++) {
            value += j;
        }
    }
    return value + n / 2;
}
```

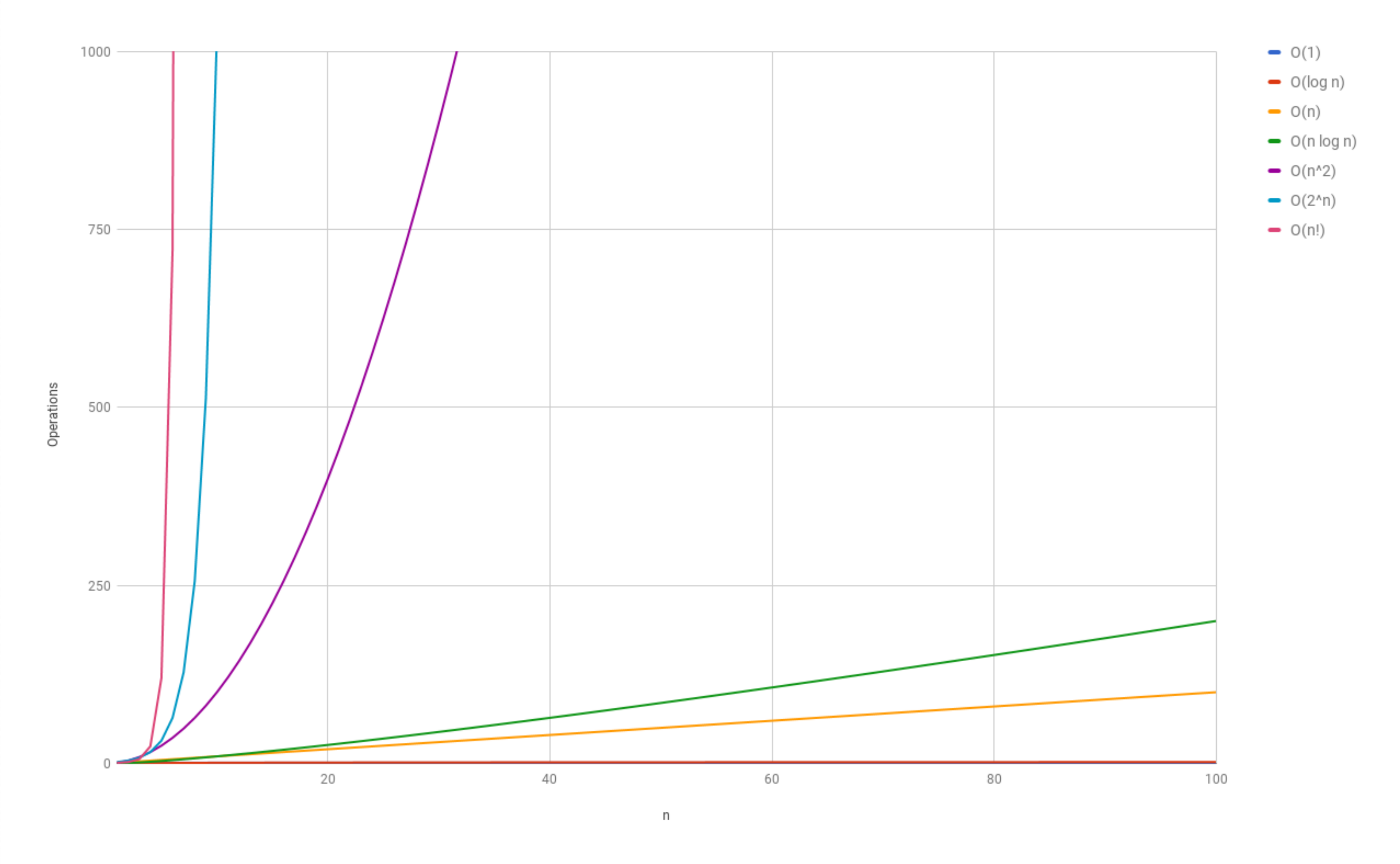
- What is the Big-O efficiency for this function?
  - O(1)
  - O(n) ✓
  - ~~O(7n)~~
  - ~~O(7n + 4)~~;
  - $O(n^2)$
  - $O(n^3)$

Complexity classes

- **complexity class**: A category of algorithm efficiency based on the algorithm's relationship to the input size N.

Class	Big-Oh	If you double N, ...	Example
constant	O(1)	unchanged	10ms
logarithmic	$O(\log_2 N)$	increases slightly	175ms
linear	O(N)	doubles	3.2 sec
log-linear	$O(N \log_2 N)$	slightly more than doubles	6 sec
quadratic	$O(N^2)$	quadruples	1 min 42 sec
cubic	$O(N^3)$	multiplies by 8	55 min
...	...	...	...
exponential	$O(2^N)$	multiplies drastically	$5 * 10^{61}$ years

Complexity classes

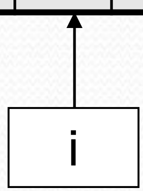


http://recursive-design.com/blog/2010/12/07/comp-sci-101-big-o-notation/ - post about a Google interview

Sequential search

- **sequential search**: Locates a target value in an array / list by examining each element from start to finish. Used in `indexOf`.
  - How many elements will it need to examine?
  - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103



Sequential search

- What is its complexity class?

```
public int indexOf(int value) {
    for (int i = 0; i < size; i++) {
        if (elementData[i] == value) {
            return i;
        }
    }
    return -1;    // not found
}
```

N

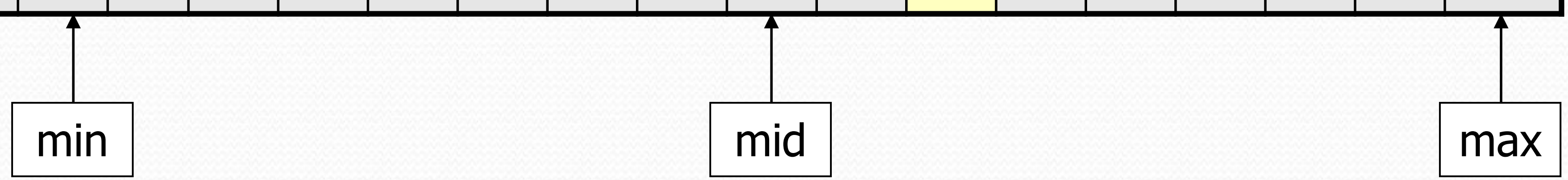
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

- On average, "only" N/2 elements are visited
    - 1/2 is a constant that can be ignored

Binary search (13.1)

- **binary search**: Locates a target value in a *sorted* array or list by successively eliminating half of the array from consideration.
  - How many elements will it need to examine?
  - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103



Binary search

- **binary search** successively eliminates half of the elements.
  - *Algorithm:* Examine the middle element of the array.
    - If it is too big, eliminate the right half of the array and repeat.
    - If it is too small, eliminate the left half of the array and repeat.
    - Else it is the value we're searching for, so stop.
  - Which indexes does the algorithm examine to find value **42**?
  - What is the runtime complexity class of binary search?

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

min (index 0) mid (index 8) max (index 16)

Binary search runtime

- For an array of size N, it eliminates ½ until 1 element remains.

  N, N/2, N/4, N/8, ..., 4, 2, 1

  - How many divisions does it take?

- Think of it from the other direction:
  - How many times do I have to multiply by 2 to reach N?

    1, 2, 4, 8, ..., N/4, N/2, N
  - Call this number of multiplications "x".

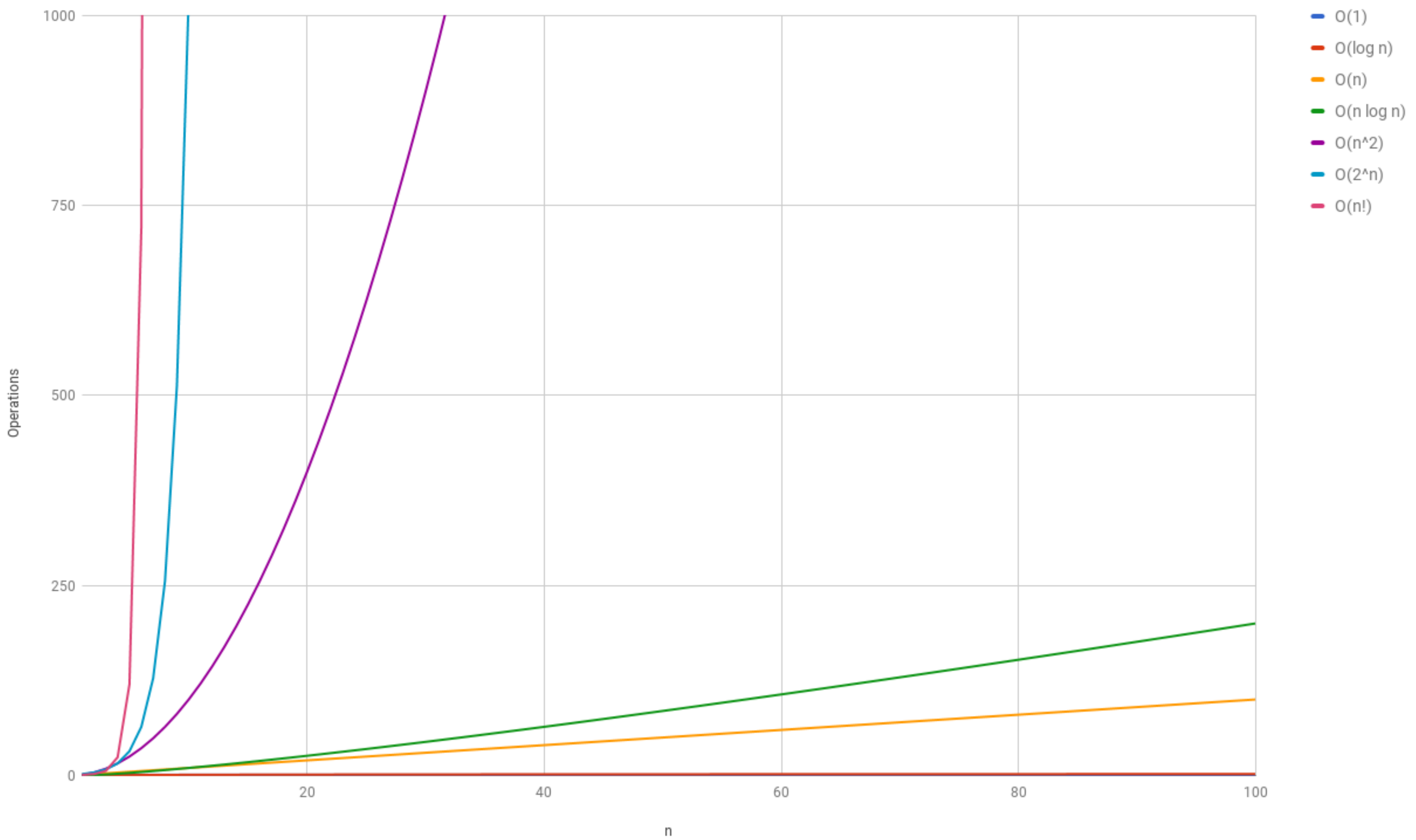
  $2^x = N$

  $\mathbf{x = \log_2 N}$

  $O(\log_2 n)$

- Binary search is in the **logarithmic** complexity class.

Complexity classes



http://recursive-design.com/blog/2010/12/07/comp-sci-101-big-o-notation/ - post about a Google interview

Collection efficiency

- Efficiency of our Java's `ArrayList` and `LinkedList` methods:

Method	**ArrayList**	**LinkedList**
`add`	O(1)*	O(1)
`add(`**index**`, `**value**`)`	O(N)	O(N)
`indexOf`	O(N)	O(N)
`get`	O(1)	O(N)
`remove`	O(N)	O(N)
`set`	O(1)	O(N)
`size`	O(1)	O(1)

Will cover Wed

* Most of the time!

Throw Back: Unique words

- Recall two weeks ago when we counted the number of unique words in a file. Our first attempt

```
public static int uniqueWords(Scanner input) {
    List<String> words = new LinkedList<String>();
    while (input.hasNext()) {
        String word = input.next();
        if (!words.contains(word)) {
            words.add(word);
        }
    }
    return words.size();
}
```

Throw Back: Unique words

- Recall two weeks ago when we counted the number of unique words in a file. Our second attempt
- We saw briefly that operations on `HashSet` are O(1)

```
public static int uniqueWords(Scanner input) {
    Set<String> words = new HashSet<String>();
    while (input.hasNext()) {
        String word = input.next();
        words.add(word);
    }
    return words.size();
}
```