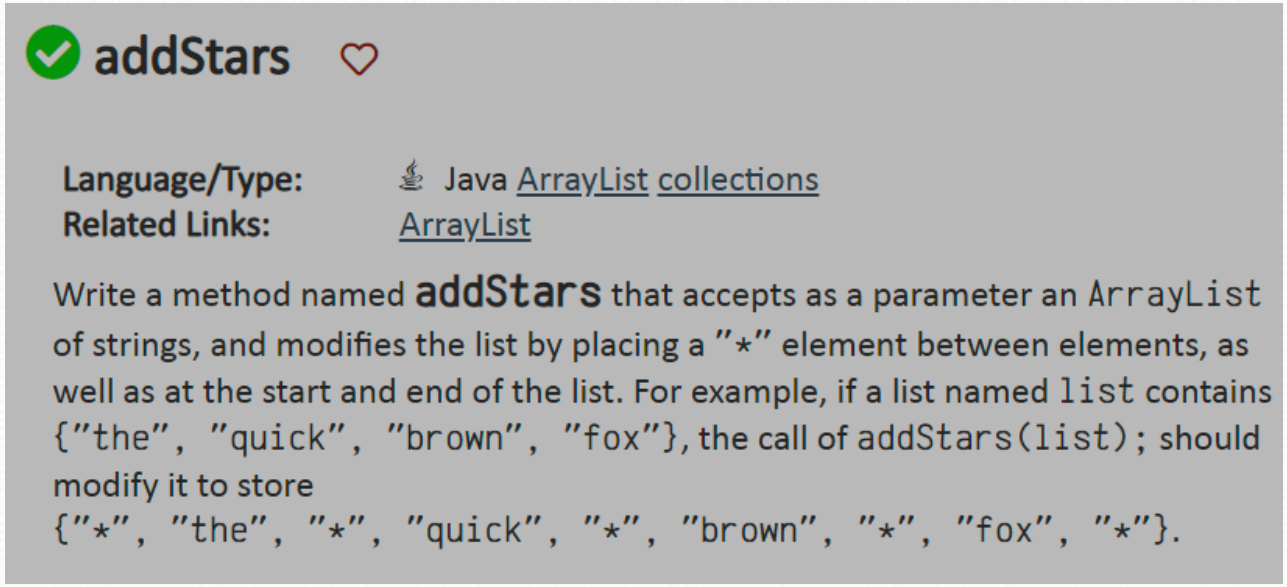
CS 142

Lecture 7: Complexity

Thanks to Marty Stepp and Stuart Reges for parts of these slides

Recall: `addStars`

- Yesterday we looked at `addStars` in lab



addStars

Language/Type: Java ArrayList collections
Related Links: ArrayList

Write a method named **addStars** that accepts as a parameter an `ArrayList` of strings, and modifies the list by placing a "*" element between elements, as well as at the start and end of the list. For example, if a list named `list` contains `{"the", "quick", "brown", "fox"}`, the call of `addStars(list);` should modify it to store
`{"*", "the", "*", "quick", "*", "brown", "*", "fox", "*"}`.

- What was hard about this problem?
- Is there any similarity between this problem and `RemovePlurals` code could help us?

Evaluating our Algorithms

- Over the last few lectures, we have discussed **sequential search** and **binary search**
  - We found we need to check way **less indexes** when using binary search – so it is **faster**
  - How can we communicate this to others without walking through the whole algorithm?
    - Generally, how can we compare the efficiency of different code solutions?

Runtime Efficiency

- **efficiency**: measure of computing resources used by code.
  - can be relative to speed (time), memory (space), etc.
  - most commonly refers to run time

- Assume the following:
  - Any single Java statement takes same amount of time to run.
  - A method call's runtime is measured by the total of the statements inside the method's body.
  - A loop's runtime, if the loop repeats N times, is N times the runtime of the statements in its body.

Efficiency examples

```
statement1;
statement2;
statement3;
```

3

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

N

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

3N

4N + 3

Efficiency examples 2

```
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?

Algorithm growth rates

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

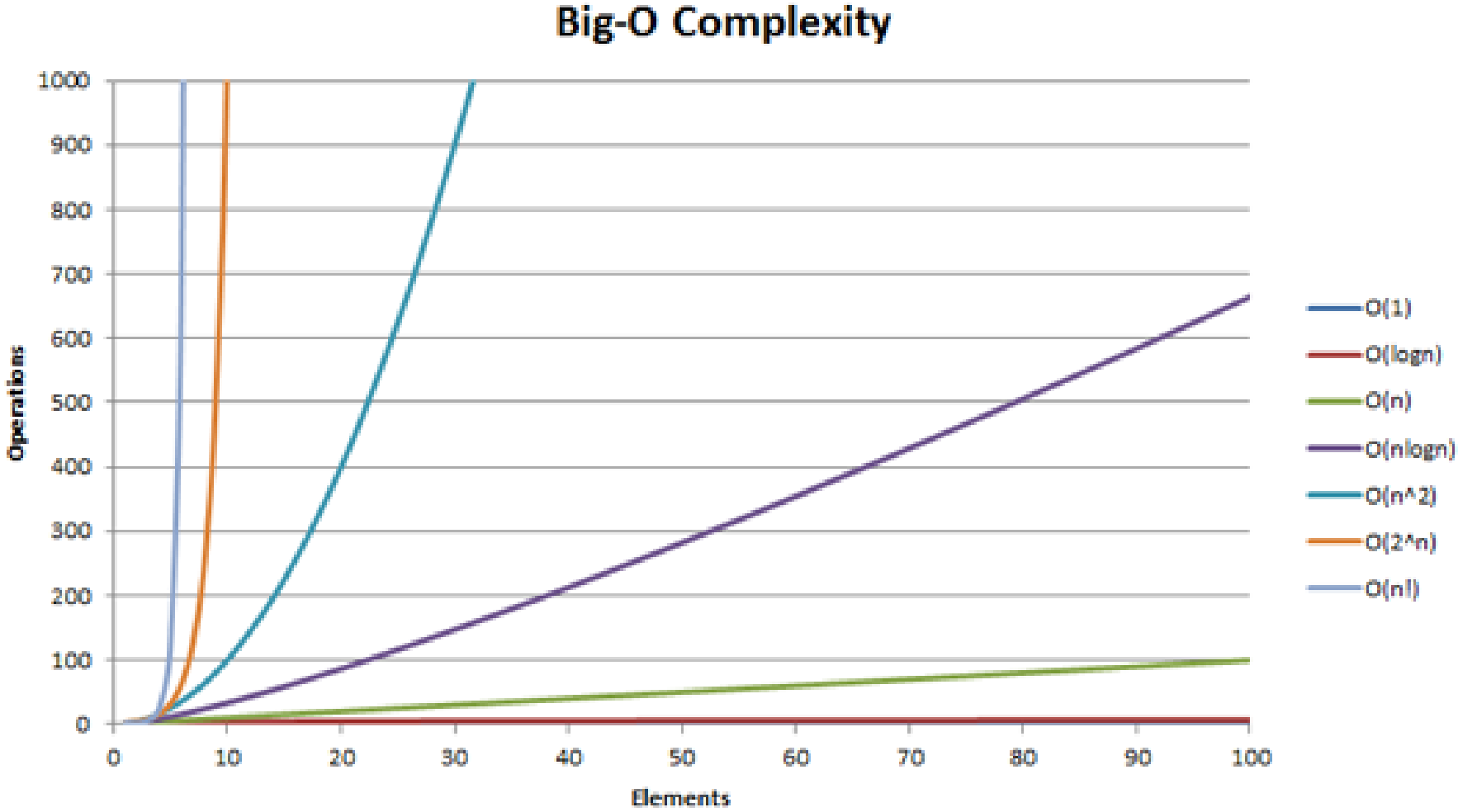
- Say an algorithm runs **$0.4N^3 + 25N^2 + 8N + 17$** statements.
  - Consider the runtime when N is *extremely large* .
  - We ignore constants like 25 because they are tiny next to N.
  - The highest-order term (N^3) dominates the overall runtime.
  - We say that this algorithm runs "on the order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

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

Collection efficiency

- Efficiency of our `ArrayIntList` or Java's `ArrayList`:

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

The software crisis

- **software engineering**: The practice of developing, designing, documenting, testing large computer programs.

- Large-scale projects face many issues:
  - programmers working together
  - getting code finished on time
  - avoiding redundant code
  - finding and fixing bugs
  - maintaining, reusing existing code

- **code reuse**: The practice of writing program code once and using it in many contexts.