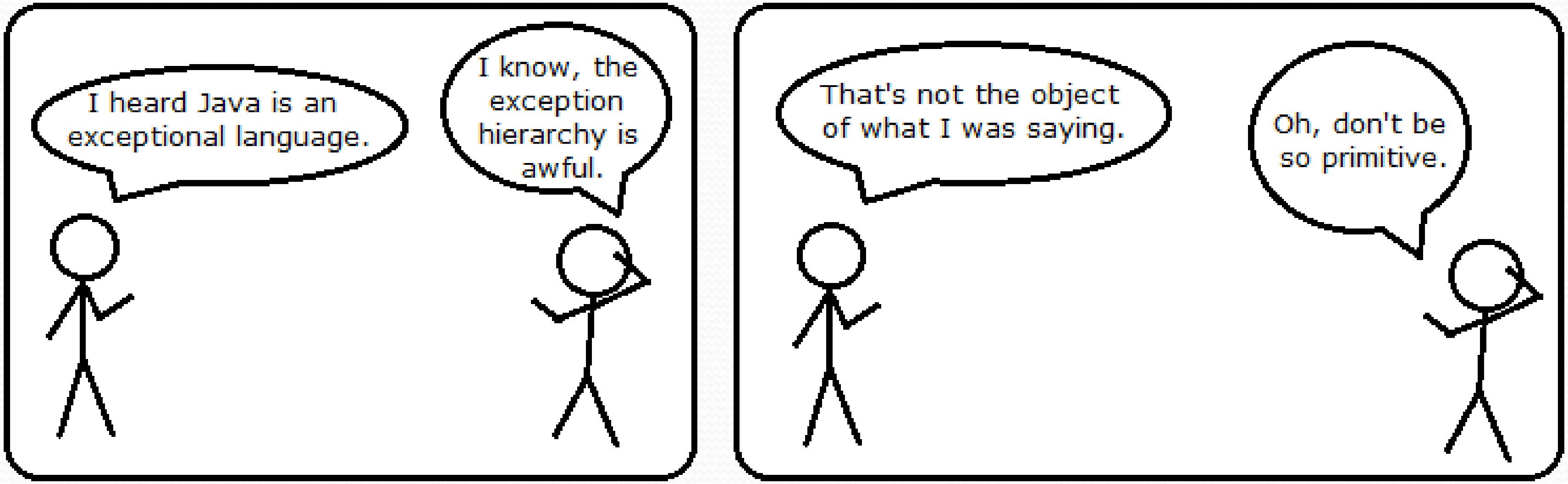
CSE 143

Lecture 5: complexity

reading: 13.1-13.2



http://www.alexsweet.co.uk/comics.php?comic=2

Interfaces

- **interface**: A list of methods that a class can promise to implement.
  - Inheritance gives you an is-a relationship *and* code sharing.
    - A `Lawyer` can be treated as an `Employee` and inherits its code.
  - Interfaces give you an is-a relationship *without* code sharing.
    - A `Rectangle` object can be treated as a `Shape` but inherits no code.
  - Always declare variables using the ***interface*** type.

```
List<String> list = new ArrayList<String>();
```

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

- 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;

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

```
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 (13.2)

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

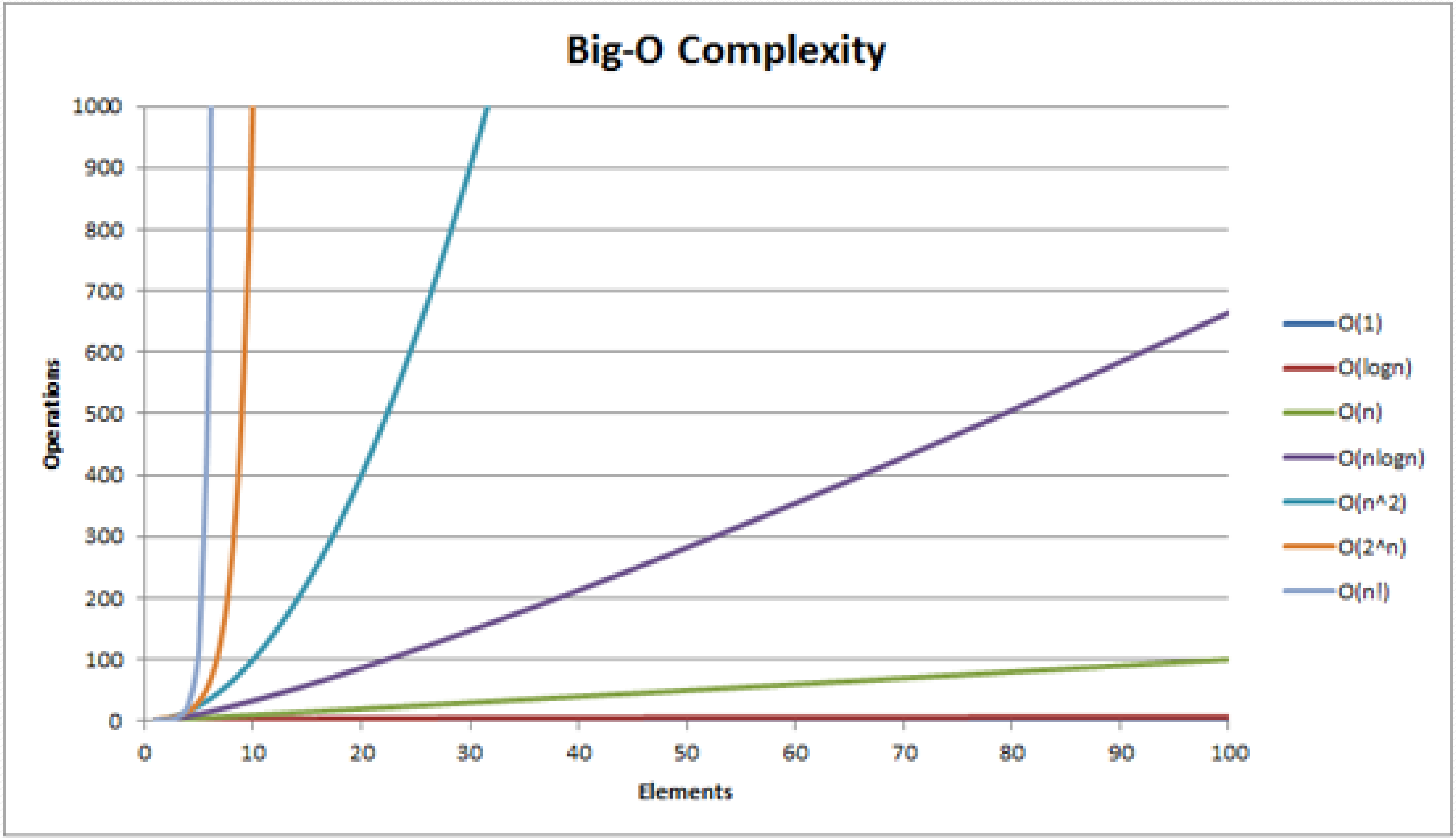
- Say an algorithm runs $\mathbf{0.4N^3 + 25N^2 + 8N + 17}$ statements.
  - Consider the runtime when N is *extremely large* .
  - We ignore constants like 25 because they are tiny next to N.
  - The highest-order term (N^3) dominates the overall runtime.
  - We say that this algorithm runs "on the order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

Max subsequence sum

- Write a method `maxSum` to find the largest sum of any contiguous subsequence in an array of integers.
  - Easy for all positives: include the whole array.
  - What if there are negatives?

index	0	1	2	3	4	5	6	7	8
value	2	1	-4	10	15	-2	22	-8	5

Largest sum: 10 + 15 + -2 + 22 = 45

  - (Let's define the max to be 0 if the array is entirely negative.)

- Ideas for algorithms?

Algorithm 1 pseudocode

```
maxSum(a):
    max = 0.
    for each starting index i:
        for each ending index j:
            sum = add the elements from a[i] to a[j].
            if sum > max,
                max = sum.

    return max.
```

index	0	1	2	3	4	5	6	7	8
value	2	1	-4	10	15	-2	22	-8	5

Algorithm 1 code

- What complexity class is this algorithm?
  - **O(N^3^)**. Takes a few seconds to process 2000 elements.

```
public static int maxSum1(int[] a) {
    int max = 0;
    for (int i = 0; i < a.length; i++) {
        for (int j = i; j < a.length; j++) {
            // sum = add the elements from a[i] to a[j].
            int sum = 0;
            for (int k = i; k <= j; k++) {
                sum += a[k];
            }
            if (sum > max) {
                max = sum;
            }
        }
    }
    return max;
}
```

Flaws in algorithm 1

- Observation: We are redundantly re-computing sums.
  - For example, we compute the sum between indexes 2 and 5:
    a[2] + a[3] + a[4] + a[5]
  - Next we compute the sum between indexes 2 and 6:
    a[2] + a[3] + a[4] + a[5] + a[6]
  - We already had computed the sum of 2-5, but we compute it again as part of the 2-6 computation.
  - Let's write an improved version that avoids this flaw.

index	0	1	2	3	4	5	6	7	8
value	2	1	-4	10	15	-2	22	-8	5

Algorithm 2 code

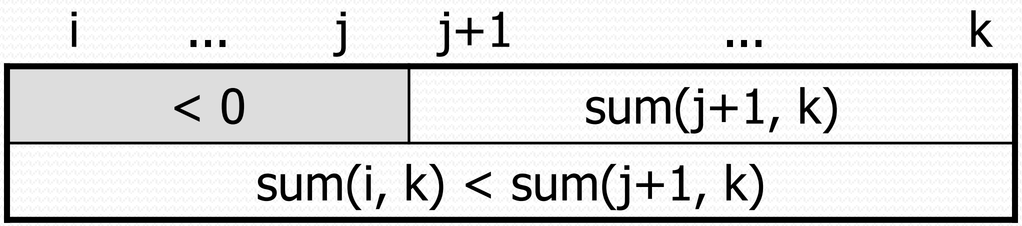
- What complexity class is this algorithm?
  - **O(N²)**. Can process tens of thousands of elements per second.

```
public static int maxSum2(int[] a) {
    int max = 0;
    for (int i = 0; i < a.length; i++) {
        int sum = 0;
        for (int j = i; j < a.length; j++) {
            sum += a[j];
            if (sum > max) {
                max = sum;
            }
        }
    }
    return max;
}
```

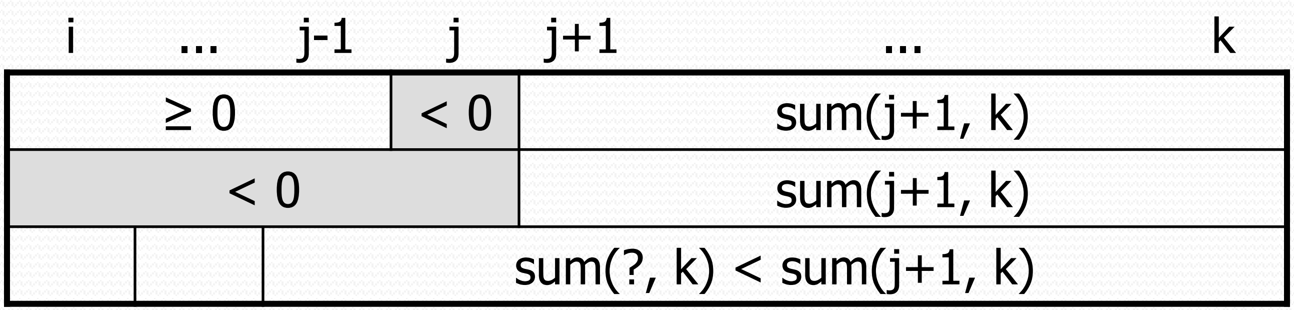
index	0	1	2	3	4	5	6	7	8
value	2	1	-4	10	15	-2	22	-8	5

A clever solution

- *Claim 1* : A max range cannot start with a negative-sum range.



- *Claim 2* : If sum(i, j-1) ≥ 0 and sum(i, j) < 0, any max range that ends at j+1 or higher cannot start at any of i through j.



  - Together, these observations lead to a very clever algorithm...

Algorithm 3 code

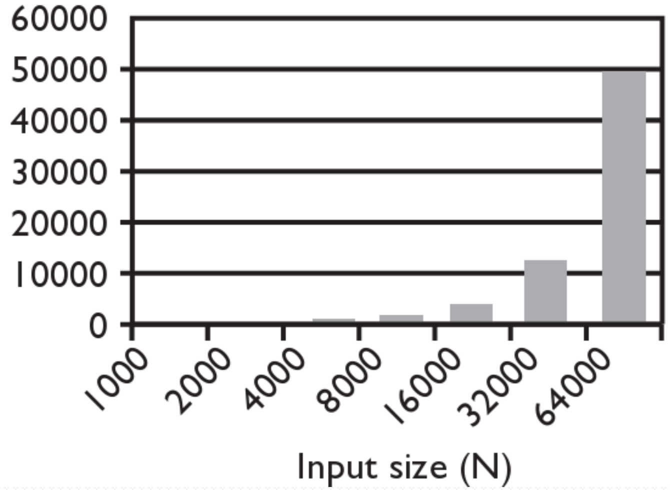
- What complexity class is this algorithm?
  - **O(N)**. Handles many millions of elements per second!

```
public static int maxSum3(int[] a) {
    int max = 0;
    int sum = 0;
    int i = 0;
    for (int j = 0; j < a.length; j++) {
        if (sum < 0) {    // if sum becomes negative, max range
            i = j;        // cannot start with any of i - j-1
            sum = 0;      // (Claim 2)
        }
        sum += a[j];
        if (sum > max) {
            max = sum;
        }
    }
    return max;
}
```

Runtime of first 2 versions

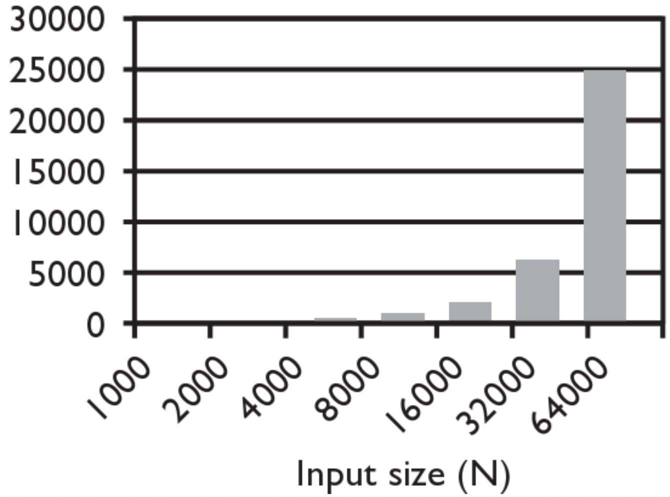
- Version 1:

N	Runtime (ms)
1000	15
2000	47
4000	203
8000	781
16000	3110
32000	12563
64000	49937



- Version 2:

N	Runtime (ms)
1000	16
2000	16
4000	110
8000	406
16000	1578
32000	6265
64000	25031



Runtime of 3rd version

- Version 3:

N	Runtime (ms)
1000	0
2000	0
4000	0
8000	0
16000	0
32000	0
64000	0
128000	0
256000	0
512000	0
1e6	0
2e6	16
4e6	31
8e6	47
1.67e7	94
3.3e7	188
6.5e7	453
1.3e8	797
2.6e8	1578