Review

- Big-oh is a way for us to quantify the running time (or space) of an algorithm.
- To measure the runtime as a function that tells us how fast/slow the alg runs as a func of the size of the input.

- $T(n)$: # of "basic operations" the alg runs.
  n ← input size

- Big-oh: gives us an upper bound on the growth of $T(n)$.

~~Let's show T(n) = 3n+4 = O(n)~~

$T(n) = 3n^2 + 4n + 2$

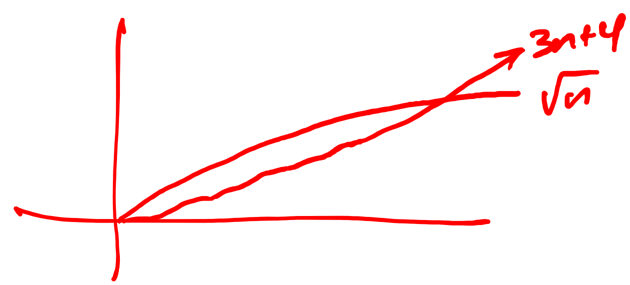
$T(n) = O(n)$ X FALSE

$T(n) \cancel{\le} c \cdot n$

True $3n+4 = O(2n)$?

True ~~False~~ $3n+4 = O(n^2)$?

False $3n+4 = O(\sqrt{n})$?



Rules of big-oh

1. Drop coefficients inside $O(\ldots)$ b/c the def'n of big-oh, includes a constant "c" already.

$O(\cancel{2}n) \rightarrow O(n)$ $\quad O(\cancel{3}n^2) \rightarrow O(n^2)$

2. If you have multiple terms added together inside the $O(\ldots)$ part, you keep only ~~on~~ the one that grows fastest.

$T(n) = 3n^2 + 2n$

$O(\cancel{3}n^2 + \cancel{2}n) \rightarrow O(\underline{n^2} + \cancel{n})$

$\rightarrow O(n^2)$

Rules of big-oh

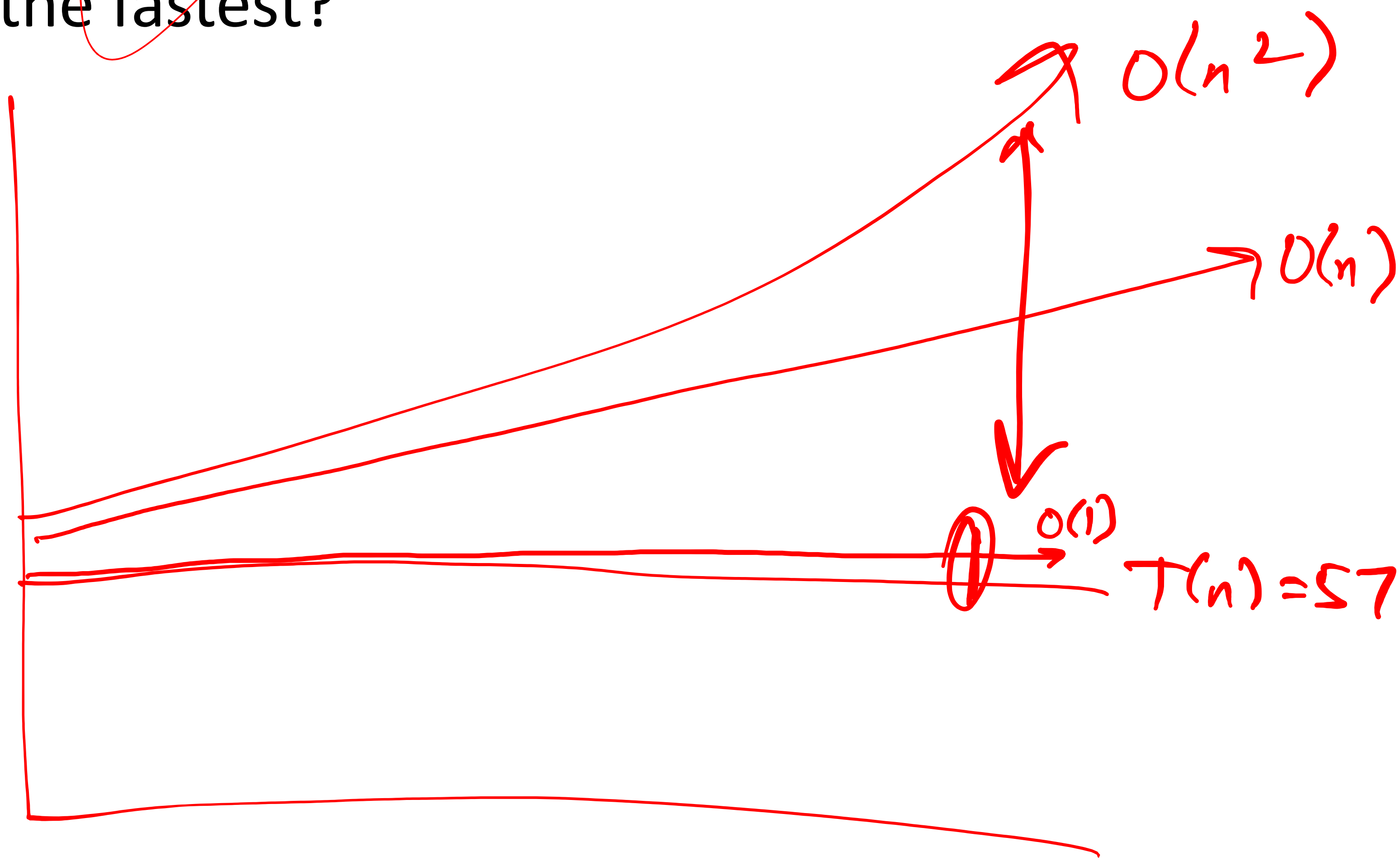
Why do we do this?

- Why not just use a stopwatch?
  - Every computer runs at a different speed.
  - Doesn't convey order of growth.

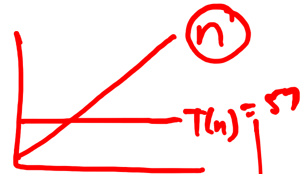
- Why not just report T(n) for an algorithm? → # of basic operations
  - "Basic operation" → vague

Why do we do this?

- Why drop coefficients and only keep the term that grows the fastest?



Examples



$T(n)$	$O(1)$	$O(n)$	$O(n^2)$
$T(n) = 57$	True	True	True
$T(n) = 50n + 10{,}000$	False	True	True
$T(n) = 10{,}000n^2 + 25n + 4000$	False	False	True

"Give the tightest big-oh bound possible"

Categories

$O(1)$ - constant time [$T(n) = 57$ (constant)]

$O(\log n)$ - logarithmic time

$O(n)$ - linear time [Double the size of the input, the time also doubles]

$O(n \cdot \log n)$ - linearithmic time / log-linear time

$O(n^2)$ - quadratic time [Double the input size, the time quadruples]

⋮

polynomials

"opposite"

Array of size 10 → 1 min

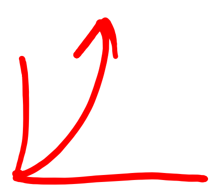
+ 1 → 2 min

$O(2^n)$ - exponential time

P-vs-NP problem

⋮

$O(n!)$ - factorial time



Graph (+ website)

Shortcuts

- You don't have to determine the exact T(n) for a section of code to compute big-oh. There are shortcuts.

figure out # of times the loop runs with respect to the input size

Loops: **(Example 2)**

```
for (int i = 0; i < n; i++) {
    System.out.println("Hello world!")
}
```

O(n)

Shortcuts

multiply their big-oh

Nested loops: **(Example 3)**

O(n)

O(n)

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n + 25; j++) {
        System.out.println("Hello world!")
    }
}
```

$O(n) \times O(n) \rightarrow O(n^2)$

Shortcuts

Add their big-oh times.

Consecutive Statements: **(Example 4)**

```
for (int i = 0; i < n; i++)
  a[i]=0;

for (int i = 0; i < n; i++) {
  for (int j = 0; j < n + 25; j++) {
    System.out.println("Hello world!")
  }
}
```

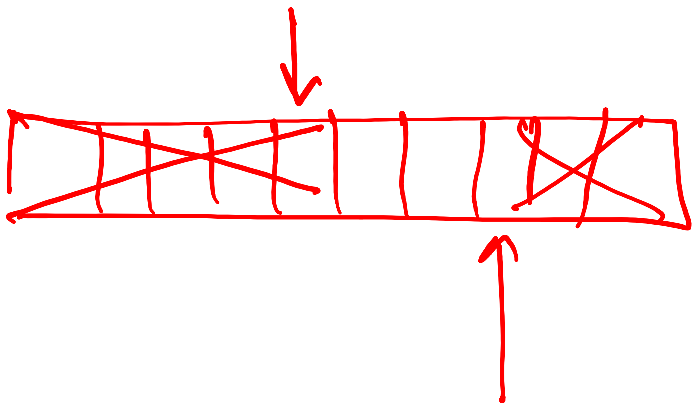
$O(n)$

$O(n^2)$

$O(n) + O(n^2) \longrightarrow O(n^2)$

Logarithmic time

- An algorithm takes logarithmic time --- **O(log n)** --- if it repeatedly cuts the size of the problem by a constant fraction (usually ½).
- Binary search is O(log n).



What is the tightest big-oh?

#1

```
sum=0;
for (int i=0; i<n; i++)
    sum++;
```

$O(n)$

#2

```
sum=0;
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        sum++;
```

$O(n^2)$

#3

```
sum=0;
for (int i=0; i<n; i++)
    for (int j=0; j<n*n; j++)
        sum++;
```

$O(n^3)$

$O(n^2)$

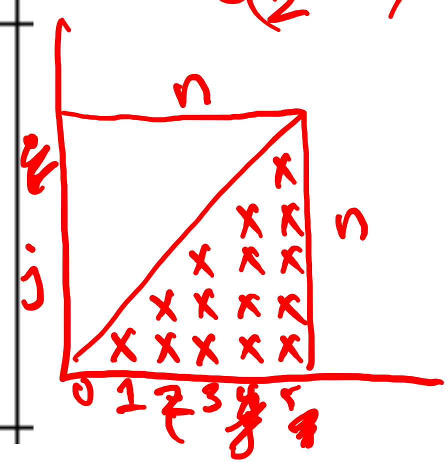
#4

```
sum=0;
for (int i=0; i<n; i++)
    for (int j=0; j<i; j++)
        sum++;
```

$O(n^2)$

n

$O(\frac{1}{2} n^2)$



#5

```
sum=0;
for (int i=0; i<n; i++)
      for (int j=0; j<i*i; j++)
            for (int k=0; k<j+100; k++)
                  sum++;
```

n, n^2, n^2

$O(n^5)$

$O(n^-)$

#6

```
x=n;
sum=0;
while(x>0){
      sum++;
      x=x/2;
}
```

$O(\log n)$

$8 \to 4 \to 2 \to 1 \to 0$

#7

```
sum=0;
for (int i=1; i<n; i*=2)
      sum++;
```

$O(\log n)$

$n = 8$

$\underbrace{i \quad 1 \to 2 \to 4 \to 8}_{\#\text{ of times}}$

#8

```
for (int i=0; i<n; i+=2){
      sum++;
}
for (int j=0; j<n; j++){
      sum++;
}
```

$O(\frac{1}{2}n)$

$O(n)$

$\to O(n)$