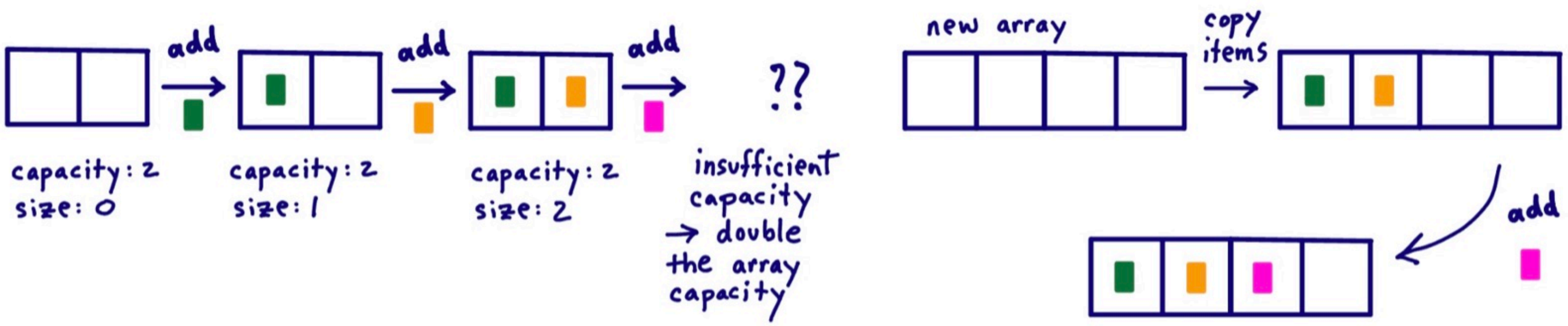
Middlebury

CSCI 201: Data Structures

Spring 2025

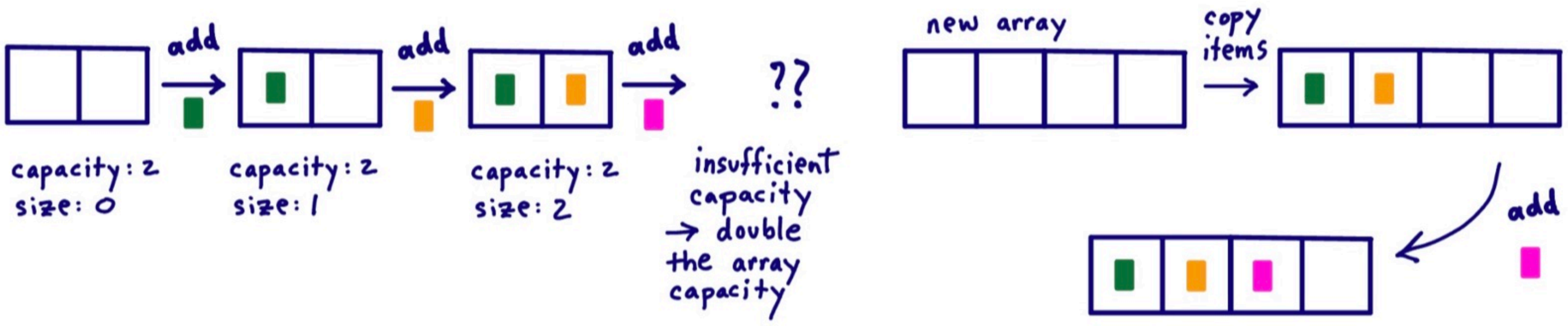
Lecture 4M: Complexity

Remember our decision to *double* the capacity of a `DIYList` when we ran out of space during a call to `add`?



Goals for today:

- Analyze the runtime cost of our `add` method for a `DIYList` as we call it many times.
- Characterize how functions grow as the inputs get very large.
- Use big-O notation to describe the running time of algorithms.



We also want our analysis to be computer-independent.

1 sec

vs

10 sec

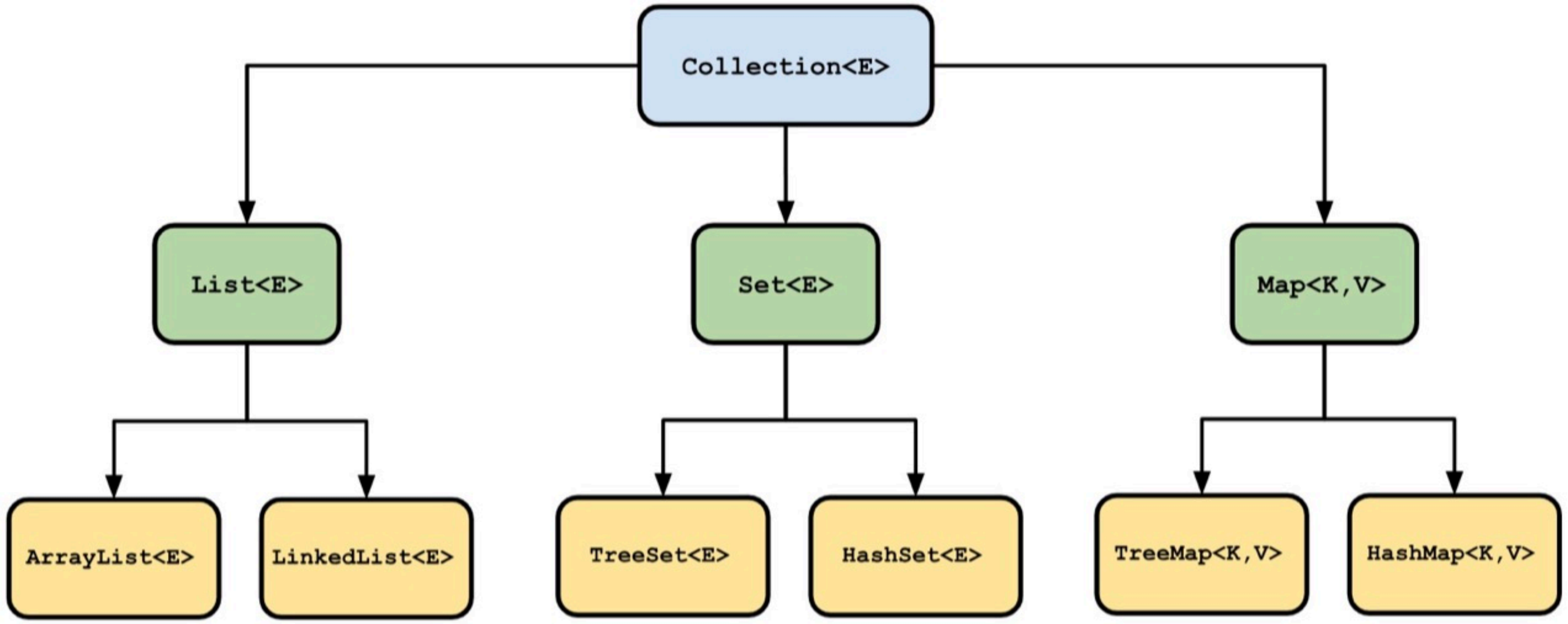
2 GHz

Two types of resources to consider:

- **Processor cycles:** number of operations per second a machine can perform.
- **Memory:** space for storing data while program is running (RAM, cache).

16 GB

The `Collections` framework describes the efficiency an implemented method should provide.



The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. The `add` operation runs in amortized constant time, that is, adding n elements requires $O(n)$ time. All of the other operations run in linear time (roughly speaking).

What kinds of things in our programs might affect the runtime?

for-loops, while-loops

conditionals

function calls

assignment =

comparison <, <=, >, >=, ==, !=

logical &&, ||

arithmetic +, -, *, /, ++, --

Analyzing how many = we're doing in the add method when *doubling* the capacity (as needed). Assume we start with a capacity of 1.

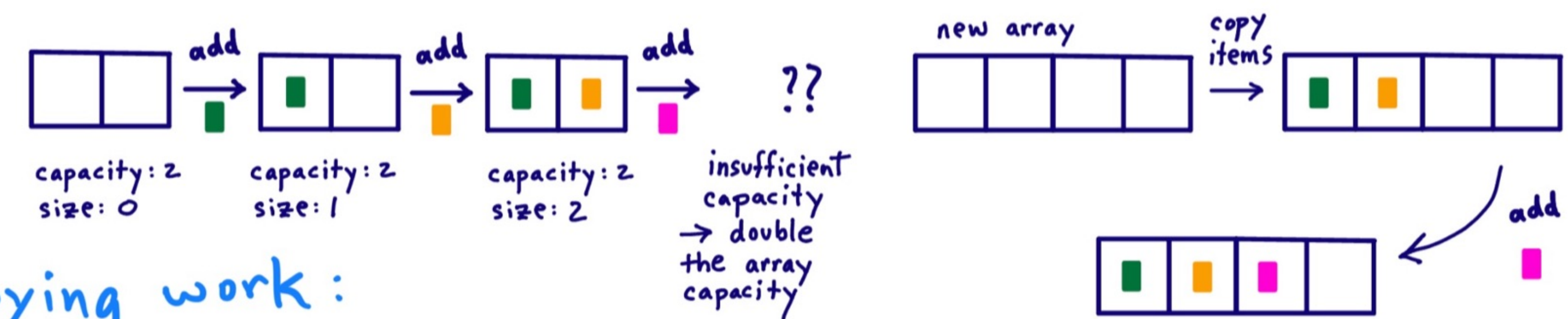
Define
n = # values
m = # resizings

```
public class DIYList {
  int size; // current number of items actually stored
  String[] items; // capacity is items.length

  public void add(String item) {
    // Is there enough space (capacity, i.e. items.length)?
    // If not, make more space and copy the old items.

    // Place item in items[size] and increment size.
  }
}
```

Doubling means $n = 2^m + 1$ so $2^m = n - 1$



Copying work:

$1 + 2^1 + 2^2 + 2^3 + \ldots + 2^m$

$= \frac{2^{m+1} - 1}{2 - 1} = 2^{m+1} - 1 = 2 \cdot 2^m - 1$

$= 2(n-1) - 1 = 2n - 3$

Adding n new values is additional n steps

$\Rightarrow$ total $3n - 3$

Two types of series we'll encounter:

1. **Geometric:**

$$1 + r + r^2 + r^3 + \dots + r^m = \frac{r^{m+1} - 1}{r - 1}$$

Here, $r = 2$

2. **Arithmetic:**

$$1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$$

Derive in CS 200!

So the total number of = when calling add n times is:

$$3n-3$$

```java
public class DIYList {
  int size; // current number of items actually stored
  String[] items; // capacity is items.length

  public void add(String item) {
    // Is there enough space (capacity, i.e. items.length)?
    // If not, make more space and copy the old items.

    // Place item in items[size] and increment size.
  }
}
```

Averaged over n calls to add (with n getting very large):

$$\frac{3n-3}{n} = 3 - \frac{3}{n} \quad \text{as } n \to \infty, \ \frac{3}{n} \to 0$$

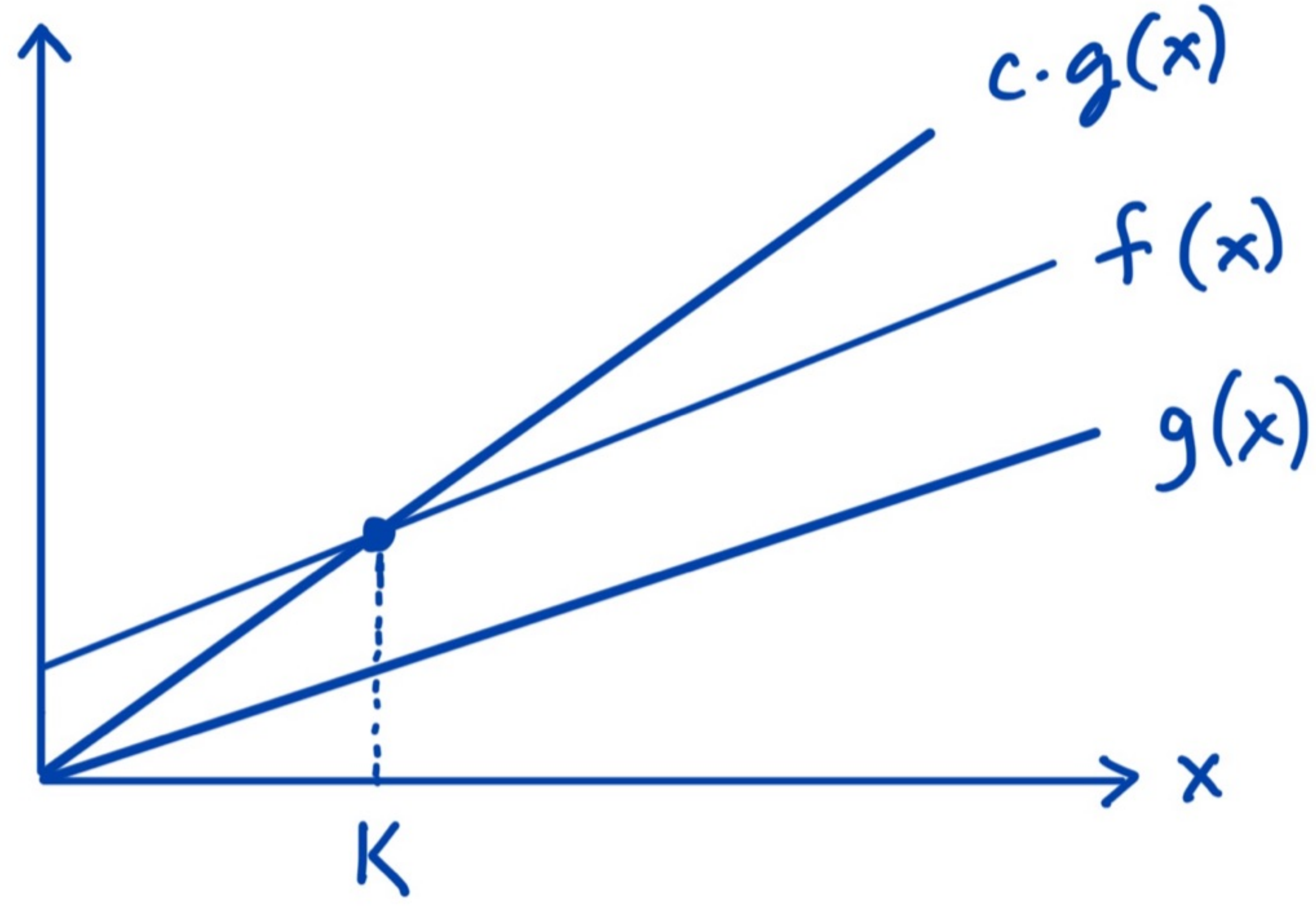
$$\boxed{= \ 3}$$

← constant time amortized over n adds

We need a better way to analyze running time of algorithms.

Big-O notation: Given functions $f,\ g$, we say that $f(x)$ is $\mathcal{O}(g(x))$ if-and-only-if there *exist* constants $c > 0$ and k such that

$$|f(x)| \leq c \cdot |g(x)|, \quad \text{for all } \ x \geq k$$



f(x) is eventually no larger than some constant multiple of g(x)

Example: Show that $x^2 + 2x + 1$ is $\mathcal{O}(x^2)$.

Show $x^2 + 2x + 1 \leq c \cdot x^2$

Rewrite as $(c-1)x^2 - 2x - 1 \geq 0$

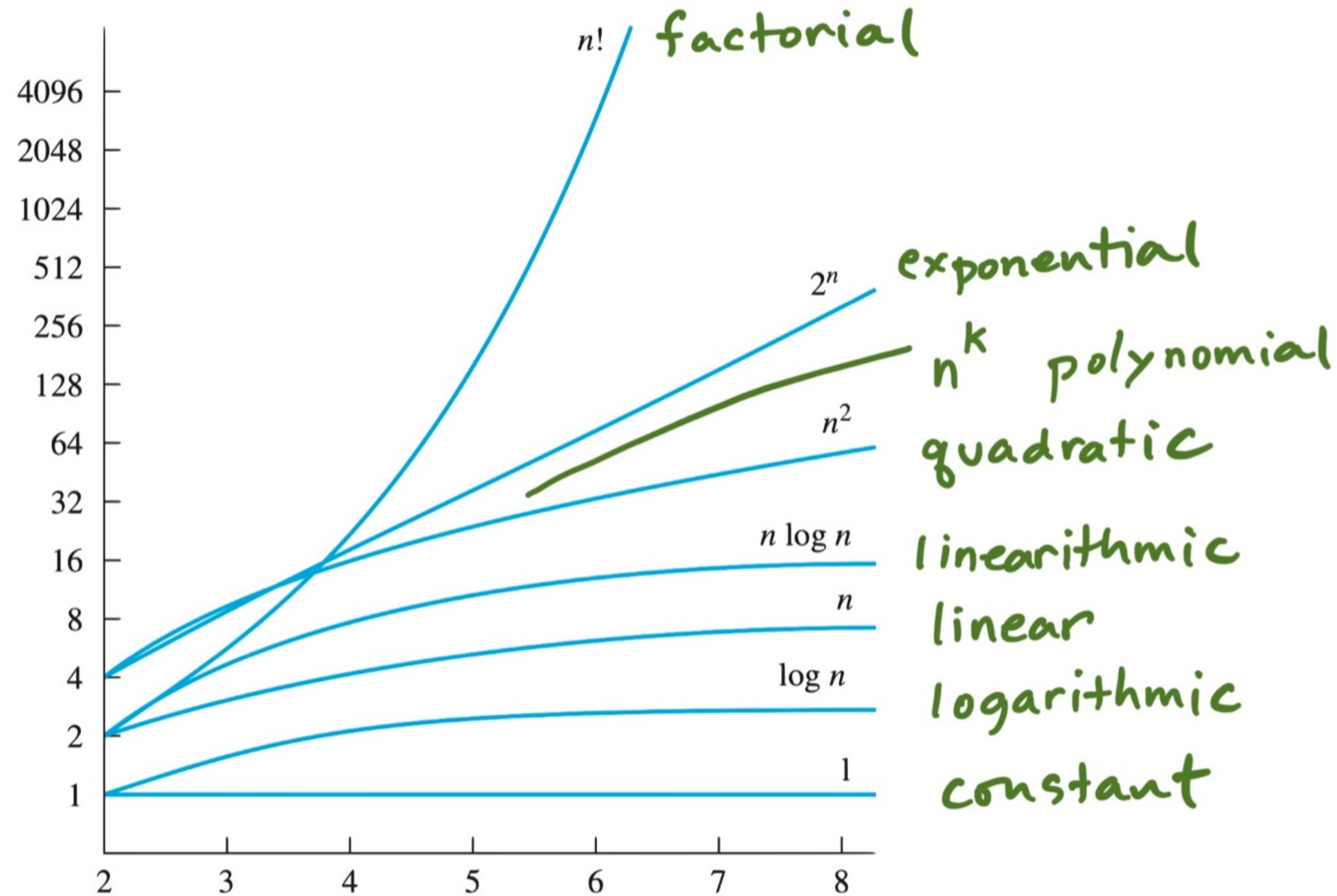
Choose $c = 2 \Rightarrow$ show $x^2 - 2x - 1 \geq 0$

Find K: try $x = 2$: $(2)^2 - 2(2) - 1 = -1 \geq 0$? no

try $x = 3$: $(3)^2 - 2(3) - 1 = 2 \geq 0$? yes

So constants $c = 2$ and $k = 3$
work to show $x^2 + 2x + 1$ is $O(x^2)$

Common functions used in big-O estimates.



(Discrete Mathematics and Its Applications 7th Ed., Rosen)

We usually want to express our algorithm runtime using the *tightest bound*.

We'll often use $T(n)$ to represent algorithm runtime in terms of input size n.

Strategy:

1. Pick out fastest growing term in $T(n)$.
2. Drop coefficients.

Eg, $1 + 100n^2$: $O(n)$? no

tightest → $O(n^2)$? yes

$O(n^3)$? yes

Determine a big-O bound for the following functions.

1. $T(n) = \cancel{1} + \cancel{5}n$: $O(n)$
2. $T(n) = \cancel{1} + \cancel{5}n^2$: $O(n^2)$
3. $T(n) = \cancel{5} + \cancel{20n} + \cancel{3}n^2$: $O(n^2)$
4. $T(n) = \frac{n^2(n^2+1)}{\cancel{2}}$: $(n^4 + \cancel{n^2}) / \cancel{2}$: $O(n^4)$
5. $T(n) = 5$: $O(1)$
6. $T(n) = n(5 + \log n)$: $\cancel{5n} + n \log n$: $O(n \log n)$

A few rules for inferring big-O bounds on algorithm runtime.

consecutive statements: $T(n) = T(s_1) + T(s_2)$

```
statement1; // performing T(s1) amount of work
statement2; // performing T(s2) amount of work
```

for loop: $T(n) = n \times T(b)$.

```
for (int i = 0; i < n; i++) {
  // some block performing T(b) amount of work
}
```

nested for loop: $T(n, m) = n \times m \times T(b)$.

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < m; j++) {
    // some block performing T(b) amount of work
  }
}
```

if statements: $T(n) = T(c) + \max(T(b_i), T(b_e))$

```
if (condition) { // condition performs T(c) amount of work
  body1; // performing T(bi) amount of work
} else {
  body2; // performing T(be) amount of work
}
```

Determine $T(n)$ (an expression for the number of operations performed by the following algorithms), then provide a big-O bound on $T(n)$.

Focus on counting ++

Example 1:

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

n

$n \cdot m$

$n \cdot m$

$T(n,m) = 2nm + n$

$O(nm)$

Eg, take $n = 32$

So $i = 32, 16, 8, 4, 2, 1$

Loop runs 6 times

Example 2:

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

n

n^2

$n^2 m$

$\cancel{2}n^2m + \cancel{n^2} + \cancel{n}$

$O(n^2m)$

Example 4:

```
int i = n;
while (i >= 1) {
  i = i / 2;
}
```

$O(\log n)$

Example 3:

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

$$\begin{array}{l} i=0 \\ i=1 \\ i=2 \\ i=3 \\ \vdots \\ i=n-1 \end{array} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ \vdots \\ n \end{bmatrix}$$

add these

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

Describe the worst-case runtime complexity of the following methods in Big-O.
Use variables to describe complexity of data structures, eg, n = nums.size(), n = list1.size(), and m = list2.size()

Complexity of isEven: $O(1)$

```
public static boolean isEven(int n) {
  return (n % 2 == 0);
}
```

1. Complexity of isPrime: $O(n)$

```
public static boolean isPrime(int n) {
  if (n < 2) {
    return false;
  }
  for (int i = 2; i < n; i++) {
    if (n % i == 0) return false;
  }
  return true;
}
```

2. Complexity of factorialOf: $O(n)$

```
public static long factorialOf(int n) {
  switch(n) {
    case 0:
    case 1: return 1;
    default: return n * factorialOf(n - 1);
  }
}
```

3. Complexity of sumProduct: $O(n^2)$

```
public static Integer sumProduct(ArrayList nums) {
  int sum = 0;
  for (int i = 0; i < nums.size(); i++) {
    for (int j = 0; j < nums.size(); j++) {
      if (i != j) {
        sum += nums.get(i) * nums.get(j);
      }
    }
  }
  return sum;
}
```

4. Complexity of checking: $O(nm)$

```
public static boolean checking(ArrayList list1, ArrayList list2) {
  int value1;
  int value2;
  for (int i = 0; i < list1.size(); i++) {
    value1 = list1.get(i);
    for (int j = 0; j < list2.size(); j++) {
      value2 = list2.get(j);
      if (value1 == value2) return true;
    }
  }
  return false;
}
```

5. Complexity of lastElement: $O(1)$

```
public static Integer lastElement(ArrayList nums) {
  if (nums.size() == 0) return null;
  else return nums.get(nums.size()-1);
}
```

6. Complexity of createPairs: $O(n^2)$

```
public static void createPairs(ArrayList nums) {
  for (int i = 0; i < nums.size(); i++) {
    for (int j = i+1; j < nums.size(); j++) {
      System.out.println(nums.get(i) + ", " + nums.get(j));
    }
  }
}
```

See you on Wednesday!

- We'll use what we covered today to analyze some sorting algorithms.
- Work on Homework 3! Implement your own `DIYArrayListString`.
- Reminder that Noah (go/noah) and Smith (go/smith) have office hours throughout the week and the 201 Course Assistants have drop-in hours in the late afternoons/evenings (go/cshelp).