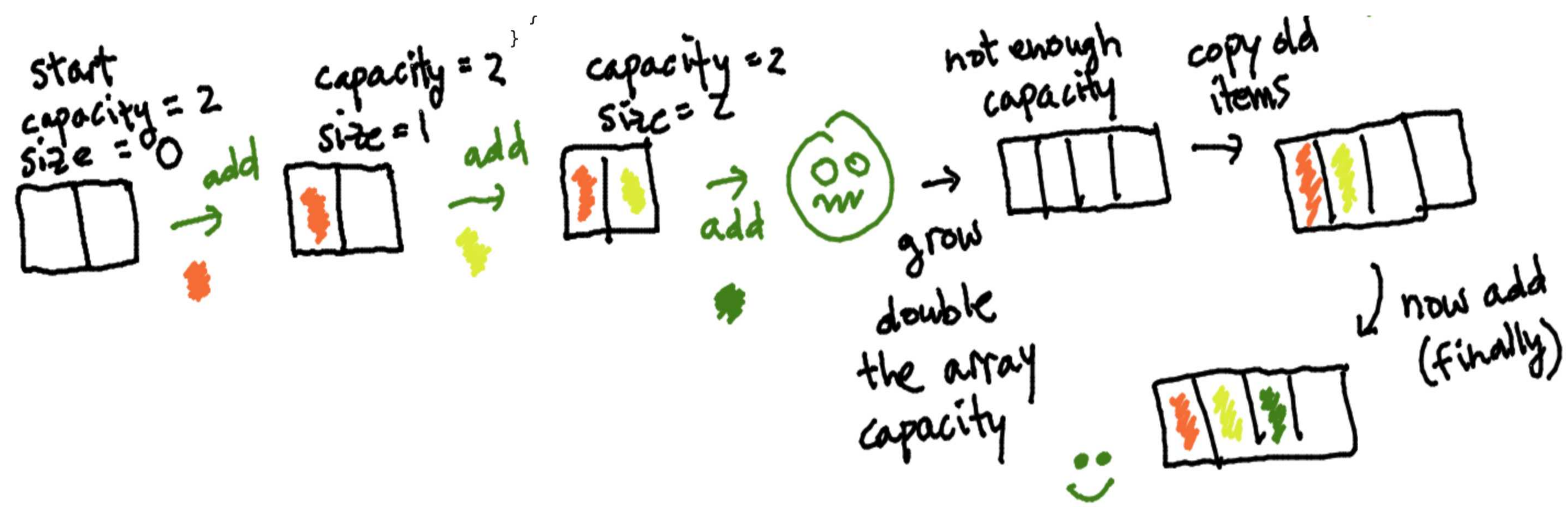
Middlebury

CSCI 201: Data Structures

Fall 2024

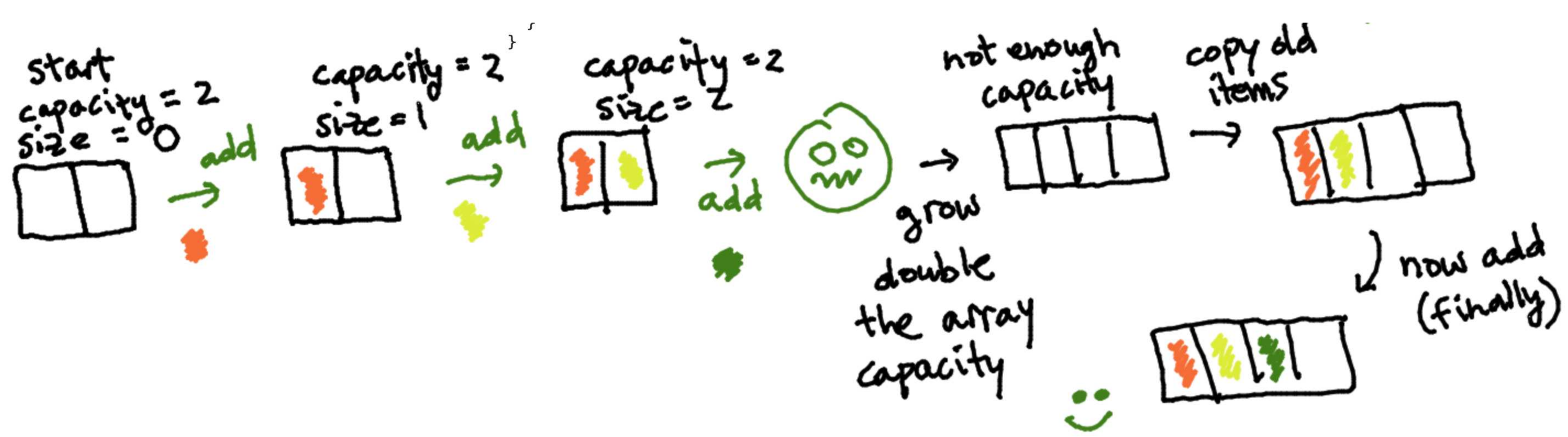
Lecture 4T: Complexity

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 really big.
- Use big-oh notation to describe the running time of algorithms.



Why? Why? Why?

Analyzing our codes will help us put the *SCIENCE* in computer science.

more in CS 200
(math foundations)

We also want our analysis to be computer-independent.

1 sec.

vs

10 sec.

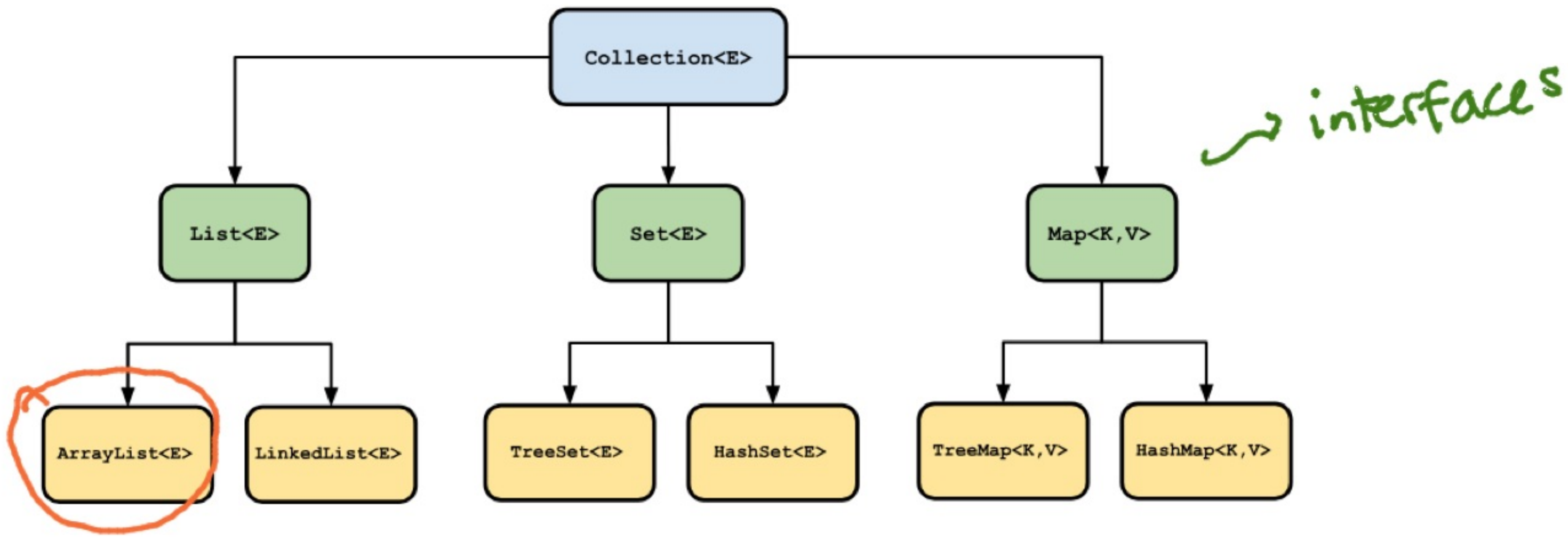
2 GHz

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

if-statements

comparisons: <, <=, >, >=, ==, !=

logical: &&, ||

assignments: =

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

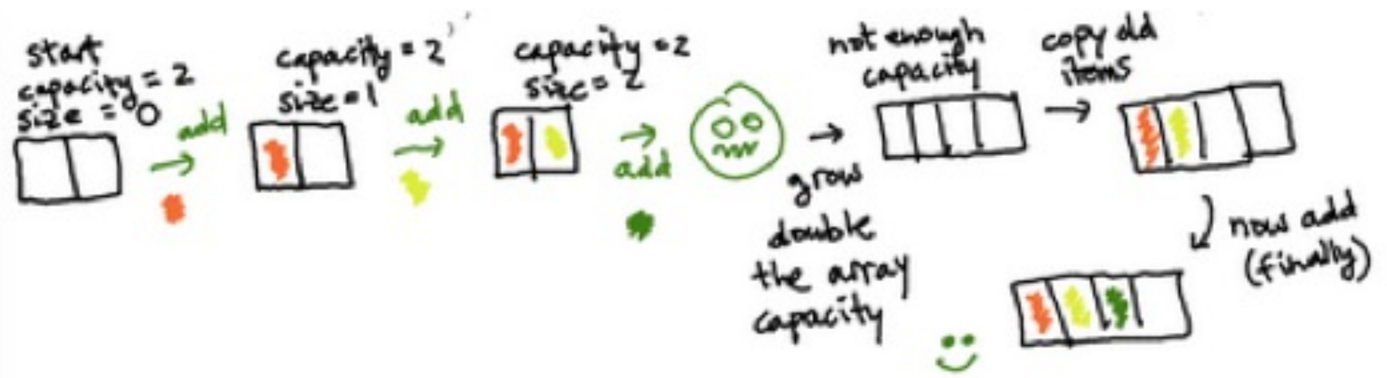
Analyzing how many **=** we're doing in the **add** method when *doubling* the capacity (as needed). Assume we start with a capacity of **1**.

```
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.
  }
}
```

what about this? (when copying)

items[size] = item

(a) called n times



$2^m = n - 1$

relationship between # resizes (m) and n? $n = 2^m + 1$

$1 + 2^1 + 2^2 + 2^3 + \dots + 2^m$] geometric series

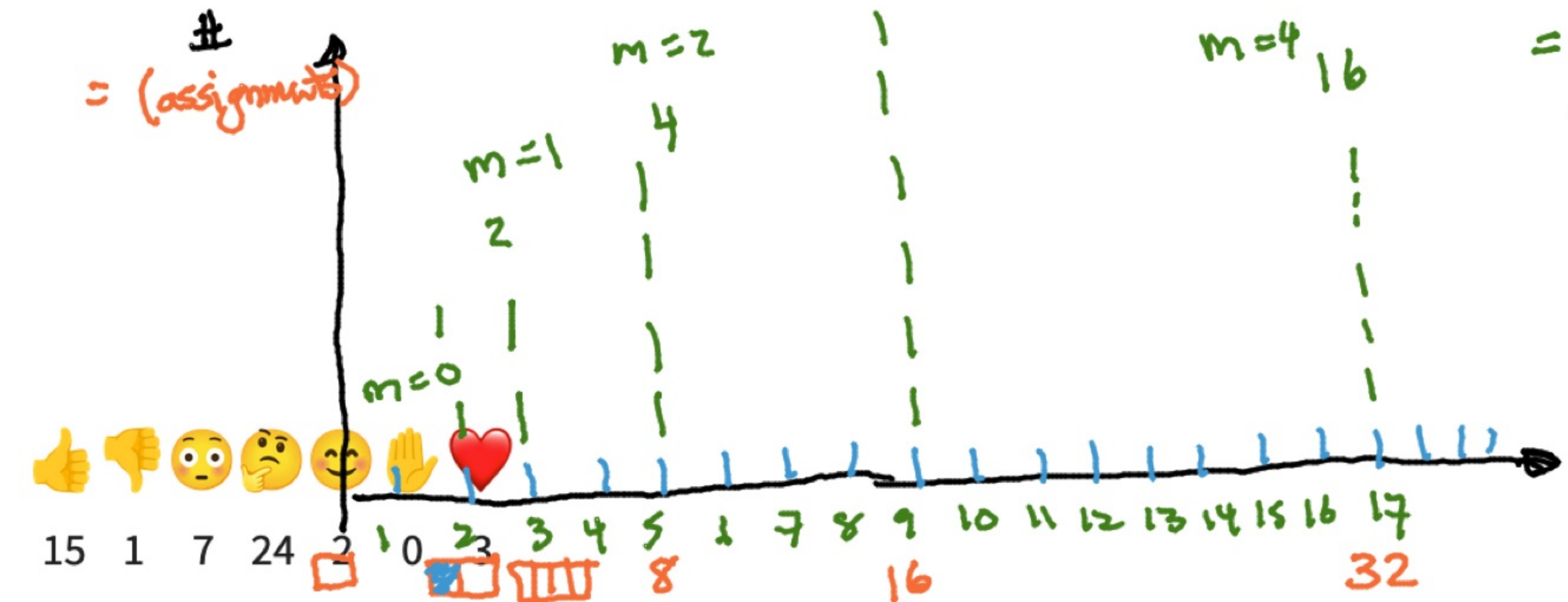
$= \frac{2^{m+1} - 1}{2 - 1} = 2^{m+1} - 1 = 2 \cdot 2^m - 1$

$= 2(n-1) - 1$

\# = for copy $= 2n - 3$

\# = for (a) $= n$

$\boxed{\text{total} = 3n - 3}$



Two types of series we'll encounter:

1. **Geometric:** $1 + r + r^2 + r^3 + \ldots + r^m = \dfrac{r^{m+1} - 1}{r - 1}$

here, $r = 2$

2. **Arithmetic:** $1 + 2 + 3 + 4 + \ldots + n = \dfrac{n(n+1)}{2}$

derived in CS 200.

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

$3n - 3$

comparison +1

```
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.
  }
}
```

3? 4? 5?

size++ +1 + n over all n calls

Averaged over n calls to add (with n getting really big):

$\frac{3n-3}{n} = 3 - \frac{3}{n}$ as $n \to \infty$ (big)

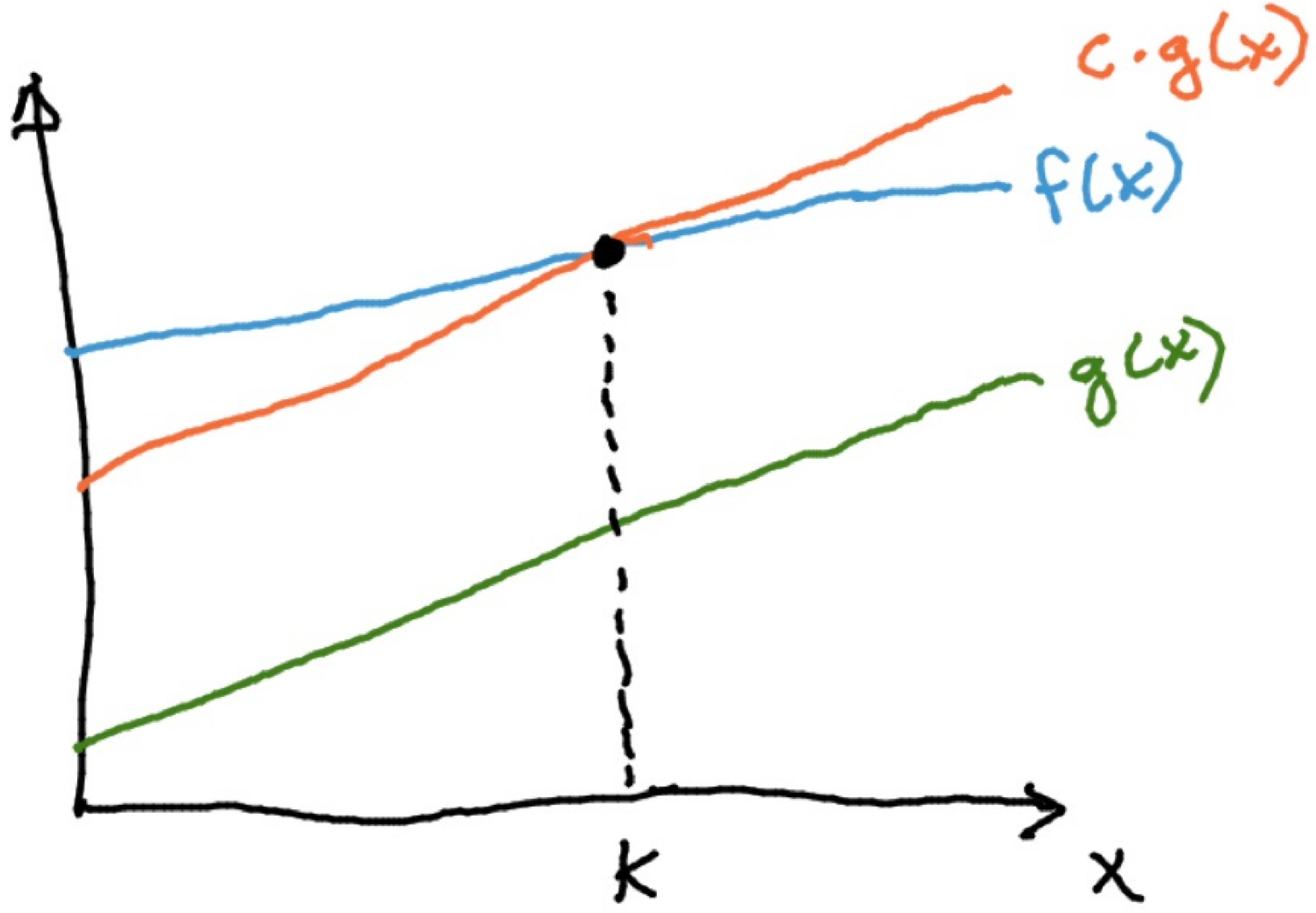
$\frac{3}{n} \to 0$

$\boxed{= 3}$

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

big-oh 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

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



in words: " 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: $x^2 + 2x + 1 \leq c \cdot x^2$ (dropping absolute value, assume $x > 0$)

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

pick c: let $c = 2$

$x^2 - 2x - 1 \geq 0$

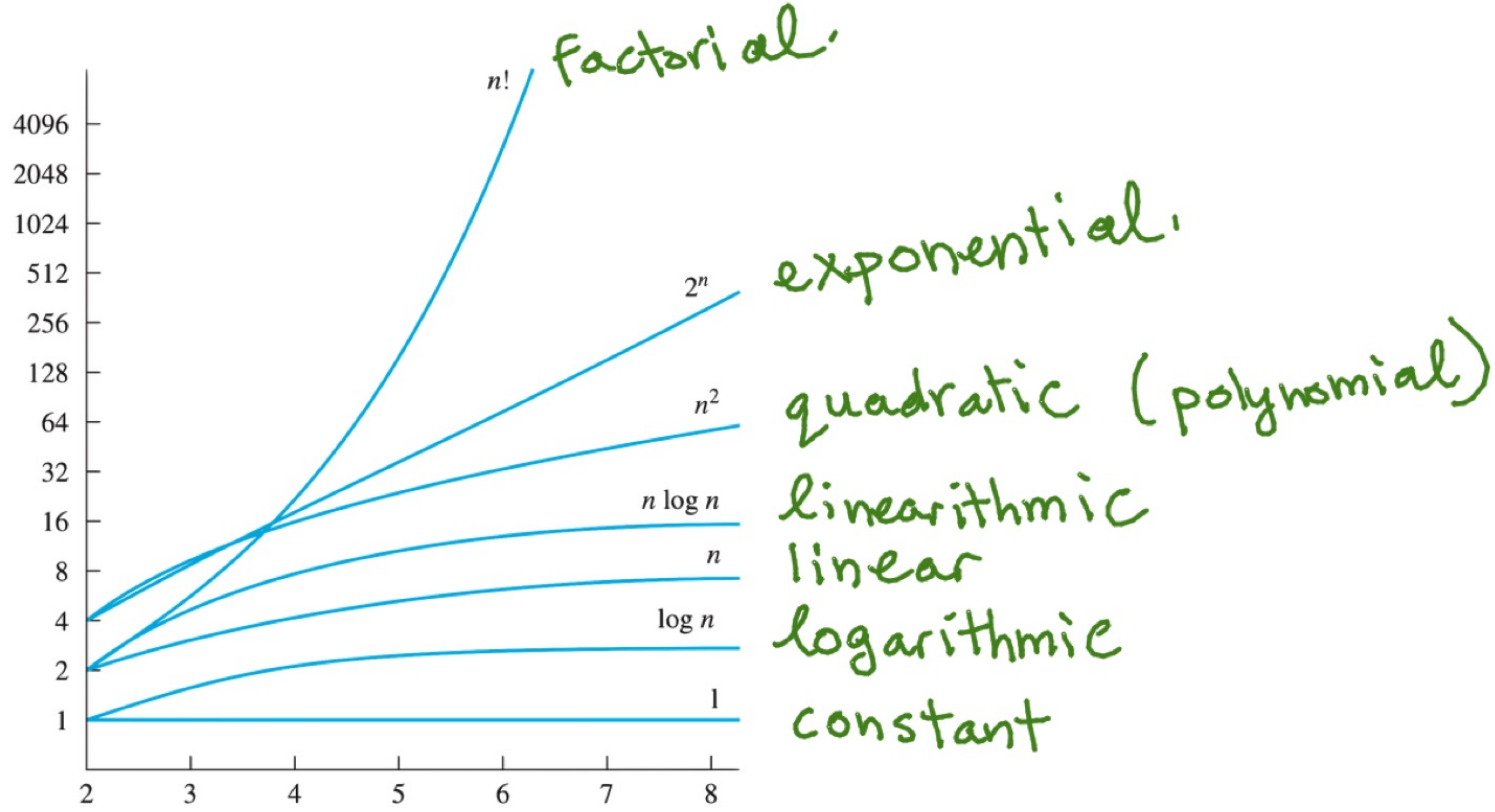
find x (k): try $x = 1$: $(1)^2 - 2(1) - 1 = -2 \geq 0$? no

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

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

:) found c, k

Common functions used in big-oh 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.

e.g. $1 + 100n^2$

$O(n)$? no

$O(n^3)$? yes but not tightest bound.

$\boxed{O(n^2)}$? yes, and lowest bound

Exercises: determine a big-oh bound for the following functions.

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

A few rules for inferring big-oh 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
}
```

focus on counting ++

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

Example 1:

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

n

$n \times m \times 1$ $\quad$ $n \times m$

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

$O(nm)$

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 $\quad$ n^2 $\quad$ n^2

n^2m $\quad$ n^2m $\quad$ oops

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

$O(n^2m)$

Example 3:

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

$i=0$	1
$i=1$	2
$i=2$	3
$i=3$	4
⋮	
$i=n-1$	n

add these together (arithmetic series)

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

$O(n^2)$

👍 👎 😳 🤔 😊 ✋ ❤️

15 0 0 14 8 0 7

See you on Thursday!

- We'll use what we covered today to analyze some sorting algorithms.
- Get started 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).
- Submit exit ticket 4T today.