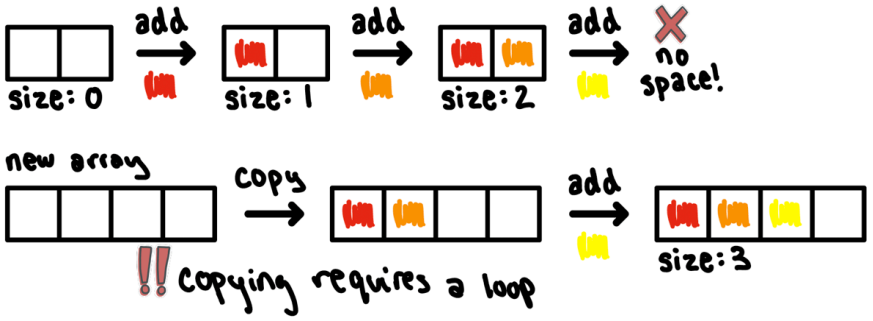
Middlebury

CSCI 201: Data Structures

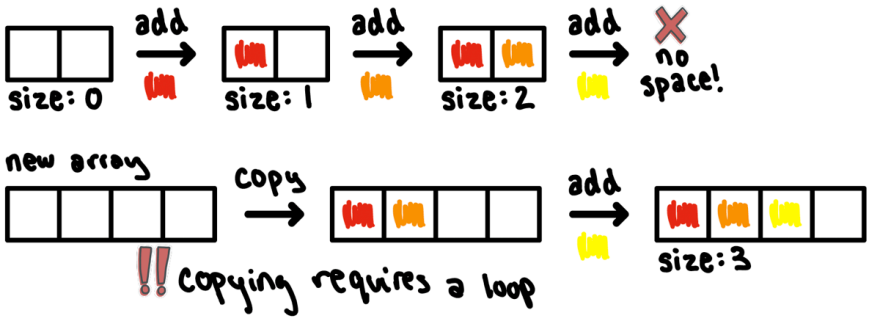
Fall 2025

Lecture 4T: Complexity

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

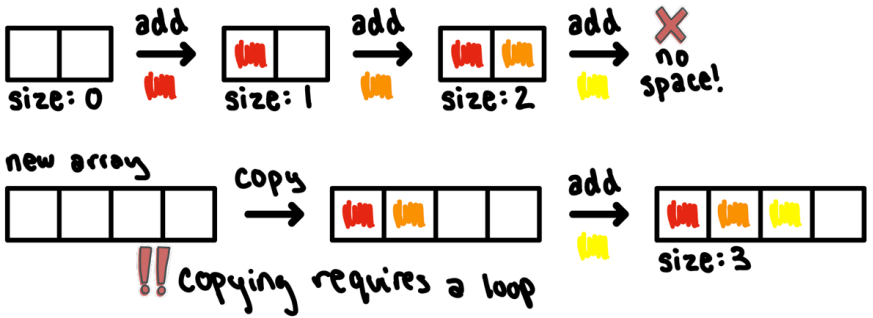


Goals for today:



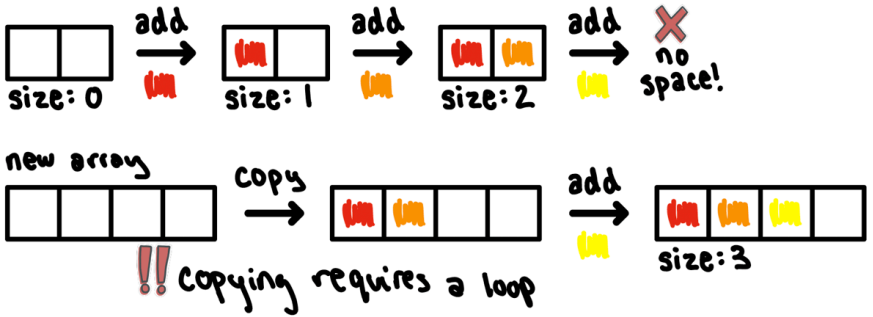
Goals for today:

- Analyze the runtime cost of our `add` method for a `DIYList` as we call it many times.



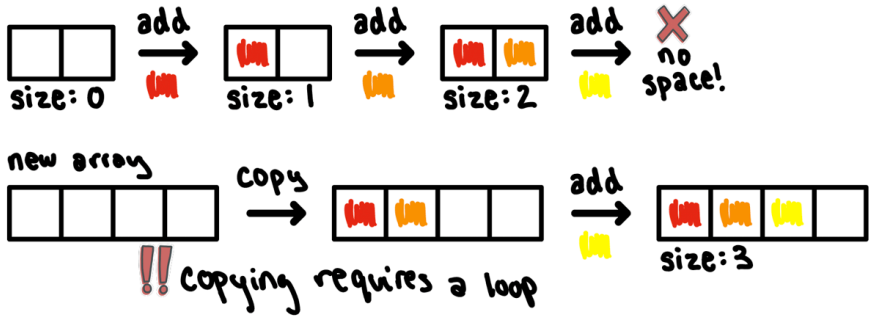
Goals for today:

- 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.



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.

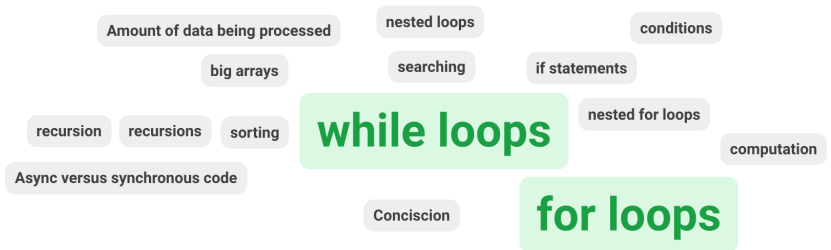
20 sec

Two types of resources to consider:

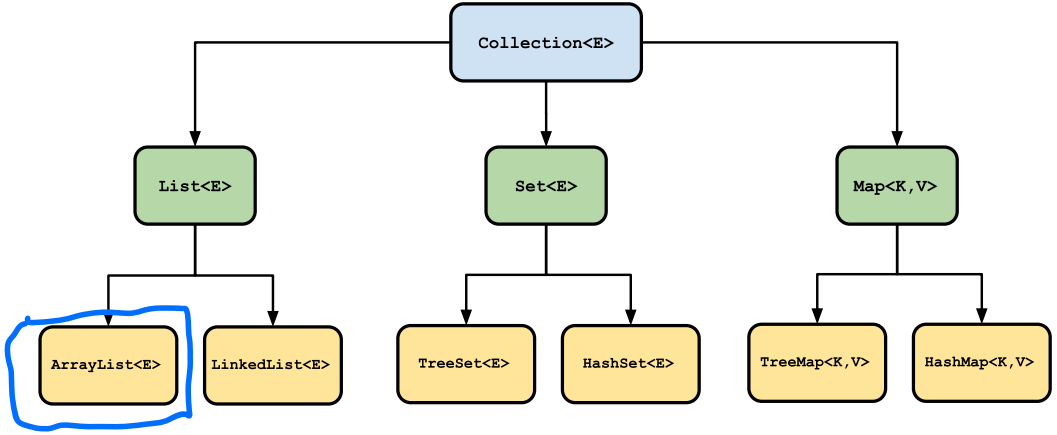
- **Processor cycles:** number of operations per second a machine can perform.
- **Memory:** space for storing data while program is running (RAM, cache).

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

Answer on sli.do - find the link on the course schedule!

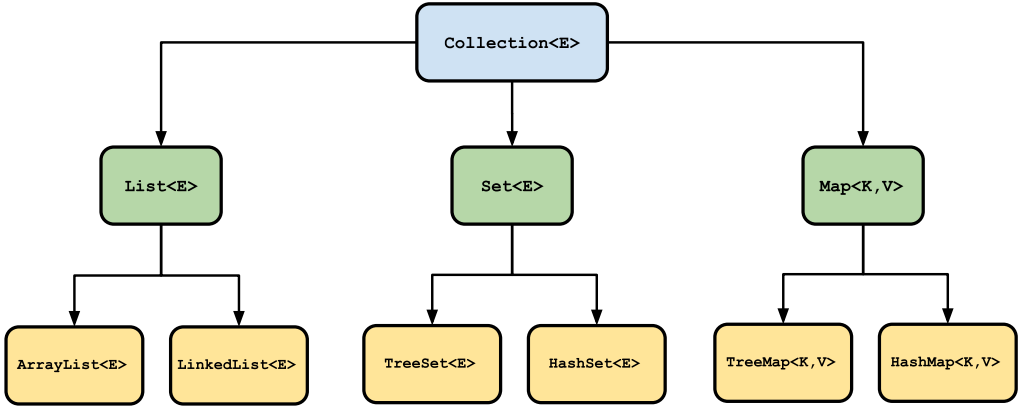


The `Collections` framework describes the efficiency an implemented method should provide.



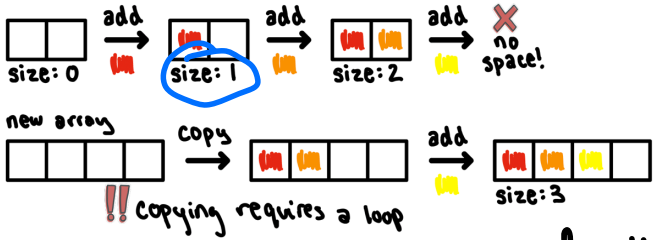
The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. The `add` operation runs in amortized constant time, that is, adding n elements requires $O(n)$ time. All of the other operations run in linear time (roughly speaking).

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).

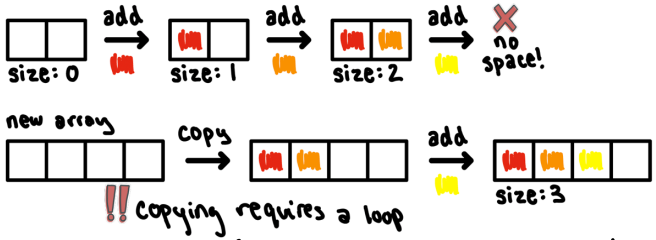
Analyzing how many = we're doing in the **add** method when *doubling* the capacity (as needed). Assume we start with a capacity of **1**.



Q0: How many = when we don't have to copy?
(adding n items)

n 2n

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



Q1: Each time we copy, is there a pattern that describes the # of items we will copy?

1, 2, 4, 8 $2^0, 2^1, 2^2, 2^3$

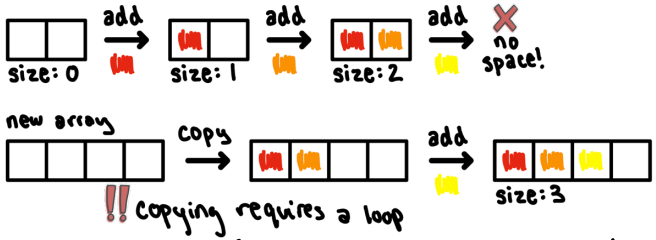
Q2: When we resize for the mth time, how many elements have we added?

$n = 2^m + 1$ $2^m = n - 1$

m=0, n=2
m=1, n=3
m=2, n=5
m=3, n=9

m	n	capacity	doubled?	= to copy
	1	1	✗	
0	2	2	✓	1
1	3	4	✓	2
	4	4	✗	
2	5	8	✓	4
	6	8	✗	
	7	8	✗	
	8	8	✗	
3	9	16	✓	8
	10	16	✗	

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



Q1: Each time we copy, is there a pattern that describes the # of items we will copy?

$1, 2, 4, 8$ $\quad 2^0, 2^1, 2^2, 2^3$

Q2: When we resize for the mth time, how many elements have we added?

$n = 2^m + 1 \quad 2^m = n-1$

Q3: How many = are we doing when we copy?

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

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

$= 2n - 3$

m	n	capacity	doubled?	= to copy
	1	1	✗	
0	2	2	✓	1
1	3	4	✓	2
	4	4	✗	
2	5	8	✓	4
	6	8	✗	
	7	8	✗	
	8	8	✗	
3	9	16	✓	8
	10	16	✗	

Two types of series we'll encounter:

1. **Geometric:**

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

Ex. $r = 2, m = 3$

$1 + 2 + 2^2 + 2^3 = 1 + 2 + 4 + 8 = 15$ $\qquad \frac{2^4 - 1}{2 - 1} = 16 - 1 = 15$

2. **Arithmetic:**

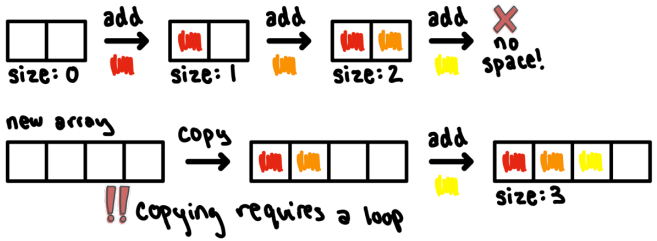
$$1 + 2 + 3 + 4 + 5 + \ldots + n = \frac{n(n+1)}{2}$$

Ex. $n = 5$

$1 + 2 + 3 + 4 + 5 = 15$ $\qquad \frac{n(n+1)}{2} = \frac{5(6)}{2} = 15$

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

$$n + 2n - 3 = 3n - 3$$



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

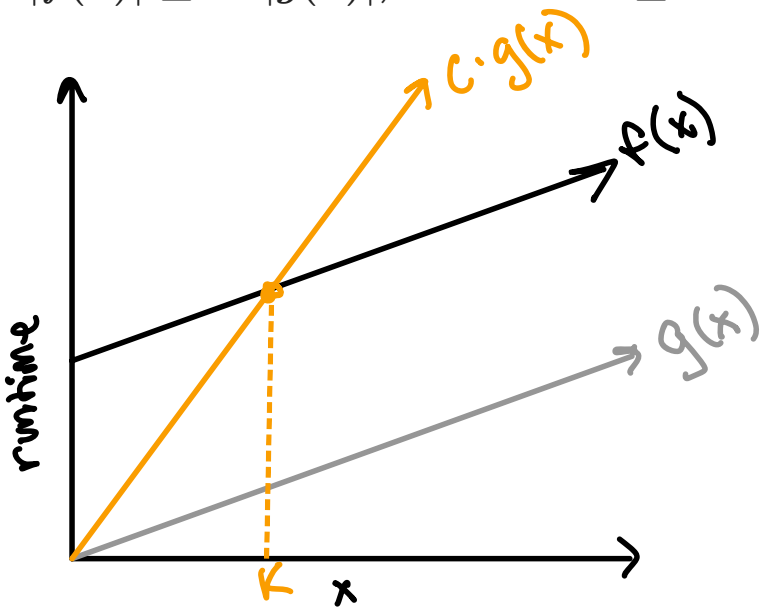
$$\frac{3n-3}{n} = \frac{3n}{n} - \frac{3}{n} = 3 - \boxed{\frac{3}{n}}$$

$$\approx 3$$

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

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



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

show $x^2 + 2x + 1 \leq c \cdot x^2$ for all $x \geq k$

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

$x \cdot x$ vs. $x \cdot 2$

choose $c = 2 \Rightarrow$ we must show

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

find k: try 2

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

$4 - 4 - 1 \geq 0$

$-1 \geq 0$ ✗

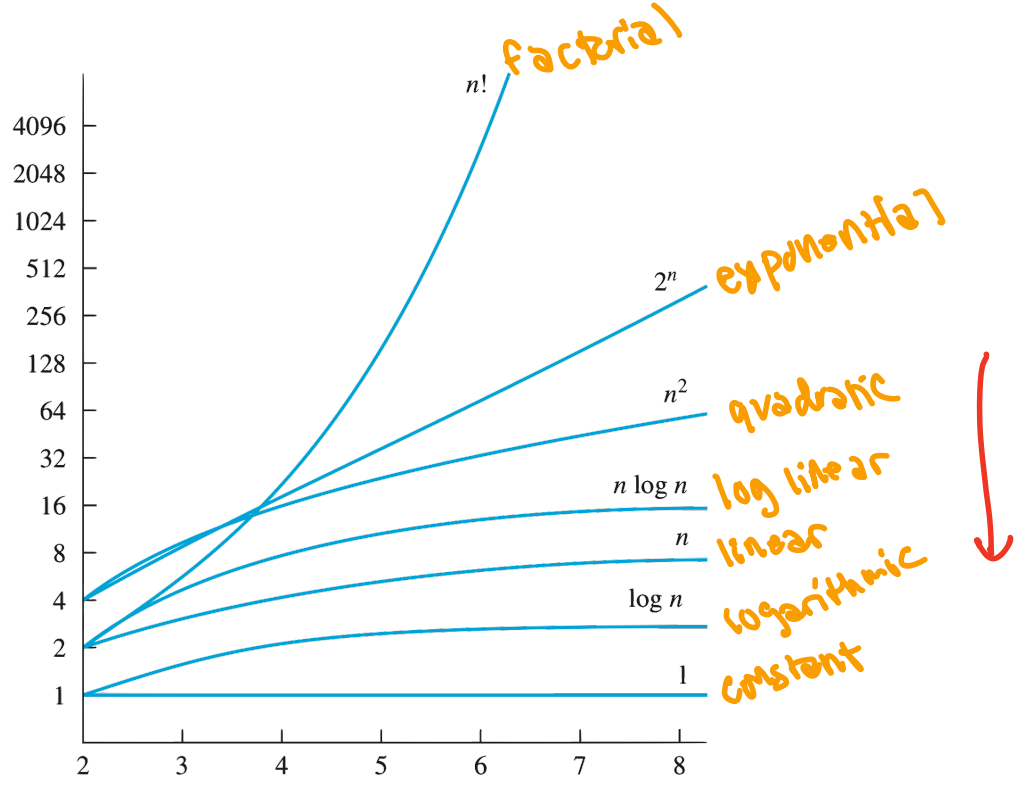
try 3

$3^2 - 3 \cdot 2 - 1 \geq 0$

$9 - 6 - 1 \geq 0$

$2 \geq 0$ ✓

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.

Determine a big-O 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. $T(n) = \frac{n^2(n^2+1)}{2}$: $O(n^4)$ $\quad \frac{n^4 + n^2}{2}$
5. $T(n) = 5$: $O(1)$
6. $T(n) = n(5 + \log n)$: $O(n \log n)$

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

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
```

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
}
```

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
  }
}
```

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 it

Example 1:

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

$n + nm + nm$

$O(nm)$

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++;
    }
  }
}
```

$n + n^2 + n^2m + n^2m$

$O(n^2m)$

Example 3:

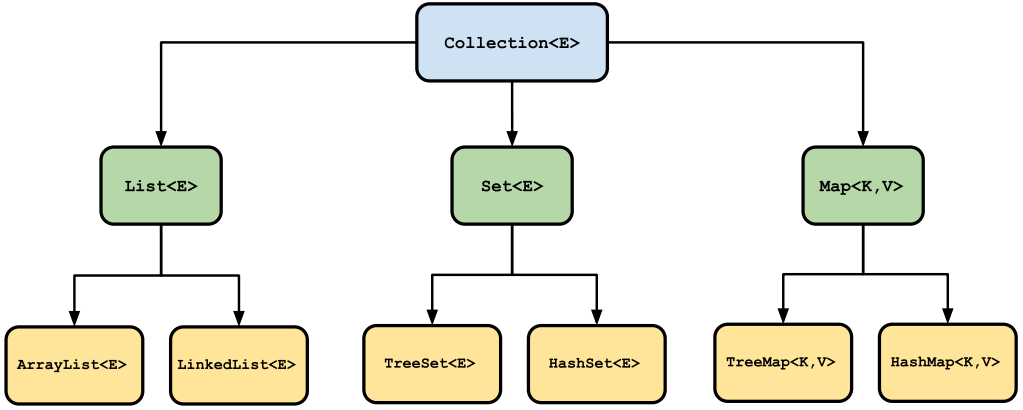
```
int sum = 0;
for (int i = 0; i < n; i++) {
  for (int j = 0; j <= i; j++) {
    sum++;
  }
}
```

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

$i = 0: 1$
$i = 1: 2$
$i = 2: 3$
$\vdots$
$i = n-1: n$

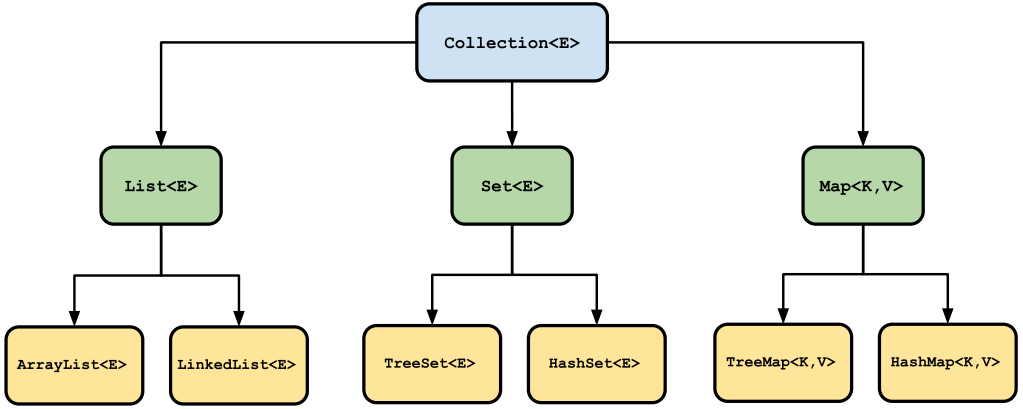
$\frac{n(n+1)}{2}$

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).

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).

See you on Thursday!

- We'll use what we covered today to analyze some sorting algorithms.
- Start studying for the programming exam and sign up for your timeslot ASAP!
- Turn in Lab 3 by 5PM tonight.
- Work on Homework 3! Implement your own `ArrayListString`.
- Reminder that Smith (go/smith) has office hours throughout the week and the 201 Course Assistants have drop-in hours in the late afternoons/evenings (go/cshelp).