CSCI 136:
Data Structures
and
Advanced Programming

Lecture 10

Recursion, part 2

Instructor: Dan Barowy

Williams

Topics

- Recursion costs
- Mathematical Induction

Your to-dos

1. Lab 3, **due Tuesday 10/4 by 10pm**
2. Read **before Mon**: Bailey, Ch 9.4–9.5.
3. Quiz 3, due **tomorrow by noon**.

Announcements

- CS Colloquium this **Friday, Sept 30 @ 2:35pm in Wege Auditorium (TCL 123)**

Sonia Roberts (Northeastern University)

Sonia is a postdoctoral research associate working on **soft sensors** based on origami and **knitted** structures for soft robots at Northeastern University as part of the Institute for Experiential Robotics.

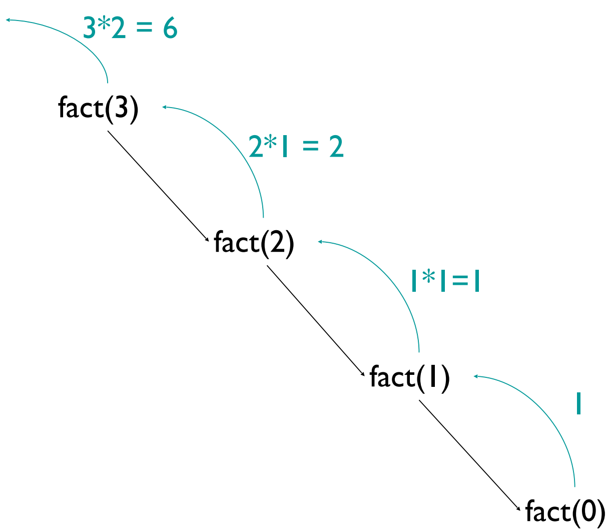
Sonia's research focuses on the morphological design and control of robots, asking questions like how detailed a model of the environment a robot needs, why a robot might need legs or wheels for different tasks, and what the trade-off is between robustness and plasticity when implementing aspects of a robot's control using morphology versus actuated degrees of freedom.

Recall: Factorial

- `n! = n × (n-1) × (n-2) × … × 1`
- Work with a partner and see if you can come up with a recursive solution.

How much does a **recusive** solution **cost**?

Graphically…

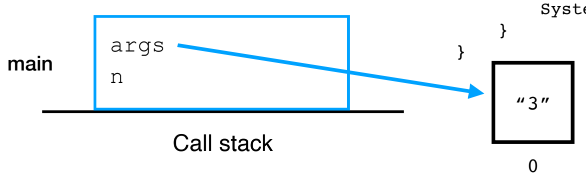



```
class Factorial {
    public static int fact(int n) {
        if (n == 0) { return 1; }
        return n * fact(n - 1);
    }

    public static void main(String[] args)
        int n = Integer.parseInt(args[0]);
        System.out.println(fact(n));
    }
}
```

Call stack

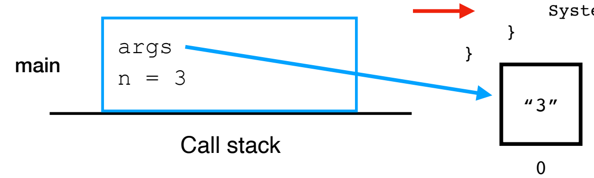
Call program with input "3".

```
class Factorial {
    public static int fact(int n) {
        if (n == 0) { return 1; }
        return n * fact(n - 1);
    }

    public static void main(String[] args)
        int n = Integer.parseInt(args[0]);
        System.out.println(fact(n));
    }
}
```



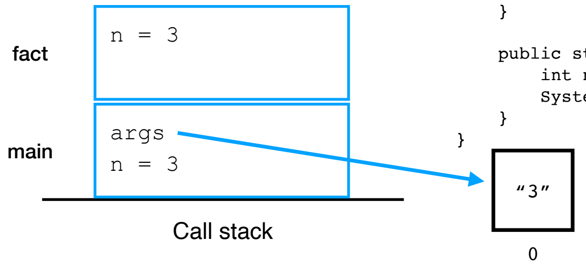
Call program with input "3".

```
class Factorial {
    public static int fact(int n) {
        if (n == 0) { return 1; }
        return n * fact(n - 1);
    }

    public static void main(String[] args)
        int n = Integer.parseInt(args[0]);
        System.out.println(fact(n));
    }
}
```

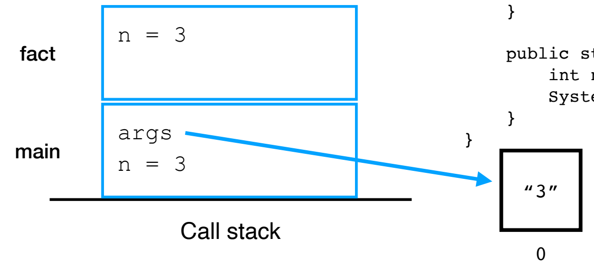


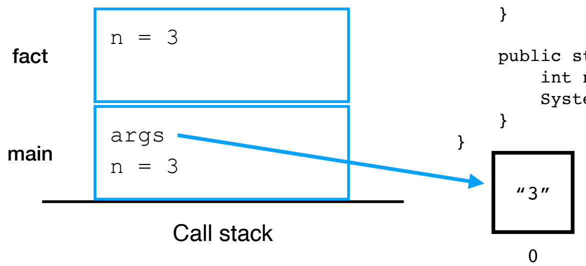
I skipped a subtlety here; did you spot it?

```
class Factorial {
    public static int fact(int n) {
        if (n == 0) { return 1; }
        return n * fact(n - 1);
    }

    public static void main(String[] args)
        int n = Integer.parseInt(args[0]);
        System.out.println(fact(n));
    }
}
```

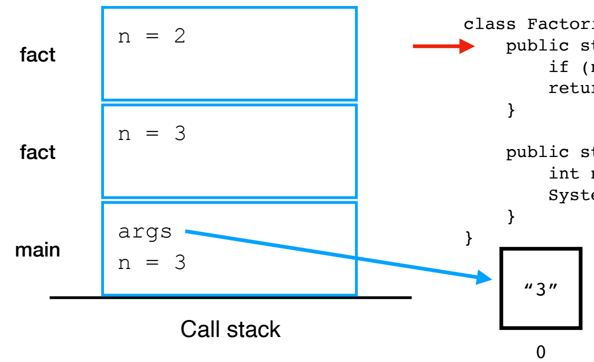



```
class Factorial {
    public static int fact(int n) {
        if (n == 0) { return 1; }
        return n * fact(n - 1);
    }

    public static void main(String[] args)
        int n = Integer.parseInt(args[0]);
        System.out.println(fact(n));
    }
}
```

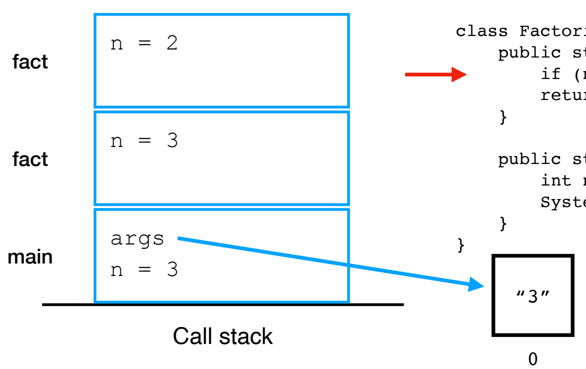




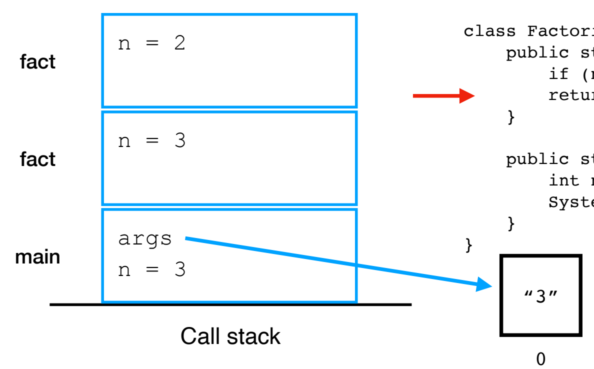

```
class Factorial {
    public static int fact(int n) {
        if (n == 0) { return 1; }
        return n * fact(n - 1);
    }

    public static void main(String[] args)
        int n = Integer.parseInt(args[0]);
        System.out.println(fact(n));
    }
}
```



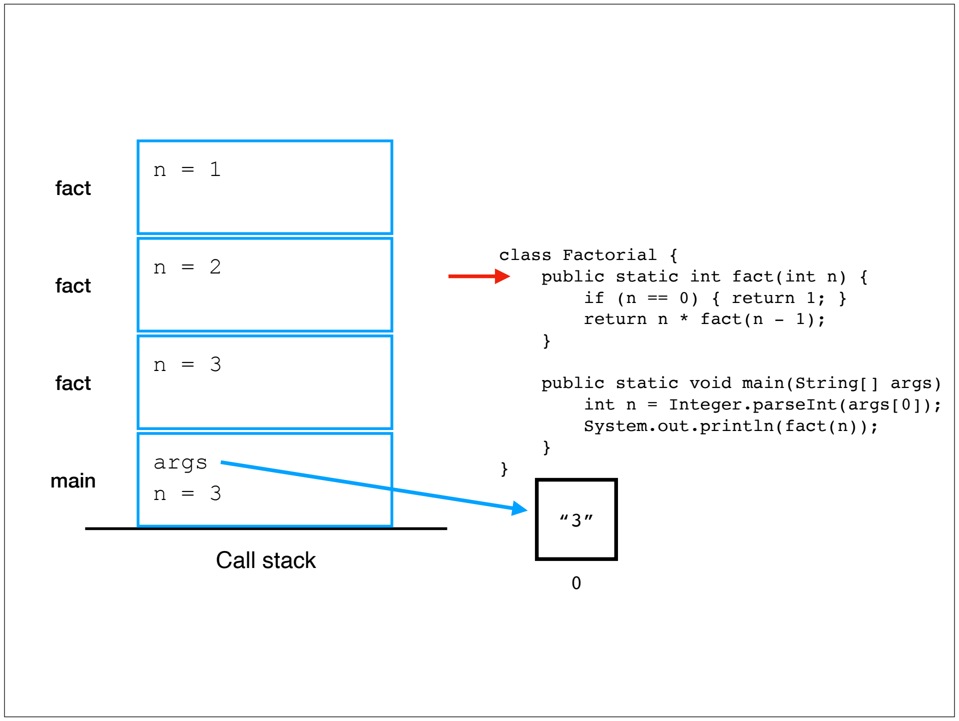

```
class Factorial {
    public static int fact(int n) {
        if (n == 0) { return 1; }
        return n * fact(n - 1);
    }

    public static void main(String[] args)
        int n = Integer.parseInt(args[0]);
        System.out.println(fact(n));
    }
}
```

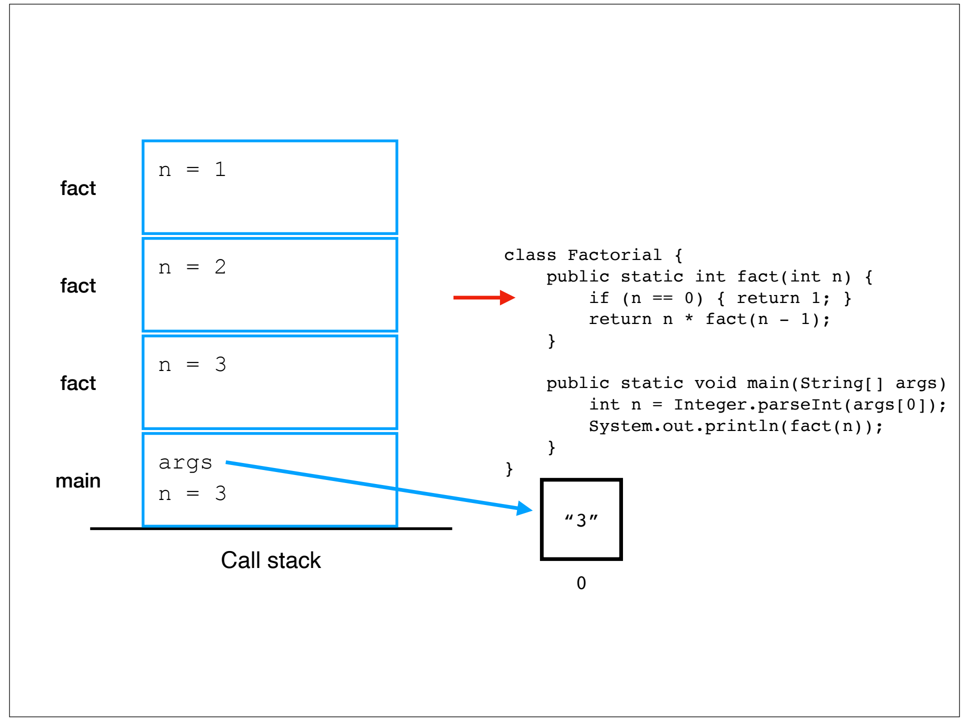


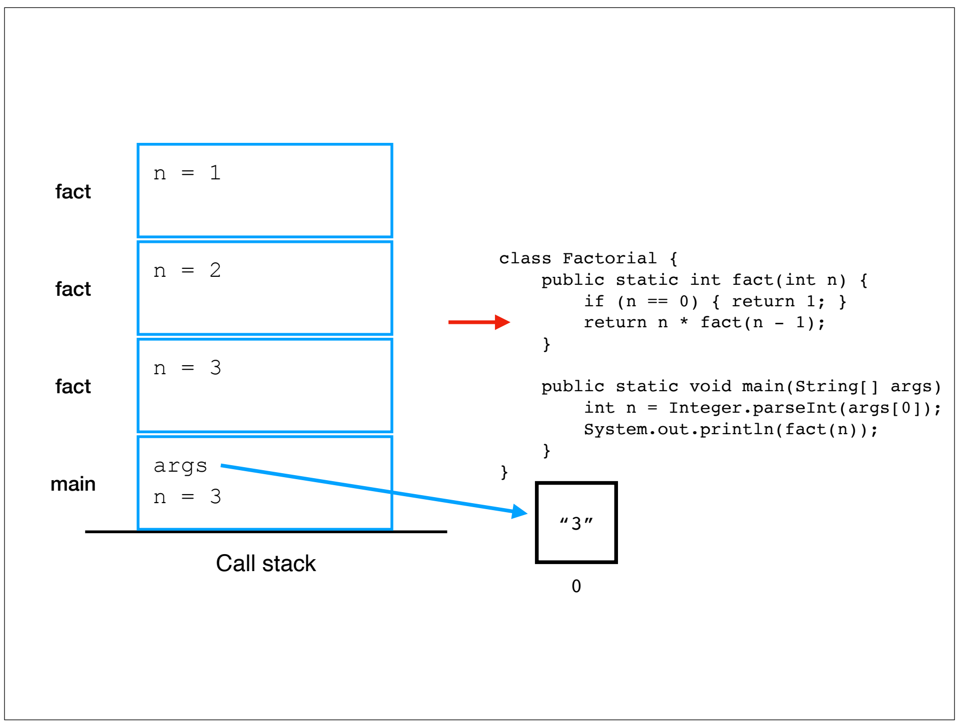

```
class Factorial {
    public static int fact(int n) {
        if (n == 0) { return 1; }
        return n * fact(n - 1);
    }

    public static void main(String[] args)
        int n = Integer.parseInt(args[0]);
        System.out.println(fact(n));
    }
}
```

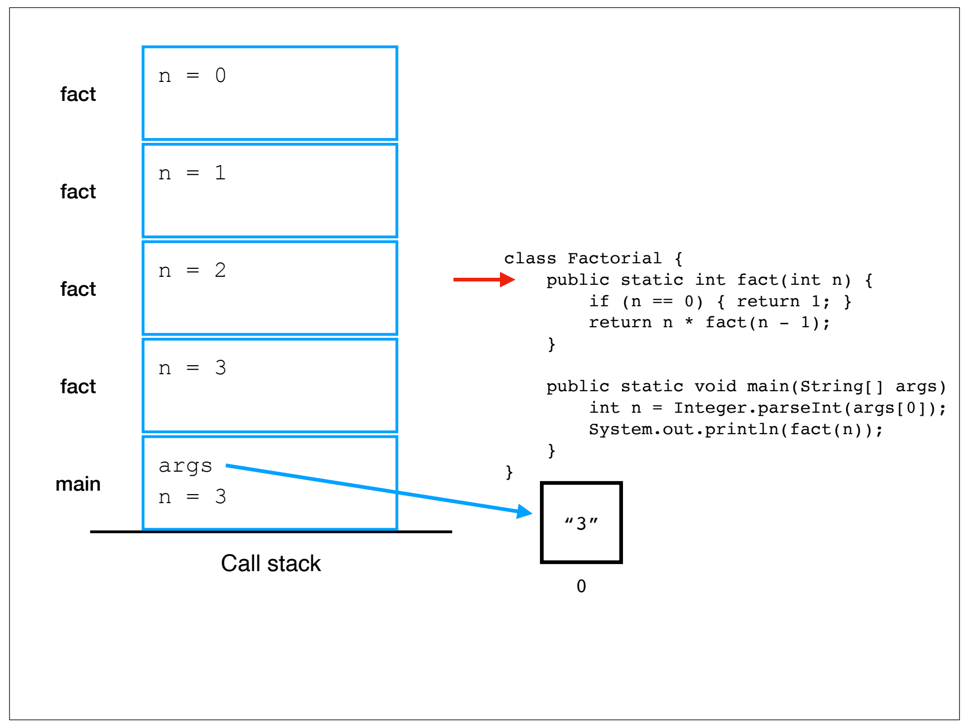


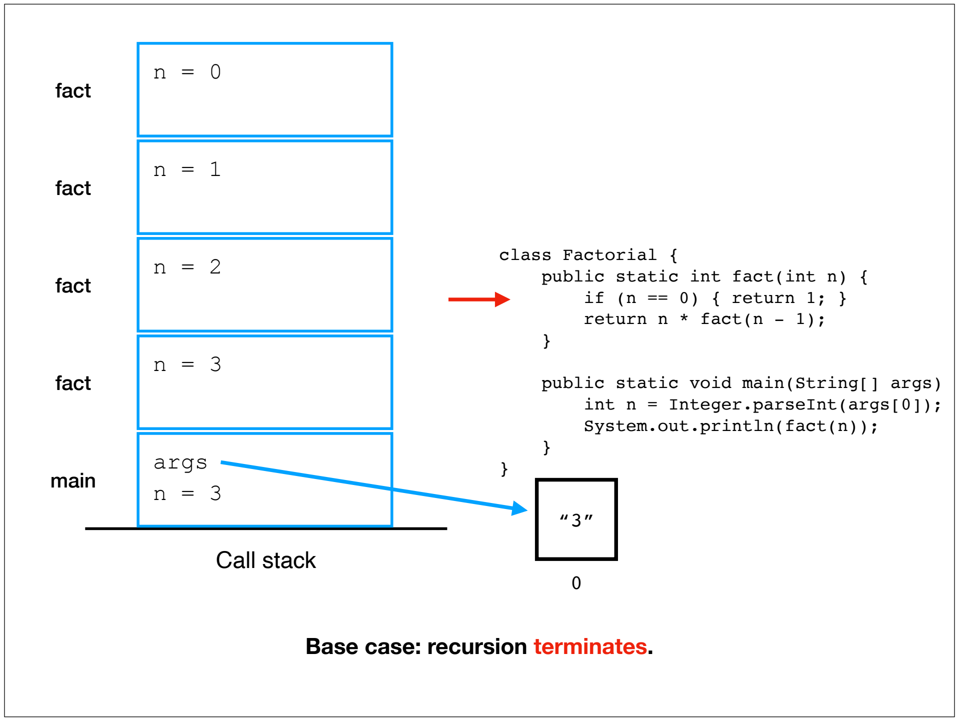

```
class Factorial {
    public static int fact(int n) {
        if (n == 0) { return 1; }
        return n * fact(n - 1);
    }

    public static void main(String[] args)
        int n = Integer.parseInt(args[0]);
        System.out.println(fact(n));
    }
}
```

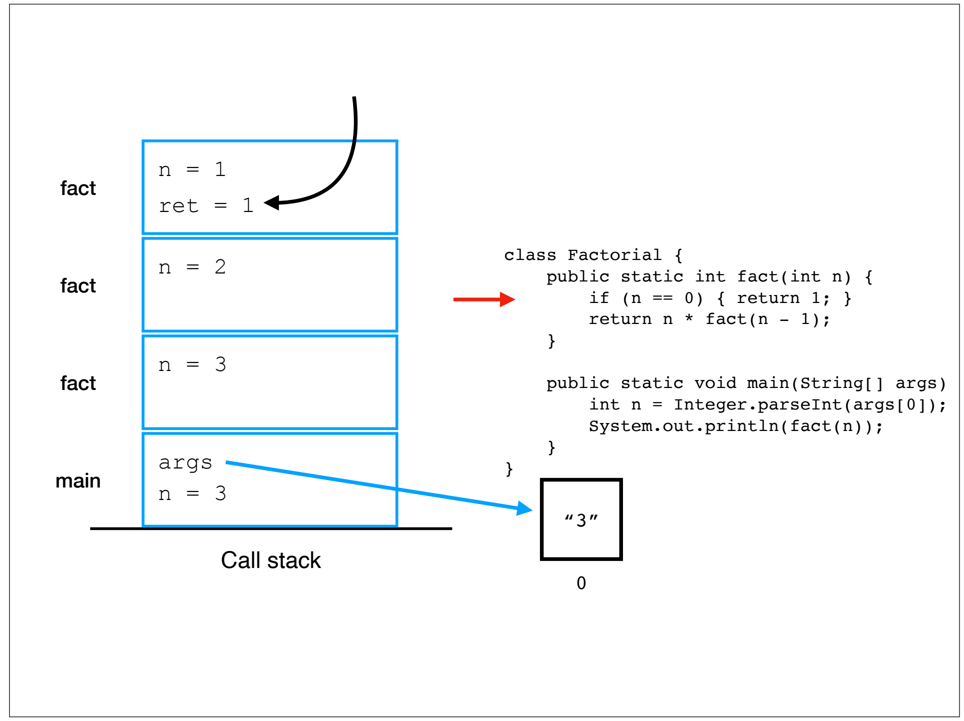

fact
n = 1
fact
n = 2
fact
n = 3
main
args
n = 3
Call stack
"3"
0
class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}

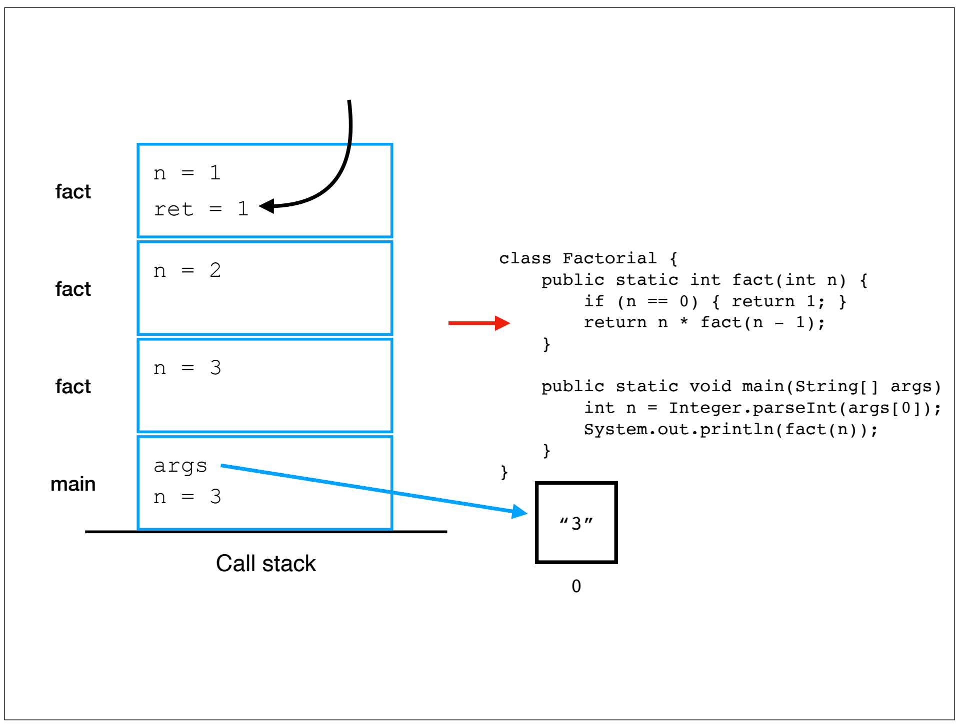

fact
n = 1
fact
n = 2
fact
n = 3
main
args
n = 3
Call stack
"3"
0
class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}

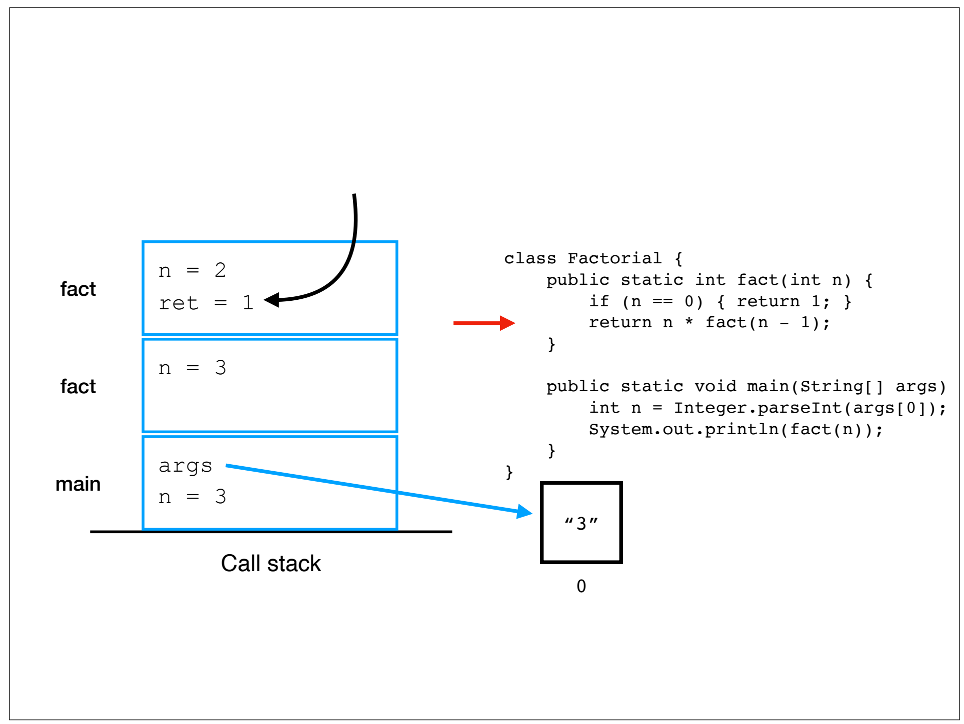

fact
n = 1
fact
n = 2
fact
n = 3
main
args
n = 3
Call stack
"3"
0
class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}

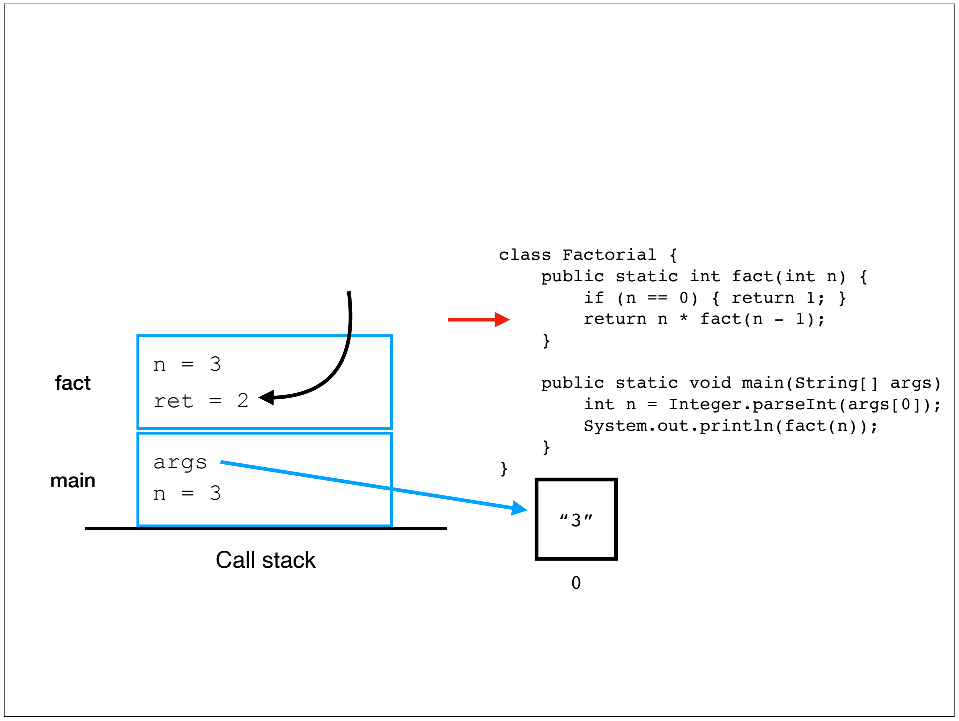

fact
n = 0
fact
n = 1
fact
n = 2
fact
n = 3
main
args
n = 3
Call stack
"3"
0
class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}

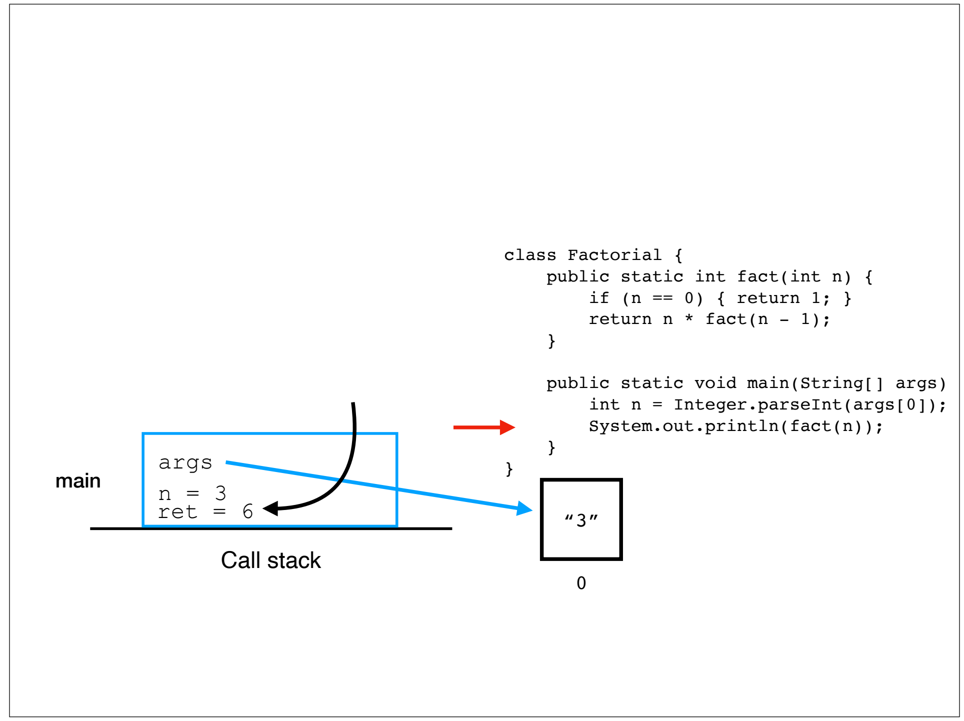

fact
n = 0
fact
n = 1
fact
n = 2
fact
n = 3
main
args
n = 3
Call stack
class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}
"3"
0
Base case: recursion terminates.

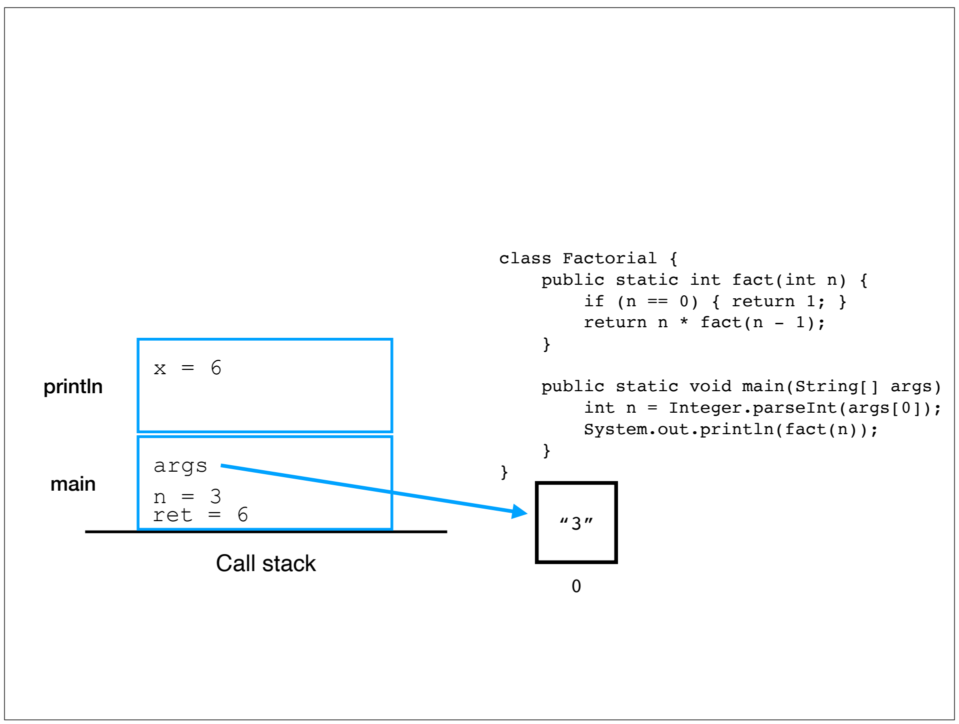

fact
n = 1
ret = 1
fact
n = 2
fact
n = 3
main
args
n = 3
Call stack
class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}
"3"
0

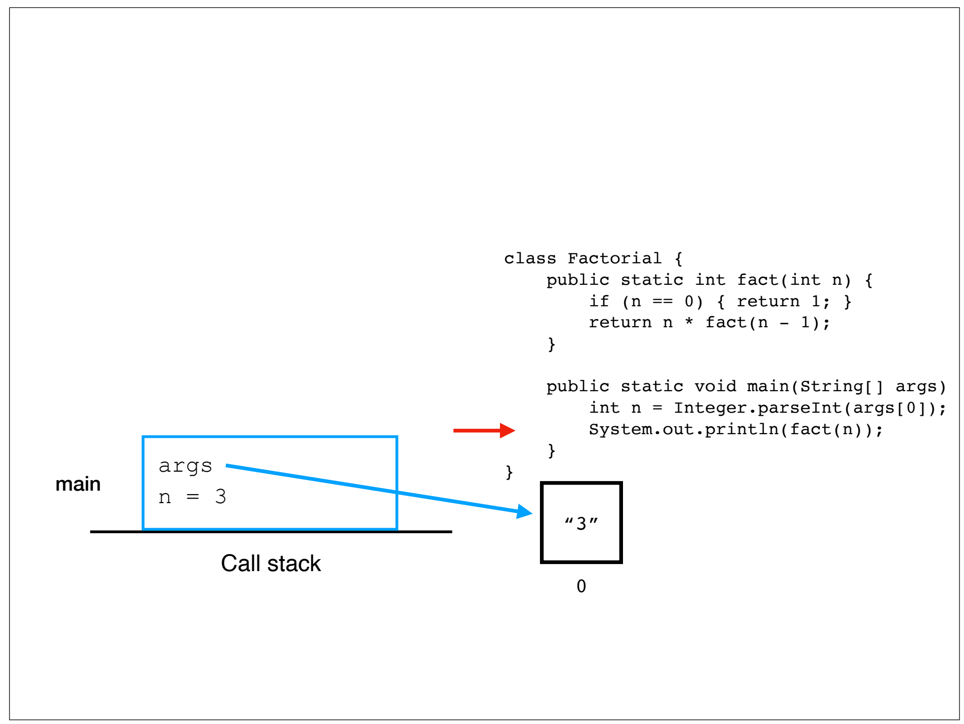

fact
n = 1
ret = 1
fact
n = 2
fact
n = 3
main
args
n = 3
Call stack
class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}
"3"
0


fact
n = 2
ret = 1
fact
n = 3
main
args
n = 3
Call stack
class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}
"3"
0

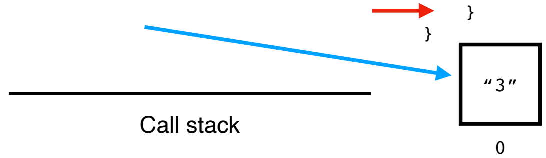

class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}
fact
n = 3
ret = 2
main
args
n = 3
"3"
0
Call stack


class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}
main
args
n = 3
ret = 6
"3"
0
Call stack


class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}
println
x = 6
main
args
n = 3
ret = 6
"3"
0
Call stack


class Factorial {
public static int fact(int n) {
if (n == 0) { return 1; }
return n * fact(n - 1);
}
public static void main(String[] args)
int n = Integer.parseInt(args[0]);
System.out.println(fact(n));
}
}
main
args
n = 3
"3"
0
Call stack

```
class Factorial {
    public static int fact(int n) {
        if (n == 0) { return 1; }
        return n * fact(n - 1);
    }

    public static void main(String[] args)
        int n = Integer.parseInt(args[0]);
        System.out.println(fact(n));
    }
}
```



Recursion tradeoffs

- Advantages
  - Often **easier** to construct recursive solution than a loop
  - Code is usually **clearer**
  - Some problems do not have **obvious** non-recursive solutions
- Disadvantages
  - **Time cost** of recursive calls
  - **Memory cost** (need to store state for each recursive call until base case is reached)

Mathematical Induction

A note about “formal methods”

If the problem “fits” the mold, there is a procedure for determining truth.

Mathematical Induction

- The **mathematical cousin** of **recursion** is **induction**
- Induction is a **proof technique**
- Purpose: to **simultaneously prove** an **infinite number** of theorems!

Principle of Mathematical Induction

Let **P(n)** be a **predicate** that is defined for **integers n**, and let **a** be a **fixed integer**.

If the following two statements are **true**:

1. **P(a)** is **true**.
2. For all integers **k ≥ a**, **if P(k)** is **true then P(k + 1)** is **true**.

then the statement

for all integers **n ≥ a**, **P(n)** is **true**

is **also true**.

Principle of Mathematical Induction (variant)

Let **P(n)** be a **predicate** that is defined for **integers n**, and let **a** be a **fixed integer**.

If the following two statements are **true**:

1. **P(a)** is **true**.
2. For all integers **k > a**, **if P(k-1)** is **true then P(k)** is **true**.

then the statement

for all integers **n ≥ a**, **P(n)** is **true**

is **also true**.

To be clear:

If you want to prove that **P(n)** is **true** for all integers **n ≥ a,**

1. You must first prove that **P(a)** is **true**.

2. Then you must prove that:

For all integers **k ≥ a**, **if P(k)** is **true then P(k+1)** is **true**.

Critically, when proving #2, **assume** that **P(k)** is **true** and **show** that **P(k+1) must also be true**.

Names for things and "form"

Hypothesis: P(n) is **true** for all integers n ≥ a,

1. Base case: P(a) is **true**.

2. Inductive step:

   For all integers k ≥ a, if P(k) is true then P(k+1) is true.

Like recursion, there is an analogy

Like recursion, there is an analogy

Example

Prove that the sum of the first n integers is:

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

Example

Put another way, prove

$$\mathbf{P(n)} : 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$

for all $n \geq 1$.

We have an unbounded number of hypotheses ("for all $n \geq 1$").

Use **mathematical induction**.

Remember the template!

Step 1: Prove **P(a)**

Step 2: Prove **P(k) ⇒ P(k+1)**

Therefore,

$$\mathbf{P(n)} : 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$

For all n ≥ 1.

Is **true**.

Example

Step 1: Prove **P(a)**

What would a good **a** be?

$$P(n) : 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$

The "simplest" instance is **a = 1**. Let's start there.

Example

Step 1: Prove **P(a)**

$$P(a) : 1 = \frac{1(1+1)}{2}$$

Is this statement true? **Yes.**

Proof: $\frac{1(1+1)}{2} = \frac{2}{2} = 1$

Example

Step 2: Prove **P(k) ⇒ P(k+1)**

Assume the following is true:

$$\textbf{P(k)} : 1 + 2 + 3 + \ldots + k = \frac{k(k+1)}{2}$$

Prove that **P(k)** implies:

$$\textbf{P(k+1)} : 1 + 2 + 3 + \ldots + (k + 1) = \frac{(k+1)((k+1)+1)}{2}$$

Example

Step 2: Prove **P(k) ⇒ P(k+1)**

$$\textbf{P(k+1)} : 1 + 2 + 3 + \ldots + (k + 1) = \frac{(k+1)((k+1)+1)}{2}$$

Let's handle the left side first.

$$1 + 2 + 3 + \ldots + (k + 1)$$

Looks familiar. Isn't it the same as:

$$(1 + 2 + 3 + \ldots + k) + (k + 1)$$

Example

Step 2: Prove **P(k) ⇒ P(k+1)**

$$(1 + 2 + 3 + \ldots + k) + (k + 1)$$

According to P(k), which is true,
it must be equal to:

$$(1 + 2 + 3 + \ldots + k) + (k + 1) = \frac{k(k+1)}{2} + (k + 1)$$

Example

Step 2: Prove **P(k) ⇒ P(k+1)**

Simplify

$$= \frac{k(k+1)}{2} + (k + 1)$$

$$= \frac{k(k+1)}{2} + \frac{2(k+1)}{2}$$

$$= \frac{k(k+1) + 2(k+1)}{2}$$

Let's stop here.
The left side is

$$= \frac{(k+1)(k+2)}{2}$$

Example

Step 2: Prove **P(k) ⇒ P(k+1)**

$$\textbf{P(k+1)} : 1 + 2 + 3 + \ldots + (k + 1) = \frac{(k+1)((k+1)+1)}{2}$$

Let's handle the right side now.

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

Simplify

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

Let's stop here.

Example

Step 2: Prove **P(k) ⇒ P(k+1)**

$$\textbf{P(k+1)} : 1 + 2 + 3 + \ldots + (k + 1) = \frac{(k+1)((k+1)+1)}{2}$$

We just showed that the left side

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

equals the right side

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

Example

Step 1: Prove **P(a)**

Step 2: Prove **P(k) ⇒ P(k+1)**

Therefore,

$$\textbf{P(n)} : 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$

For all $n \geq 1$.

Is **true**.

Recap & Next Class

Today:

- Recursion costs
- Mathematical induction

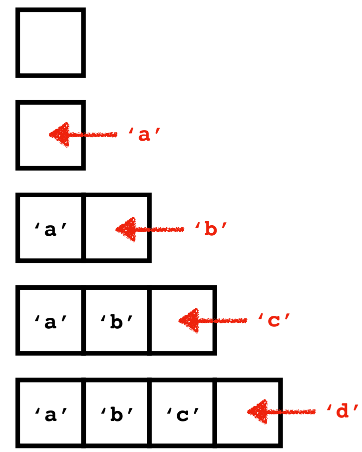
Next class:

- Vector doubling
- ADTs
- Lists

Expanding vectors: why double?

Why is the **array doubling** strategy for Vector **better** than expanding the array **one element at a time**?

One-at-a-time expansion



Initial array.

Insert element.

New array; **copy** previous; **insert** element.

New array; **copy** previous; **insert** element.

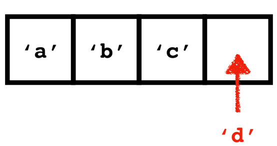
New array; **copy** previous; **insert** element.

. . .

. . .

Insertion into an array

How much does **array insertion** cost?



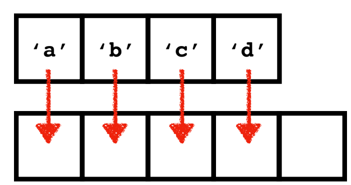
It costs **O(1)**.

In fact, lookup and insertion both cost **O(1)**.

Tradeoff: arrays are fixed size.

Copying an array

How much does an **array copy** cost?

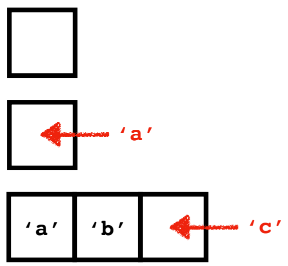


It costs **O(1)** × **m**, where **m** is the size of the original array.

≈ **O(m)**

One-at-a-time expansion costs?

(in the worst case, each time)



Initial array.

Insert element.

New array; **copy** previous; **insert** element.

O(m) + **O(1)** ≈ **O(m)**, where **m** is the size of the original array.

Cost is **dominated by the size of the array** being copied.

How many copies?

of copies for one-at-a-time expansion:

	1	**+**	**2**	**+**	**3**	**+**	**…**	**+(n-1)**
`add()`	2nd elem.		3rd elem.		4th elem.			nth elem.

Recall theorem: $1 + 2 + 3 + \ldots + k = k(k+1)/2$

Sub n-1 for k: $(n-1)((n-1)+1)/2 = n(n-1)/2$

$= (n^2-n)/2$

One-at-a-time expansion costs ≈ **O(n²)**

How many copies?

of copies for doubling expansion:

	1	**+**	**2**	**+**	**4**	**+**	**…**	**+(n/2)**
`add()`	up to 2nd elem.		up to 4th elem.		up to 8th elem.			up to nth elem.

Neat theorem: $1 + 2 + 4 + \ldots + 2^{k-1} = 2^k-1$

Suppose $n = 2^k$.

Then $1 + \ldots + n/2 = 1 + \ldots + 2^k/2$

$= 1 + \ldots + 2^{k-1} = 2^k-1 = n-1$

Doubling expansion costs ≈ **O(n)**

Which is faster?

💩 One-at-a-time expansion costs ≈ **O(n²)** 💩

😎 Doubling expansion costs ≈ **O(n)** 😎

Doubling is Vin Diesel-approved.

Activity

Prove: n cents can be obtained by using only 3-cent and 8-cent coins, for all $n \geq 15$.