Week 5 - Wednesday

COMP 2100

Last time

- What did we talk about last time?
- Exam 1!
- Before that:
  - Review
- Before that:
  - Linked lists
  - Implementing stacks with linked lists
  - Implementing queues with linked lists

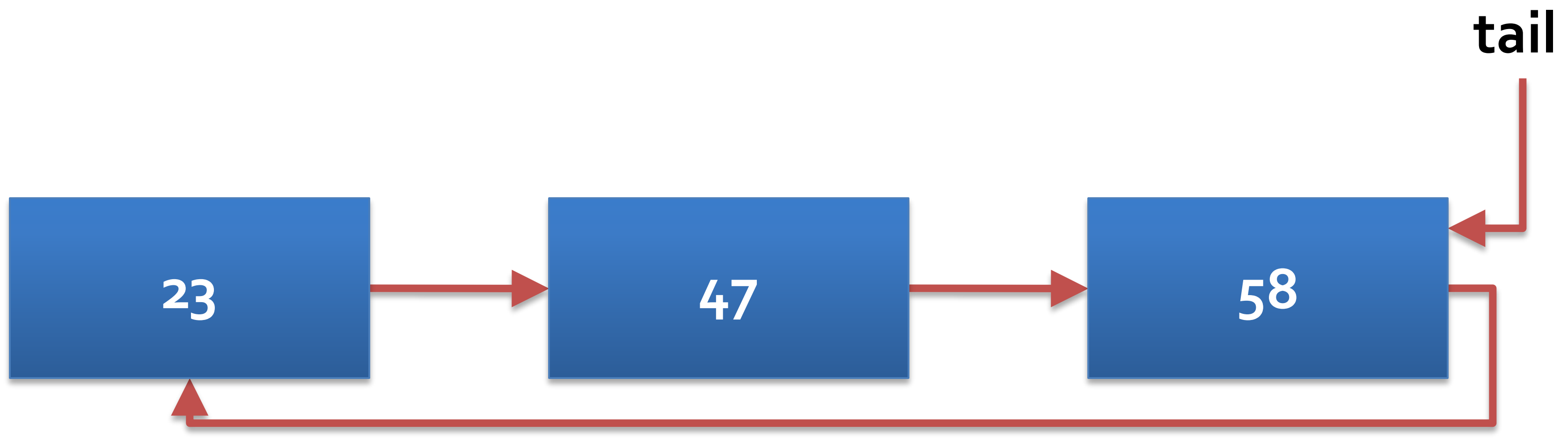
Questions?

Project 2

Other kinds of linked lists

Circular linked lists

- Linked lists can be made circular such that the last node points back at the head node
- This organization is good for situations in which we want to cycle through all of the nodes in the list repeatedly

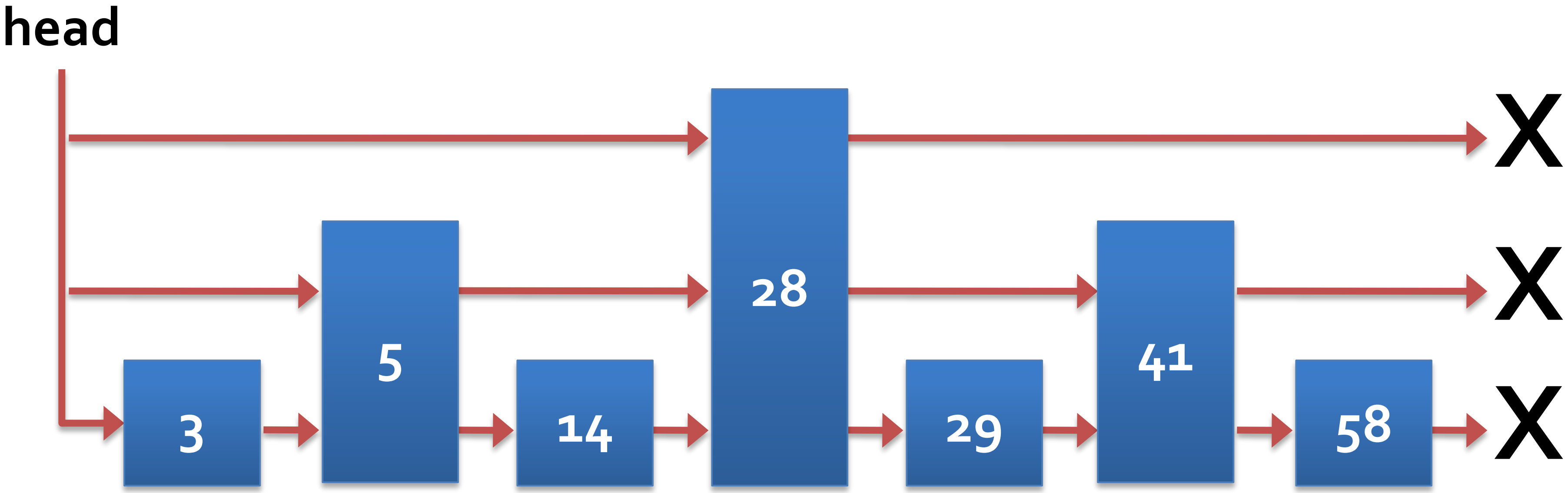


Performance of a circular linked list

- Insert at front (or back)
  - $\Theta(1)$
- Delete at front
  - $\Theta(1)$
  - Delete at back costs $\Theta(\boldsymbol{n})$ unless we used doubly linked lists
- Search
  - $\Theta(\boldsymbol{n})$

Skip lists

- We can design linked lists with multiple pointers in some nodes
- We want ½ of the nodes to have 1 pointer, ¼ of the nodes to have 2 pointers, 1/8 of the nodes to have 3 pointers...



Performance of skip lists

- If ordered, search is
  - $\Theta(\log \boldsymbol{n})$
- Go to index is
  - $\Theta(\log \boldsymbol{n})$
- Insert at end
  - $\Theta(\log \boldsymbol{n})$
- Delete
  - Totally insane, at least $\Theta(\boldsymbol{n})$
- Trees end up being a better alternative

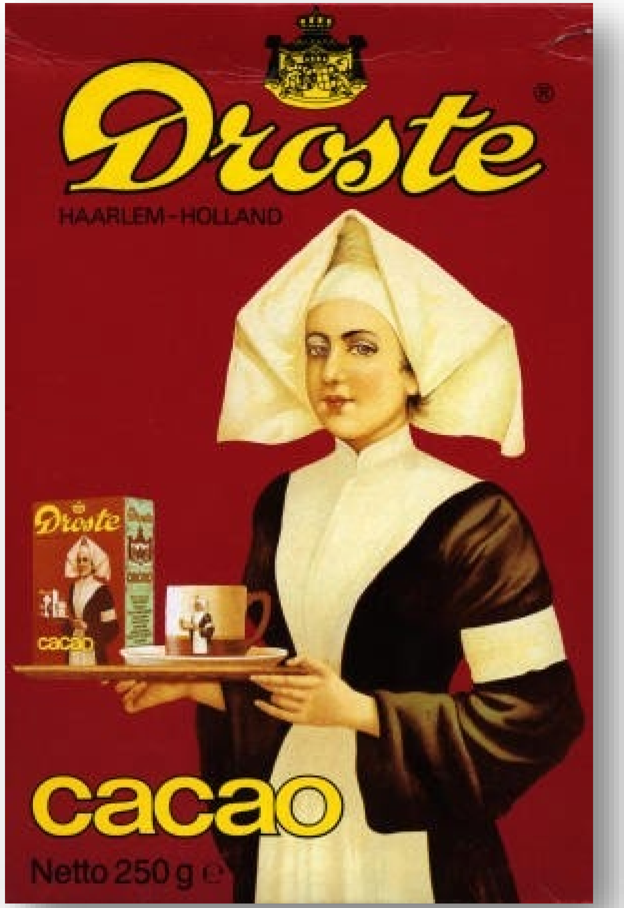
Self organizing lists

- Maybe we want to make items that are used frequently easy to get at
- Several different approaches, mostly based on finding items repeatedly:
  - **Move to front:** After finding an item, put it in the front
  - **Transpose:** After finding an item, move it up by one
  - **Count:** Keep the list ordered by how often you get a particular item (requires a counter in each node)
  - **Ordering:** Sort the list according to some feature of the data

Recursion

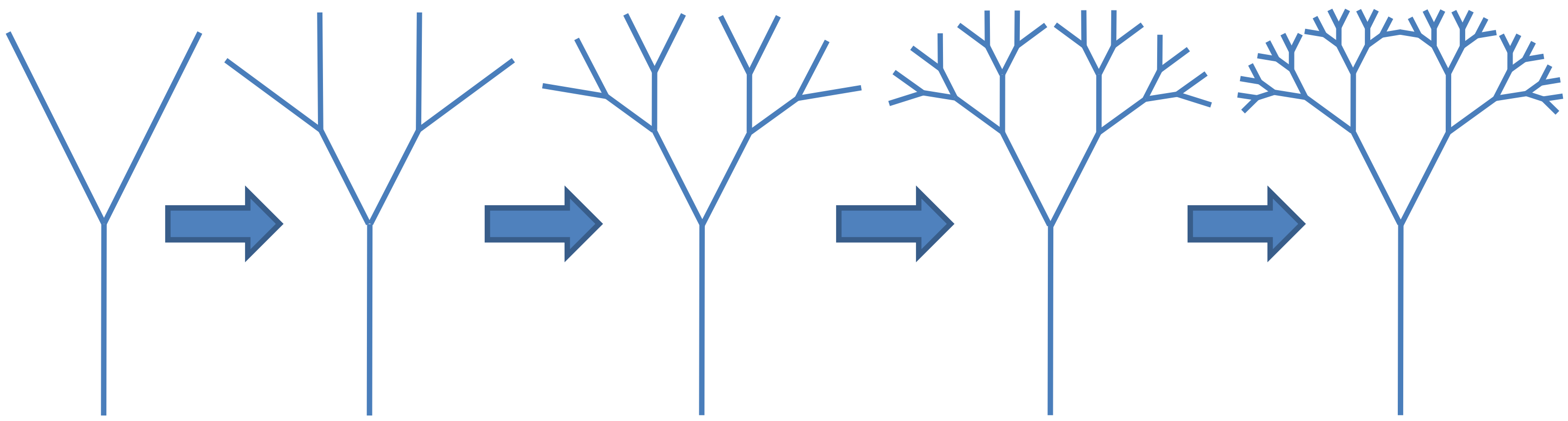
What is recursion?

- Defining something in terms of itself
- To be useful, the definition must be based on progressively simpler definitions of the thing being defined



Bottom up

- It is possible to define something recursively from the bottom up
- We start with a simple pattern and repeat the pattern, using a copy of the pattern for each part of the starting pattern



Top down

Explicitly:

- $n! = (n)(n-1)(n-2) \ldots (2)(1)$

Recursively:

- $n! = (n)(n-1)!$
- $1! = 1$

- $6! = 6 \cdot 5!$
  - $5! = 5 \cdot 4!$
    - $4! = 4 \cdot 3!$
      - $3! = 3 \cdot 2!$
        - $2! = 2 \cdot 1!$
          - $1! = 1$
- $6! = 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 720$

Examples in acronyms

- PHP
  - PHP: Hypertext Processor
    - (PHP: Hypertext Processor): Hypertext Processor
      - …
- XINU
  - XINU Is Not Unix
    - (XINU Is Not Unix) Is Not Unix
      - …

Useful recursion

Two parts:

- Base case(s)
  - Tells recursion when to stop
  - For factorial, $n = 1$ or $n = 0$ are examples of base cases
- Recursive case(s)
  - Allows recursion to progress
  - "Leap of faith"
  - For factorial, $n > 1$ is the recursive case

Solving Problems with Recursion

Approach for problems

- Top down approach
- Don't try to solve the whole problem
- Deal with the next step in the problem
- Then make the "leap of faith"
- Assume that you can solve any smaller part of the problem

Walking to the door

- Problem: You want to walk to the door
- Base case (if you reach the door):
  - You're done!
- Recursive case (if you aren't there yet):
  - Take a step toward the door

Problem

Implementing factorial

- Base case ($n \leq 1$):
  - $1! = 0! = 1$
- Recursive case ($n > 1$):
  - $n! = n(n-1)!$

Code for factorial

```
public static long factorial(int n) {

  if( n <= 1 ){
    return 1;
  } else {
    return n*factorial( n - 1 );
  }

}
```

Base Case

Recursive Case

Count the zeroes

- Given an integer, count the number of zeroes in its representation
- Example:
  - 13007804
  - 3 zeroes

Recursion for zeroes

- Base cases (number less than 10):
  - 1 zero if it is 0
  - No zeroes otherwise
- Recursive cases (number greater than or equal to 10):
  - One more zero than the rest of the number if the last digit is 0
  - The same number of zeroes as the rest of the number if the last digit is not 0

Code for zeroes

```
public static int zeroes(int n) {

  if (n == 0) {
     return 1;
  } else if (n < 10) {
     return 0;
  } else if (n % 10 == 0) {
     return 1 + zeroes(n / 10);
  } else {
     return zeroes( n / 10 );
  }

}
```

Base Cases

Recursive Cases

Searching in a sorted array

- Given an array of integers in (ascending) sorted order, find the index of the one you are looking for
- Useful problem with practical applications
- Recursion makes an efficient solution obvious
- Play the **High-Low** game

Recursion for binary search

- Base cases:
  - The number isn't in the range you are looking at. Return -1.
  - The number in the middle of the range is the one you are looking for. Return its index.
- Recursion cases:
  - The number in the middle of the range is too low. Look in the range above it.
  - The number in middle of the range is too high. Look in the range below it.

Code for binary search

```
public static int search( int[] array,
  int n, int start, int end) {
  int midpoint = (start + end)/2;
  if (start >= end) {
     return -1;
  } else if (array[midpoint] == n) {
     return midpoint;
  } else if (array[midpoint] < n) {
     return search( array, n,
               midpoint + 1, end );
  } else {
     return search(array, n, start,
            midpoint);
  }
}
```

Base Cases

Recursive Cases

Time for binary search

- Each recursive call splits the range in half
- In the worst case, we will have to keep splitting the range in half until we have a single number left
- We want to find the number of times that we have to multiply n by ½ before we get 1
  - $n(\frac{1}{2})^x = 1$
  - $n = 2^x$
  - $x = \log_2(n)$

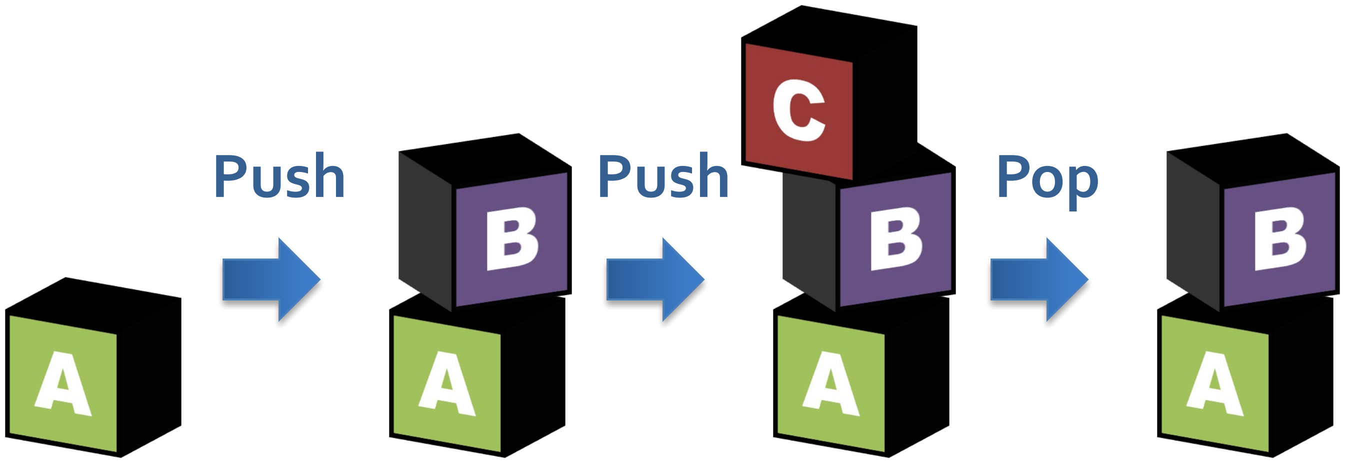
How Does Recursion Work Inside The Computer?

All this math is great, but…

- How does it actually work inside a computer?
- Is there a problem with calling a method inside the same method?
- How does the computer keep track of which method is which?

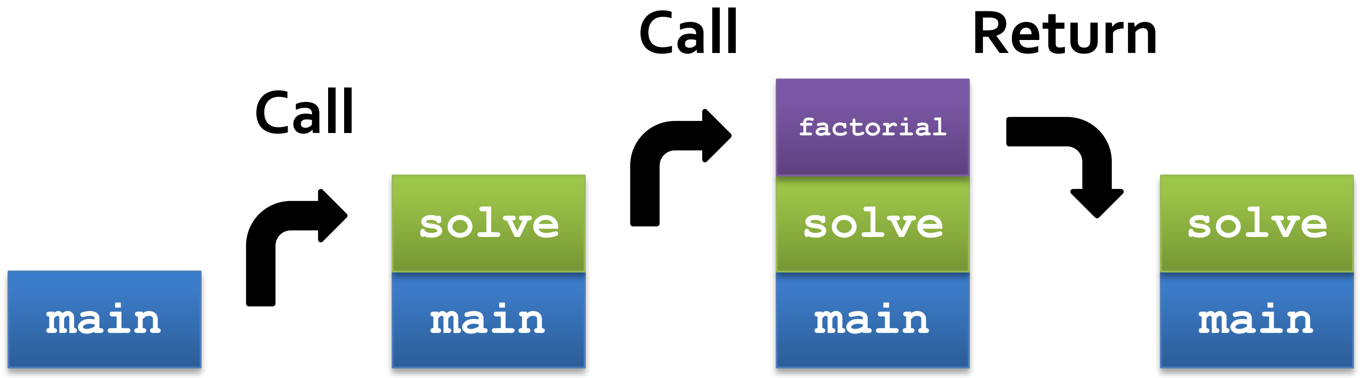
The stack

- As you know, a stack is a FILO data structure used to store and retrieve items in a particular order
- Just like a stack of blocks:



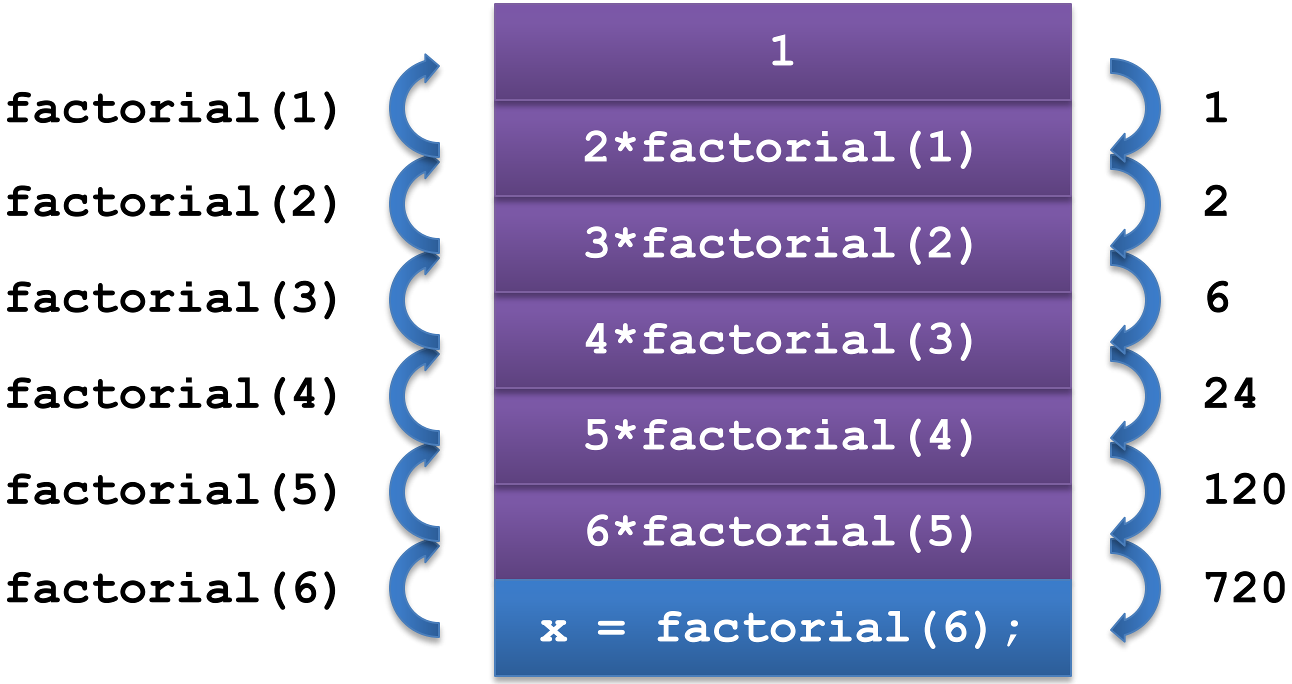
Stack for functions

- In the same way, the local variables for each function are stored on the stack
- When a function is called, a copy of that function is **pushed** onto the stack
- When a function returns, that copy of the function **pops** off the stack



Example with factorial

- Each copy of factorial has a value of ***n*** stored as a local variable
- For 6! :



Issues of Efficiency

When to use recursion?

- Recursion is a great technique
- One of its strengths is in writing concise code to solve a problem
- Some recursive solutions are very efficient
- Some are not
- It pays to be aware of both

Summation

- Find the sum of the integers 1 through ***n***

- Example: ***n*** = 8
- sum(8) = 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1
- sum(8) = 36

Recursion for Summing

- Base case (***n*** = 1):
  - $$\sum_{i=1}^{1} i = 1$$
- Recursive case (***n*** > 1):
  - $$\sum_{i=1}^{n} i = n + \sum_{i=1}^{n-1} i$$

Code for summing

```
public static int sum( int n ) {

    if (n == 1) {
        return 1;
    } else {
        return n + sum( n – 1 );
    }

}
```

Base Case

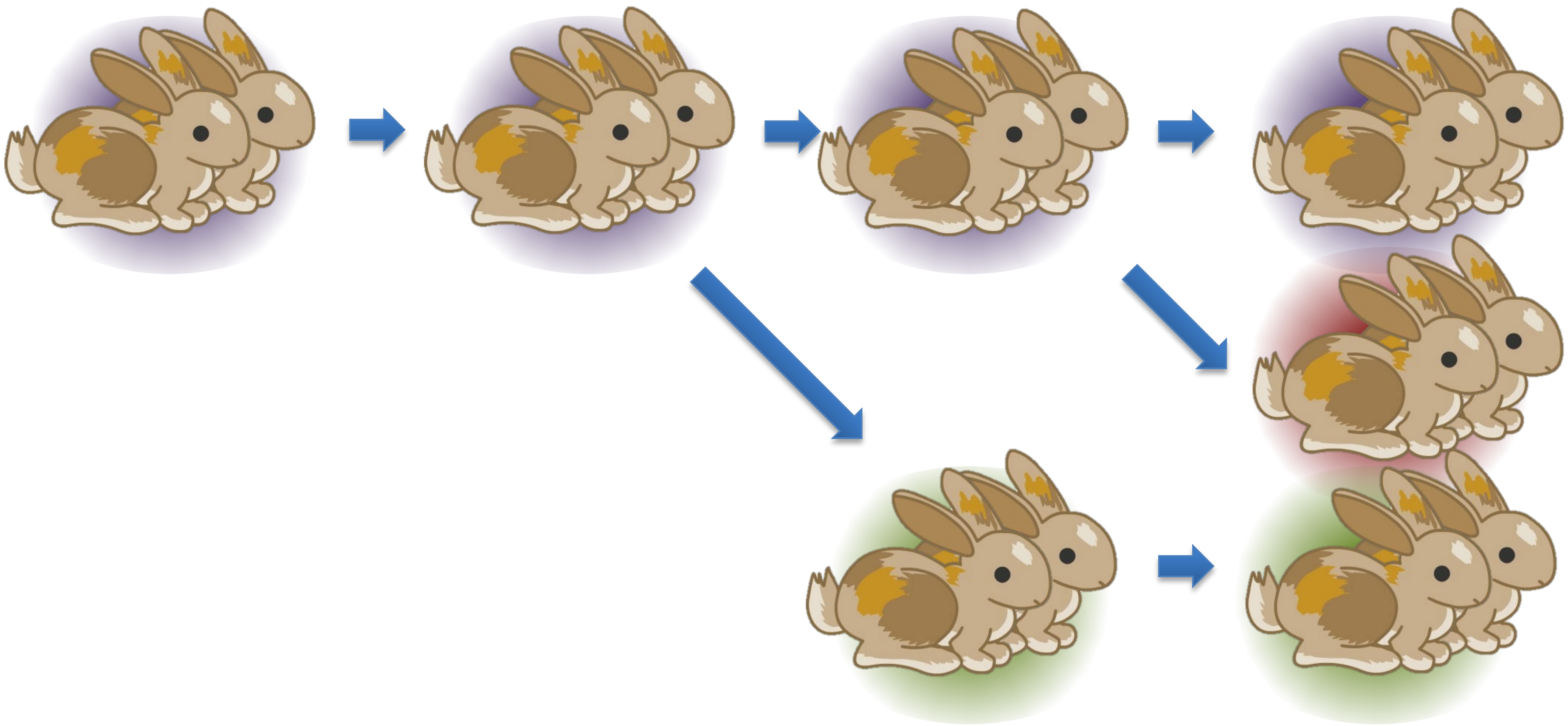
Recursive Case

Why not recursion?

- Recursive summing takes linear time (summing ***n*** takes ***n*** function calls)
- Is there another way to find this sum?
- Closed form equation
  - $$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$
  - Constant time!
  - Remember the story of young Gauss

Fibonacci

- The sequence: 1 1 2 3 5 8 13 21 34 55…
- Studied by Leonardo of Pisa to model the growth of rabbit populations



Fibonacci problem

- Find the ***n***th term of the Fibonacci sequence
- Simple approach of summing two previous terms together
- Example: ***n*** = 7
- **1 1 2 3 5 8 13**

  1 2 3 4 5 6 7

Upcoming

Next time…

- More on recursive running time
- Symbol tables

Reminders

- Read section 3.1
- Keep working on Project 2
- **Office hours from 4-5 today are cancelled due to a meeting**