Recursion

notes Chapter 8

Printing n of Something

- suppose you want to implement a method that prints out n copies of a string

```
public static void printIt(String s, int n) {
  for(int i = 0; i < n; i++) {
    System.out.print(s);
  }
}
```

A Different Solution

- alternatively we can use the following algorithm:
  1. if n == 0 done, otherwise
     I. print the string once
     II. print the string (n – 1) more times

```
public static void printItToo(String s, int n) {
  if (n == 0) {
    return;
  }
  else {
    System.out.print(s);
    printItToo(s, n - 1);    // method invokes itself
  }
}
```

Recursion

- a method that calls itself is called a *recursive* method
- a recursive method solves a problem by repeatedly reducing the problem so that a base case can be reached

```
printItToo("*", 5)
*printItToo ("*", 4)
**printItToo ("*", 3)
***printItToo ("*", 2)
****printItToo ("*", 1)
*****printItToo ("*", 0) base case
*****
```

Notice that the number of times the string is printed decreases after each recursive call to printIt

Notice that the base case is eventually reached.

Infinite Recursion

- if the base case(s) is missing, or never reached, a recursive method will run forever (or until the computer runs out of resources)

```
public static void printItForever(String s, int n) {
  // missing base case; infinite recursion
  System.out.print(s);
  printItForever(s, n - 1);
}
```

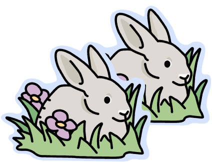
```
printItForever("*", 1)
* printItForever("*", 0)
** printItForever("*", -1)
*** printItForever("*", -2) ...........
```

Climbing a Flight of n Stairs

- not Java

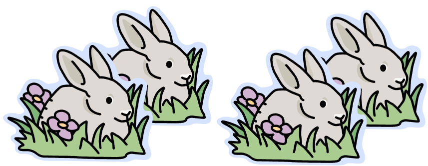
```
climb(n) :
if n == 0
  done
else
  step up 1 stair
  climb(n – 1);
end
```

Rabbits



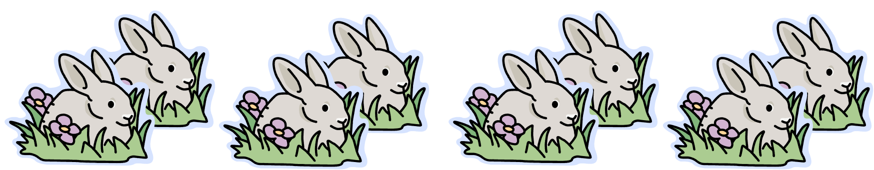
Month 0: 1 pair

0 additional pairs



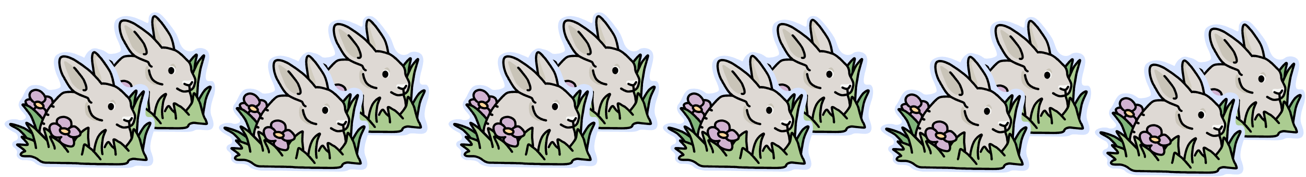
Month 1: first pair makes another pair

1 additional pair



Month 2: each pair makes another pair; oldest pair dies

1 additional pair



2 additional pairs

Month 3: each pair makes another pair; oldest pair dies

Fibonacci Numbers

- the sequence of additional pairs
  - **`0, 1, 1, 2, 3, 5, 8, 13, ...`**

  are called Fibonacci numbers

- base cases
  - **`F(0) = 0`**
  - **`F(1) = 1`**
- recursive definition
  - **`F(n) = F(n – 1) +  F(n – 2)`**

Recursive Methods & Return Values

- a recursive method can return a value
- example: compute the nth Fibonacci number

```
public static int fibonacci(int n) {
  if (n == 0) {
    return 0;
  }
  else if (n == 1) {
    return 1;
  }
  else {
   int f = fibonacci(n - 1) + fibonacci(n - 2);
   return f;
  }
}
```

Recursive Methods & Return Values

- example: write a recursive method **`countZeros`** that counts the number of zeros in an integer number **`n`**
  - **`10305060700002L`** has 8 zeros

- trick: examine the following sequence of numbers
  1. **`10305060700002`**
  2. **`10305060700000`**
  3. **`1030506070000`**
  4. **`103050607000`**
  5. **`103050607`**
  6. **`1030506 ...`**

Recursive Methods & Return Values

- not Java:

```
countZeros(n) :
if the last digit in n is a zero
  return 1 + countZeros(n / 10)
else
  return countZeros(n / 10)
```

- don't forget to establish the base case(s)
  - when should the recursion stop? when you reach a single digit (not zero digits; you never reach zero digits!)
    - base case #1 : `n == 0`
      - `return 1`
    - base case #2 : `n != 0 && n < 10`
      - `return 0`

```java
public static int countZeros(long n) {

  if(n == 0L) {  // base case 1
    return 1;
  }
  else if(n < 10L) {  // base case 2
    return 0;
  }

  boolean lastDigitIsZero = (n % 10L == 0);
  final long m = n / 10L;
  if(lastDigitIsZero) {
    return 1 + countZeros(m);
  }
  else {
    return countZeros(m);
  }
}
```

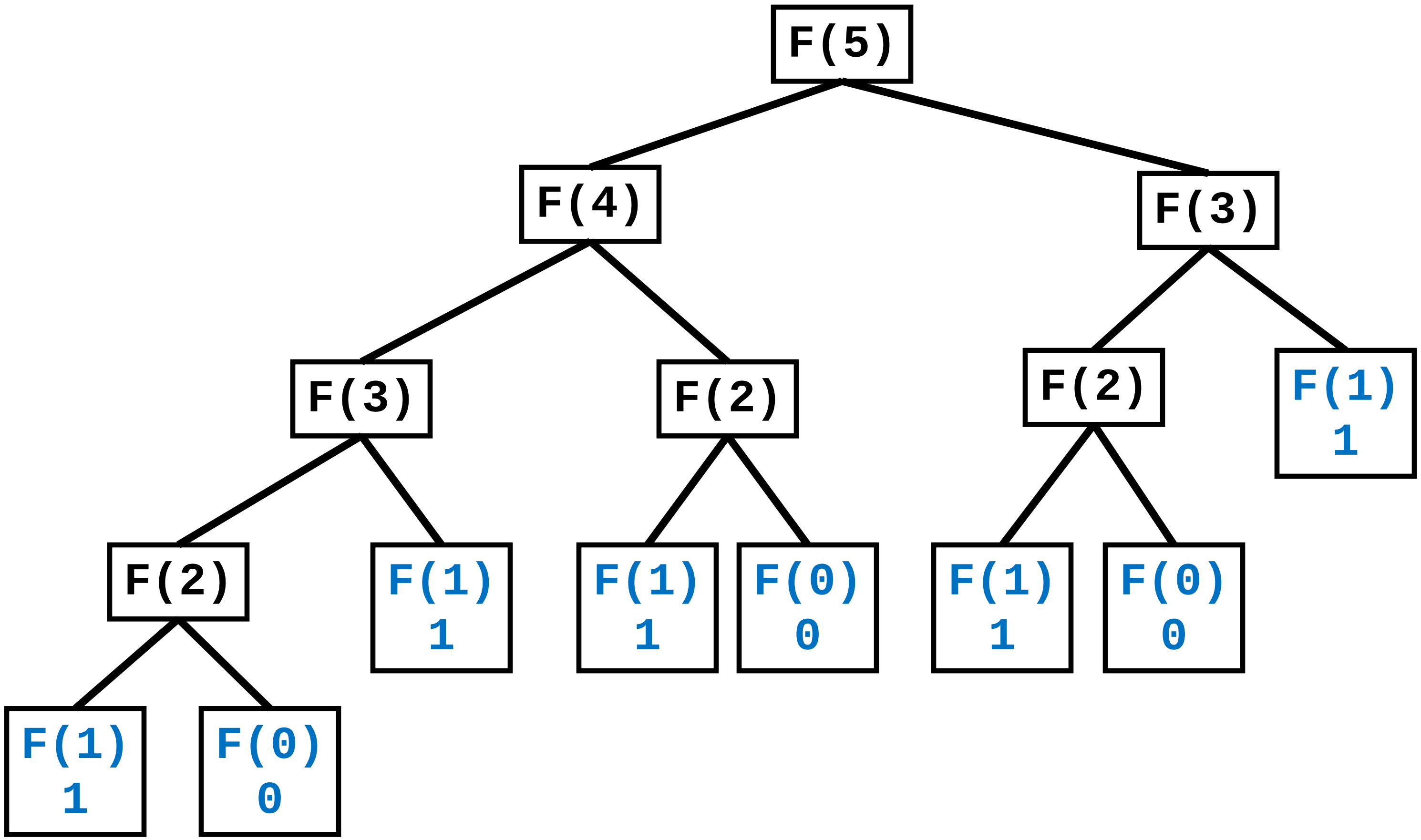
countZeros Call Stack

`callZeros( 800410L )`

last in first out

`callZeros( 8L )`	`0`
`callZeros( 80L )`	`1 + 0`
`callZeros( 800L )`	`1 + 1 + 0`
`callZeros( 8004L )`	`0 + 1 + 1 + 0`
`callZeros( 80041L )`	`0 + 0 + 1 + 1 + 0`
`callZeros( 800410L )`	`1 + 0 + 0 + 1 + 1 + 0`
	`= 3`

Fibonacci Call Tree



Compute Powers of 10

- write a recursive method that computes 10^{n} for any integer value **n**
- recall:
  - $10^{0} = 1$
  - $10^{n} = 10 * 10^{n-1}$
  - $10^{-n} = 1 / 10^{n}$

```
public static double powerOf10(int n) {
  if (n == 0) {
    // base case
    return 1.0;
  }
  else if (n > 0) {
    // recursive call for positive n
    return 10.0 * powerOf10(n - 1);
  }
  else {
    // recursive call for negative n
    return 1.0 / powerOf10(-n);
  }
}
```

Proving Correctness and Termination

- to show that a recursive method accomplishes its goal you must prove:
  1. that the base case(s) and the recursive calls are correct
  2. that the method terminates

Proving Correctness

- to prove correctness:
  1. prove that each base case is correct
  2. assume that the recursive invocation is correct and then prove that each recursive case is correct

printItToo

```
public static void printItToo(String s, int n) {
  if (n == 0) {
    return;
  }
  else {
    System.out.print(s);
    printItToo(s, n - 1);
  }
}
```

Correctness of printItToo

1. (prove the base case) If `n == 0` nothing is printed; thus the base case is correct.
2. Assume that `printItToo(s, n-1)` prints the string `s` exactly `(n – 1)` times. Then the recursive case prints the string `s` exactly `(n – 1)+1 = n` times; thus the recursive case is correct.

Proving Termination

- to prove that a recursive method terminates:
  1. define the size of a method invocation; the size must be a non-negative integer number
  2. prove that each recursive invocation has a smaller size than the original invocation

Termination of printIt

1. **`printIt(s, n)`** prints **`n`** copies of the string **`s`**; define the size of **`printIt(s, n)`** to be **`n`**
2. The size of the recursive invocation **`printIt(s, n-1)`** is **`n-1`** (by definition) which is smaller than the original size **`n`**.

countZeros

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

  if(n == 0L) {  // base case 1
    return 1;
  }
  else if(n < 10L) {  // base case 2
    return 0;
  }

  boolean lastDigitIsZero = (n % 10L == 0);
  final long m = n / 10L;
  if(lastDigitIsZero) {
    return 1 + countZeros(m);
  }
  else {
    return countZeros(m);
  }
}
```

Correctness of countZeros

1. (base cases) If the number has only one digit then the method returns **`1`** if the digit is zero and **`0`** if the digit is not zero; therefore, the base case is correct.
2. (recursive cases) Assume that **`countZeros(n/10L)`** is correct (it returns the number of zeros in the first **`(d – 1)`** digits of **`n`**). If the last digit in the number is zero, then the recursive case returns **`1 +`** the number of zeros in the first **`(d – 1)`** digits of **`n`**, otherwise it returns the number of zeros in the first **`(d – 1)`** digits of **`n`**; therefore, the recursive cases are correct.

Termination of countZeros

1. Let the size of **`countZeros(n)`** be **`d`** the number of digits in the number **`n`**.
2. The size of the recursive invocation **`countZeros(n/10L)`** is **`d-1`**, which is smaller than the size of the original invocation.

Decrease and Conquer

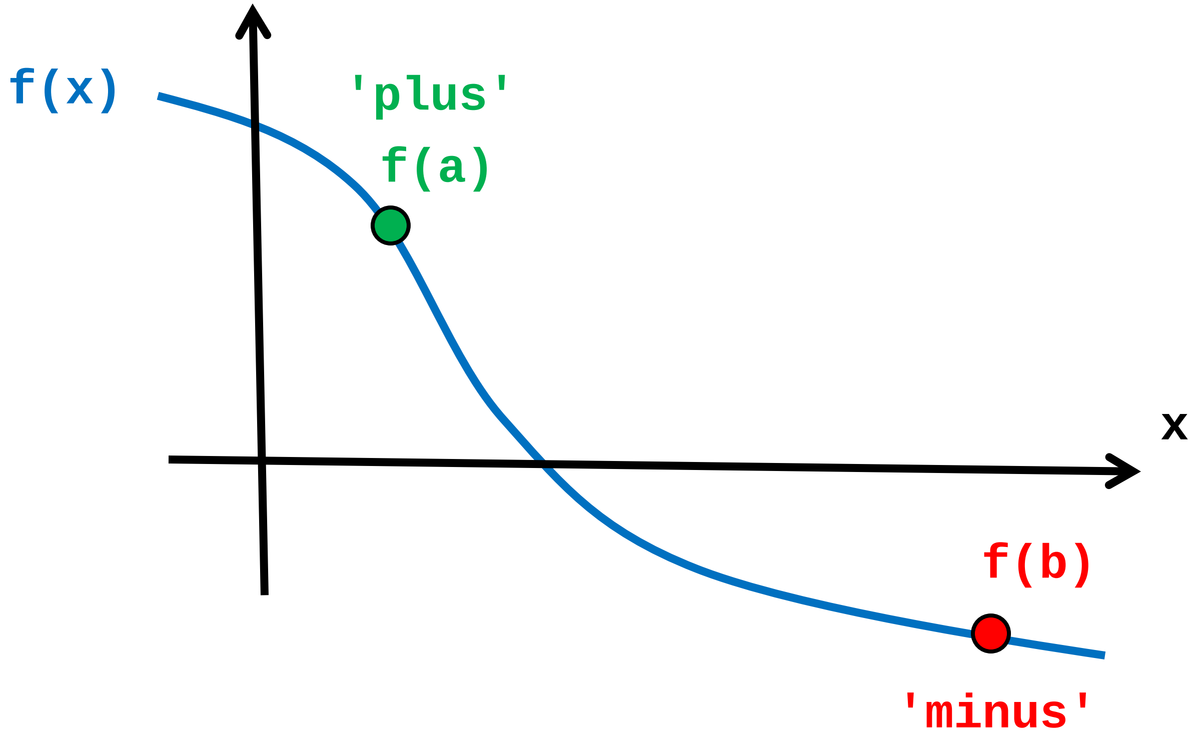
- a common strategy for solving computational problems
  - solves a problem by taking the original problem and converting it to *one* smaller version of the same problem
    - note the similarity to recursion
- decrease and conquer, and the closely related divide and conquer method, are widely used in computer science
  - allow you to solve certain complex problems easily
  - help to discover efficient algorithms

Root Finding

- suppose you have a mathematical function `f(x)` and you want to find x_0 such that $f(x_0) = 0$
  - why would you want to do this?
  - many problems in computer science, science, and engineering reduce to optimization problems
    - find the shape of an automobile that minimizes aerodynamic drag
    - find an image that is similar to another image (minimize the difference between the images)
    - find the sales price of an item that maximizes profit
  - if you can write the optimization criteria as a function `g(x)` then its derivative `f(x) = dg/dx = 0` at the minimum or maximum of `g` (as long as `g` has certain properties)

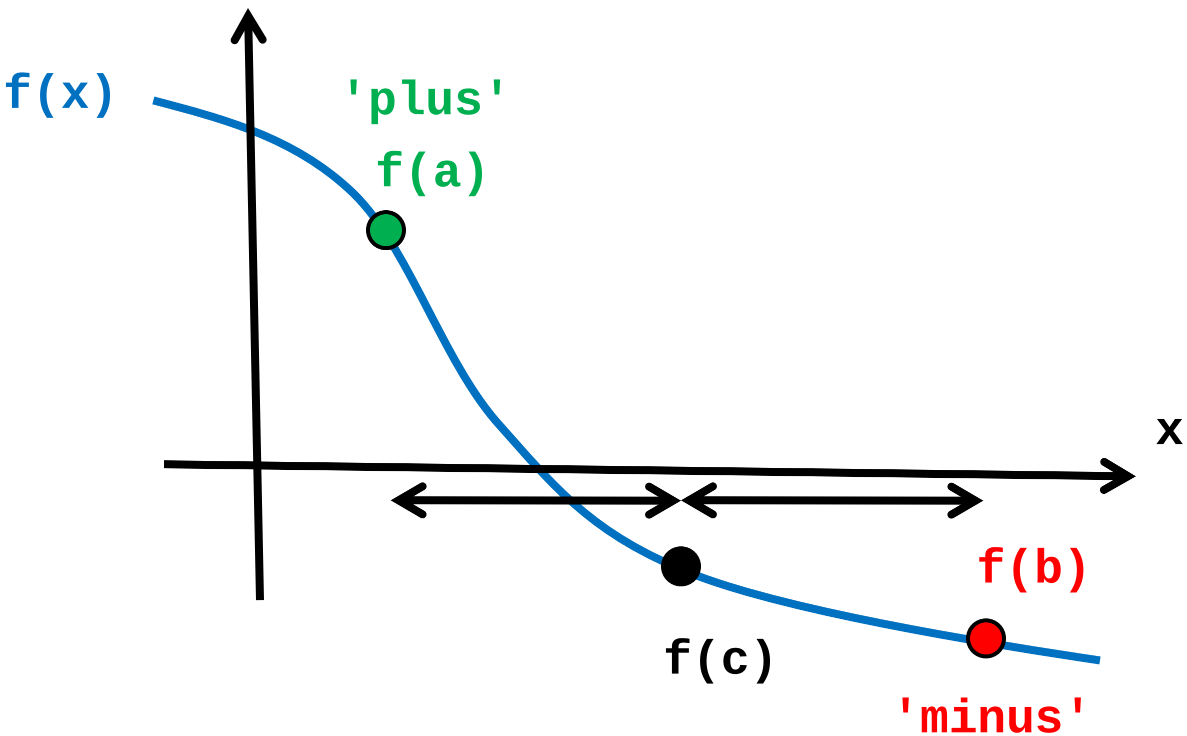
Bisection Method

- suppose you can evaluate `f(x)` at two points `x = a` and `x = b` such that
  - `f(a) > 0`
  - `f(b) < 0`



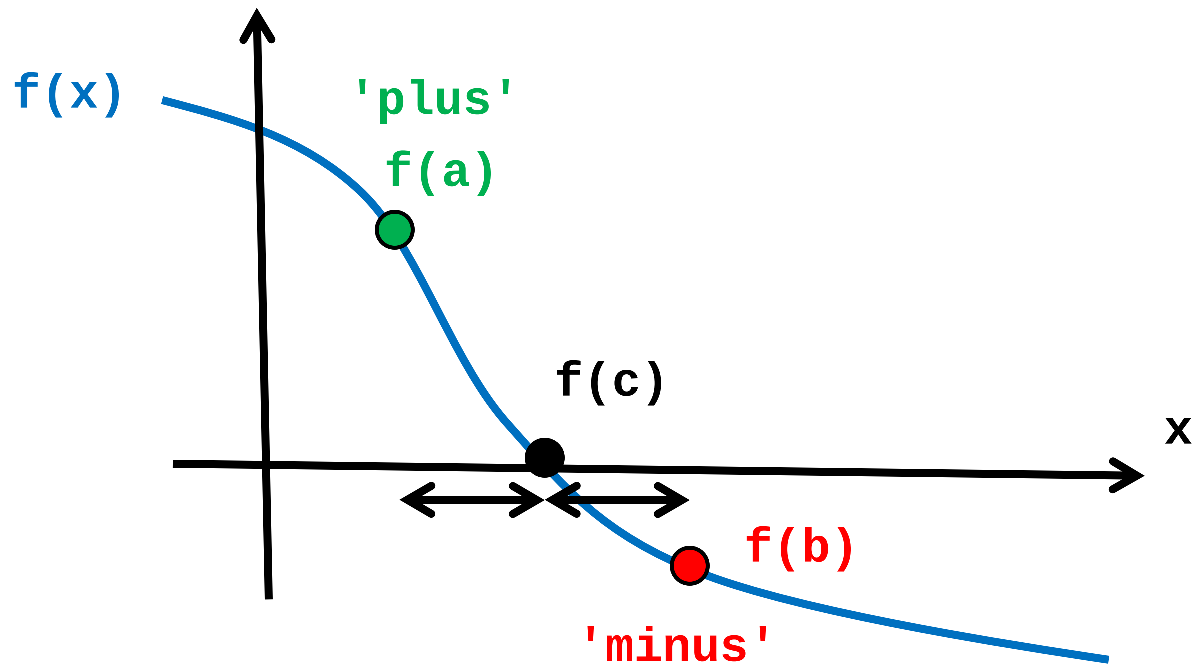
Bisection Method

- evaluate **`f(c)`** where **`c`** is halfway between **`a`** and **`b`**
  - if **`f(c)`** is close enough to zero done



Bisection Method

- otherwise **`c`** becomes the new end point (in this case, `'minus'`) and recursively search the range `'plus'` – `'minus'`



```
public class Bisect {

  // the function we want to find the root of
  public static double f(double x) {
    return Math.cos(x);
  }
```

```
public static double bisect(double xplus, double xminus,
                            double tolerance) {
   // base case
   double c = (xplus + xminus) / 2.0;
   double fc = f(c);
   if( Math.abs(fc) < tolerance ) {
     return c;
   }
   else if (fc < 0.0) {
     return bisect(xplus, c, tolerance);
   }
   else {
     return bisect(c, xminus, tolerance);
   }
 }
```

```
    public static void main(String[] args)
    {
          System.out.println("bisection returns: " +
                               bisect(1.0, Math.PI, 0.001));
          System.out.println("true answer      : "
                               + Math.PI / 2.0);
    }
}

prints:

bisection returns: 1.5709519476855602
true answer      : 1.5707963267948966
```

Divide and Conquer

- bisection works by recursively finding which half of the range `'plus'` – `'minus'` the root lies in
  - each recursive call solves the same problem (tries to find the root of the function by guessing at the midpoint of the range)
  - each recursive call solves *one* smaller problem because half of the range is discarded
    - bisection method is decrease and conquer
- divide and conquer algorithms typically recursively divide a problem into several smaller sub-problems until the sub-problems are small enough that they can be solved directly