Implementing Recursion

Based on slides by Prof. Burton Ma

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

```
printIt("*", 5)
*printIt("*", 4)
**printIt("*", 3)
***printIt("*", 2)
****printIt("*", 1)
*****printIt("*", 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);
}
```

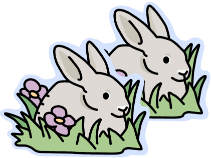
```
printIt("*", 1)
* printIt("*", 0)
** printIt("*", -1)
*** printIt("*", -2) ...........
```

Climbing a Flight of n Stairs

- Not Java

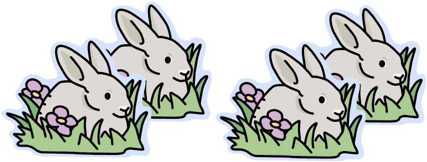
```
climb(n) :
if n == 0
  done
else
  step up 1 stair
  climb(n – 1);
end
```

Rabbits



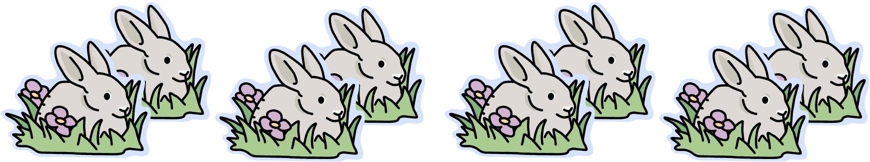
Month 0: 1 pair

0 additional pairs



Month 1: first pair makes another pair

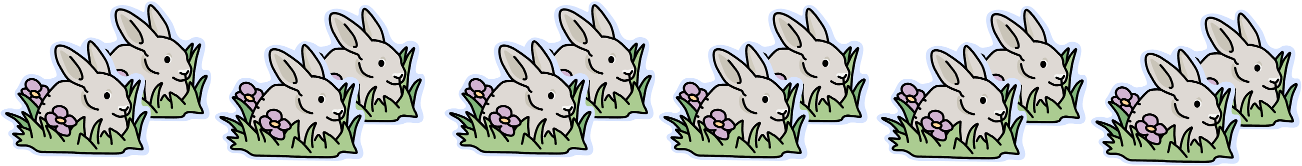
1 additional pair



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. **`1030506070000`**
3. **`103050607000`**
4. **`10305060700`**
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

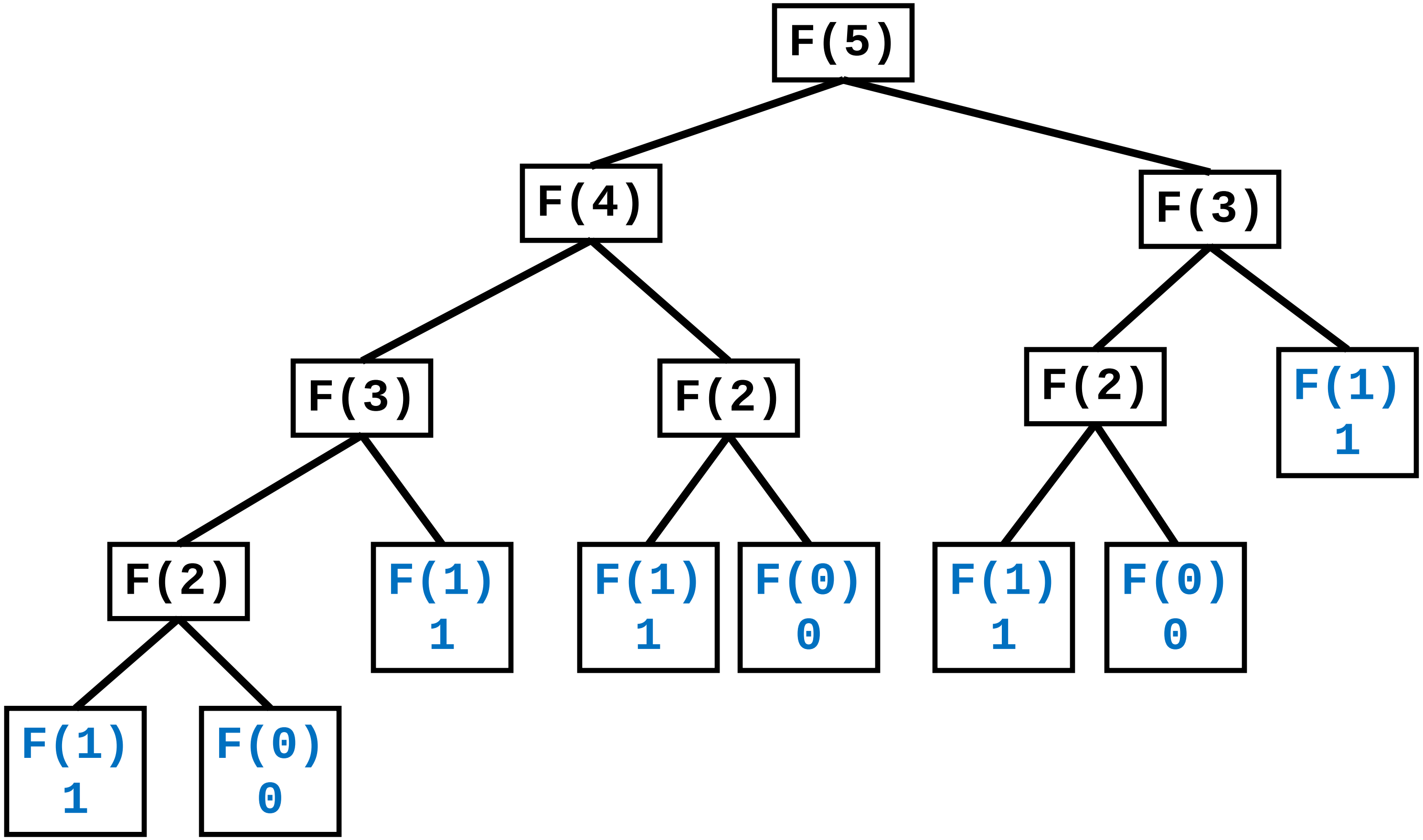
`callZeros( 800410L )`

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

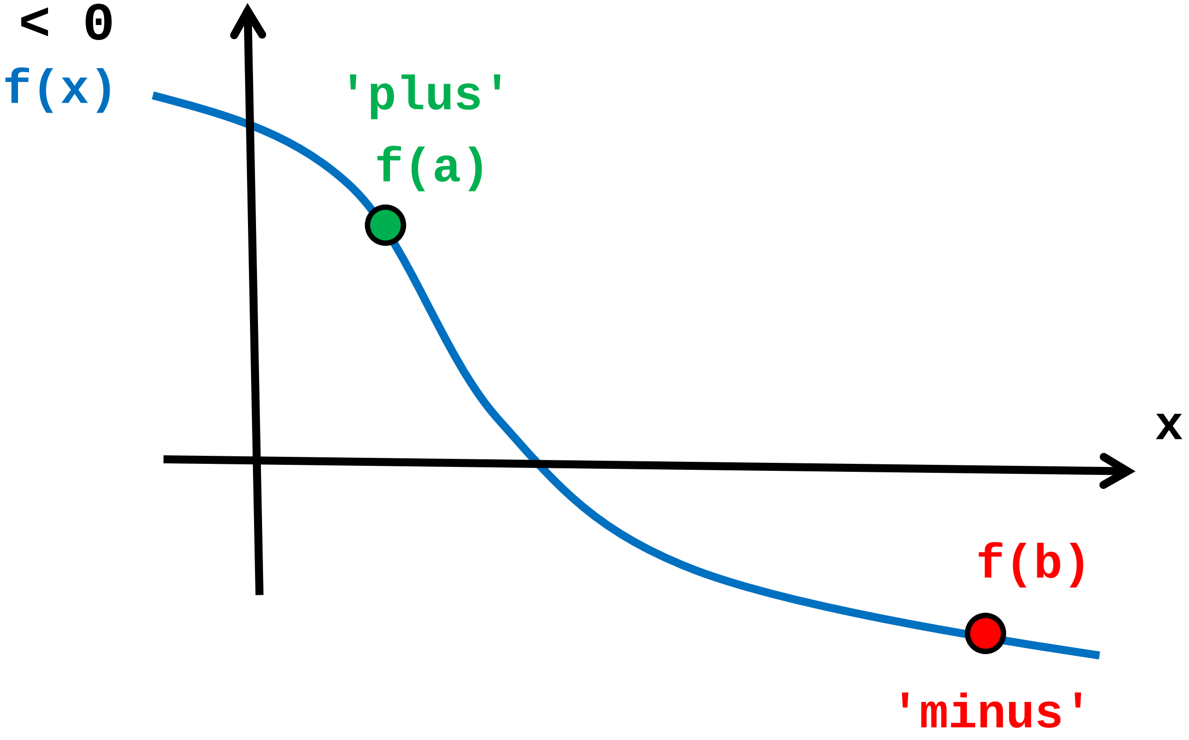
- A common strategy for solving computational problems
  - Solves a problem by taking the original problem and converting it to *one* smaller version of the same problem
    - Note the similarity to recursion
- Decrease and conquer, and the closely related divide and conquer method, are widely used in 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 $\mathbf{x_0}$ such that $\mathbf{f(x_0) = 0}$
  - 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)

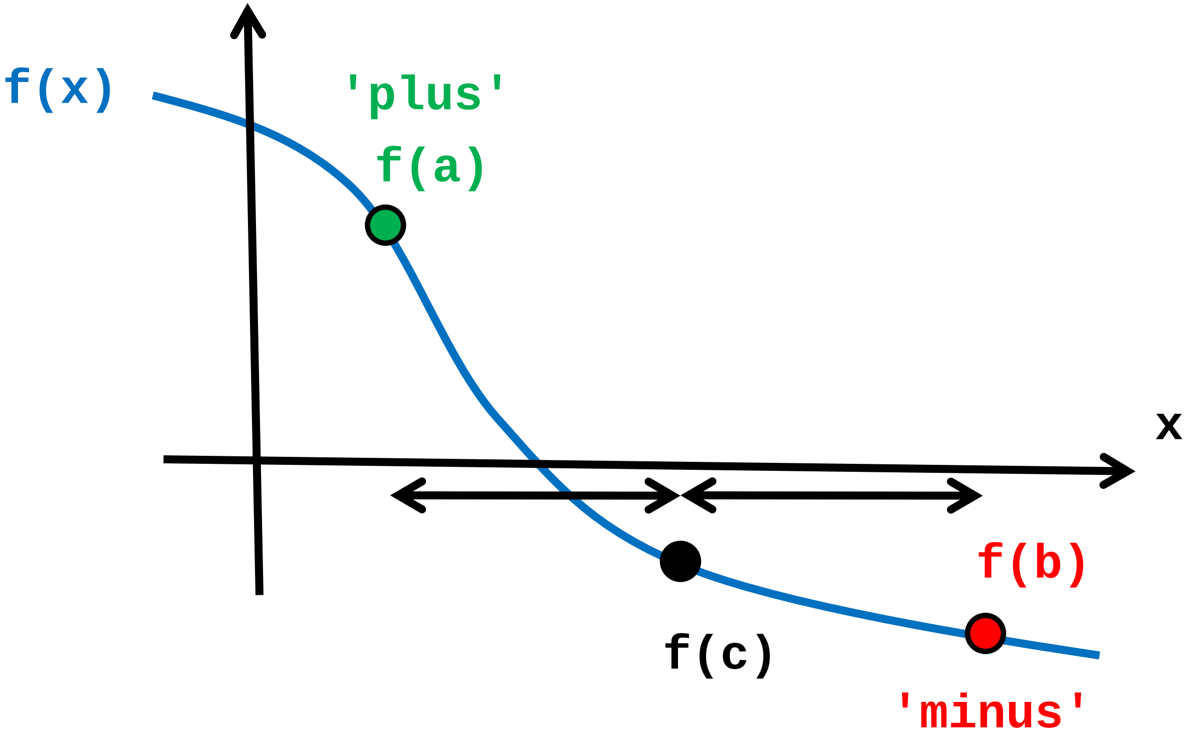
Bisection Method

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



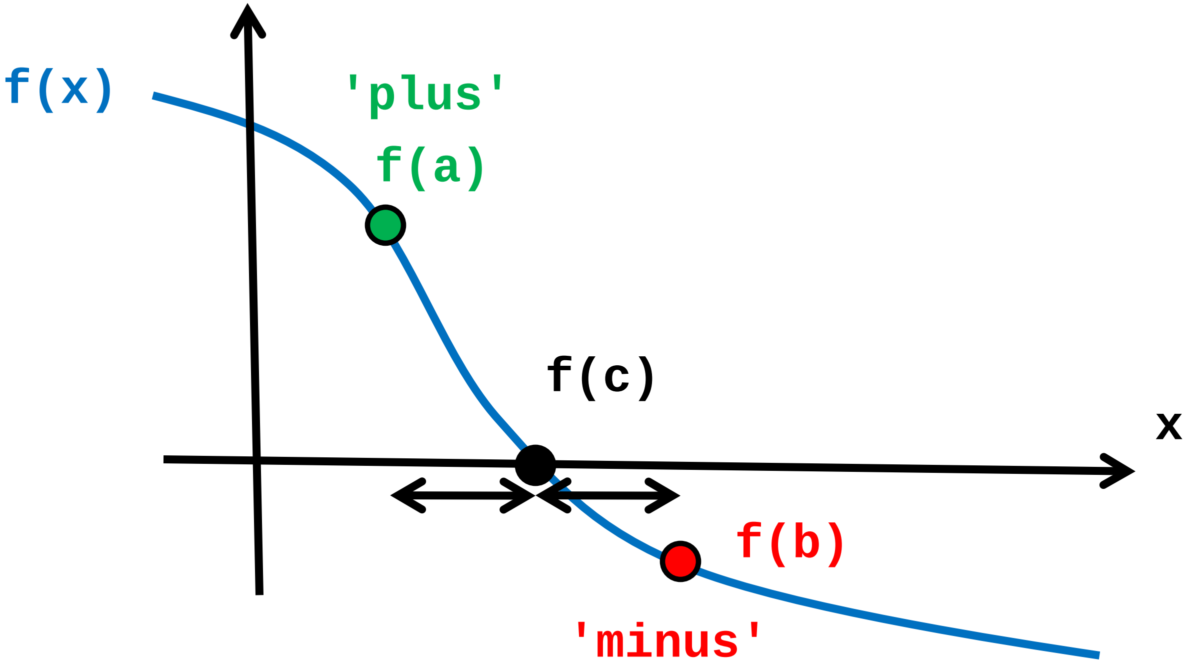
Bisection Method

- 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:

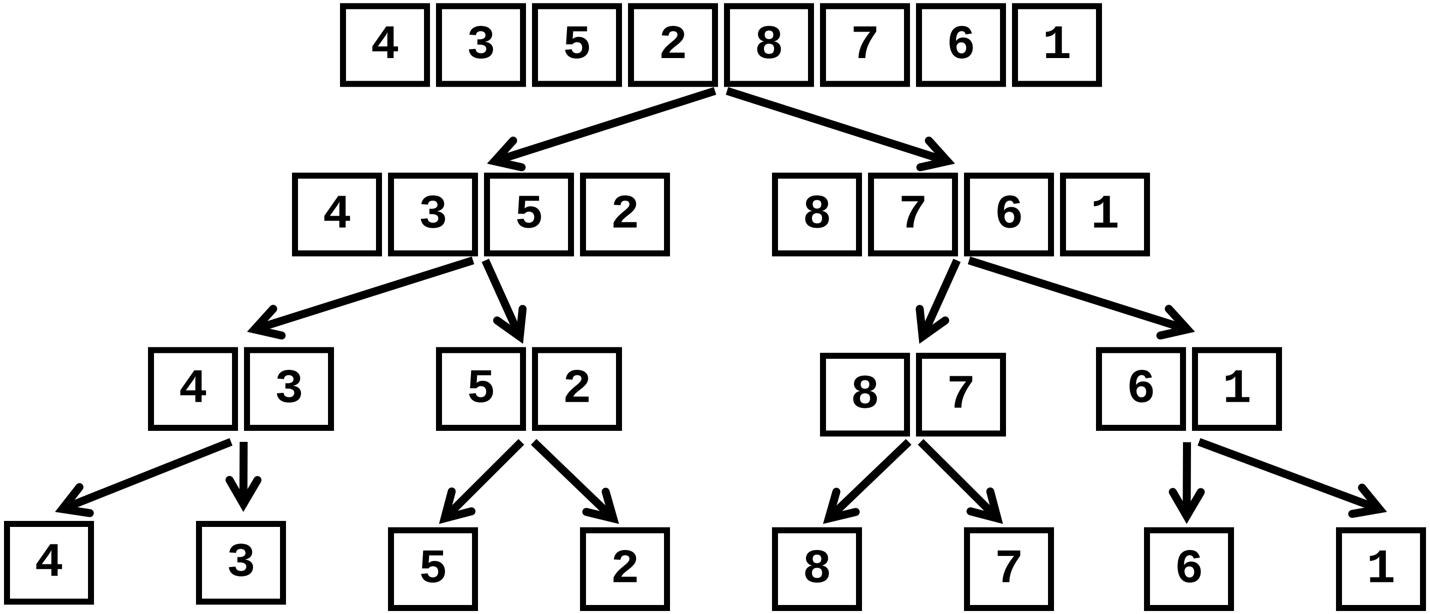
```
bisection returns: 1.5709519476855602
true answer      : 1.5707963267948966
```

Divide and Conquer

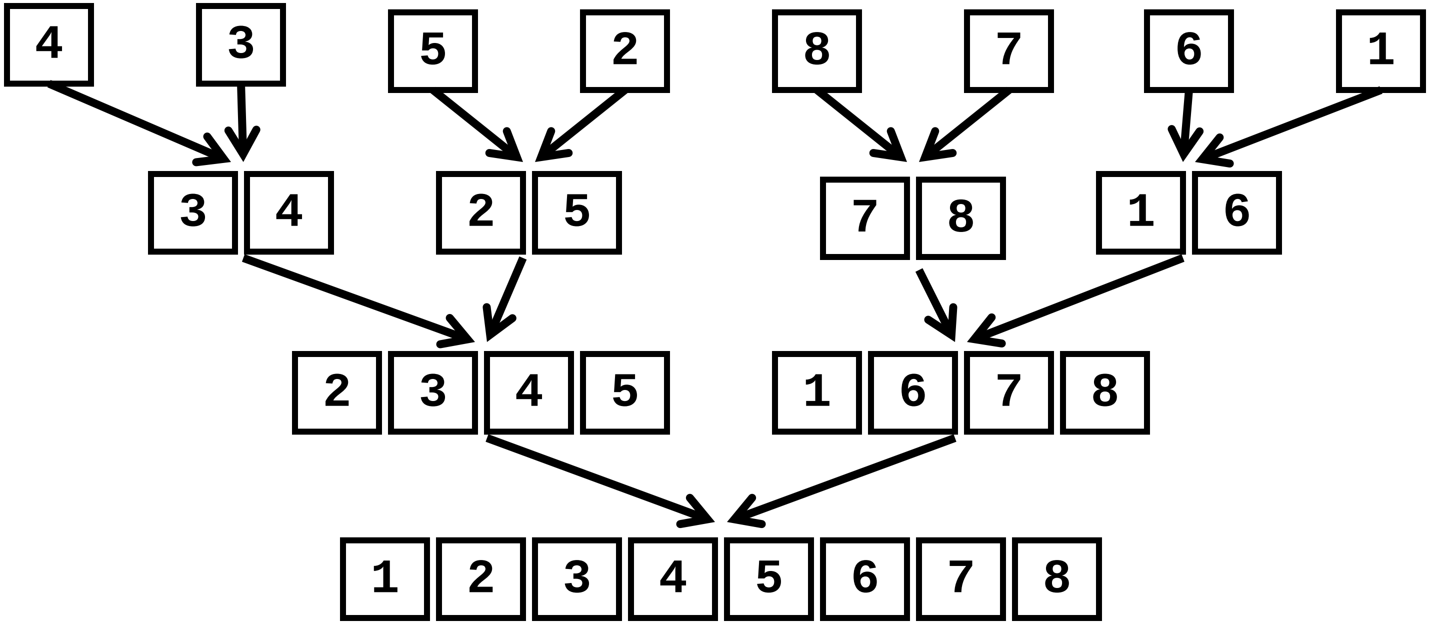
- Bisection works by recursively finding which half of the range `'plus'` – `'minus'` the root lies in
  - Each recursive call solves the same problem (tries to find the root of the function by guessing at the midpoint of the range)
  - Each recursive call solves *one* smaller problem because half of 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

Merge Sort

- Merge sort is a divide and conquer algorithm that sorts a list of numbers by recursively splitting the list into two halves

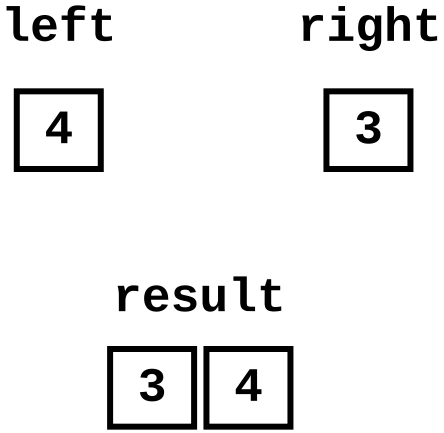


- The split lists are then merged into sorted sub-lists



Merging Sorted Sub-lists

- Two sub-lists of length 1



left

4

right

3

result

3 4

1 Comparison

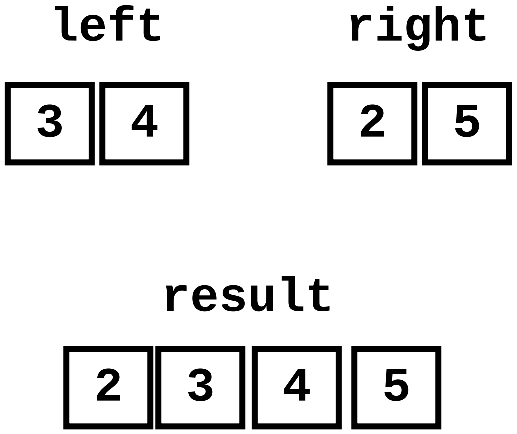
2 Copies

```
LinkedList<Integer> result = new LinkedList<Integer>();

int fL = left.getFirst();
int fR = right.getFirst();
if (fL < fR) {
  result.add(fL);
  left.removeFirst();
}
else {
  result.add(fR);
  right.removeFirst();
}
if (left.isEmpty()) {
  result.addAll(right);
}
else {
  result.addAll(left);
}
```

Merging Sorted Sub-lists

- Two sub-lists of length 2



left

3	4

right

2	5

result

2	3	4	5

3 Comparisons

4 Copies

```
LinkedList<Integer> result = new LinkedList<Integer>();

while (left.size() > 0 && right.size() > 0 ) {
  int fL = left.getFirst();
  int fR = right.getFirst();
  if (fL < fR) {
    result.add(fL);
    left.removeFirst();
  }
  else {
    result.add(fR);
    right.removeFirst();
  }
}
if (left.isEmpty()) {
  result.addAll(right);
}
else {
  result.addAll(left);
}
```

Merging Sorted Sub-lists

- Two sub-lists of length 4

`left`

2	3	4	5

`right`

1	6	7	8

`result`

1	2	3	4	5	6	7	8

5 Comparisons
8 Copies

Simplified Complexity Analysis

- In the worst case merging a total of **n** elements requires

  n – 1 comparisons **+**

  n copies

  = 2n – 1 total operations

- We say that the worst-case complexity of merging is the order of *O(n)*
  - *O(...)* is called Big O notation
  - Notice that we don't care about the constants 2 and 1

- Formally, a function *f(n)* is an element of *O(n)* if and only if there is a positive real number *M* and a real number *m* such that

  $$| f(n) | < Mn \text{ for all } n > m$$

- Is $\mathbf{2}n - \mathbf{1}$ an element of *O(n)*?
  - Yes, let $M = \mathbf{2}$ and $m = \mathbf{0}$,
    then $\mathbf{2}n - \mathbf{1} < \mathbf{2}n$ for all $n > \mathbf{0}$

Informal Analysis of Merge Sort

- Suppose the running time (the number of operations) of merge sort is a function of the number of elements to sort
  - Let the function be *T(n)*
- Merge sort works by splitting the list into two sub-lists (each about half the size of the original list) and sorting the sub-lists
  - This takes **2***T(n/***2***)* running time
- Then the sub-lists are merged
  - This takes *O(n)* running time
- Total running time *T(n)* = **2***T(n/***2***)* + *O(n)*

Solving the Recurrence Relation

$T(n)$ approaches...

$$
\begin{aligned}
T(n) &\rightarrow \mathbf{2}T(n/\mathbf{2}) + O(n) \\
&\approx \mathbf{2}T(n/\mathbf{2}) + n \\
&= \mathbf{2}[\ \mathbf{2}T(n/\mathbf{4}) + n/\mathbf{2}\ ] + n \\
&= \mathbf{4}T(n/\mathbf{4}) + \mathbf{2}n \\
&= \mathbf{4}[\ \mathbf{2}T(n/\mathbf{8}) + n/\mathbf{4}\ ] + \mathbf{2}n \\
&= \mathbf{8}T(n/\mathbf{8}) + \mathbf{3}n \\
&= \mathbf{8}[\ \mathbf{2}T(n/\mathbf{16}) + n/\mathbf{8}\ ] + \mathbf{3}n \\
&= \mathbf{16}T(n/\mathbf{16}) + \mathbf{4}n \\
&= \mathbf{2}^kT(n/\mathbf{2}^k) + kn
\end{aligned}
$$

Solving the Recurrence Relation

$$T(n) \quad = \qquad \mathbf{2}^k T(\underline{n/\mathbf{2}^k}) + kn$$

- For a list of length $\mathbf{1}$ we know $T(\mathbf{1}) = \mathbf{1}$
  - If we can substitute $T(1)$ into the right-hand side of $T(n)$ we might be able to solve the recurrence

$$\underline{\;n/\mathbf{2}^k} = \mathbf{1} \;\Rightarrow\; \mathbf{2}^k = n \Rightarrow k = \log(n)$$

Solving the Recurrence Relation

$$
\begin{aligned}
T(n) &= \mathbf{2}^{\log(n)} T(n/\mathbf{2}^{\log(n)}) + n \log(n) \\
&= n\, T(\mathbf{1}) + n \log(n) \\
&= n + n \log(n) \\
&\in n \log(n)
\end{aligned}
$$

Is Merge Sort Efficient?

- Consider a simpler (non-recursive) sorting algorithm called insertion sort

```
// to sort an array a[0]..a[n-1]                not Java!
for i = 0 to (n-1) {
   k = index of smallest element in sub-array a[i]..a[n-1]
   swap a[i] and a[k]
}
```

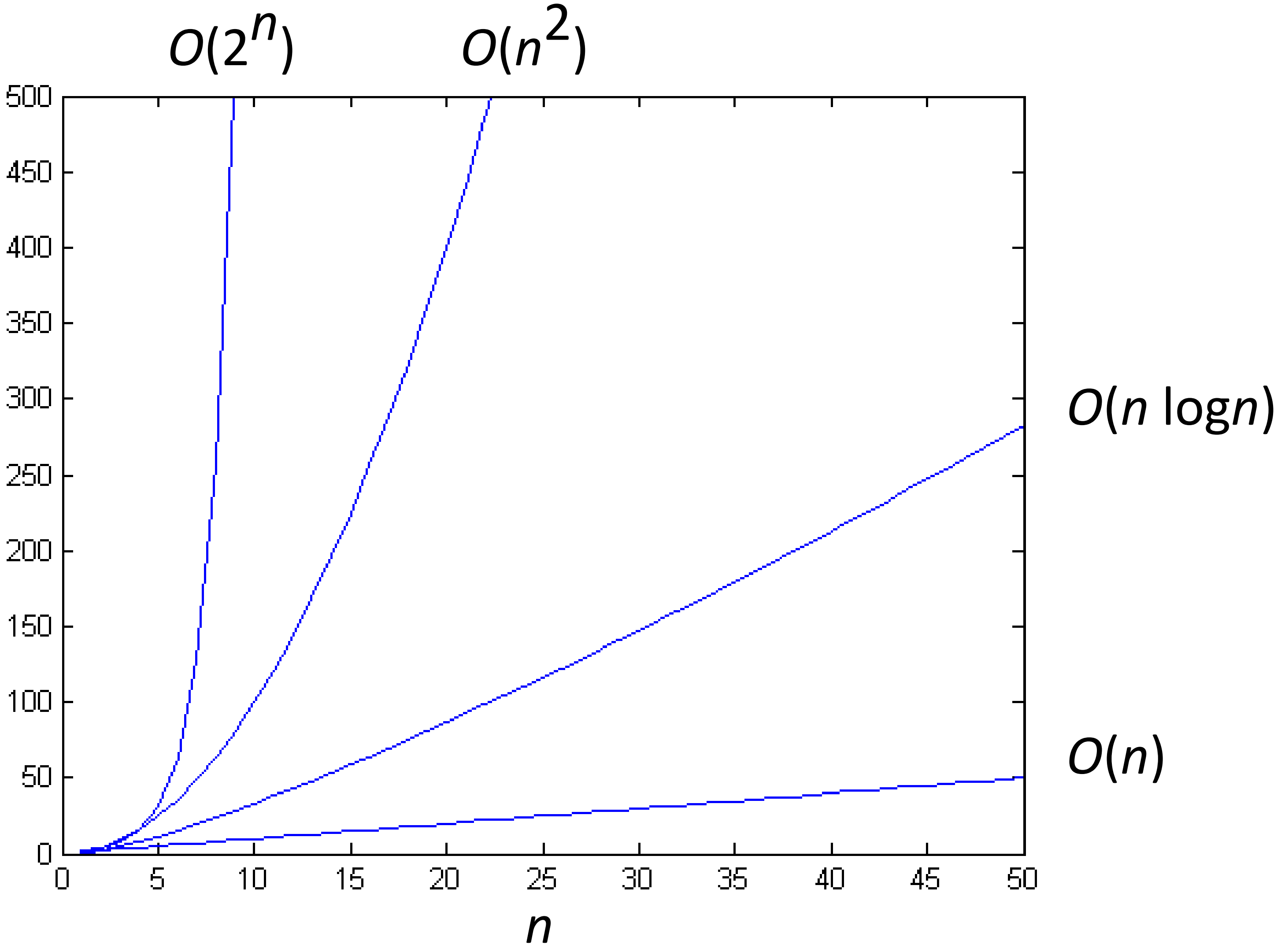
```
for i = 0 to (n-1) {                             not Java!
   for j = (i+1) to (n-1) {
      if (a[j] < a[i]) {
         k = j;
      }
   }
   tmp = a[i];   a[i] = a[k];   a[k] = tmp;
}
```

1 comparison +
1 assignment

3 assignments

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{n-1}\left(\left(\sum_{j=(i+1)}^{n-1} 2\right) + 3\right) \\
&= \sum_{i=0}^{n-1}\left(2\left(n - i - 1\right)\right) \quad + \quad 3n \\
&= 2\sum_{i=0}^{n-1} n \quad - \quad 2\sum_{i=0}^{n-1} i \quad - \quad 2\sum_{i=0}^{n-1} 1 \quad + \quad 3n \\
&= 2n^2 \quad - \quad 2\frac{n\left(n-1\right)}{2} \quad - \quad 2n \quad + \quad 3n \\
&= 2n^2 \quad - \quad n^2 \quad + \quad n \quad - \quad 2n \quad + \quad 3n \\
&= n^2 \quad + \quad 2n \quad \in O\left(n^2\right)
\end{aligned}
$$

Comparing Rates of Growth



Comments

- Big O complexity tells you something about the running time of an algorithm as the size of the input, *n*, approaches infinity
  - We say that it describes the limiting, or asymptotic, running time of an algorithm
- For small values of *n* it is often the case that a less efficient algorithm (in terms of big O) will run faster than a more efficient one
  - Insertion sort is typically faster than merge sort for short lists of numbers