CSCI 136
Data Structures &
Advanced Programming

Lecture 7

Fall 2019

Instructors: Bill &Sam

Last Time

- Vector Implementation
- Condition Checking
    - Pre- and post-conditions

Today

- Problem set 1 and handout
- Assertions
- Asymptotic Growth & Measuring Complexity (from previous slide deck)
- Introduction to Recursion & Induction
  - (maybe)

Pre and Post Conditions

- Recall `charAt(int index)` in Java String class
- What are the pre-conditions for charAt?
  - 0 <= index < length()
- What are the post-conditions?
  - Method returns char at position index in string
- We put pre and post conditions in comments above most methods

```
/* pre: 0 ≤ index < length
 * post: returns char at position index
 */
public char charAt(int index) { … }
```

Pre and Post Conditions

- Pre and post conditions "form a contract"
- Post-condition is guaranteed if method is called when pre-condition is true
- Examples:
  - `s.charAt(s.length() - 1)`: index < length, so valid
  - `s.charAt(s.length() + 1)`: index > length, not valid
- These conditions document requirements that user of method should satisfy
- But, as comments, they are not enforced

Other Examples

- Other places pre and post conditions are useful

```
    // Pre: other is of type Card
    // Post: Returns true if suits and ranks match
    public boolean equals(Object other) {
       Card oc = (Card) other;
       return this.getRank() == oc.getRank() &&
              this.getSuit() == oc.getSuit();
}
```

Assert Class

- Pre- and post-condition comments are important for *documenting* code.
- Better if the program could catch error and "gracefully" halt (with useful information)
- The Assert class (in structure5 package) allows us to programmatically check for pre- and post-conditions

Assert Class

The Assert class contains the methods

```
public static void pre(boolean test, String message);
public static void post(boolean test, String message);
public static void condition(boolean test, String message);
public static void fail(String message);
```

If the boolean test is NOT satisfied, an exception is raised, the message is printed and the program halts

Assert Examples

The Vector class uses Assert in many places

```
// Pre: initialCapacity >= 0
public Vector(int initialCapacity) {
      Assert.pre(initialCapacity >= 0,"Capacity
            must not be negative");

// Pre: 0 <= index && index < size()
public E elementAt(int index) {
      Assert.pre(0 <= index && index < size(),"index
            is within bounds");
```

General Rules about Assert

1. State pre/post conditions in comments
2. Check conditions in code using “Assert”
3. Use Fail in unexpected cases (such as the default block of a switch statement)

- Any questions?
- You can start using Assertions in Lab 2

The Java assert keyword

- An alternative to Duane's Assert class
- Added in Java 1.4
- Two variants
  - assert boolean_expression
    - Throws an AssertionError if the expression is false
  - assert boolean_expression : other_expression
    - In addition, prints value of other_expression

Measuring Computational Cost

Consider these two code fragments…

```
for (int i=0; i < arr.length; i++)
   if (arr[i] == x) return "Found it!";
```

…and…

```
for (int i=0; i < arr.length; i++)
   for (int j=0; j < arr.length; j++)
     if( i !=j && arr[i] == arr[j]) return "Match!";
```

How long does it take to execute each block?

Measuring Computational Cost

- How can we measure the amount of work needed by a computation?
  - Absolute clock time
    - Problems?
      - Different machines have different clocks
      - Too much other stuff happening (network, OS, etc)
      - Hardware changes can have significant effects
      - Not consistent. Need lots of tests to predict future behavior

Measuring Computational Cost

- Counting computations
  - Count *all* computational steps?
  - Count how many “expensive” operations were performed?
  - Count number of times “x” happens?
    - For a specific event or action “x”
    - i.e., How many times a certain variable changes
- Question: How accurate do we need to be?
  - 64 vs 65? 100 vs 105? Does it really matter??

An Example

```
// Pre: array length n > 0
public static int findPosOfMax(int[] arr) {
      int maxPos = 0 // A wild guess
      for(int i = 1; i < arr.length; i++)
            if (arr[maxPos] < arr[i]) maxPos = i;
      return maxPos;
}
```

- Can we count steps exactly?
  - ”if” makes it hard
- Idea: Overcount: assume “if” block always runs
- Overcounting gives *upper bound* on run time
- Can also undercount for lower bound
- Overcount: 4(n-1) + 4; undercount: 3(n-1) + 4

Measuring Computational Cost

- Rather than keeping exact counts, we want to know the *order of magnitude* of occurrences
  - 60 vs 600 vs 6000, *not* 65 vs 68
  - n, *not* 4(n-1) + 4
- We want to make comparisons without looking at details and without running tests
- Avoid using specific numbers or values
- Look for overall trends as data grows

Measuring Computational Cost

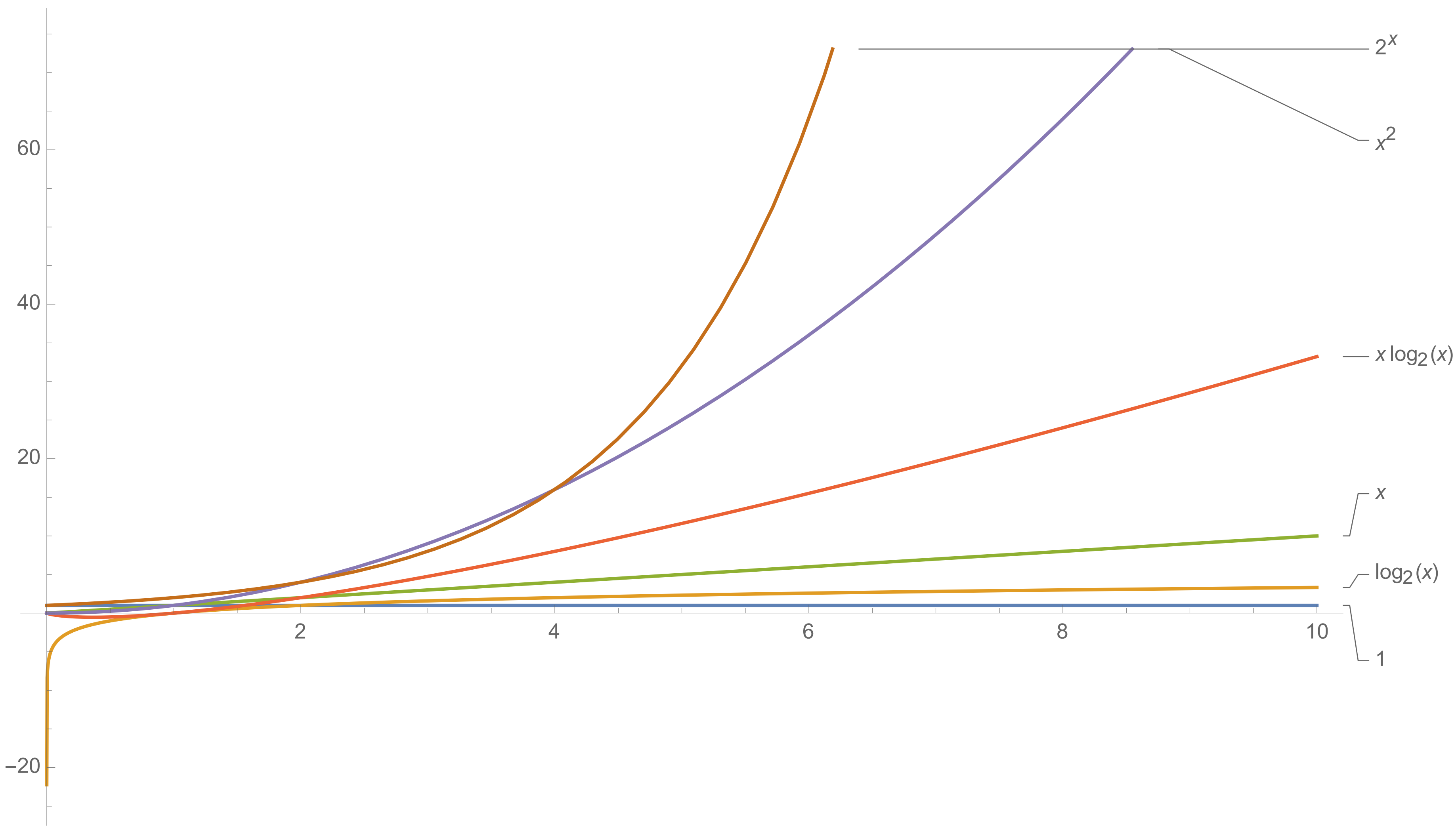
- How does algorithm scale with problem size?
  - E.g.: If I double the size of the problem instance, how much longer will it take to solve:
    - Find maximum: $(n-1) \rightarrow (2n-1)$ (≈ **twice as** long)
    - Bubble sort: $\frac{n(n-1)}{2} \rightarrow \frac{2n(2n-1)}{2}$ (≈ 4 times as long)
    - Subset sum: $2^{n-1} \rightarrow 2^{2n-1}$ (2^n times as long!!!)
    - Etc.
- We will also measure amount of space used by an algorithm using the same ideas….

Function Growth

Consider the following functions, for $x \geq 1$

- $f(x) = 1$
- $g(x) = \log_2 x$ // Reminder: if $x = 2^n$, $\log_2 x = n$
- $h(x) = x$
- $m(x) = x \log_2 x$
- $n(x) = x^2$
- $p(x) = x^3$
- $r(x) = 2^x$

Function Growth



Function Growth

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Function Growth & Big-O

- Rule of thumb: ignore multiplicative constants
- Examples:
  - Treat n and n/2 as same order of magnitude
  - $n^2/1000$, $2n^2$, and $1000n^2$ are "pretty much" just n^2
  - $a_0 n^k + a_1 n^{k-1} + a_1 n^{k-2} + a_k$ is roughly n^k
- The key is to find the most *significant* or *dominant* term
- Ex: $\lim_{x\to\infty} (3x^4 - 10x^3 - 1)/x^4 = 3$ (Why?)
  - So $3x^4 - 10x^3 - 1$ grows "like" x^4

Asymptotic Bounds (Big-O Analysis)

- A function f(n) is *O(g(n))* if and only if there exist positive constants c and n_0 such that

$$|f(n)| \leq c \cdot g(n) \text{ for all } n \geq n_0$$

- $c \cdot g$ is "at least as big as" f **for large n**
  - Up to a multaplicative constant c!
- Example:
  - $f(n) = n^2/2$ is $O(n^2)$
  - $f(n) = 1000n^3$ is $O(n^3)$
  - $f(n) = n/2$ is $O(n)$

Determining “Best” Upper Bounds

- We typically want the *most conservative* upper bound when we estimate running time
  - And among those, the *simplest*
- Example: Let $f(n) = 3n^2$
  - $f(n)$ is $O(n^2)$
  - $f(n)$ is $O(n^3)$
  - $f(n)$ is $O(2^n)$ (see next slide)
  - $f(n)$ is NOT $O(n)$ (!!)
- “Best” upper bound is $O(n^2)$
- We care about **c** and $\mathbf{n_0}$ in practice, but focus on size of **g** when designing algorithms and data structures

What's n_0? Messy Functions

- Example: Let $f(n) = 3n^2 - 4n + 1$. $f(n)$ is $O(n^2)$
  - Well, $3n^2 - 4n + 1 \leq 3n^2 + 1 \leq 4n^2$, for $n \geq 1$
  - So, for $c = 4$ and $n_0 = 1$, we satisfy Big-O definition
- Example: Let $f(n) = n^k$, for any fixed $k \geq 1$.

  $f(n)$ is $O(2^n)$
  - Harder to show: Is $n^k \leq c\, 2^n$ for some $c > 0$ and large enough n?
  - It is if and only if $\log_2(n^k) \leq \log_2(2^n)$, that is, iff $k \log_2(n) \leq n$.
  - That is iff $k \leq n/\log_2(n)$. But $n/\log_2(n) \to \infty$ as $n \to \infty$
  - This implies that for some n_0 on $n/\log_2(n) \geq k$ if $n \geq n_0$
  - Thus $n \geq k \log_2(n)$ for $n \geq n_0$ and so $2^n \geq n^k$

Input-dependent Running Times

- Algorithms may have different running times for different input values
- Best case (typically not useful)
  - BubbleSort already sorted array: O(n)
  - Find item in first place that we look: O(1)
- Worst case (generally useful, sometimes misleading)
  - Don't find item in list: O(n)
  - BubbleSort array that's in reverse order: $O(n^2)$
- Average case (useful, but often hard to compute)
  - Linear search O(n)
  - QuickSort random array O(n log n) ← We'll sort soon

Vector Operations : Worst-Case

For n = Vector size (*not* capacity!):

- $O(1)$: size(), capacity(), isEmpty(), get(i), set(i), firstElement(), lastElement()
- $O(n)$: indexOf(), contains(), remove(elt), remove(i)
- What about add methods?
  - If Vector doesn't need to grow
    - add(elt) is $O(1)$ but add(elt, i) is $O(n)$
    - Otherwise, depends on ensureCapacity() time
    - Time to compute newLength : $O(\log n)$
    - Time to copy array: $O(n)$
    - $O(\log n) + O(n)$is $O(n)$

Vector: Add Method Complexity

Suppose we grow the Vector's array by a fixed amount d. How long does it take to add n items to an empty Vector?

- The array will be copied each time its capacity needs to exceed a multiple of d
  - At sizes 0, d, 2d, …, n/d
- Copying an array of size kd takes ckd steps for some constant c, giving a total of

$$\sum_{k=1}^{n/d} c \cdot k \cdot d = c \cdot d \sum_{k=1}^{n/d} k = c \cdot d \cdot \frac{\left(n/d\right)\left(n/d + 1\right)}{2} = O(n^2)$$

Vector: Add Method Complexity

Suppose we want to grow the Vector's array by doubling. How long does it take to add *n* items to an empty Vector?

- The array will be copied each time it's capacity needs to exceed a power of 2.
  - At sizes $0, 1, 2, 4, 8, \dots, 2^{\lfloor \log_2 n \rfloor}$
- Copying an array of size 2^k takes $c2^k$ steps for some constant c, giving a total of:

$$\sum_{k=1}^{\log_2 n} c \cdot 2^k = c \sum_{k=1}^{\log_2 n} 2^k = c \cdot (2^{1+\log_2 n} - 1) = O(n)$$

Common Complexities

For n = measure of problem size:

- $O(1)$: constant time and space
- $O(\log n)$: divide and conquer algorithms, binary search
- $O(n)$: linear dependence, simple list lookup
- $O(n \log n)$: divide and conquer sorting algorithms
- $O(n^2)$: matrix addition, selection sort
- $O(n^3)$: matrix multiplication
- $O(n^{12})$: Original AKS primality test for n-bit integers
- $O(2^n)$: subset sum, graph 3-coloring, satisfiability, ...
- $O(n!)$: traveling salesman problem (in fact $O(n^2 2^n)$)

Recursion

- General problem solving strategy
  - Break problem into smaller pieces (sub-problems)
  - Sub-problems are typically smaller versions of same problem

Recursion

- Many algorithms are recursive
  - Can be easier to understand, prove correctness, or determine efficiency
- Today we will review recursion and then talk about techniques for reasoning about recursive algorithms

Factorial

- $n! = n \bullet (n-1) \bullet (n-2) \bullet \ldots \bullet 1$
- How can we implement this?
  - We could use a for loop…

```
int product = 1;
for(int i = 1;i <= n; i++)
      product *= i;
```

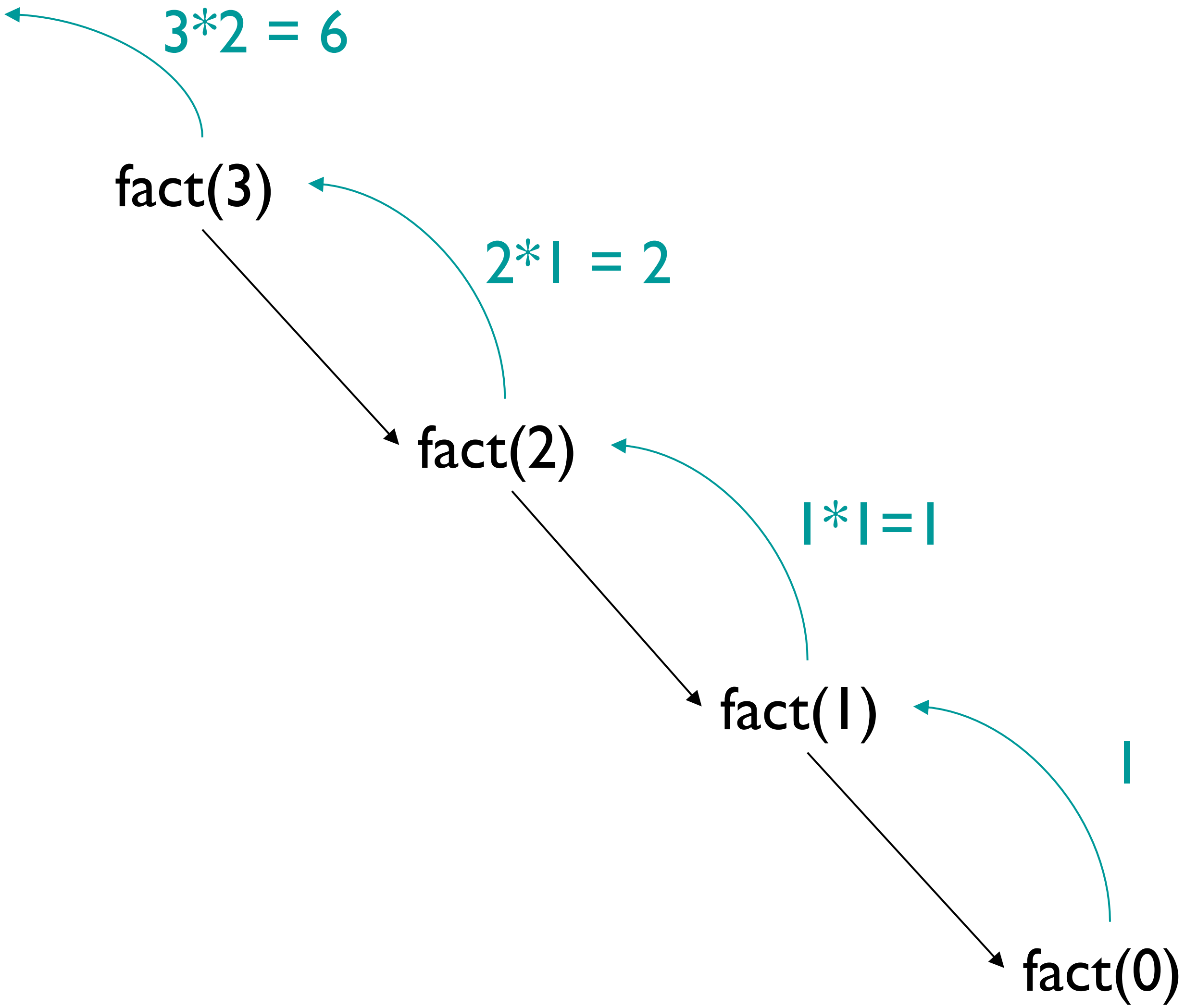
- But we could also write it recursively….

Factorial

- n! = n • (n-1) • (n-2) • … • 1
- Recursive definition (what "…" really means!)
  - n! = n • (n-1)!
  - 0! = 1

```
// Pre: n >= 0
public static int fact(int n) {
      if (n==0) return 1;
      else return n*fact(n-1);
}
```

Factorial



Factorial

- In recursion, we always use the same basic approach
- What's our base case? [Sometimes "cases"]
  - n=0; fact(0) = 1
- What's the recursive relationship?
  - n>0; fact(n) = n • fact(n-1)

Fibonacci Numbers

- 1, 1, 2, 3, 5, 8, 13, ...
- Definition
  - $F_0 = 1, F_1 = 1$
  - For $n > 1$, $F_n = F_{n-1} + F_{n-2}$
- Inherently recursive!
- It appears almost everywhere
  - Growth: Populations, plant features
  - Architecture
  - Data Structures!

fib.java

```
public class fib{
   // pre: n is non-negative
    public static int fib(int n) {
       if (n==0 || n == 1) {
          return 1;
       }
       else {
          return fib(n – 1) + fib(n – 2);
       }
    }

    public static void main(String args[]) {
       System.out.println(fib(Integer.valueOf(args[0]).intValue()));
    }

}
```

Demo: RecursiveMethods.java….

Question: Why is fib so slow?!

Towers of Hanoi

- Demo
- Base case:
  - One disk: Move from start to finish
- Recursive case (n disks):
  - Move smallest n-1 disks from start to temp
  - Move bottom disk from start to finish
  - Move smallest n-1 disks from temp to finish
- Let's try to write it....

Recursion Tradeoffs

- Advantages
  - Often easier to construct recursive solution
  - Code is usually cleaner
  - Some problems do not have obvious non-recursive solutions
- Disadvantages
  - Overhead of recursive calls
  - Can use lots of memory (need to store state for each recursive call until base case is reached)
    - E.g. recursive fibonacci method

Alternate contains() for Vector

```
// Helper method: returns true if elt has index in range from..to
public boolean contains(E elt, int from, int to) {
   if (from > to)
        return false; // Base case: empty range
   else
        return elt.equals(elementData[from]) ||
               contains(elt, from+1, to);
}

public boolean contains(E elt) {
   return contains(elt, 0, size()-1); }
```

- What's the time complexity of contains?
  - O(to – from + 1) = O(n) (n is the portion of the array searched)
  - Why?
    - Bootstrapping argument! True for: to – from = 0, to – from = 1, …
- Let's formalize this bootstrapping idea....

Mathematical Induction

- The mathematical cousin of recursion is induction
- Induction is a proof technique
- Reflects the structure of the natural numbers
- Use to simultaneously prove an infinite number of theorems!

Mathematical Induction

- Example: Prove that for every n ≥ 0

$$P_n : \sum_{i=0}^{n} i = 0 + 1 + \ldots + n = \frac{n(n+1)}{2}$$

- Proof by induction:
  - Base case: P_n is true for n = 0 (just check it!)
  - Induction step: If P_n is true for some n≥0, then P_{n+1} is true.

$$P_{n+1}: 0 + 1 + \ldots + n + (n+1) = \frac{(n+1)\big((n+1)+1\big)}{2} = \frac{(n+1)(n+2)}{2}$$

Check: $0 + 1 + \ldots + n + (n+1) = \frac{n(n+1)}{2} + (n+1) = \frac{(n+1)(n+2)}{2}$

  - First equality holds by assumed truth of P_n!

Mathematical Induction

Principle of Mathematical Induction (Weak)

Let P(0), P(1), P(2), ... Be a sequence of statements, each of which could be either true or false. Suppose that

1. P(0) is true, and
2. For all n **≥ 0, if P(n) is true, then so is** P(n+1).

Then all of the statements are true!

Note: Often Property 2 is stated as

2. For all n > 0, if P(n-1) is true, then so is P(n).

   Apology: I do this a lot, as you'll see on future slides!

Mathematical Induction

- Prove: $$\sum_{i=0}^{n} 2^i = 2^0 + 2^1 + 2^2 + \ldots + 2^n = 2^{n+1} - 1$$

- Prove: $$0^3 + 1^3 + \ldots + n^3 = (0 + 1 + \ldots + n)^2$$

Proof: $0^3 + 1^3 + \ldots + n^3 = (0 + 1 + \ldots + n)^2$

Note: I'm doing the n-1 → n version

$$
\begin{aligned}
(0^3 + 1^3 + \ldots n^3) &= (0^3 + 1^3 + \ldots + (n-1)^3) + n^3 \\
\text{Induction☞} &= (0 + 1 + \ldots + (n-1))^2 + n^3 \\
&= \left(\frac{n(n-1)}{2}\right)^2 + n^3 \\
&= n^2\left(\frac{(n-1)^2 + 4n}{4}\right) \\
&= n^2\left(\frac{n^2 + 2n + 1}{4}\right) \\
&= n^2\left(\frac{(n+1)^2}{4}\right) \\
&= \left(\frac{n(n+1)}{2}\right)^2 \\
&= (0 + 1 + \ldots + n)^2
\end{aligned}
$$

What about Recursion?

- What does induction have to do with recursion?
  - Same form!
    - Base case
    - Inductive case that uses simpler form of problem
- Example: factorial
  - Prove that fact(n) requires n multiplications
    - Base case: n = 0 returns 1, 0 multiplications
    - Assume true for all k<n, so fact(k) requires k multiplications.
    - fact(n) performs one multiplication (n*fact(n-1)). We know that fact(n-1) requires n-1 multiplications. 1+n-1 = n, therefore fact(n) requires n multiplications.

Counting Method Calls

- Example: Fibonacci
  - Prove that fib(n) makes at least fib(n) calls to fib()
    - Base cases: n = 0: 1 call; n = 1; 1 call
    - Assume that for some n**≥2, fib(n-1) makes at least** n-1 calls to fib() and fib(n-2) makes at least fib(n-2) calls to fib().
    - Claim: Then fib(n) makes at least fib(n) calls to fib()
      - 1 initial call: fib(n)
      - By induction: At least fib(n-1) calls for fib(n-1)
      - And as least fib(n-2) calls for fib(n-2)
      - Total: 1 + fib(n-1) + fib(n-2) > fib(n-1) + fib(n-2) = fib(n) calls
    - Note: Need two base cases!
  - One can show by induction that for n > 10: fib(n) > $(1.5)^n$
  - Thus the number of calls grows exponentially!
  - We can visualize this with a *method call graph*….

Mathematical Induction : Version 2

Principle of Mathematical Induction (Weak)

Let P_0, P_1, P_2, ... Be a sequence of statements, each of which could be either true or false. Suppose that

1. P_0 and P_1 are true, and
2. For all n **≥ 2, if P**$_{n-1}$ and P_{n-2} are true, then so is P_n.

Then all of the statements are true!

Other versions:

- Can have k > 2 base cases
- Doesn't need to start at 0

Example: Binary Search

- Given an array a[] of positive integers in increasing order, and an integer x, find location of x in a[].
  - Take "indexOf" approach: return -1 if x is not in a[]

```
protected static int recBinarySearch(int a[], int value,
             int low, int high) {
  if (low > high) return -1;
  else {
      int mid = (low + high) / 2;       //find midpoint
      if (a[mid] == value) return mid;  //first comparison
                                        //second comparison
      else if (a[mid] < value)          //search upper half
      return recBinarySearch(a, value, mid + 1, high);
       else                             //search lower half
            return recBinarySearch(a, value, low, mid - 1);
  }
```

Binary Search takes O(log n) Time

Can we use induction to prove this?

- Claim: If n = high - low + 1, then recBinSearch performs at most c (1+ log n) operations, where c is *twice* the number of statements in recBinSearch
- Base case: n = 1: Then low = high so only c statements execute (method runs twice) and c **≤** c(1+log 1)
- Assume that claim holds for some n **≥ 1, does it** hold for n+1? [Note: n+1 > 1, so low < high]
- Problem: Recursive call is *not* on n---it's on n/2.
- Solution: We need a better version of the PMI….

Mathematical Induction

Principle of Mathematical Induction (Strong)

Let P(0), P(1), P(2), ... Be a sequence of statements, each of which could be either true or false. Suppose that, for some k **≥ 0**

1. P(0), P(1), ... , P(k) are true, and
2. For every n **≥ k, if P(1), P(2), ... , P(n)** are true, then so is P(n+1).

Then all of the statements are true!

Binary Search takes O(log n) Time

Try again now:

- Assume that for some n **≥ 1, the claim** holds *for all* k **≤ n, does claim hold for** n+1?
- Yes! Either
  - x = a[mid], so a constant number of operations are performed, or
  - RecBinSearch is called on a sub-array of size n/2, and by induction, at most c(1 + log (n/2)) operations are performed.
    - This gives a total of at most c + c(1 + log(n/2)) = c + c(log(2) + log(n/2)) = c + c(log n) = c(1 + log n) statements

Wait…what???

- Prove: All horses are the same color.
- Base case: n = 1. Clear
- Induction (n>1): Assume we have a set X of n horses. Let x and y be two of the horses. X – {x} is a set of n-1 horses, so (by induction) they are all the same color. Similarly, all horses in X – {y} are the same color. Now pick z in X, z **≠ x,y. Then z** is in X-{x} and z is in X-{y}, so all all horses are the same color (as z)!
- Question: What went wrong?