Administrative Details

• Lab 3 Today
  • Declare your partner (or independence) by 10am
    – One repository where both people have access
    – Beware of merge conflicts!
  • Questions about warm-up problems?
    – We’ll go over at start of lab, but does anyone feel like they have a good solution?
Last Time

- Measuring Growth
  - Big-O
    - We care about trends
    - Goal: determine how performance scales with input size.
    - Best, worst, and average cases
Today

• Applying $O()$ to Compute Complexity
  • Finish Vector growing examples

• Recursion

• Mathematical Induction
Vector Operations: Worst-Case

Let \( n \) = Vector size (*not* capacity!):

- **O(1) operations (cost is same regardless of size):**
  - `size()`, `capacity()`, `isEmpty()`, `get(i)`, `set(i)`, `firstElement()`, `lastElement()`

- **O(n) operations (cost grows proportionally to size):**
  - `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 copy array: \( O(n) \)
Vectors: 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, \( d \), 2\( d \), \( \ldots \), \( n/d \).
- Copying an array of size \( kd \) takes \( ckd \) steps for some constant \( c \), giving a total of

\[
\sum_{k=1}^{n/d} ckd = cd \sum_{k=1}^{n/d} k = cd \left( \frac{n}{d} \right) \left( \frac{n}{d} + 1 \right) / 2 = O(n^2)
\]
Vectors: Add Method Complexity

Suppose we 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 its capacity needs to exceed a power of 2
  - At sizes 0, 1, 2, 4, 8 ..., \( n/2 \)
- The total number of elements are copied when \( n \) elements are added is:
  - \( 1 + 2 + 4 + \ldots + n/2 = n-1 = O(n) \)
- Very cool! (So cool that we’ll prove it later)
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 scan (e.g., list lookup)
- $O(n \log n)$: divide and conquer sorting algorithms
- $O(n^2)$: matrix addition, selection sort
- $O(n^3)$: matrix multiplication
- $O(n^k)$: cell phone switching algorithms
- $O(2^n)$: 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 sub-problems of same type
  • Solve sub-problems
  • Combine sub-problem solutions into solution for original problem
    • Recursive leap of faith!
Recursion

• Many algorithms are recursive
  • Can be easier to understand (and prove correctness/state efficiency of) than iterative versions
  • They feel elegant

• Today we will review recursion and then talk about techniques for reasoning about recursive algorithms
Think Recursively

- In recursion, we always use the same basic approach.
- What’s our base case? [Sometimes “cases”]
  - n=0? list.isEmpty()?
- What’s the recursive relationship?
  - How can we use the solution to a smaller version of the problem to answer the question?
Factorial

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

- But we could also write it recursively
  - $n! = n \cdot (n-1)!$
  - $0! = 1$
Factorial

1 * 1 = 1
2 * 1 = 2
3 * 2 = 6

fact(3) → fact(2) → fact(1) → fact(0)
public class Fact{

    // Pre: n >= 0
    public static int fact(int n) {
        // base case
        if (n==0) {
            return 1;
        }
        // recursive leap of faith
        else {
            return n*fact(n-1);
        }
    }

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

    // pre: n is non-negative
    public static int fib(int n) {
        // base case
        if (n == 0 || n == 1) {
            return 1;
        }
        // recursive leap of faith
        else {
            return fib(n - 1) + fib(n - 2);
        }
    }

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

- **Advantages**
  - Often easier to construct recursive solution
  - Code is usually cleaner (so *elegant*!)
  - 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) // Base case: empty range
        return false;
    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....