Administrative Details

- Lab 3 Today!
  - You *may* work with a partner
  - Come to lab with a plan!
  - Try to answer questions before lab
Last Time

• Note: Storing null values in Lists
• More on Doubly-Linked List
  • Lab this week: Doubly Linked Lists with dummy nodes
• Abstract Classes and Inheritance
  • Return of the Card Classes!
• The Structure5 Universe to date
Today

• Measuring Growth
  • Big-O
• Introduction to Recursion
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)
      – Not consistent. Need lots of tests to predict future behavior
Measuring Computational Cost

• A better way: 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
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 → (2n) – 1 (≈ twice as long)
    • Bubble sort: n(n-1)/2 → 2n(2n – 1)/2 (≈ 4 times as long)
    • Subset sum: 2^{n-1} → 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

- $2^x$
- $x^2$
- $x \log_2(x)$
- $x$
- $\log_2(x)$
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 $n^2$
  - $a_0n^k + a_1n^{k-1} + a_2n^{k-2} + \ldots a_k$ is roughly $n^k$
- The key is to find the most significant or dominant term
- Ex: $\lim_{x \to \infty} \frac{3x^4 - 10x^3 - 1}{x^4} = 3$ (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
  \[
  |f(n)| \leq c \cdot g(n) \text{ for all } n \geq n_0
  \]
- \( g \) is “at least as big as” \( f \) for large \( n \)
  - Up to a multiplicative 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 smallest upper bound when we estimate running time

• 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 \( n_0 \) 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 $n^k \leq c \cdot 2^n$ 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)
  - Sort already sorted array in $O(n)$
  - Find item in first place that we look $O(1)$
- Worst case (generally useful, sometimes misleading)
  - Don’t find item in list $O(n)$
  - Reverse order sort $O(n^2)$
- 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 \( n = \text{Vector size (not capacity!)} \):

- \( O(1) \): 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: \( O( \log_2(n) ) \)
    - Time to copy array: \( O(n) \)
    - \( O(\log_2(n)) + O(n) \) is \( O(n) \)
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 … $2^{\log_2 n}$
  • Copying an array of size $2^k$ takes $c 2^k$ steps for some constant $c$, giving a total of
    $$\sum_{k=1}^{\log_2 n} c 2^k = c \sum_{k=1}^{\log_2 n} 2^k = c \left(2^{\log_2 n+1} - 1\right) = O(n)$$
• Very cool!