CSCI 136
Data Structures &
Advanced Programming

Measuring Complexity

Measuring Complexity

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 clock rates
      - Too much other stuff happening (network, OS, etc)
      - 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

Measuring Computational Cost

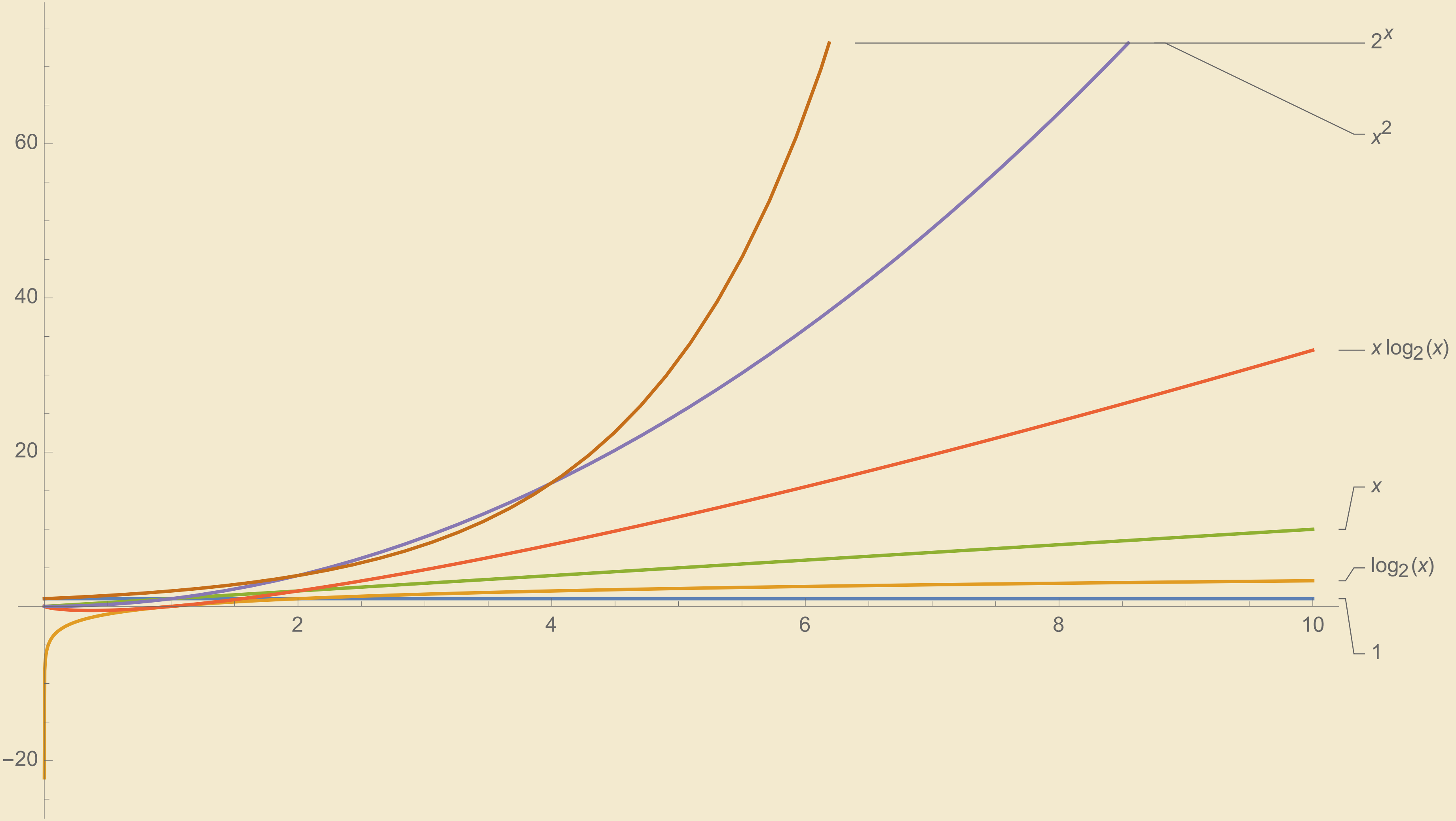
- How do number of operations scale with problem size?
  - E.g.: If I double the size of the problem instance, how much longer will it take to solve:
    - Find maximum: $n - 1 \rightarrow (2n) - 1$ ($\approx$ twice as long)
    - Bubble sort: $n(n-1)/2 \rightarrow 2n(2n - 1)/2$ ($\approx$ 4 times as 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

- 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
  - $a_0n^k + a_1n^{k-1} + a_2n^{k-2} + \cdots a_k$ is roughly n^k
- The key is to understand the relative magnitudes (ratio of magnitudes) of functions
- Ex: $3x^4 -10x^3 -1)/x^4 \cong 3$ (Why?)
  - So $3x^4 -10x^3 -1$ grows "like" x^4

Function Growth

Why does $3x^4$ -10x^3 -1 grows "like" x^4?

$$\frac{3x^4 - 10x^3 - 1}{x^4} = 3 - \frac{10}{x} - \frac{1}{x^4}$$

As $x \to \infty$, note that $3 - \frac{10}{x} - \frac{1}{x^4} \to 3$

Ratio of the functions is $\cong$ constant as x grows

Function Growth

Example: $3x^4 - 10x^3 - 1$ grows much more slowly than x^5

$$\frac{3x^4 - 10x^3 - 1}{x^5} = \frac{3}{x} - \frac{10}{x^2} - \frac{1}{x^5}$$

As $x \to \infty$, note that $\frac{3}{x} - \frac{10}{x^2} - \frac{1}{x^5} \to 0$

Ratio of the functions is $\cong 0$ as x grows

Function Growth

Example: $3x^4$ -$10x^3$ -1 grows much more quickly than x^3

$$\frac{3x^4 - 10x^3 - 1}{x^3} = 3x - 10 - \frac{1}{x^3}$$

As $x \to \infty$, note that $3x - 10 - \frac{1}{x^3} \to 3x - 10 \to \infty$

Ratio of the functions grows large as x grows

Asymptotic Bounds

How can we capture this idea?

What *is* the idea?

- If $\frac{f(x)}{g(x)} \cong 0$ as $x \to \infty$ then $g(x)$ *grows [much] faster than* $f(x)$
- If, for some constant c > 0, $\frac{f(x)}{g(x)} \cong$ c as $x \to \infty$, then $g(x)$ *grows at the same rate as* $f(x)$
- If $\frac{f(x)}{g(x)} \to \infty$ as $x \to \infty$ then $g(x)$ *grows [much] more slowly than* $f(x)$
- Let's make this precise….

Asymptotic Bounds (Big-O)

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

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

- Notes
  - $c \cdot g(n)$ is “at least as big as” f(n) *for large n*
  - Ratios are replaced by inequality
  - Absolute value?
    - Capture idea that -f(n) grows *in magnitude* at the same rate as f(n)

Asymptotic Bounds (Big-O)

- Examples:
  - $f(n) = n^2/2$ is $O(n^2)$
    - Here $g(n) = n^2$
    - $n^2/2 \leq c\ n^2$ for $c = ½$ and all $n \geq 0$ (so $n_0=0$)
  - $f(n) = 1000n^3$ is $O(n^3)$
    - Here $g(n) = n^3$
    - $1000n^3 \leq c\ n^3$ for $c = 1000$ and all $n \geq 0$ (so $n_0=0$)
  - $f(n) = (n+5)/2$ is $O(n)$
    - Here $g(n) = n$
    - $(n+5)/2 \leq c\ n$ for $c = 1$ and all $n \geq 5$ (so $n_0=5$)

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 $\mathbf{c}$ and $\mathbf{n_0}$ in practice, but focus on size of $\mathbf{g}$ when designing algorithms and data structures

Input-dependent Running Times

- Algorithms may have different running times for different inputs of the same size
- Best case (typically not useful)
  - Find item in first place that we look: O(1)
- Worst case (generally useful) ← This is us!
  - Don't find item in list: O(n)
  - Looking for duplicates when there are none: $O(n^2)$
- Average case (useful, but often hard to compute)
  - Linear search O(n)
  - QuickSort random array O(n log n) ← We'll sort soon

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 $\log_2(n^k) \leq \log_2(2^n)$, that is, if $k \log_2(n) \leq n$.
  - That is if $k \leq n/\log_2(n)$.
  - But calculus tells us tha $n/\log_2(n) \rightarrow \infty$ as $n \rightarrow \infty$
  - 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 $2^n \geq n^k$

Presentation Ends Here

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 : $O(\log_2(n))$
    - Time to copy array: O(n)
    - $O(\log_2(n)) + O(n)$ is O(n)

Vector: Add Method Complexity

Suppose we grow the Vector's array by a fixed abount 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*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, …, $2^{\log_2 n}$
- 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 $O(n^2 2^n)$)