CSCI 136:
Data Structures
and
Advanced Programming

Lecture 11

Asymptotic analysis, part 2

Instructor: Dan Barowy

Williams

Outline

Study tip

Big-O notation

Announcements

Feedback: **`Assert.pre`** / **`Assert.post`**

Who here knows where to find docs?

`structure5` documentation on book website

Have you ever been frustrated because you don't even understand what the professor is asking?

Life skill #8

Understanding the problem is half the battle.

Specification problem

Our APIs are lists of methods, along with brief English-language descriptions of what the methods are supposed to do. Ideally, an API would clearly articulate behavior for all possible inputs, including side effects, and then we would have software to check that implementations meet the specification. Unfortunately, a fundamental result from theoretical computer science, known as the *specification problem*, says that this goal is actually *impossible* to achieve. Briefly, such a specification would have to be written in a formal language like a programming language, and the problem of determining whether two programs perform the same computation is known, mathematically, to be *unsolvable*. (If you are interested in this idea, you can learn much more about the nature of unsolvable problems and their role in our understanding of the nature of computation in a course in theoretical computer science.) Therefore, we resort to informal descriptions with examples, such as those in the text surrounding our APIs.

—Sedgwick and Wayne, *Computer Science: An Introductory Approach*

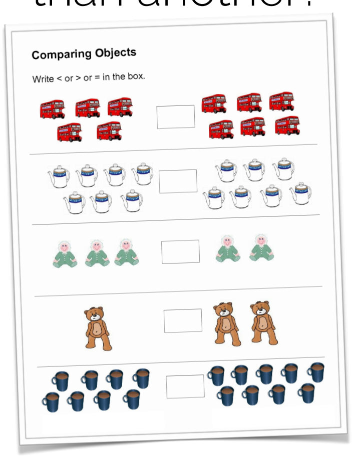
Life skill #8

Understanding the problem is half the battle.

You can work with **anyone** to understand problems!

(but only work **with your partner to solve** them)

How do we decide if one algorithm is better (time/space) than another?



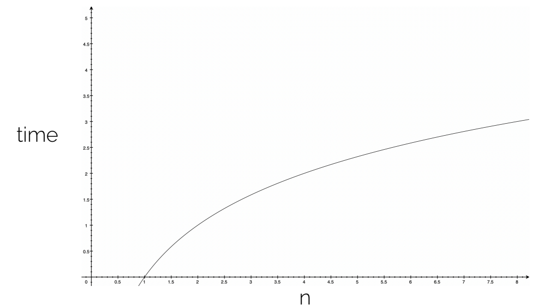
Why can't we just measure "wall time"?

Recall: directly measuing is problematic

- Other things are happening at the same time
- Total running time usually varies by input
- Different computers may produce different results!

Big idea:

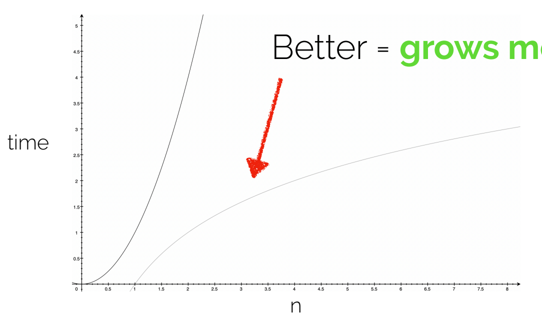
{Time, Space} cost in terms of **n**, where n is the size of the input.



I.e., cost is a function of n.

Big idea:

This form makes comparisons **easy**.



Better = **grows more slowly**

One program is **clearly better** than the other.

Overcounting Example

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

// line 1 cost: c_1
// line 2 cost: nc_2
// line 3 cost: nc_3
// line 4 cost: nc_4
// line 6 cost: c_5

Total cost: $c_1 + nc_2 + nc_3 + nc_4 + c_5$

$= c_1 + n(c_2 + c_3 + c_4) + c_5$

$= n(c_2 + c_3 + c_4) + c_1 + c_5$

$\approx O(n)$

Overcounting gives us an **upper** bound.

Undercounting Example

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

// line 1 cost: c_1
// line 2 cost: nc_2
// line 3 cost: nc_4
// line 4 cost: **zero**
// line 6 cost: c_5

Total cost: $c_1 + nc_2 + nc_4 + 0 + c_5$

$= c_1 + n(c_2 + c_4) + c_5$

$= n(c_2 + c_4) + c_1 + c_5$

$\approx O(n)$

Undercounting gives us a **lower** bound.

What did we learn?

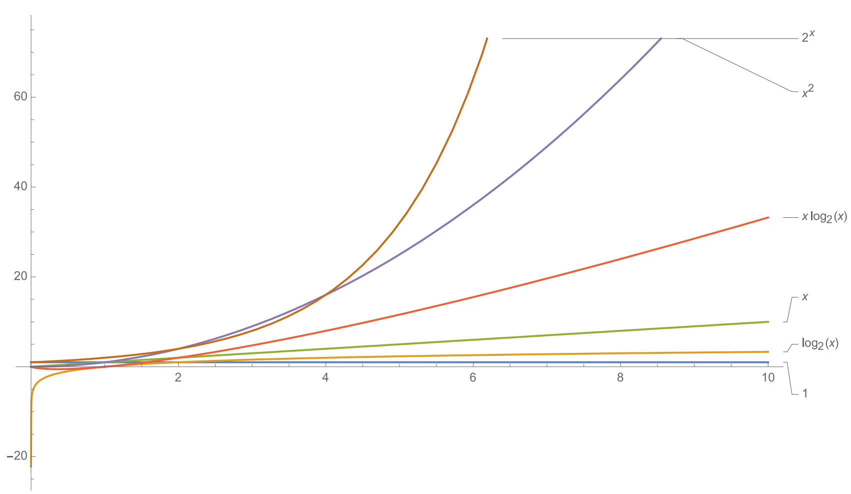
```
// Pre: array length n > 0
public static int findPosOfMax(int[] arr) {
        int maxPos = 0
        for(int i = 1; i < arr.length; i++)
                if (arr[maxPos] < arr[i]) {
                    maxPos = i;
                }
        return maxPos;
}
```

Upper bound: O(**n**)

Lower bound: O(**n**)

Function's run time is "linear", no matter what.

Focus is on **order of magnitude**



Cases

We can do this analysis for the **best**, **average**, and **worst** cases.

When the case is left unstated, we **usually mean** "worst case."

Function growth

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

- $f(x) = 1$
- $g(x) = \log_2(x)$
- $h(x) = x$
- $m(x) = x \log_2(x)$
- $n(x) = x^2$
- $p(x) = x^3$
- $r(x) = 2^x$

// Reminder: if $x=2^n$, $\log_2(x) = n$

i.e., log is the **inverse** of exponentiation

Rule of thumb

- **Ignore additive and multiplicative constants**
- Examples:
  - $n + 1$ is essentially n
  - n and $n/2$ are the same order of magnitude
  - $n^2/1000$, $2n^2$, and $1000n^2$ are "pretty much" just n^2
  - $a_0n^k + a_1n^{k-1} + a_2n^{k-2} + \dots a_k$ is roughly n^k
- The key is to find the **most significant** or **dominant term**

More precisely: **take the limit**

- Suppose we discover cost is $3x^4 - 10x^3 - 1$
- What is the dominant term?
- How do we know?
- Ex: $\lim_{x\to\infty} (3x^4 - 10x^3 - 1)/x^4$
  - $= \lim_{x\to\infty} 3 - 10/x - 1/x^4 = 3$
  - So $3x^4 - 10x^3 - 1$ grows "like" x^4

Big-O notation

Let **f** and **g** be real-valued functions that are defined on the same set of real numbers. Then **f** is of order **g**, written **f(n) is O(g(n))**, if and only if there exists a positive real number **c** and a real number $\mathbf{n_0}$ such that for all **n** in in the common domain of **f** and **g**,

$$|f(n)| \leq c \times |g(n)|, \text{ whenever } n > n_0.$$

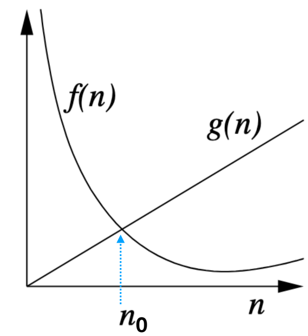
We read this as: "**f(n) is O(g(n))**"
as "**f** of **n** is big-oh of **g** of **n**."

Big-O notation

$|f(n)| \leq c \times |g(n)|$, whenever $n > n_0$.

- $c \times g$ is "at least as big as" **f for large n**
  - for some 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 $O(n)$

f(n) is O(g(n))



Because there is **some point n_0** after which f(n) is **always closer to the horizontal axis** (forever).

Grapher example

$f(n) = n^2/2$ is $O(n^2)$

"Best" upper bounds

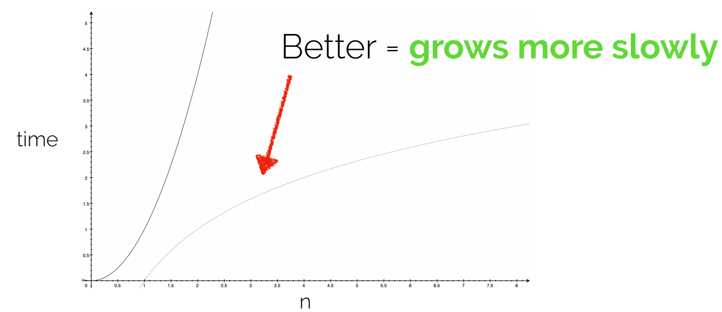
- We typically want the **most conservative** upper bound when we estimate running time
  - And among those, **the simplest**
- Example: Let $f(n) = 3n^2$
  - f(n) is $O(n^2)$
  - f(n) is $O(n^3)$
  - f(n) is $O(2^n)$ (see next slide)
  - **f(n) is NOT O(n) (!!)**
- "Best" upper bound is $O(n^2)$
- We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

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)

Why is this important?

We want an easy comparison between program costs as a function of input size **n**.



One program is clearly better than the other.

Something to think about

Why is the **array doubling** strategy for Vector **better** than expanding the array **one element at a time**?

Recap & Next Class

Today we learned:

Intro to asymptotic analysis

Next class:

Big-O notation