CSCI 136:
Data Structures
and
Advanced Programming

Lecture 8

Asymptotic analysis

Instructor: Dan Barowy

Williams

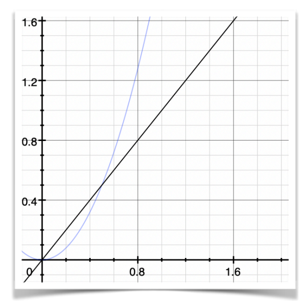
Topics

- Time and space
- Pre/post conditions

Your to-dos

1. Lab 3, **due Tuesday 3/1 by 10pm** ([random] partner lab!)
2. Read **before Fri**: Bailey, Ch 7.1–7.2.

Asymptotic analysis



We measure **time** and **space** similarly.

(I'll focus on **time** today)

How do we know if an algorithm is faster than another?

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

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

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

Let's just count "steps", then

- If we count steps, then…
  - what is a "**step**"?
  - what about steps **inside loops**?

A little context

- How **accurate** do we need to be?
  - If one algorithm takes **64 steps** and another **128 steps**, do we need to know the precise number?

What we do

Instead of precisely counting steps, we usually develop an **approximation** of a program's **time** or **space complexity**.

This approximation **ignores tiny details** and focuses on the big picture: *how do time and space requirements grow as a function of the size of the input?*

Some things that cost "one step"

Accessing an element of an array.

```
arr[5]
```

Assigning a value to a variable.

```
int x = 10;
```

Reading a class field.

```
foo.some_data;
```

Elementary mathematical operations.

```
x + 1
y * z
```

Returning something.

```
return x;
```

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;
}
```

- Can we count steps exactly? Do we even want to?
  - `if` complicates counting
- Idea: **overcount**: assume `if` block always runs
  - in the worst case, it does
- Overcounting gives **upper bound** on run time
- Can also **undercount** for **lower bound**

Overcounting Example

```
// pre: array length n > 0
public static int findPosOfMax(int[] arr) {
      int maxPos = 0 .................................... line 1 cost: c1
      for (int i = 1; i < arr.length; i++) ............. line 2 cost: nc2
            if (arr[maxPos] < arr[i]) ................... line 3 cost: nc3
                  maxPos = i; ........................... line 4 cost: nc4
      return maxPos; ................................... line 5 cost: c5
}
```

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

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

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

$\approx O(n)$

(as you shall see)

Focus is on **order of magnitude**

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

Average case analysis is interesting and extremely useful, but it's beyond the scope of this course.

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 n_0 such that for all **n** in in the common domain of **f** and **g**,

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

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

English, please!

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

Intuition:
"f(n) is **bounded from above** by g(n)."

What we want: some **g(n)** that is both:
- Always bigger than **f(n)** (after some value n_0)
- Close to **f(n)**

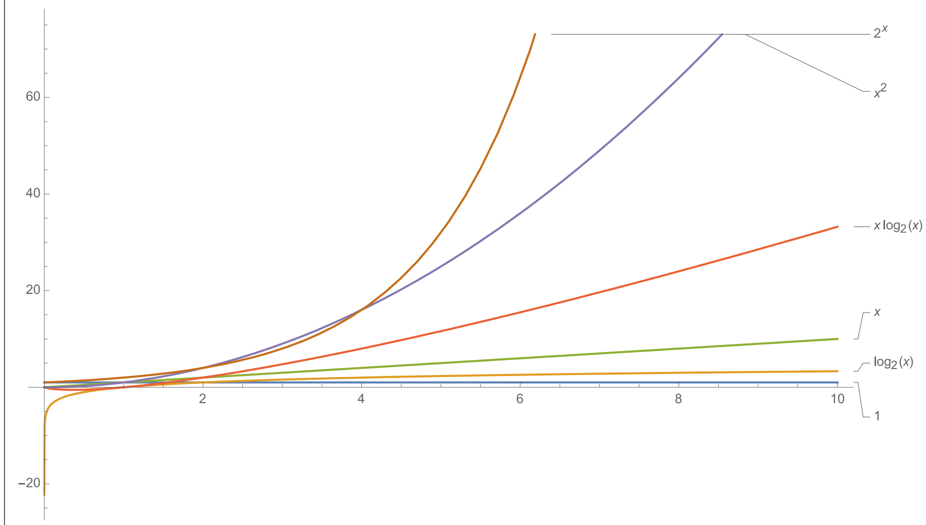
If so, f is **O(g(n))**.

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 & 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
  - $a_0 n^k + a_1 n^{k-1} + a_2 n^{k-2} + \cdots a_k$ is roughly n^k
- The key is to find the **most significant** or **dominant term**
- Ex: $\lim_{x\to\infty} (3x^4 - 10x^3 - 1) = x^4$ (Why?)
  - So $3x^4 - 10x^3 - 1$ grows "like" x^4

Why base of log doesn't matter

- In CS, we generally use **$\log_2$**
- But for asymptotic analysis, **the base does not matter**.
- Proof:

  $\log_2(x) = \log_{10}(x)/\log_{10}(2)$

  $\log_{10}(2) \cdot \log_2(x) = \log_{10}(x)$

  $c \cdot \log_2(x) = \log_{10}(x)$

  $\log_2$ and $\log_{10}$ are **asymptotically the same!**

Think about the following for next class

Example

```
x + 1
```

What does this operation do?
(i.e., what is our desired post-condition?)

Recap & Next Class

Today:

- Time and space analysis
- Pre/post conditions

Next class:

- More pre/post conditions
- Recursion