CSE 373: Data Structures and Algorithms

Lecture 3: Math Review/Asymptotic Analysis

Announcements

- Programming Project #1
  - Getting Help
    - General Questions → Message board
      - Feel free to answer/respond yourselves
      - Please no code/specifics – general ideas only
    - Specific/Implementation Questions → Office Hours (on course website) or email cse373-staff AT cs DOT washington DOT edu (read by myself and the three TAs)
  - No turnin yet
  - Using `sox`
- Want to add CSE 373? See me after class.

Motivation

- So much data!!
  - Human genome: $3.2 * 10^9$ base pairs
    - If there are $6.8 * 10^9$ on the planet, how many base pairs of human DNA?
  - Earth surface area: $1.49 * 10^8$ km^2
    - How many photos if taking a photo of each m^2?
    - For every day of the year ($3.65 * 10^2$)?
- But aren't computers getting faster and faster?

Why algorithm analysis?

- As problem sizes get bigger, analysis is becoming *more* important.
- The difference between good and bad algorithms is getting bigger.
- Being able to analyze algorithms will help us identify good ones without having to program them and test them first.

Measuring Performance: Empirical Approach

- Implement it, run it, time it (averaging trials)
  - Pros?
  - Cons?

Measuring Performance: Empirical Approach

- Implement it, run it, time it (averaging trials)
  - Pros?
    - Find out how the system effects performance
    - Stress testing – how does it perform in dynamic environment
    - No math!
  - Cons?
    - Need to implement code
    - Can be hard to estimate performance
    - When comparing two algorithms, all other factors need to be held constant (e.g., same computer, OS, processor, load)

Measuring Performance: Analytical Approach

- Use a simple model for basic operation costs

- Computational Model
  - has all the basic operations:
    +, -, *, / , =, comparisons
  - fixed sized integers (e.g., 32-bit)
  - infinite memory
  - all basic operations take exactly one time unit (one CPU instruction) to execute

Measuring Performance: Analytical Approach

- Analyze steps of algorithm, estimating amount of work each step takes
  - Pros?
    - Independent of system-specific configuration
    - Good for estimating
    - Don't need to implement code
  - Cons?
    - Won't give you info exact runtimes optimizations made by the architecture (i.e. cache)
    - Only gives useful information for large problem sizes
    - In real life, not all operations take exactly the same time and have memory limitations

Analyzing Performance

- General “rules” to help measure how long it takes to do things:

Basic operations	Constant time
Consecutive statements	Sum of timesx
Conditionals	Test, plus larger branch cost
Loops	Sum of iterations
Function calls	Cost of function body
Recursive functions	Solve recurrence relation…

Efficiency examples

```
statement1;
statement2;
statement3;
```

?

```
for (int i = 1; i <= N; i++) {
    statement4;
}
```

?

```
for (int i = 1; i <= N; i++) {
    statement5;
    statement6;
    statement7;
}
```

?

?

Efficiency examples

```
statement1;
statement2;
statement3;
```

3

```
for (int i = 1; i <= N; i++) {
     statement4;
}
```

N

```
for (int i = 1; i <= N; i++) {
     statement5;
     statement6;
     statement7;
}
```

3N

4N + 3

Efficiency examples 2

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {
        statement1;
    }
}

for (int i = 1; i <= N; i++) {
    statement2;
    statement3;
    statement4;
    statement5;
}
```

?

?

?

Efficiency examples 2

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {
        statement1;
    }
}

for (int i = 1; i <= N; i++) {
    statement2;
    statement3;
    statement4;
    statement5;
}
```

N^2

$4N$

$N^2 + 4N$

- How many statements will execute if N = 10? If N = 1000?

Relative rates of growth

- most algorithms' runtime can be expressed as a *function* of the input size *N*

- **rate of growth**: measure of how quickly the graph of a function rises

- goal: distinguish between fast- and slow-growing functions
  - we only care about very large input sizes
    (for small sizes, most any algorithm is fast enough)
  - this helps us discover which algorithms will run more quickly or slowly, for large input sizes

- most of the time interested in worst case performance; sometimes look at best or average performance

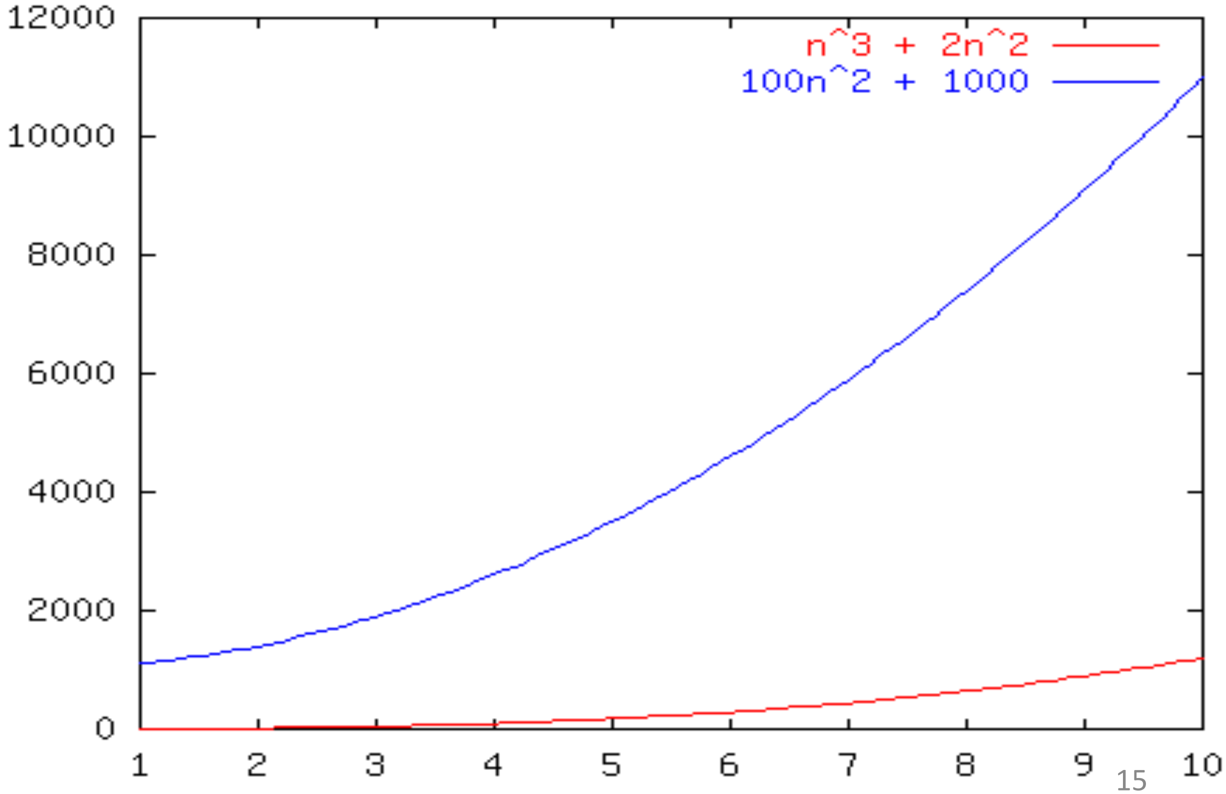
Growth rate example

Consider these graphs of functions.
Perhaps each one represents an algorithm:

$n^3 + 2n^2$

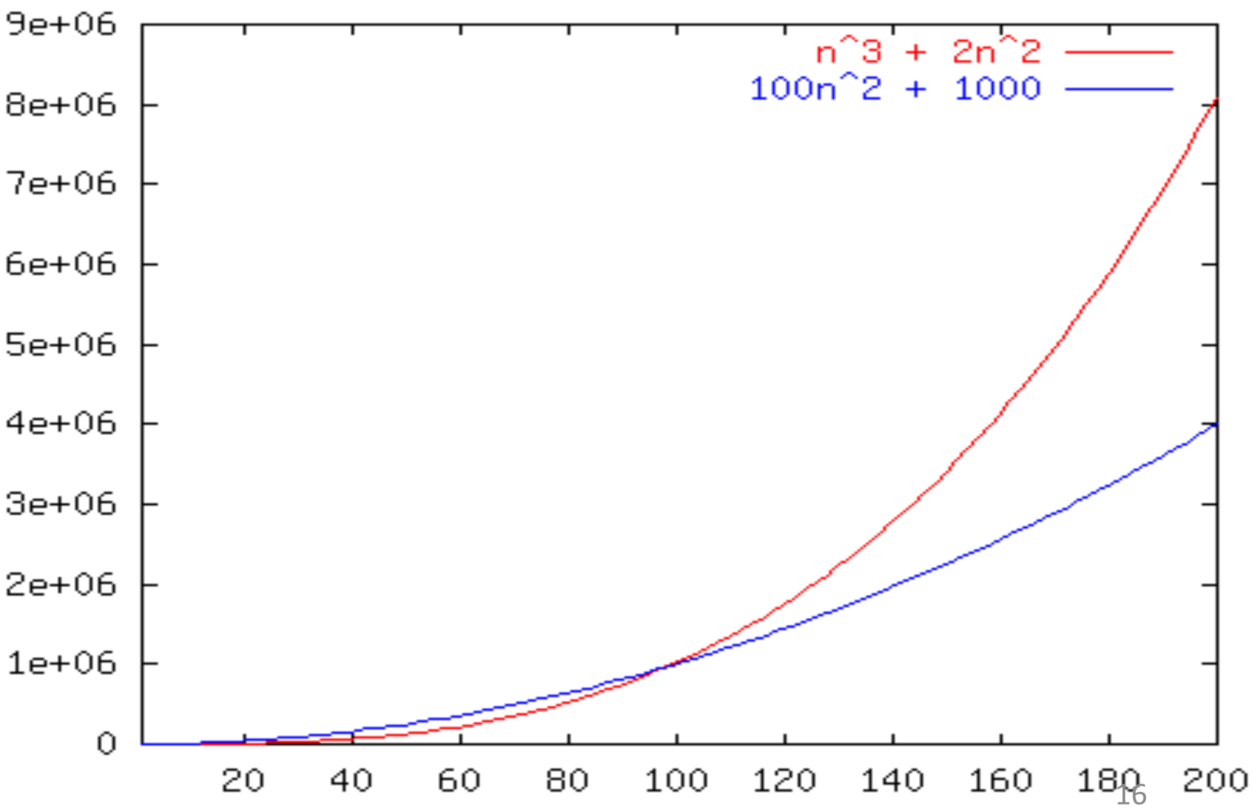
$100n^2 + 1000$

- Which grows faster?



Growth rate example

- How about now?



Big-Oh notation

- Defn:
  $T(N) = O(f(N))$
  if there exist positive constants c , n_0 such that:
  $T(N) \leq c \cdot f(N)$ for all $N \geq n_0$

- idea: We are concerned with how the function grows when N is large. We are not picky about constant factors: coarse distinctions among functions

- Lingo: "T(N) grows no faster than f(N)."

Big-Oh example problems

- $n = O(2n)$?
- $2n = O(n)$?
- $n = O(n^2)$?
- $n^2 = O(n)$?
- $n = O(1)$?
- $100 = O(n)$?
- $214n + 34 = O(2n^2 + 8n)$?

Preferred big-Oh usage

- pick tightest bound. If $f(N) = 5N$, then:

  $f(N) = O(N^5)$

  $f(N) = O(N^3)$

  $f(N) = O(N \log N)$

  $f(N) = O(N)$ ← preferred

- ignore constant factors and low order terms

  $T(N) = O(N)$, *not* $T(N) = O(5N)$

  $T(N) = O(N^3)$, *not* $T(N) = O(N^3 + N^2 + N \log N)$

  – Wrong: $f(N) \leq O(g(N))$

  – Wrong: $f(N) \geq O(g(N))$

Show $f(n) = O(n)$

Claim: $n^2 + 100n = O(n^2)$

Proof: Must find c, n_0 such that for all $n > n_0$,

$n^2 + 100n <= cn^2$

Efficiency examples 3

```
sum = 0;
for (int i = 1; i <= N * N; i++) {
    for (int j = 1; j <= N * N * N; j++) {
        sum++;
    }
}
```

? ? ?

Efficiency examples 3

```
sum = 0;
for (int i = 1; i <= N * N; i++) {
    for (int j = 1; j <= N * N * N; j++) {
        sum++;
    }
}
```

N^3

N^2

$N^5 + 1$

- So what is the Big-Oh?

Math background: Exponents

- Exponents
  - X^Y , or "X to the Y^{th} power";
    X multiplied by itself Y times

- Some useful identities
  - $X^A X^B = X^{A+B}$
  - $X^A / X^B = X^{A-B}$
  - $(X^A)^B = X^{AB}$
  - $X^N+X^N = 2X^N$
  - $2^N+2^N = 2^{N+1}$

Efficiency examples 4

```
sum = 0;
for (int i = 1; i <= N; i += c) {
   sum++;
}
```

?

?

Efficiency examples 4

```
sum = 0;
for (int i = 1; i <= N; i += c) {
   sum++;
}
```

N/c

N/c + 1

- What is the Big-Oh?
  - Intuition: Adding to the loop counter means that the loop runtime grows linearly when compared to its maximum value *n*.

Efficiency examples 5

```
sum = 0;
for (int i = 1; i <= N; i *= c) {
   sum++;
}
```

?

?

- Intuition: Multiplying the loop counter means that the maximum value *n* must grow exponentially to linearly increase the loop runtime

Efficiency examples 5

```
sum = 0;
for (int i = 1; i <= N; i *= c) {
   sum++;
}
```

$\log_c N$

$\log_c N + 1$

- What is the Big-Oh?

Math background: Logarithms

- Logarithms
  - *definition*: $X^A = B$ if and only if $\log_X B = A$
  - *intuition*: $\log_X B$ means:
    "the power X must be raised to, to get B"

  - In this course, a logarithm with no base implies base 2.
    $\log B$ means $\log_2 B$

- Examples
  - $\log_2 16 = 4$ (because $2^4 = 16$)
  - $\log_{10} 1000 = 3$ (because $10^3 = 1000$)

Logarithm identities

Identities for logs with addition, multiplication, powers:

- $\log (AB) = \log A + \log B$
- $\log (A/B) = \log A - \log B$
- $\log (A^B) = B \log A$

Identity for converting bases of a logarithm:

- $\log_A B = \dfrac{\log_C B}{\log_C A} \quad A, B, C > 0, A \neq 1$
  - example:
    $\log_4 32 = (\log_2 32) / (\log_2 4)$
    $= 5 / 2$

Logarithm problem solving

- When presented with an expression of the form:
  - $\log_a X = Y$

  and trying to solve for X, raise both sides to the a power.
  - $X = a^Y$

- When presented with an expression of the form:
  - $\log_a X = \log_b Y$

  and trying to solve for X, find a common base between the logarithms using the identity on the last slide.
  - $\log_a X = \log_a Y / \log_a b$

Logarithm practice problems

- Determine the value of *x* in the following equation.
  - $\log_7 x + \log_7 13 = 3$

- Determine the value of *x* in the following equation.
  - $\log_8 4 - \log_8 x = \log_8 5 + \log_{16} 6$