CSE 326 Lecture 2: Mathematical Background

- Today, we will review:
  - Logs and exponents
  - Series
  - Recursion
  - Big-Oh notation for analysis of algorithms
- Covered in Chapters 1 and 2 of the text

Logs and exponents

- We will be dealing mostly with binary numbers (base 2)
- **Definition**: $\log_X B = A$ means $X^A = B$
- Any base is equivalent to base 2 within a constant factor:

$$\log_X B = \frac{\log_2 B}{\log_2 X}$$

- Why?

Logs and exponents

- We will be dealing mostly with binary numbers (base 2)
- Definition: $\log_X B = A$ means $X^A = B$
- Any base is equivalent to base 2 within a constant factor:

$$\log_X B = \frac{\log_2 B}{\log_2 X}$$

- Why?
- Because: Let $R = \log_2 B$, $S = \log_2 X$, and $T = \log_X B$,

  ➪ $2^R = B$, $2^S = X$, and $X^T = B$

  Then, $2^R = B = X^T = 2^{ST}$ i.e. $R = ST$ and therefore, $T = R/S$.

Properties of logs

- We will assume logs to base 2 unless specified otherwise
- $\log AB = ?$
- $\log A/B = ?$
- $\log A^B = ?$

Properties of logs

- We will assume logs to base 2 unless specified otherwise
- log AB = log A + log B (note: log AB ≠ log A•log B)
- log A/B = log A – log B (note: log A/B ≠ log A / log B)
- log A^B = B log A (note: log A^B ≠ $(\log A)^B$ = $\log^B A$)

More on logs

- log log X < log X < X for all X > 1
  - log log X = Y means $2^{2^Y} = X$
  - log X grows slower than X; called a "sub-linear" function
- log 1 = 0, log 2 = 1, log 1024 = 10

Arithmetic Series

- $S(N) = 1 + 2 + \ldots + N = \sum_{i=1}^{N} i = ?$
- Note: S(1) = 1, S(2) = 3, S(3) = 6, S(4) = 10, …
  - Is there a pattern?

Arithmetic Series

- $S(N) = 1 + 2 + \ldots + N = \sum_{i=1}^{N} i = ?$
- Is S(N) = N(N+1)/2 ?
  - Prove by induction (base case: N = 1, S(N) = 1(2)/2 = 1)
  - Assume true for N = k: S(k) = k(k+1)/2
  - Suppose N = k+1.
  - S(k+1) = 1 + 2 + …+ k + (k+1) = S(k) + (k+1)
    = k(k+1)/2 + (k+1) = (k+1)(k/2 + 1) =
    (k+1)(k+2)/2. ✓
- $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$

Arithmetic Series

- $S(N) = 1 + 2 + \ldots + N = \sum_{i=1}^{N} i = ?$

- $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$ Why is this formula useful?

Yes, why indeed? (yawn)

A Sneak Preview of Algorithm Analysis

- Consider the following program segment:

```
for (i = 1; i <= N; i++)
  for (j = 1; j <= i; j++)
    <print "Hey, wassup?">   // pseudocode for Java/C++ print
```

- How many times is the "print" statement executed?
  - Or, How many wassup's will you see?

A Sneak Preview of Algorithm Analysis

- The program segment being analyzed:

```
for (i = 1; i <= N; i++)
   for (j = 1; j <= i; j++)
      <print "Hey, wassup?">
```

- Inner loop executes "print" i times in the i^{th} iteration
- There are N iterations in the outer loop (i goes from 1 to N)
- Total number of times "print" is executed = $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$

A Sneak Preview of Algorithm Analysis

- The program segment being analyzed:

```
for (i = 1; i <= N; i++)
   for (j = 1; j <= i; j++)
      <print "Hey, wassup?">
```

- Total number of times "print" is executed = $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$
- Running time of the program is proportional to N(N+1)/2 for all N.

Congrats - You just analyzed your first program!

Other Important Series (know them well!)

- Sum of squares: $\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3}$ for large N
- Sum of exponents: $\sum_{i=1}^{N} i^k \approx \frac{N^{k+1}}{|k+1|}$ for large N and k ≠ -1
- Harmonic series (k = -1): $\sum_{i=1}^{N} \frac{1}{i} \approx \log_e N$ for large N
  - $\log_e N$ (or ln N) is the natural log of N
- Geometric series: $\sum_{i=0}^{N} A^i = \frac{A^{N+1}-1}{A-1}$

Recursion

- A function that calls itself is said to be recursive
  - E.g. Recursive procedure "sum" in the first lecture
- Recursion may be a natural way to program certain functions that involve repetitive calculations (as compared to iteration by "for" or "while" loops)
- Classic example: Fibonacci numbers F_n

1, 1, 2, 3, 5, 8, 13, 21, 34, …

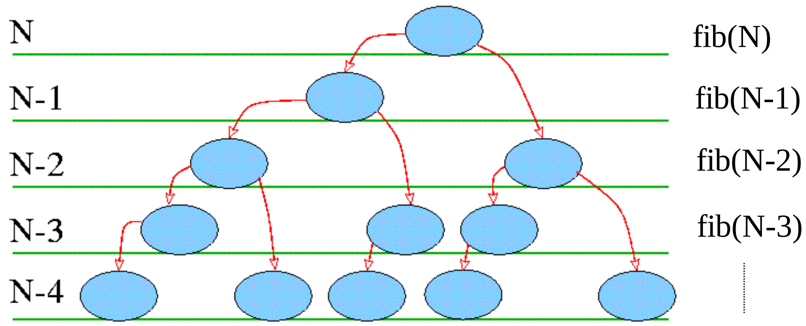
  - First two are: $F_0 = F_1 = 1$
  - Rest are sum of preceding two
  - $F_n = F_{n-1} + F_{n-2}$ (n > 1)

Leonardo Pisano Fibonacci (1170-1250)

Recursive Procedure for Fibonacci Numbers

- ```
  public static int fib(int i) {
   if (i < 0) return 0;  //invalid input
   if (i == 0 || i == 1) return 1;  //base cases
   else return fib(i-1)+fib(i-2);
   }
  ```
- Easy to write: looks like the definition of F_n
- But, can you spot a big problem?

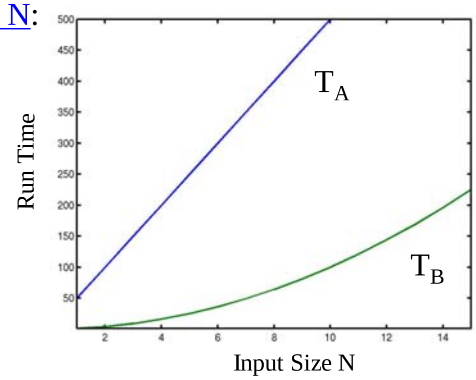
Recursive Calls of Fibonacci Procedure



- Wastes precious time by re-computing fib(N-i) multiple times, for i = 2, 3, 4, etc.!
  ```

Iterative Procedure for Fibonacci Numbers

- ```
  public static int fib_iter(int i) {
        int fib0 = 1, fib1 = 1, fibj = 1;
        if (i < 0) return 0;  //invalid input
        for (int j = 2; j <= i; j++) { //calculate all fib nos. up to i
              fibj = fib0 + fib1;
              fib0 = fib1;
              fib1 = fibj;
        }
        return fibj;
  }
  ```
- More variables and more bookkeeping but avoids repetitive calculations and saves time.
  - How much time is saved over the recursive procedure?
  - Answer in next class…

Recursion Summary

- Recursion may simplify programming, but beware of generating large numbers of calls
  - Function calls can be expensive in terms of time and space
  - There is a hidden space cost associated with the system's stack
- Be sure to get the base case(s) correct!
- Each step must get you closer to the base case
- You may use induction to prove your program is correct
  - See example in previous lecture
  ```

Motivation for Big-Oh Notation

- Suppose you are given two algorithms A and B for solving a problem
- Here is the *running time* $T_A(N)$ and $T_B(N)$ of A and B as a function of input size N:

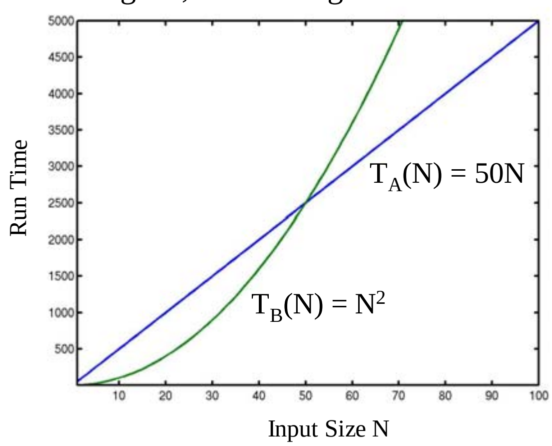


Which algorithm

would you choose?

Motivation for Big-Oh Notation (cont.)

- For large N, the running time of A and B is:



Now which

algorithm would

you choose?

Motivation for Big-Oh: Asymptotic Behavior

- In general, what really matters is the "asymptotic" performance as $N \to \infty$, regardless of what happens for small input sizes N.
- Performance for small input sizes may matter in practice, if you are sure that small N will be common
  - This is usually not the case for most applications
- Given functions $T_1(N)$ and $T_2(N)$ that define the running times of two algorithms, we need a way to decide which one is better (i.e. asymptotically smaller)
  - Big-Oh notation

Big-Oh Notation

- $T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq cf(N)$ for $N \geq n_0$.
- We say that T(N) is "big-oh" of f(N) (or, order of f(N))
- Example 1: Suppose $T(N) = 50N$. Then, $T(N) = O(N)$
  - Why?

Big-Oh Example 2

- $T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq cf(N)$ for $N \geq n_0$.
- We say that T(N) is "big-oh" of f(N) (or, order of f(N))
- Example 1: Suppose $T(N) = 50N$. Then, $T(N) = O(N)$
  - Choose $c = 50$ and $n_0 = 1$ (many other choices work too!)
- Example 2: Suppose $T(N) = 50N+11$. Then, $T(N) = O(N)$
  - Why?

Big-Oh Example 3

- $T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq cf(N)$ for $N \geq n_0$.
- Example 2: Suppose $T(N) = 50N+11$. Then, $T(N) = O(N)$
  - Why?
  - $T(N) = 50N+11 \leq 50N+11N = 61N$ for $N \geq 1$.
  - So, $c = 61$ and $n_0 = 1$ works
- Example 3: $T_A(N) = N+1$, $T_B(N) = N^2$.

  Show that $T_A(N) = O(T_B(N))$: what works for c and n_0?

Big-Oh Example 3

- $T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \le cf(N)$ for $N \ge n_0$.

- Example 3: $T_A(N) = N+1$, $T_B(N) = N^2$.

  $T_A(N) = O(T_B(N))$: choose $c = 1$ and $n_0 = 2$ or

  choose $c = 2$ and $n_0 = 1$ or

  choose $c = 326$ and $n_0 = 322$ etc.

  but not: $c = 0.5$ and $n_0 = 2$ or

  $c = 1$ and $n_0 = 1$

Big-Oh Example 4

- $T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \le cf(N)$ for $N \ge n_0$.

- Example 4: $T(N) = \dfrac{N(N+1)}{2}$

  Is $T(N) = O(N)$? $O(N^2)$? $O(N^3)$?

Big-Oh Example 4

- T(N) = O(f(N)) if there are positive constants c and n_0 such that $T(N) \le cf(N)$ for $N \ge n_0$.

- Example 4: $T(N) = \dfrac{N(N+1)}{2}$

T(N) = O(N^2)

$$T(N) = \frac{N(N+1)}{2} = \frac{N^2}{2} + \frac{N}{2} \le N^2 + N \le 2N^2 \text{ for } N \ge 0$$

(so, choose c = 2 and n_0 = 1)

(Note: T(N) is also O(N^3)! Why?)

Example of Application to Run Time Analysis

- Recall: Our dumb printing program segment:

```
for (i = 1; i <= N; i++)
   for (j = 1; j <= i; j++)
      <print "Hey, wassup?">
```

- Running time is proportional to number of times print statement is executed =

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2} = O(N^2)$$

- Runs in “Quadratic time”

Common functions we will encounter…

Increasing cost ↓

Name	Big-Oh
Constant	O(1)
Log log	O(log log N)
Logarithmic	O(log N)
Log squared	$O((\log N)^2)$
Linear	O(N)
N log N	O(N log N)
Quadratic	$O(N^2)$
Cubic	$O(N^3)$
Exponential	$O(2^N)$

} Polynomial time (Linear through Cubic)

Next Lecture: Using Big-Oh for Algorithm Analysis

To do:

Finish reading Chapters 1 and 2

Start (and Finish!) Homework #1