Fundamentals

CSE 373 - Data Structures

April 8, 2002

Readings and References

- Reading
  - › Chapters 1-2, *Data Structures and Algorithm Analysis in C*, Weiss
- Other References

Mathematical Background

- Today, we will review:
  - › Logs and exponents
  - › Series
  - › Recursion
  - › Motivation for Algorithm Analysis

Powers of 2

- Many of the numbers we use will be powers of 2
- Binary numbers (base 2) are easily represented in digital computers
  - › each "bit" is a 0 or a 1
  - › $2^0=1$, $2^1=2$, $2^2=4$, $2^3=8$, $2^4=16$, $2^8=256$, …
  - › an n-bit wide field can hold 2^n positive integers:
    - $0 \le k \le 2^n-1$

Unsigned binary numbers

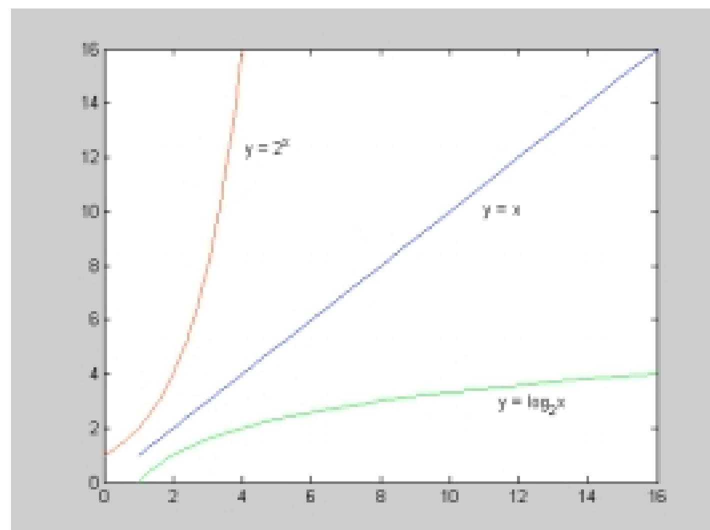
- Each bit position represents a power of 2
- For unsigned numbers in a fixed width field
  - › the minimum value is 0
  - › the maximum value is 2^n-1, where n is the number of bits in the field
- Fixed field widths determine many limits
  - › 5 bits = 32 possible values ($2^5 = 32$)
  - › 10 bits = 1024 possible values ($2^{10} = 1024$)

Binary, Hex, and Decimal

2^8=256	2^7=128	2^6=64	2^5=32	2^4=16	2^3=8	2^2=4	2^1=2	2^0=1	Hex_{16}	$Decimal_{10}$
							1	1	0x3	3
					1	0	0	1	0x9	9
					1	0	1	0	0xA	10
					1	1	1	1	0xF	15
				1	0	0	0	0	0x10	16
				1	1	1	1	1	0x1F	31
		1	1	1	1	1	1	1	0x7F	127
	1	1	1	1	1	1	1	1	0xFF	255

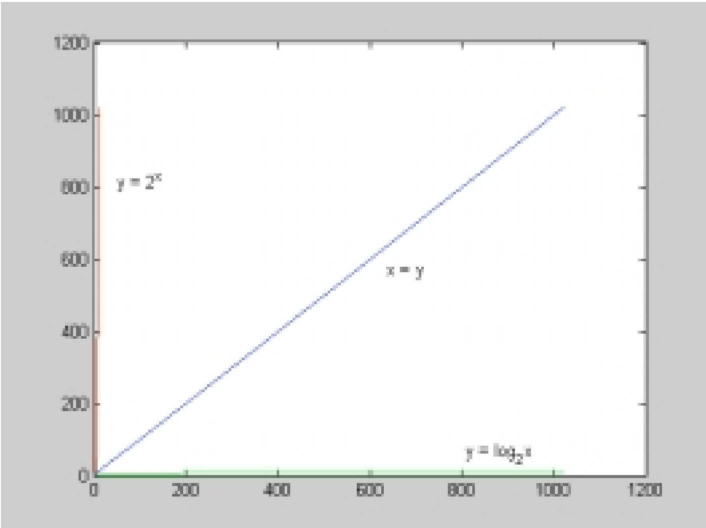
Logs and exponents

- Definition: $\log_2 x = y$ means $x = 2^y$
  - › the log of x, base 2, is the value y that gives $x = 2^y$
  - › $8 = 2^3$, so $\log_2 8 = 3$
  - › $65536 = 2^{16}$, so $\log_2 65536 = 16$
- Notice that $\log_2 x$ tells you how many bits are needed to hold x values
  - › 8 bits holds 256 numbers: 0 to 2^8-1 = 0 to 255
  - › $\log_2 256 = 8$




```
x = 0:.1:4
y = 2.^x
plot(x,y,'r')
hold on
plot(y,x,'g')
plot(y,y,'b')
```

2^x and $\log_2 x$

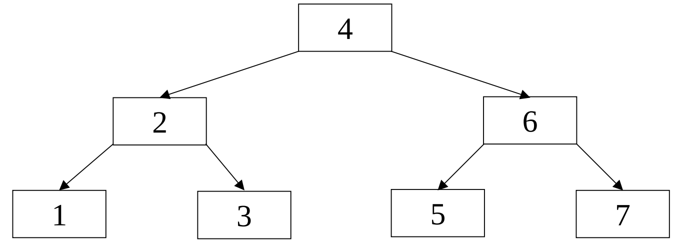



```
x = 0:10
y = 2.^x
plot(x,y,'r')
hold on
plot(y,x,'g')
plot(y,y,'b')
```

2^x and $\log_2 x$

Example: $\log_2 x$ and tree depth

- 7 items in a binary tree, $3 = \lfloor \log_2 7 \rfloor + 1$ levels



Properties of logs (of the mathematical kind)

- We will assume logs to base 2 unless specified otherwise
- log AB = log A + log B
  - › $A=2^{\log_2 A}$ and $B=2^{\log_2 B}$
  - › $AB = 2^{\log_2 A} \bullet 2^{\log_2 B} = 2^{\log_2 A + \log_2 B}$
  - › so $\log_2 AB = \log_2 A + \log_2 B$
  - › note: $\log AB \neq \log A \bullet \log B$

Other log properties

- log A/B = log A – log B
- $\log (A^B) = B \log A$
- log log X < log X < X for all X > 0
  - › log log X = Y means $2^{2^Y} = X$
  - › log X grows slower than X
    - called a “sub-linear” function

A log is a log is a log

- Any base x log is equivalent to base 2 log within a constant factor

$$B = 2^{\log_2 B}$$
$$x = 2^{\log_2 x}$$

$$\log_x B = \log_x B$$
$$x^{\log_x B} = B$$
$$(2^{\log_2 x})^{\log_x B} = 2^{\log_2 B}$$
$$2^{\log_2 x \log_x B} = 2^{\log_2 B}$$
$$\log_2 x \log_x B = \log_2 B$$
$$\log_x B = \frac{\log_2 B}{\log_2 x}$$

Arithmetic Series

- $S(N) = 1 + 2 + \ldots + N = \sum_{i=1}^{N} i$
- The sum is
  - › S(1) = 1
  - › S(2) = 1+2 = 3
  - › S(3) = 1+2+3 = 6
- $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$

Why is this formula useful?

Quicky Algorithm Analysis

- Consider the following program segment:

```
for (i = 1; i <= N; i++)
  for (j = 1; j <= i; j++)
   printf("Hello\n");
```

- How many times is "printf" executed?
  - › Or, How many Hello's will you see?

What is actually being executed?

- The program segment being analyzed:

```
for (i = 1; i <= N; i++)
  for (j = 1; j <= i; j++)
   printf("Hello\n");
```

- Inner loop executes "printf" i times in the i^{th} iteration
  - › j goes from 1 to i
- There are N iterations in the outer loop
  - › i goes from 1 to N

Lots of hellos

- Total number of times "printf" is executed =
  $$1+2+3+\ldots=\sum_{i=1}^{N} i=\frac{N(N+1)}{2}$$
- Congratulations - You've just analyzed your first program!
  - › Running time of the program is proportional to N(N+1)/2 for all N
  - › Proportional to N^2

Recursion

- Classic (bad) example: Fibonacci numbers F_n

1, 1, 2, 3, 5, 8, 13, 21, 34, …

Leonardo Pisano Fibonacci (1170-1250)

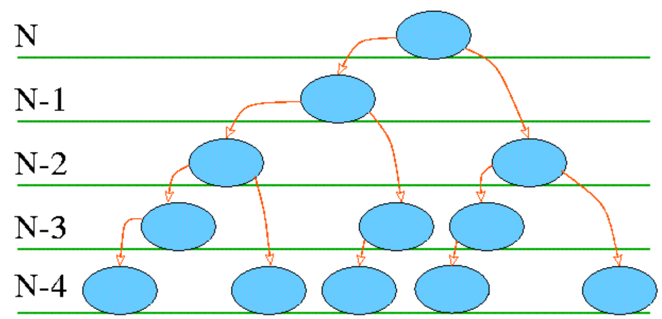
- › First two are defined to be 1
- › Rest are sum of preceding two
- › $F_n = F_{n-1} + F_{n-2}$ (n > 1)

Recursive Procedure for Fibonacci Numbers

```
int fib(int i) {
  if (i < 0) return 0;
  if (i == 0 || i == 1)
   return 1;
  else
   return fib(i-1)+fib(i-2);
  }
```

- Easy to write: looks like the definition of F_n
- But, can you spot the big problem?

Recursive Calls of Fibonacci Procedure



- Re-computes fib(N-i) multiple times!

Iterative Procedure for Fibonacci Numbers

```
int fib_iter(int i) {
   int fib0 = 1, fib1 = 1, fibj = 1;
   if (i < 0) return 0;
   for (int j = 2; j <= i; j++) {
         fibj = fib0 + fib1;
         fib0 = fib1;
         fib1 = fibj;
   }
   return fibj;
  }
```

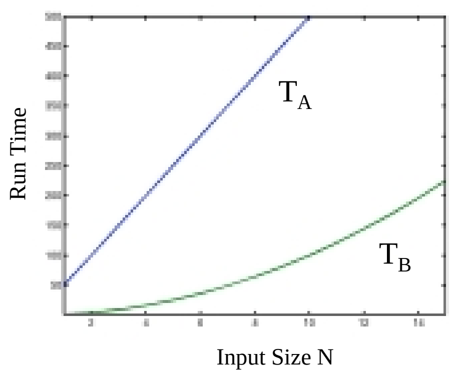
- More variables and more bookkeeping but avoids repetitive calculations and saves time.

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
- Be sure to get the base case(s) correct!
- Each step must get you closer to the base case

Motivation for Algorithm Analysis

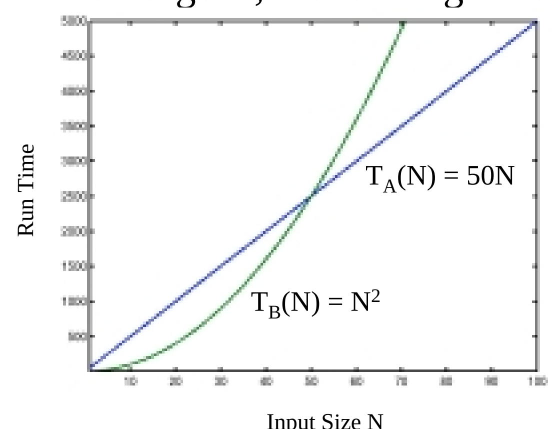
- Suppose you are given two algos A and B for solving a problem
- The running times $T_A(N)$ and $T_B(N)$ of A and B as a function of input size N are given



Which is better?

More Motivation

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



Now which algorithm would you choose?

Asymptotic Behavior

- The "asymptotic" performance as $N \to \infty$, regardless of what happens for small input sizes N, is generally most important
- Performance for small input sizes may matter in practice, if you are <u>sure</u> that <u>small</u> N will be common <u>forever</u>
- We will compare algorithms based on how they scale for large values of N