Fundamentals

CSE 373
Data Structures
Lecture 5
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:
    • $0 \leq k \leq 2^n - 1$
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 and Decimal

<table>
<thead>
<tr>
<th>$2^8 = 256$</th>
<th>$2^7 = 128$</th>
<th>$2^6 = 64$</th>
<th>$2^5 = 32$</th>
<th>$2^4 = 16$</th>
<th>$2^3 = 8$</th>
<th>$2^2 = 4$</th>
<th>$2^1 = 2$</th>
<th>$2^0 = 1$</th>
<th>Decimal$_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>127</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>255</td>
</tr>
</tbody>
</table>
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
  › \( \log_2 256 = 8 \)
$2^x$ and $\log_2 x$
\[ y = 2^x \]

plot(x, y, 'r')
hold on
plot(y, x, 'g')
plot(y, y, 'b')

\[ x = 0:10 \]

\[ y = 2^x \]

plot(x, y, 'r')
hold on
plot(y, x, 'g')
plot(y, y, 'b')

\[ 2^x \text{ and } \log_2{x} \]
Floor and Ceiling

\[ \lfloor X \rfloor \quad \text{Floor function: the largest integer } \leq X \]

\[
\begin{align*}
\lfloor 2.7 \rfloor &= 2 \\
\lfloor -2.7 \rfloor &= -3 \\
\lfloor 2 \rfloor &= 2 \\
\end{align*}
\]

\[ \lceil X \rceil \quad \text{Ceiling function: the smallest integer } \geq X \]

\[
\begin{align*}
\lceil 2.3 \rceil &= 3 \\
\lceil -2.3 \rceil &= -2 \\
\lceil 2 \rceil &= 2 \\
\end{align*}
\]
Facts about Floor and Ceiling

1. $X - 1 < \lfloor X \rfloor \leq X$
2. $X \leq \lceil X \rceil < X + 1$
3. $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$ if $n$ is an integer
Example: $\log_2 x$ and tree depth

- 7 items in a binary tree, $3 = \left\lfloor \log_2 7 \right\rfloor + 1$ 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 $B = 2^{\log_2 B}$
  
  › $AB = 2^{\log_2 A} \cdot 2^{\log_2 B} = 2^{\log_2 A + \log_2 B}$
  
  › so $\log_2 AB = \log_2 A + \log_2 B$

  › note: $\log AB \neq \log A \cdot \log B$
Other log properties

- \( \log \frac{A}{B} = \log A - \log B \)
- \( \log (A^B) = B \log A \)
- \( \log \log X < \log X < X \) 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

\[
B = 2^{\log_2 B}
\]

\[
x^{\log_2 B} = B
\]

\[
(2^{\log_2 x})^{\log_2 B} = 2^{\log_2 B}
\]

\[
2^{\log_2 x \log_2 B} = 2^{\log_2 B}
\]

\[
\log_2 x \log_2 B = \log_2 B
\]

\[
\log_x B = \frac{\log_2 B}{\log_2 x}
\]
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?
Algorithm Analysis

• Consider the following program segment:

\[
x := 0; \\
\text{for } i = 1 \text{ to } N \text{ do} \\
\quad \text{for } j = 1 \text{ to } i \text{ do} \\
\quad \quad x := x + 1;
\]

• What is the value of \( x \) at the end?
Analyzing the Loop

• Total number of times $x$ is incremented is executed =

$$1 + 2 + 3 + ... = \sum_{i=1}^{N} i = \frac{N(N+1)}{2}$$

• Congratulations - You’ve just analyzed your first program!
  › Running time of the program is proportional to $N(N+1)/2$ for all $N$
  › $O(N^2)$
Analyzing Mergesort

Mergesort(p : node pointer) : node pointer {
   Case {
      p = null : return p; //no elements
      p.next = null : return p; //one element
      else
         d : duo pointer; // duo has two fields first,second
         d := Split(p);
         return Merge(Mergesort(d.first),Mergesort(d.second));
   }
}

T(n) is the time to sort n items.
T(0), T(1) ≤ c
T(n) ≤ T(⌊n/2⌋) + T(⌈n/2⌉) + dn
Mergesort Analysis

Upper Bound

\[ T(n) \leq 2T(n/2) + dn \quad \text{Assuming } n \text{ is a power of } 2 \]
\[ \leq 2(2T(n/4) + dn/2) + dn \]
\[ = 4T(n/4) + 2dn \]
\[ \leq 4(2T(n/8) + dn/4) + 2dn \]
\[ = 8T(n/8) + 3dn \]
\[ \vdots \]
\[ \leq 2^k T(n/2^k) + kdn \]
\[ = nT(1) + kdn \quad \text{if } n = 2^k \]
\[ \leq cn + dn \log_2 n \]
\[ = O(n \log n) \]
Recursion Used Badly

- Classic example: Fibonacci numbers $F_n$
  
  $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$

  - $F_0 = 0$, $F_1 = 1$ (Base Cases)
  - 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

\[
\text{fib}(n : \text{integer}) : \text{integer} \{
\quad \text{Case} \{
\quad \quad n \leq 0 : \text{return } 0;
\quad \quad n = 1 : \text{return } 1;
\quad \quad \text{else} : \text{return } \text{fib}(n-1) + \text{fib}(n-2);
\quad \}\}
\]

- 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!
Fibonacci Analysis

Lower Bound

T(n) is the time to compute fib(n).

T(0), T(1) ≥ 1

T(n) ≥ T(n-1) + T(n-2)

It can be shown by induction that T(n) ≥ φ^{n-2}

where

φ = \frac{1 + \sqrt{5}}{2} ≈ 1.62
Iterative Algorithm for Fibonacci Numbers

fib_iter(n : integer): integer {
fib0, fib1, fibresult, i : integer;
fib0 := 0; fib1 := 1;
case { _
    n < 0 : fibresult := 0;
    n = 1 : fibresult := 1;
    else :
        for i = 2 to n do {
            fibresult := fib0 + fib1;
            fib0 := fib1;
            fib1 := fibresult;
        }
    }
return fibresult;
}
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 algorithms 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

![Graph showing the run times of T_A(N) = 50N and T_B(N) = N^2]

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 sure that small $N$ will be common forever.

• We will compare algorithms based on how they scale for large values of $N$. 
Order Notation

• Mainly used to express upper bounds on time of algorithms. “n” is the size of the input.

• \( T(n) = O(f(n)) \) if there are constants \( c \) and \( n_0 \) such that \( T(n) \leq c f(n) \) for all \( n \geq n_0 \).
  
  › 10000n + 10 \( n \log_2 n \) = \( O(n \log n) \)
  
  › .00001 \( n^2 \neq O(n \log n) \)

• Order notation ignores constant factors and low order terms.
Why Order Notation

• Program performance may vary by a constant factor depending on the compiler and the computer used.
• In asymptotic performance \( (n \to \infty) \) the low order terms are negligible.
Some Basic Time Bounds

• Logarithmic time is $O(\log n)$
• Linear time is $O(n)$
• Quadratic time is $O(n^2)$
• Cubic time is $O(n^3)$
• Polynomial time is $O(n^k)$ for some $k$.
• Exponential time is $O(c^n)$ for some $c > 1$. 
Kinds of Analysis

- **Asymptotic** – uses order notation, ignores constant factors and low order terms.
- **Upper bound vs. lower bound**
- **Worst case** – time bound valid for all inputs of length $n$.
- **Average case** – time bound valid on average – requires a distribution of inputs.
- **Amortized** – worst case time averaged over a sequence of operations.
- **Others** – best case, common case, cache miss