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 in Computer Science are powers of 2
- Binary numbers (base 2) are easily represented in digital computers
  - each "bit" is a 0 or a 1
  - \(2^0=1, 2^1=2, 2^2=4, 2^3=8, 2^4=16,\ldots, 2^{10}=1024\) (1K)
  - an n-bit wide field can hold \(2^n\) positive integers:
    - \(0 \leq k \leq 2^n - 1\)
Unsigned binary numbers

- 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
  - The value is $\sum_{i=0}^{i=n-1} a_i 2^i$
- Each bit position represents a power of 2 with $a_i = 0$ or $a_i = 1$
Logs and exponents

- Definition: $\log_2 x = y$ means $x = 2^y$
  - $8 = 2^3$, 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
  - $\log_2 256 = 8$
$x$, $2^x$ and $\log_2 x$
$2^x$ and $\log_2 x$
Floor and Ceiling

\[ \left\lfloor X \right\rfloor \]  Floor function: the largest integer \( \leq X \)

\[ \left\lfloor 2.7 \right\rfloor = 2 \quad \left\lceil -2.7 \right\rceil = -3 \quad \left\lceil 2 \right\rceil = 2 \]

\[ \left\lceil X \right\rceil \]  Ceiling function: the smallest integer \( \geq X \)

\[ \left\lceil 2.3 \right\rceil = 3 \quad \left\lceil -2.3 \right\rceil = -2 \quad \left\lceil 2 \right\rceil = 2 \]
Facts about Floor and Ceiling

1. \( X - 1 < \lfloor X \rfloor \leq X \)
2. \( X \leq \lceil X \rceil < X + 1 \)
3. \( \left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil = n \) if \( n \) is an integer
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

\[
\begin{align*}
B &= 2^{\log_x B} \\
x &= 2^{\log_x x} \\
(2^{\log_x x})^{\log_x B} &= 2^{\log_x B} \\
2^{\log_x x \log_x B} &= 2^{\log_x B} \\
\log_x x \log_x B &= \log_x B \\
\log_x B &= \frac{\log_2 B}{\log_2 x}
\end{align*}
\]
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 when you analyze algorithms?
Algorithm Analysis

- Consider the following program segment:

  \[
  x := 0;
  \]

  \[
  \text{for } i = 1 \text{ to } N \text{ do}
  \]

  \[
  \text{for } j = 1 \text{ to } i \text{ do}
  \]

  \[
  x := x + 1;
  \]

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

- Total number of times x is incremented is the number of “instructions” executed
  \[1+2+3+\ldots=\sum_{i=1}^{N} i = \frac{N(N+1)}{2}\]

- You’ve just analyzed the 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

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

- Classic example: Fibonacci numbers $F_n$

  0,1, 1, 2, 3, 5, 8, 13, 21, …

  - $F_0 = 0$, $F_1 = 1$ (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

\[ \text{fib}(n : \text{integer}) : \text{integer} \{ \]
  Case \{ 
    n < 0 : return 0; 
    n = 1 : return 1; 
    else : return \text{fib}(n-1) + \text{fib}(n-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 \( \text{fib}(N-i) \) multiple times!
Fibonacci Analysis
Lower Bound

$T(n)$ is the time to compute $\text{fib}(n)$.
$T(0), T(1) \geq 1$
$T(n) \geq T(n - 1) + T(n - 2)$

It can be shown by induction that $T(n) \geq \phi^{n-2}$
where
$\phi = \frac{1 + \sqrt{5}}{2} \approx 1.62$
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

\[ T_A(N) = 50N \]
\[ T_B(N) = N^2 \]

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 (one more time)

- 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 ($n \rightarrow \infty$) 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 (80%-20%) etc.