Mathematical Background 2

CSE 373
Data Structures

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 \(\sum_{i=0}^{i=n-1} a_i \cdot 2^i\)
  » 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 \(2^8-1 = 255\)
  » \(\log_2 256 = 8\)
Floor and Ceiling

- **Floor function**: the largest integer \( \leq X \)
  - \( [2.7] = 2 \)
  - \( [-2.7] = -3 \)
  - \( [2] = 2 \)

- **Ceiling function**: the smallest integer \( \geq X \)
  - \( \lceil 2.3 \rceil = 3 \)
  - \( \lceil -2.3 \rceil = -2 \)
  - \( [2] = 2 \)

Facts about Floor and Ceiling

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

Properties of logs (of the mathematical kind)

- We will assume logs to base 2 unless specified otherwise
- **Logarithm properties**:
  - \( \log AB = \log A + \log B \)
  - \( \log A/B = \log A - \log B \)
  - \( \log (A^B) = B \log A \)
  - \( \log \log X < \log X < X \) for all \( X > 0 \)
    - \( \log \log X = Y \) means \( 2^Y = X \)
    - \( \log X \) grows slower than \( X \)
      - called a 'sub-linear' function

Other log properties

1. \( \log A/B = \log A - \log B \)
2. \( \log (A^B) = B \log A \)
3. \( \log \log X < \log X < X \) for all \( X > 0 \)
   - \( \log \log X = Y \) means \( 2^Y = X \)
   - \( \log X \) 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
  - \( B = 2^{\log_{x} B} \)
  - \( X = 2^{\log_{x} X} \)
  - \( \log_{x} B = \log B / \log_{2} x \)
  - \( \log_{x} A = \log A / \log_{2} x \)
  - \( \log_{x} (A/B) = \log_{x} A - \log_{x} B \)
  - \( \log_{x} (A^B) = B \log_{x} A \)
  - \( \log_{x} x = 1 \)
  - \( \log_{x} 1 = 0 \)
  - \( \log_{x} 2 = \log_{2} x \)
  - \( \log_{x} (A/B) = \log_{x} A - \log_{x} B \)
  - \( \log_{x} (A^B) = B \log_{x} A \)
Arithmetic Series

- S(N) = 1 + 2 + ... + N = \sum_{i=1}^{N} i
- The sum is:
  - S(1) = 1
  - S(2) = 1 + 2 = 3
  - S(3) = 1 + 2 + 3 = 6
- \[ \sum_{i=1}^{N} i = \frac{N(N+1)}{2} \]

Algorithm Analysis

- Consider the following program segment:
  ```plaintext
  x := 0;
  for i = 1 to N do
      for j = 1 to i do
          x := x + 1;
  ```
- Why is this formula useful when you analyze algorithms?
- 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 + ... + \( \sum_{i=1}^{N} \frac{N(N+1)}{2} \)
- 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

```plaintext
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));
    }
}
```

Recursion Used Badly

- Classic example: Fibonacci numbers \( F_n \)
- \( F_0 = 0, F_1 = 1 \) (Base Cases)
- Rest are sum of preceding two
  \( F_n = F_{n-1} + F_{n-2} \) \( n > 1 \)
Recursive Procedure for Fibonacci Numbers

```pseudocode
fib(n : integer) : integer {
    Case {
        n < 0 : return 0;
        n = 1 : return 1;
        else : return fib(n-1) + 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 $fib(N-i)$ multiple times!

Fibonacci Analysis

Lower Bound

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

$T(0), T(1) \geq 1$

$T(n) \geq T(n-1) + T(n-2)$

It can be shown by induction that $T(n) \geq \phi^n$ where

$\phi = \frac{1+\sqrt{5}}{2} \approx 1.62$

Iterative Algorithm for Fibonacci Numbers

```pseudocode
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 $B$.

Now which algorithm would you choose?

Asymptotic Behavior

• The “asymptotic” performance as $N \rightarrow \infty$, 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.