CISC 320 Introduction to Algorithms
Fall 2003

Course Overview

Administrative

- Course syllabus
- Course webpage: http://www.cis.udel.edu/~lliao/cis320f03
- Office hours: Tuesdays and Thursdays 1:30-3:00pm
- TA: Huizhuan Wu (wuhuizhu@cis.udel.edu)
  - Tuesdays 8:30-10:30AM, 115B Pearson Hall
- Your name and email
- A random ID will be assigned to you for the purpose of posting grades.
Subject: Industry perspective on CS
Date: 12 Jul 2003 13:48:26 -0700

I had a conversation 2 weeks ago with a colleague at Intel.
His group was looking for a few CS people last year. He said that
almost everyone he interviewed was "completely useless: they know
nothing about algorithms, analysis, or any kind of mathematical
reasoning; the only thing they know is Java." This seems to match
what I hear a lot of people say around here about recent CS grads.
It's like they go to college for 4 years and come out knowing one
thing.

Has CIS switched from C++ to Java? That would be a mistake if any of
them want to work for Intel -- I think the assumption here is if your
primary language is Java, then you're worthless. (BTW, most of our
work is a mix of C++ and C, with the occasional Perl script here and
there.)

Kevin
--
Overview: Designing Algorithms

- Designing paradigms
  - Brute Force
  - Divide and Conquer
  - Greedy
  - Dynamic programming
  - Randomized
  - Parallel

- Bottom line
  - Contents
    - a list of algorithms
    - Able to recite the main idea
      - convince others and yourself it works
      - complexity
    - Able to write (pseudo) code for it
  - Methods
    - Several design techniques
    - Proof techniques
    - Abstract thinking
How to solve it? (Polya)

- What is the problem?
- Can we solve it anyway (Brute force)?
  - Is our solution correct?
    - All inputs
    - Special inputs (boundaries and/or limits)
- Can we solve it more efficiently?
  - Have we used all info available?
    - Need specific data structure to store the input and/or intermediaries?
  - Can we divide and conquer, be greedy, do dynamic programming, etc.?
  - Can we improve average-case, if not worst-case, performance? Amortizing? Do we need to randomize to enforce a favorable average-case performance?
  - …

Overview: Designing Algorithms

- How to measure time complexity?
  - In terms of seconds? (a faster CPU takes less time)
  - In terms of machine instructions, i.e., instructions/second (MIPS)? (hard to estimate number of machine instructions based on psuedo code, also compiler dependent)
  - In terms of lines of code executed.
Overview: Example

Example: Sequential search of an unordered array E

```c
int seqSearch (int[] E, int n, int k)
1    int ans, index;
2    ans = -1;
3    for (i = 0; i < n; i++) {
4        if(k == E[i])
5            ans = I;
6            break;
7    return ans;
```

Worst-case: \( T_{wc}(n) = \max\{T(I) | I \in D\} = 3 + 2n \)

Overview: Example (cont’d)

- **Average case**: \( T_{avg}(n) = \sum_{I \in D} Pr(I)T(I) \)
  - \( S_i \) is the event that \( E[i] = k \)
  - \( T_{succ}(n) = \sum_{0 \leq i \leq (n-1)} Pr(S_i|succ)T(S_i) \)
    - \( = \sum_{0 \leq i \leq (n-1)} (1/n) (2i +4) \)
    - \( = 3 + n \)
  - \( T_{avg}(n) = Pr(succ) T_{succ}(n) + Pr(fail) T_{fail}(n) \)
    - \( = 50\% (3+n) + 50\% (3+2n) \)
    - \( = 3 + (3/2)n \)
Lessons learned:
- Worst-case: the most commonly used
- Average: harder to analyze
- Best-case: not that useful

Overview: Big O, etc.
- Asymptotic behavior of functions

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>33n</td>
<td>46n lg n</td>
<td>13 n²</td>
<td>3.4 n³</td>
<td>2^n</td>
</tr>
<tr>
<td>Input</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>.00033</td>
<td>.015</td>
<td>.013</td>
<td>.0034</td>
<td>.001</td>
</tr>
<tr>
<td>100</td>
<td>.003</td>
<td>.03</td>
<td>.13</td>
<td>3.4</td>
<td>4x 10^{16}</td>
</tr>
<tr>
<td>1,000</td>
<td>.033</td>
<td>.45</td>
<td>13</td>
<td>.94hr.</td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td>.33</td>
<td>6.1</td>
<td>22min.</td>
<td>39days</td>
<td></td>
</tr>
<tr>
<td>100,000</td>
<td>3.3</td>
<td>1.3min.</td>
<td>1.5days</td>
<td>108yr.</td>
<td></td>
</tr>
</tbody>
</table>
Overview: Big Oh, etc.

- $O(g(n)) = \{ f(n): \text{there exist positive constant } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0 \}$

- $\Omega(g(n)) = \{ f(n): \text{there exist positive constant } c \text{ and } n_0 \text{ such that } 0 \leq c \cdot g(n) \leq f(n) \text{ for all } n \geq n_0 \}$

- $\Theta(g(n)) = \{ f(n): \text{there exist positive constant } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ for all } n \geq n_0 \}$

Overview: Big Oh, etc.

- Typical asymptotic complexities are
  - $\Theta(lg(n))$, logarithmic (sub-linear)
  - $\Theta(n)$, linear
  - $\Theta(n \cdot lg(n))$, $n \cdot lg \cdot n$
  - $\Theta(n^2)$, quadratic
  - $\Theta(n^3)$, cubic
  - $\Theta(2^n)$, exponential
Overview: Big Oh, etc.

- **Examples:**
  - $3n^2 + 5n + 24 = \Theta(n^2)$
  - Is $3n^2 + 5n + 24$ in $O(n^2)$?
  - Is $15n + 102$ in $O(n^2)$?
  - Is $n^3 + n \log(n)$ in $O(n^2)$?
  - Drop low-order terms
  - Ignore constant in the leading term

Overview: Big Oh, etc.

- **Transitivity**
  - $f \in O(g), g \in O(h) \Rightarrow f \in O(h)$
- "Reflex-symmetry"
  - $f \in O(g) \Leftrightarrow g \in \Omega(f)$
  - $f \in \Theta(g) \Rightarrow g \in \Theta(f)$
  - $O(f+g) = O(\max(f,g))$
Spectrum of computational complexity

- Undecidable
- Superexponential
- Exponential
- Intractable
- Polynomial
- \(n^3\)
- \(n^2\)
- \(n \log n\)
- \(n\)
- Sublinear
- NP-complete problems
- Hilbert’s Tenth problem
- Matrix multiplication
- Sorting
- Tractable

Complexity of problems = necessary and sufficient work to solve the problem.

- Lower bounds of time complexities of algorithms
  - Lower bound: the least work (or steps) needed; but no guarantee of sufficiency; the higher, the better.
  - Upper bound: the most work (or steps) needed; from known algorithms that solve the problem; the lower, the better.
  - Complexity of the problem where the lower and upper bounds meet.
  - Optimal algorithms: time complexities = the complexity of the problem.
Example: finding the largest entry in an array
Example: matrix multiplication
\[ C_{ij} = \sum_{0 \leq k \leq (n-1)} A_{ik} B_{kj} \]

- Brute force \((n^3)\)
- Strassen’s algorithm \((n^{2.81})\)
- complexity of matrix multiplication is not known.

Searching an ordered array
- Alg1
  - Early stop => improve the average-case, no impact on the worst-case
- Alg2
  - Search every j entry => \(n/j + j\) comparisons
- Alg3
  - Optimize on j => \(T_{wc}(n) = 2\sqrt{n}\) comparisons
- Alg4: binary search => \(T_{wc}(n) = \lg n\)
  - This is an optimal algorithm.
Theorem 1.16  Complexity of searching an ordered array of integers via comparisons is \( \lg(n+1) \), where \( n \) is the size.

Proof

- Decision tree (a binary tree) is to visualize operation flow of an algorithm
- Max # of comparisons = \( p \), # of nodes on the Longest path of the decision tree
- max # of nodes, \( N \leq 1 + 2 + 4 + \ldots + 2^{p-1} = 2^p - 1 \)
- \( p \geq \lg(N + 1) \)
- \( N \geq n \), where \( n \) is the array size. (because every entry in array must appear at least once on the decision tree for the algorithm to work correctly)