A letter from industry

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. If 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

---

What is an algorithm?

- Step-by-step, computer executable, solution to a “computable” problem.
- Yes, non-computable problems do exist.
  - E.g., Turing’s Halting Problem
  - Computability is a subject of study of CISC 401
- Computable problems are many: -)
  - The Human Genome Project: sequence assembly, gene identification ...
  - The Internet: information search – Google it.
  - E-Commerce: data security (RSA Public-Key cryptography)
  - RSA won the Turing Award in 2002. R is Rivest, one of the authors of our textbook.
  - Airline Scheduling:
    - ...

Logistics (by problem types) adopted

- Part 1: Sorting algorithms and Selection algorithms
- Part 3: Graph algorithms
- Part 4: NP-completeness; Approximation algorithms; Parallel algorithms

Workload

- Five homework assignments (8% each)
  - Four “paper-and-pencil” and one programming
- Two exams (30% each)
  - One is on Oct. 18, and the other is given during the final
  - Mainly facts problems, e.g., describing an algorithm learned in the class
Contents
- Algorithms
  - Able to recite the main ideas
  - Convince others and yourself it works
  - Complexity
  - Other characteristics, e.g., on-line v.s. off-line.
- Methods
  - Several design techniques
  - Proof techniques (induction, decision trees)
  - Ways of abstract thinking

Able to write (pseudo) code for it

Methods

Several design techniques

Proof techniques (induction, decision trees)

Ways of abstract thinking

Analyzing Algorithms

Correctness
- Can be proved (recursion and induction)
- Approximation

Complexity (efficiency)
- Time and Space
- Worst-case, Average-case, and Best case
- Asymptotical
- Optimality

Simplicity and Clarity

Correctness
- Can be proved (recursion and induction)
- Approximation

Complexity (efficiency)
- Time and Space
- Worst-case, Average-case, and Best case
- Asymptotical
- Optimality

Simplicity and Clarity

How to solve it? (Polya’s book)

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?
  - Should 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?
  - ...

Optimality

An algorithm is Optimal: if its time complexity = the complexity of the problem.

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

- Lower bound: the least work (or steps) needed, but no guarantee to solve the problem.
- Upper bound: the sufficient work (or steps) needed, from known algorithms that solve the problem.
- Complexity of the problem is given by the lower and upper bounds that meet each other.
- Complexity of the problem is unknown when there is a gap between the tightest known lower and upper bounds.
  - E.g., multiplication of two n-by-n matrices: Tightest lower bound is $O(n^2)$ and tightest upper bound is $O(n^{2.376})$.

Example 1: finding the largest entry in an array.
- We do not know the complexity because the problem is not well-defined. E.g., if the array is already sorted?

Example 2: matrix multiplication

- Brute force ($n^3$)
- Strassen’s algorithm ($n^{2.81}$)
- Complexity of matrix multiplication is not yet known.
Example 3: Sequential search of an unordered array E

```c
int seqSearch (int[] E, int n, int k)
{
    int ans, index;
    ans = -1;
    for (i = 0; i < n; i++) {
        if (k == E[i])
            ans = i;
        break;
    }
    return ans;
}
```

Number of comparisons (execution of line 4)

Worst-case: n

Best-case: 1

Average-case: ???

---

Example 4: 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
  - Optimizing on j => T_{qc}(n) = 2\cdot n comparisons

- Alg4: binary search => T_{wc}(n) = \lg n
  - This is an optimal algorithm. Why?
  - The binary search is not an on-line algorithm

---

Complexity of searching an ordered array of integers via comparisons is \(\lg(n+1)\), where \(n\) is the array size.

**Proof**

- Decision tree (a binary tree) is to visualize operation flow of an algorithm
- Let \(p\) = max \# of comparisons = \# of nodes on the longest path of the decision tree
- Let \(N = \text{max \# of nodes in decision tree. } N \leq 1 + 2 + 4 + \ldots + 2^{p-1} = 2^p - 1\) (given the height, a balanced binary tree can hold more nodes than unbalanced).
- \(p \geq \lg (N + 1)\)
- \(N \geq n\), 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)
- Note: this does not say every node will actually be visited during a run of the algorithm.

---

Spectrum of computational complexity

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Hilbert’s Tenth Problem</th>
<th>Turing’s Halting Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undecidable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Super-exponential</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intractable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Polynomial</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(n^1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(n^2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(n\log n)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sublinear</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>