Overview: Analyzing Algorithms

- Correctness
  - Can be proved (recursion and induction)
- Complexity
  - Time and Space
  - Worst-case, Average-case, and Best case
- Asymptotical
- Optimality
- Simplicity and Clarity

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 oder intermediaries?
  - Can we divide and conquer, be greedy, 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 pseudo code, also compiler dependent)
  - In terms of lines of code executed.

Example: 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;
```

Worst-case: \( T_w (n) = \max \{ T(I) \mid I = 3 + 2n \} \)

Overview: Example (cont’d)

Average case: \( T_{avg}(n) = \sum_{i=0}^{n} Pr(I)T(I) \)

- \( S_i \) is the event that \( E[i] = k \)
- \( T_{suc}(n) = \sum\limits_{i=0}^{n} Pr(S_i)suc)T(S_i) = \sum\limits_{i=0}^{n}(1/n) (2i + 4) = 3 + n \)
- \( T_{avg}(n) = Pr(suc)T_{suc}(n) + Pr(fail)T_{fail}(n) = 50\% (3+n) + 50\% (3+2n) = 3 + (3/2)n \)

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>Algorithm</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 n^2</td>
<td>3.4 n^3</td>
<td>2^n</td>
</tr>
<tr>
<td>Input</td>
<td>0.0003</td>
<td>0.015</td>
<td>0.013</td>
<td>0.034</td>
<td>0.001</td>
</tr>
<tr>
<td>10</td>
<td>0.003</td>
<td>0.03</td>
<td>13</td>
<td>3.4</td>
<td>4x 10^{10}</td>
</tr>
<tr>
<td>1,000</td>
<td>0.033</td>
<td>45</td>
<td>13</td>
<td>345y.</td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td>33</td>
<td>6.1</td>
<td>22min.</td>
<td>38days</td>
<td></td>
</tr>
<tr>
<td>100,000</td>
<td>3.3</td>
<td>1.3min.</td>
<td>1.5days</td>
<td>188yr.</td>
<td></td>
</tr>
</tbody>
</table>
Overview: Big Oh, etc.

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

Typical asymptotic complexities are

- $O(n \log(n))$, logarithmic (sub-linear)
- $O(n)$, linear
- $O(n \log(n))$, $n \cdot \log(n)$
- $O(n^2)$, quadratic
- $O(n^3)$, cubic
- $O(2^n)$, exponential

Examples:

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

Overview: Big Oh, etc.

Transitivity

- $f \in O(g)$, $g \in O(h) \Rightarrow f \in O(h)$
- “Reflexive symmetry”
- $f \in O(g) \Leftrightarrow g \in \Omega(f)$
- $f \in \Theta(g) \Rightarrow g \in \Theta(f)$
- $O(f+g) = O(\max(f, g))$

Overview: Big Oh, etc.

Spectrum of computational complexity

<table>
<thead>
<tr>
<th>Intractable</th>
<th>Undecidable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polynomial</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Matrix multiplication</td>
<td>$n \log n$</td>
</tr>
<tr>
<td>Sorting</td>
<td>$n$</td>
</tr>
<tr>
<td>Sublinear</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NP-complete problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intractable</td>
</tr>
<tr>
<td>Exponential</td>
</tr>
<tr>
<td>Superexponential</td>
</tr>
</tbody>
</table>

Optimality

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

- Lower bounds: 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

\[ C = \sum_{i=0}^{n-1} A_i \cdot B_i \]

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

Optimality: when lower bound is NOT known

Searching an ordered array

- Alg1
  - Early stop -> improve the average-case, no impact on the worst-case
- Alg2
  - Search every entry \( n / j + j \) comparisons
- Alg3
  - Optimize on \( j \) \( T_{opt}(n) = 2 \cdot n \) comparisons
- Alg4: binary search \( T_{opt}(n) = \log n \)
  - This is an optimal algorithm.

Optimality: when lower bound IS known

Theorem 1.16 Complexity of searching an ordered array of integers via comparisons is \( \log(n+1) \), 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, \( N \leq 1 + 2 + 4 + \ldots + 2^{k-1} = 2^k - 1 \)
- \( p \geq \log(N+1) \)
- \( n \geq n \), 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)