Lecture 20: Analysis of Algorithms

Overview

Ex 1: N-body Simulation.
- Simulate gravitational interactions among N bodies.
- Brute force: \( N^2 \) steps.
- Barnes-Hut: \( N \log N \) steps, enables new research.

Ex 2: Discrete Fourier transform.
- Break down waveform of N samples into periodic components.
  - Applications: DVD players, JPEG, analysis of astronomical data, medical imaging, nonlinear Schroedinger equation, ...
- Brute force: \( N^2 \) steps.
- FFT algorithm: \( N \log N \) steps, enables new technology.

Ex 3: Sorting.
- Rearrange N items in ascending order.
- Fundamental information processing abstraction.

Andrew Appel
PU '81

Jon von Neumann
IAS 1945

Better Machines vs. Better Algorithms

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Better Machine</th>
<th>Better Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost</td>
<td>$$$ or more.</td>
<td>$ or less.</td>
</tr>
<tr>
<td>Applicability</td>
<td>Makes &quot;everything&quot; run faster.</td>
<td>May not apply to some problems.</td>
</tr>
<tr>
<td>Improvement</td>
<td>Incremental quantitative improvements.</td>
<td>Dramatic quantitative improvements possible.</td>
</tr>
</tbody>
</table>

Ex: sorting a billion items with quicksort on home PC is 100x faster than insertion sorting it on supercomputer.

<table>
<thead>
<tr>
<th>Computer</th>
<th>Comparisons Per Second</th>
<th>Insertion</th>
<th>Quicksort</th>
</tr>
</thead>
<tbody>
<tr>
<td>home PC</td>
<td>( 10^7 )</td>
<td>3 centuries</td>
<td>3 hours</td>
</tr>
<tr>
<td>super</td>
<td>( 10^{12} )</td>
<td>2 weeks</td>
<td>instant</td>
</tr>
</tbody>
</table>
Case Study: Sorting

Sorting problem:
- Given N items, rearrange them in ascending order.
- Applications: databases, data compression, computational biology, computer graphics, ...

Helper Functions

**Is string \( s \) lexicographically less than string \( t \)?**

```java
static boolean less(String s, String t) {
    int n = Math.min(s.length(), t.length());
    for (int i = 0; i < n; i++) {
        if (s.charAt(i) < t.charAt(i)) return true;
        else if (s.charAt(i) > t.charAt(i)) return false;
    }
    return (s.length() < t.length());
}
```

**Swap string references \( a[i] \) and \( a[j] \).**

```java
static void exch(String[] a, int i, int j) {
    String swap = a[i];
    a[i] = a[j];
    a[j] = swap;
}
```

Insertion Sort

**Insertion sort**
- Brute-force sorting solution.
- Move left-to-right through array.
- Exchange next element with larger elements to its left, one-by-one.

```java
static void insertionsort(String[] a) {
    for (int i = 0; i < a.length; i++) {
        for (int j = i; j > 0; j--) {
            if (less(a[j], a[j-1]))
                exch(a, j, j-1);
            else break;
        }
    }
}
```

Estimating the Running Time

**Total run time.** Sum over all instructions: frequency \( \times \) cost.  
**Frequency.** Determined by algorithm and input.  
**Cost.** Determined by compiler and machine. 

**Easier alternative.**
(i) Analyze asymptotic growth.  
(ii) For medium N, run and measure time.  
(iii) For large N, use (i) and (ii) to predict time. 

**Asymptotic growth rates.**
- Estimate time as a function of input size \( N \).
  - \( N, N \log N, N^2, N^3, 2^N, N! \)
  - Typically ignore lower order terms and leading coefficients.  
  - Ex. \( 6N^3 + 17N^2 + 56 \) is asymptotically proportional to \( N^3 \)
Insertion Sort: Analysis

(i) Analyze asymptotic growth of insertion sort.
- Depends on number of elements \( N \) to sort.
- Depends on specific input.
- Depends on how long compare and exchange operation takes.

Worst case.
- Elements in reverse sorted order.
  - \( i^{th} \) iteration requires \( i - 1 \) compare and exchange operations
  - total = \( 0 + 1 + 2 + \ldots + N-2 + N-1 = N(N-1)/2 \)

Best case.
- Elements in sorted order already.
  - \( i^{th} \) iteration requires only 1 compare operation
  - total = \( 0 + 1 + 1 + \ldots + 1 = N - 1 \)

Average case.
- Elements are randomly ordered.
  - \( i^{th} \) iteration requires \( i/2 \) comparison on average
  - total = \( 0 + 1/2 + 2/2 + \ldots + (N-1)/2 = N(N-1)/4 \)
Estimating the Running Time: Insertion Sort

(ii) Collect empirical data for medium N.
   • Use System.currentTimeMillis to get number of milliseconds since January 1, 1970.

```java
long start = System.currentTimeMillis();
insertionsort(a);
long stop = System.currentTimeMillis();
double elapsed = (stop - start) / 1000.0;
```

<table>
<thead>
<tr>
<th>N</th>
<th>comparisons</th>
<th>running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>6.1 million</td>
<td>0.7 seconds</td>
</tr>
<tr>
<td>10,000</td>
<td>23 million</td>
<td>4 seconds</td>
</tr>
<tr>
<td>20,000</td>
<td>98 million</td>
<td>25 seconds</td>
</tr>
<tr>
<td>40,000</td>
<td>400 million</td>
<td>130 seconds</td>
</tr>
<tr>
<td>80,000</td>
<td>1.6 billion</td>
<td>570 seconds</td>
</tr>
</tbody>
</table>

Data source: First N words of Charles Dickens’s life work.
Machine: Dell 2.26GHz PC with 1GB memory.

Insertion Sort: Number of Comparisons

(iii) Predict for large N.
   • Quadratic: 2x factor in input size ⇒ 4x factor in comparisons.

Lesson. Supercomputer can’t rescue bad algorithm.

<table>
<thead>
<tr>
<th>computer</th>
<th>comparisons per second</th>
<th>Input Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>thousand</td>
</tr>
<tr>
<td>home</td>
<td>10^7</td>
<td>instant</td>
</tr>
<tr>
<td>super</td>
<td>10^12</td>
<td>instant</td>
</tr>
</tbody>
</table>

Quicksort

Quicksort.
   • Partition array so that:
     - some partitioning element a[m] is in its final position
     - no larger element to the left of m
     - no smaller element to the right of m

How to speed up the program?
   • Make less run substantially faster.
   • Call less substantially fewer times.
Quick sort.  
- Partition array so that:
  - some partitioning element \( a[m] \) is in its final position
  - no larger element to the left of \( m \)
  - no smaller element to the right of \( m \)
- Sort each "half" recursively.

**Quick sort: Java Implementation**

- Partition array so that:
  - some partitioning element \( a[m] \) is in its final position
  - no larger element to the left of \( m \)
  - no smaller element to the right of \( m \)
- Sort each "half" recursively.

```
public void quicksort(String[] a, int left, int right) {
    if (right <= left) return;
    int i = partition(a, left, right);
    quicksort(a, left, i-1);
    quicksort(a, i+1, right);
}
```

**Quick sort: Implementing Partition**

How do we partition in-place efficiently?

```
static int partition(String[] a, int left, int right) {
    int i = left - 1;
    int j = right;
    while (true) {
        while (less(a[++i], a[right]));  // find item on left to swap
            if (j == left) break;
        if (i >= j) break;  // check if pointers cross
        exch(a, i, j);  // swap
    }
    exch(a, i, right);  // swap with partitioning element
    return i;  // return index where crossing occurs
}
```
QuickSort: Analysis

Average case running time.
- Roughly $2N \log_2 N$ comparisons.  
  see Sedgewick book or COS 226 for proof
- Assumption: partition on random element (instead of rightmost one).

Worst case running proportional to $N^2$.
- More likely that you are struck by lightning and meteor at same time.

Numerical experiments roughly match predictions.
- A bit better because of many equal keys.

<table>
<thead>
<tr>
<th>N</th>
<th>comparisons</th>
<th>running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>200,000</td>
<td>4.0 million</td>
<td>0.9 seconds</td>
</tr>
<tr>
<td>400,000</td>
<td>8.3 million</td>
<td>2.2 seconds</td>
</tr>
<tr>
<td>1 million</td>
<td>23 million</td>
<td>5.8 seconds</td>
</tr>
<tr>
<td>2 million</td>
<td>47 million</td>
<td>13 seconds</td>
</tr>
<tr>
<td>4 million</td>
<td>97 million</td>
<td>31 seconds</td>
</tr>
</tbody>
</table>

Lesson. Good algorithm can be much more powerful than supercomputer.

<table>
<thead>
<tr>
<th>Computer</th>
<th>Comparisons Per Second</th>
<th>Insertion</th>
<th>Quicksort</th>
</tr>
</thead>
<tbody>
<tr>
<td>home PC</td>
<td>$10^7$</td>
<td>3 centuries</td>
<td>3 hours</td>
</tr>
<tr>
<td>super</td>
<td>$10^{12}$</td>
<td>2 weeks</td>
<td>instant</td>
</tr>
</tbody>
</table>

N = 1 billion

Computational Complexity of Problems

Computational complexity. Framework to study efficiency of algorithms for solving a particular problem X.

Upper bound. Cost guarantee provided by some algorithm for X.
Lower bound. Proven limit on cost guarantee of any algorithm for X.
Optimal algorithm. Algorithm with best cost guarantee for X.

Example 1: X = sorting.
- Measure costs in terms of comparisons.
- Upper bound = $N \log^2 N$ with mergesort.
- Lower bound = $N \log^2 N - N \log_2 e$.
- Optimal algorithm = mergesort.

Theory simplifies and provides guidelines.
Computational Complexity of Problems

Computational complexity. Framework to study efficiency of algorithms for solving a particular problem $X$.

Upper bound. Cost guarantee provided by some algorithm for $X$.
Lower bound. Proven limit on cost guarantee of any algorithm for $X$.
Optimal algorithm. Algorithm with best cost guarantee for $X$.

Example 2: $X =$ Euclidean TSP.
- Measure cost in terms of arithmetic operations.
- Upper bound $= 2^N$ by dynamic programming.
- Lower bound $= N$.
- Optimal algorithm = ask again in 50 years.

Essence of computational complexity: closing the gap.

Summary

How can I evaluate the performance of my algorithm?
- Computational experiments.
- Analysis of algorithms.

What if it’s not fast enough?
- Understand why.
- Buy a faster computer.
- Find a better algorithm in a textbook.
- Discover a new algorithm.

Sobering philosophical thoughts.
- In theory, most problems are undecidable.
- In practice, most remaining problems are intractable.
- Analysis of algorithms helps us improve the ones we use.