Algorithms

- What is an algorithm?

- So far we have been expressing our algorithms in Java code
  - Pseudocode is a more informal notational system

- Coming up with solution is just the first problem
- For many problems, there may be several competing algorithms
- Computational complexity
  - Rigorous and useful framework for comparing algorithms and predicting performance

Linear Growth

- Grade school addition
  - Work is proportional to number of digits N
  - Linear growth: kN for some constant k

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
  +---+---|
  | 4 | 2 |
  =---+---|
  | 1 | 2 | 0 |

  - How many reads? How many writes? How many operations?

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>+</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>
=---+---+---+---|
| 1 | 1 | 1 | 2 | 0 |

N = 2

N = 4

Quadratic Growth

- Grade school multiplication
  - Work is proportional to square of number of digits N
  - Quadratic growth: kN^2 for some constant k

  | 7 | 8 |
  *---|---|
  | 4 | 2 |
  =---+---|
  | 1 | 5 | 6 |
  +---+---+---|
  | 3 | 1 | 2 | 0 |
  +---+---+---+---|
  | 3 | 2 | 7 | 6 |

  - How many reads? How many writes? How many operations?

N = 2

N = 4

Sorting

- Given n items, rearrange them so that they are in increasing order
- A key recurring problem
- Many different methods, how do we choose?
- Given a set of cards, describe how you would sort them:

- Given a set of words, describe how you would sort them in alphabetical order?
Why Does It Matter?

<table>
<thead>
<tr>
<th>Run time (nanoseconds)</th>
<th>1.3 (N^3)</th>
<th>10 (N^2)</th>
<th>47 (N \log_2 N)</th>
<th>48 (N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to solve a problem of size</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>1.3 seconds</td>
<td>10 msec</td>
<td>0.4 msec</td>
<td>0.048 msec</td>
</tr>
<tr>
<td>10,000</td>
<td>22 minutes</td>
<td>1 second</td>
<td>6 msec</td>
<td>0.48 msec</td>
</tr>
<tr>
<td>100,000</td>
<td>15 days</td>
<td>1.7 minutes</td>
<td>78 msec</td>
<td>4.8 msec</td>
</tr>
<tr>
<td>million</td>
<td>41 years</td>
<td>2.8 hours</td>
<td>0.94 seconds</td>
<td>48 msec</td>
</tr>
<tr>
<td>10 million</td>
<td>41 millennia</td>
<td>1.7 weeks</td>
<td>11 seconds</td>
<td>0.48 seconds</td>
</tr>
</tbody>
</table>

Max size problem solved in one second
- 920
- 10,000
- 1 million
- 21 million
- 3,600
- 77,000
- 49 million
- 1.3 billion
- 14,000
- 600,000
- 2.4 trillion
- 76 trillion
- 41,000
- 2.9 million
- 50 trillion
- 1,800 trillion

Orders of Magnitude

<table>
<thead>
<tr>
<th>Seconds</th>
<th>Equivalent</th>
<th>Meters Per Second</th>
<th>Imperial Units</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 second</td>
<td>(10^{-10})</td>
<td>1.2 in / decade</td>
<td>Continental drift</td>
</tr>
<tr>
<td>10</td>
<td>10 seconds</td>
<td>(10^{-8})</td>
<td>1 ft / year</td>
<td>Hair growing</td>
</tr>
<tr>
<td>(10^2)</td>
<td>1.7 minutes</td>
<td>(10^{-6})</td>
<td>3.4 in / day</td>
<td>Glacier</td>
</tr>
<tr>
<td>(10^3)</td>
<td>17 minutes</td>
<td>(10^{-4})</td>
<td>1.2 ft / hour</td>
<td>Gastro-intestinal tract</td>
</tr>
<tr>
<td>(10^4)</td>
<td>2.8 hours</td>
<td>(10^{-2})</td>
<td>2 ft / minute</td>
<td>Ant</td>
</tr>
<tr>
<td>(10^5)</td>
<td>1.1 days</td>
<td>1</td>
<td>2.2 mi / hour</td>
<td>Human walk</td>
</tr>
<tr>
<td>(10^6)</td>
<td>1.6 weeks</td>
<td>(10^0)</td>
<td>220 mi / hour</td>
<td>Propeller airplane</td>
</tr>
<tr>
<td>(10^7)</td>
<td>3.8 months</td>
<td>(10^2)</td>
<td>370 mi / min</td>
<td>Space shuttle</td>
</tr>
<tr>
<td>(10^8)</td>
<td>3.1 years</td>
<td>(10^4)</td>
<td>620 mi / sec</td>
<td>Earth in galactic orbit</td>
</tr>
<tr>
<td>(10^9)</td>
<td>3.1 decades</td>
<td>(10^6)</td>
<td>62,000 mi / sec</td>
<td>1/3 speed of light</td>
</tr>
<tr>
<td>(10^{10})</td>
<td>3.1 centuries</td>
<td>(10^8)</td>
<td>2 (10^{10}) thousand</td>
<td></td>
</tr>
</tbody>
</table>

Powers of 2

- \(2^{10}\) thousand
- \(2^{20}\) million
- \(2^{30}\) billion

Historical Quest for Speed

Multiplication: \(a \times b\).
- Naïve: add \(a\) to itself \(b\) times. \(N 2^N\) steps
- Grade school. \(N^2\) steps
- Divide-and-conquer (Karatsuba, 1962). \(N^{1.58}\) steps
- Ingenuity (Schönhage and Strassen, 1971). \(N \log N \log \log N\) steps

Greatest common divisor: \(gcd(a, b)\).
- Naïve: factor \(a\) and \(b\), then find \(gcd(a, b)\). \(2^N\) steps
- Euclid (20 BCE): \(gcd(a, b) = gcd(b, a \mod b)\). \(N\) steps

Better Machines vs. Better Algorithms

New machine.
- Costs $$$ or more.
- Makes "everything" finish sooner.
- Incremental quantitative improvements (Moore’s Law).
- May not help much with some problems.

New algorithm.
- Costs $ or less.
- Dramatic qualitative improvements possible! (million times faster)
- May make the difference, allowing specific problem to be solved.
- May not help much with some problems.
Impact of Better Algorithms

Example 1: N-body-simulation.
- Simulate gravitational interactions among N bodies.
  - physicists want N = # atoms in universe
- Brute force method: N^2 steps.

Example 2: Discrete Fourier Transform (DFT).
- Breaks down waveforms (sound) into periodic components.
  - foundation of signal processing
  - CD players, JPEG, analyzing astronomical data, etc.
- Grade school method: N^3 steps.
  - FFT algorithm: N log N steps, enables new technology.

Case Study: Sorting

Sorting problem:
- Given N items, rearrange them so that they are in increasing order.
- Among most fundamental problems.

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

Profiling Insertion Sort Analytically

How long does insertion sort take?
- 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 + ... + 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 + ... + 1 = N -1
How long does insertion sort take?
- Depends on number of elements N to sort.
- Depends on specific input.
- Depends on how long compare and exchange operation takes.

Average case.
- Elements are randomly ordered.
  - \( i \)th iteration requires \( i/2 \) comparison on average
  - total = \( 0 + 1/2 + 2/2 + \ldots + (N-1)/2 = N(N-1)/4 \)
  - check with profile: 249,750 vs. 256,313

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

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

Insertion sort is quadratic. On arizona: 1 second for \( N = 10,000 \).
- How long for \( N = 100,000? \) 100 seconds (100 times as long).
- \( N = 1 \) million? 2.78 hours (another factor of 100).
- \( N = 1 \) billion? 317 years (another factor of 10⁶).
Sorting Case Study: mergesort

Insertion sort (brute-force)
Mergesort (divide-and-conquer)
  - Divide array into two halves.
  - Sort each half separately.
  - Merge two halves to make sorted whole.

How do we merge efficiently?

Profiling Mergesort Analytically

How long does mergesort take?
- Bottleneck = merging (and copying).
  - merging two files of size N/2 requires N comparisons
- T(N) = comparisons to mergesort N elements.

How much space?
- Can’t do "in-place" like insertion sort.
- Need auxiliary array of size N.
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 \( m \)

Sort each "half" recursively.

```
void quicksort(char a[], int left, int right) {
    int m;
    if (right > left) {
        m = partition(a, left, right);
        quicksort(a, left, m - 1);
        quicksort(a, m + 1, right);
    }
}
```
Profiling Quicksort Analytically

Intuition.
- Assume all elements unique.
- Assume we always select median as partition element.
- $T(N) = \text{# comparisons.}$

\[
T(N) = \begin{cases} 
0 & \text{if } N = 1 \\
\frac{2T(N/2)}{\text{sorting both halves}} + \frac{N}{\text{partitioning}} & \text{otherwise}
\end{cases}
\]

If $N$ is a power of 2.
$\Rightarrow T(N) = N \log_2 N$

Analysis "almost true" if you partition on random element.

Can you find median in $O(N)$ time?
$\checkmark$ Yes, see COS 226/423.


Profiling Quicksort Analytically

Partition on median element.
$\checkmark$ Proportional to $N \log_2 N$ in best and worst case.

Partition on rightmost element.
$\checkmark$ Proportional to $N^2$ in worst case.
$\checkmark$ Already sorted file: takes $N^2/2 + N/2$ comparisons.

Partition on random element.
$\checkmark$ Roughly $2N \log_2 N$ steps.
$\checkmark$ Choose random partition element.

Check profile.
- $2N \log_2 N$: 13815 vs. 12372 (5708 + 6664).
- Running time for $N = 100,000$ about 1.2 seconds.
- How long for $N = 1$ million?
  - slightly more than 10 times (about 12 seconds)

Sorting Analysis Summary

Running time estimates:
- Home pc executes $10^8$ comparisons/second.
- Supercomputer executes $10^{12}$ comparisons/second.

<table>
<thead>
<tr>
<th>Computer</th>
<th>Insertion Sort ($N^2$)</th>
<th>Quicksort ($N \log N$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>thousand</td>
<td>million</td>
</tr>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
</tr>
</tbody>
</table>

Lesson: good algorithms are more powerful than supercomputers.

Design, Analysis, and Implementation of Algorithms

Algorithm.
- "Step-by-step recipe" used to solve a problem.
- Generally independent of programming language or machine on which it is to be executed.

Design.
- Find a method to solve the problem.

Analysis.
- Evaluate its effectiveness and predict theoretical performance.

Implementation.
- Write actual code and test your theory.
Comparison of Different Sorting Algorithms

<table>
<thead>
<tr>
<th>Attribute</th>
<th>insertion</th>
<th>quicksort</th>
<th>mergesort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst case complexity</td>
<td>$N^2$</td>
<td>$N^2$</td>
<td>$N \log_2 N$</td>
</tr>
<tr>
<td>Best case complexity</td>
<td>$N$</td>
<td>$N \log_2 N$</td>
<td>$N \log_2 N$</td>
</tr>
<tr>
<td>Average case complexity</td>
<td>$N^2$</td>
<td>$N \log_2 N$</td>
<td>$N \log_2 N$</td>
</tr>
<tr>
<td>Already sorted</td>
<td>$N$</td>
<td>$N^2$</td>
<td>$N \log_2 N$</td>
</tr>
<tr>
<td>Reverse sorted</td>
<td>$N^2$</td>
<td>$N^2$</td>
<td>$N \log_2 N$</td>
</tr>
<tr>
<td>Space</td>
<td>$N$</td>
<td>$N$</td>
<td>$2N$</td>
</tr>
<tr>
<td>Stable</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Sorting algorithms have different performance characteristics.
- Other choices: BST sort, bubblesort, heapsort, shellsort, selection sort, shaker sort, radix sort, distribution sort, solitaire sort, hybrid methods.
- Which one should I use?
  > Depends on application.

Computational Complexity

<table>
<thead>
<tr>
<th>Attribute</th>
<th>insertion</th>
<th>quicksort</th>
<th>mergesort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst case complexity</td>
<td>$N^2$</td>
<td>$N^2$</td>
<td>$N \log_2 N$</td>
</tr>
<tr>
<td>Best case complexity</td>
<td>$N$</td>
<td>$N \log_2 N$</td>
<td>$N \log_2 N$</td>
</tr>
<tr>
<td>Average case complexity</td>
<td>$N^2$</td>
<td>$N \log_2 N$</td>
<td>$N \log_2 N$</td>
</tr>
<tr>
<td>Already sorted</td>
<td>$N$</td>
<td>$N^2$</td>
<td>$N \log_2 N$</td>
</tr>
<tr>
<td>Reverse sorted</td>
<td>$N^2$</td>
<td>$N^2$</td>
<td>$N \log_2 N$</td>
</tr>
<tr>
<td>Space</td>
<td>$N$</td>
<td>$N$</td>
<td>$2N$</td>
</tr>
<tr>
<td>Stable</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Framework to study efficiency of algorithms.
- Depends on machine model, average case, worst case.
- UPPER BOUND = algorithm to solve the problem.
- LOWER BOUND = proof that no algorithm can do better.
- OPTIMAL ALGORITHM: lower bound = upper bound.

Example: sorting.
- Measure costs in terms of comparisons.
- Upper bound = $N \log_2 N$ (mergesort).
  - quicksort usually faster, but mergesort never slow
- Lower bound = $N \log_2 N - N \log_2 e$
  (applies to any comparison-based algorithm).
  - Why?

Computational Complexity

Caveats.
- Worst or average case may be unrealistic.
- Costs ignored in analysis may dominate.
- Machine model may be restrictive.

Complexity studies provide:
- Starting point for practical implementations.
- Indication of approaches to be avoided.

Summary

How can I evaluate the performance of a proposed algorithm?
- Computational experiments.
- Complexity theory.

What if it’s not fast enough?
- Use a faster computer.
  - performance improves incrementally
- Understand why.
- Discover a better algorithm.
  - performance can improve dramatically
  - not always easy / possible to develop better algorithm