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>
<tr>
<td>+</td>
<td>4</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

$N = 2$

<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>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

$N = 4$

- How many reads? How many writes? How many operations?
Quadratic Growth

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

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

\[
\begin{array}{c|c|c|c}
N = 2 & 7 & 8 & \times 4 \\
& * & 2 & 2 \\
1 & 5 & 6 & \hline
3 & 1 & 2 & 0 \\
\hline
3 & 2 & 7 & 6
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c}
N = 4 & 4 & 2 & 7 & 8 & \times 6 & 8 & 4 & 2 \\
& & & & & 8 & 5 & 5 & 6 \\
1 & 7 & 1 & 1 & 2 & 0 & \hline
3 & 4 & 2 & 2 & 4 & 0 & 0 & 0 \\
2 & 5 & 6 & 6 & 8 & 0 & 0 & 0 \\
\hline
2 & 9 & 2 & 7 & 0 & 0 & 7 & 6
\end{array}
\]
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><strong>Time to solve a problem of size</strong></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>
<tr>
<td><strong>Max size problem solved in one</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>second</td>
<td>920</td>
<td>10,000</td>
<td>1 million</td>
<td>21 million</td>
</tr>
<tr>
<td>minute</td>
<td>3,600</td>
<td>77,000</td>
<td>49 million</td>
<td>1.3 billion</td>
</tr>
<tr>
<td>hour</td>
<td>14,000</td>
<td>600,000</td>
<td>2.4 trillion</td>
<td>76 trillion</td>
</tr>
<tr>
<td>day</td>
<td>41,000</td>
<td>2.9 million</td>
<td>50 trillion</td>
<td>1,800 trillion</td>
</tr>
<tr>
<td><strong>N multiplied by 10, time multiplied by</strong></td>
<td>1,000</td>
<td>100</td>
<td>10+</td>
<td>10</td>
</tr>
</tbody>
</table>
## Orders of Magnitude

<table>
<thead>
<tr>
<th>Seconds</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 second</td>
</tr>
<tr>
<td>10</td>
<td>10 seconds</td>
</tr>
<tr>
<td>$10^2$</td>
<td>1.7 minutes</td>
</tr>
<tr>
<td>$10^3$</td>
<td>17 minutes</td>
</tr>
<tr>
<td>$10^4$</td>
<td>2.8 hours</td>
</tr>
<tr>
<td>$10^5$</td>
<td>1.1 days</td>
</tr>
<tr>
<td>$10^6$</td>
<td>1.6 weeks</td>
</tr>
<tr>
<td>$10^7$</td>
<td>3.8 months</td>
</tr>
<tr>
<td>$10^8$</td>
<td>3.1 years</td>
</tr>
<tr>
<td>$10^9$</td>
<td>3.1 decades</td>
</tr>
<tr>
<td>$10^{10}$</td>
<td>3.1 centuries</td>
</tr>
<tr>
<td>...</td>
<td>forever</td>
</tr>
<tr>
<td>$10^{21}$</td>
<td>age of universe</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Meters Per Second</th>
<th>Imperial Units</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-10}$</td>
<td>1.2 in / decade</td>
<td>Continental drift</td>
</tr>
<tr>
<td>$10^{-8}$</td>
<td>1 ft / year</td>
<td>Hair growing</td>
</tr>
<tr>
<td>$10^{-6}$</td>
<td>3.4 in / day</td>
<td>Glacier</td>
</tr>
<tr>
<td>$10^{-4}$</td>
<td>1.2 ft / hour</td>
<td>Gastro-intestinal tract</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>2 ft / minute</td>
<td>Ant</td>
</tr>
<tr>
<td>1</td>
<td>2.2 mi / hour</td>
<td>Human walk</td>
</tr>
<tr>
<td>$10^2$</td>
<td>220 mi / hour</td>
<td>Propeller airplane</td>
</tr>
<tr>
<td>$10^4$</td>
<td>370 mi / min</td>
<td>Space shuttle</td>
</tr>
<tr>
<td>$10^6$</td>
<td>620 mi / sec</td>
<td>Earth in galactic orbit</td>
</tr>
<tr>
<td>$10^8$</td>
<td>62,000 mi / sec</td>
<td>1/3 speed of light</td>
</tr>
</tbody>
</table>

### Powers of 2

<table>
<thead>
<tr>
<th>$2^{10}$</th>
<th>thousand</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{20}$</td>
<td>million</td>
</tr>
<tr>
<td>$2^{30}$</td>
<td>billion</td>
</tr>
</tbody>
</table>
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.
- Appel (1981). \( N \log N \) steps, enables new research.

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: \( N^2 \) 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 = $0 + 1 + 2 + \ldots + N-1 = N \frac{(N-1)}{2}$

![Diagram of insertion sort process with elements E, F, G, H, I, J, D, C, B, A in different states: unsorted, active, sorted]
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.

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

\[\begin{array}{cccccccc}
A & B & C & D & E & F & G & H & I & J \\
\text{unsorted} & & & & & & \text{active} & & & \text{sorted}
\end{array}\]
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.

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
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: $N (N - 1) / 2$.

Best case: $N - 1$.

Average case: $N (N - 1) / 4$. 
Estimating the Running Time

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 \(10^6\)).
Sorting Case Study: mergesort

Insertion sort (brute-force)
Mergesort (divide-and-conquer)
  - Divide array into two halves.
  - Sort each half separately. How do we sort half size files?
    - Any sorting algorithm will do.
    - Use mergesort recursively!

```
MERGESORT

MERGESORT

EERMRS

EMORT
```

divide

sort
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?

```
MERGESORT
MERGESORT
MERGESORT
MERGESORT
MERGESORT
MERGESORT
MERGESORT
MERGESORT
MERGESORT
MERGESORT
MERGESORT
MERGESORT
MERGESORT
MERGESORT
MERGESORT
MERGESORT

divide
sort
merge
```
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.

\[
T(N) = \begin{cases} 
0 & \text{if } N = 1 \\
2T(N/2) + N & \text{otherwise}
\end{cases}
\]
Profiling Mergesort Analytically

\[ T(N) = \begin{cases} 
0 & \text{if } N = 1 \\
2T\left(\frac{N}{2}\right) + \frac{N}{\text{merging}} & \text{otherwise} 
\end{cases} \]
Profiling Mergesort Analytically

How long does mergesort take?
- Bottleneck = merging (and copying).
  - merging two files of size $N/2$ requires $N$ comparisons
- $N \log_2 N$ comparisons to sort ANY array of $N$ elements.
  - even already sorted array!

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 \( m \)
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 \( m \)
- Sort each "half" recursively.
Sorting Case Study: quicksort

Insertion sort (brute-force)
Mergesort (divide-and-conquer)
Quicksort (conquer-and-divide)

- 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.

```c
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);
    }
}
```
Sorting Case Study: quicksort

Insertion sort (brute-force)
Mergesort (divide-and-conquer)
Quicksort (conquer-and-divide)

- 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.
- How do we partition efficiently?
  - \( N - 1 \) comparisons
  - easy with auxiliary array
  - better solution: use no extra space!
Profiling Quicksort Analytically

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

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

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?
- Yes, see COS 226/423.

Profiling Quicksort Analytically

Partition on median element.

- Proportional to $N \log_2 N$ in best and worst case.

Partition on rightmost element.

- Proportional to $N^2$ in worst case.
- Already sorted file: takes $N^2/2 + N/2$ comparisons.

Partition on random element.

- Roughly $2 \, N \log_e N$ steps.
- Choose random partition element.

Check profile.

- $2 \, N \log_e N$: 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>thousand</th>
<th>million</th>
<th>billion</th>
</tr>
</thead>
<tbody>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
<td>317 years</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
<td>1.6 weeks</td>
</tr>
</tbody>
</table>

Insertion Sort ($N^2$)

<table>
<thead>
<tr>
<th>computer</th>
<th>thousand</th>
<th>million</th>
<th>billion</th>
</tr>
</thead>
<tbody>
<tr>
<td>home</td>
<td>instant</td>
<td>0.3 sec</td>
<td>6 min</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>instant</td>
<td>instant</td>
</tr>
</tbody>
</table>

QuickSort ($N \log N$)

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.
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

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