Space and Time Complexity

- For a given problem, may be a variety of algorithms that you could implement (and a variety of data structures you could use).
- For each algorithm, want to consider the time complexity (and occasionally, the space (memory) complexity)
Exact Time Complexity Analysis is Hard

- It is difficult to work with *exactly* because it is typically very complicated.
- It is cleaner and easier to talk about *upper and lower bounds* of the function.
- Remember that we ignore constants.
  - This makes sense since running our algorithm on a machine that is twice as fast will affect the running time by a multiplicative constant of 2, we are going to have to ignore constant factors anyway.
- Asymptotic notation (O, Θ, Ω) are the best that we can practically do to deal with the complexity of functions.

Asymptotic Analysis of Algorithms

- **Best case**: (Ω) is the function defined by the minimum number of steps taken on any instance of size n.
  - Too easy to cheat with best case.
  - We do not rely it on much.
- **Average case**: (θ) is the function defined by an average number of steps taken on any instance of size n.
  - Usually *very hard* to compute the average running time.
  - Very time consuming.
- **Worst case**: (O) is the function defined by the maximum number of steps taken on any instance of size n.
  - Fairly easy to analyze.
  - Often close to the average running time.
Bounding Functions (Informally)

- $f(n) = \mathcal{O}(g(n))$ means $c \times g(n)$ is an upper bound on $f(n)$.
- $f(n) = \Omega(g(n))$ means $c \times g(n)$ is a lower bound on $f(n)$.
- $f(n) = \Theta(g(n))$ means $c_1 \times g(n)$ is an upper bound on $g(n)$ and $c_2 \times g(n)$ is a lower bound on $f(n)$.

$c$, $c_1$, and $c_2$ are all constants independent of $n$.

Examples of $\mathcal{O}$, $\Omega$, and $\Theta$
Formal Definitions – Big Oh

- $f(n) = O(g(n))$ if there are positive constants $c$ and $n_0$ such that to the right of $n_0$, the value of $f(n)$ always lies on or below $c \cdot g(n)$.

- Think of the equality (=) as meaning in the set of functions.

- Examples:
  - $f(n) = 4n + 3$, $f(n) = O( )$
  - $f(n) = 3n^2 - 100n + 6 = O( )$
  - $f(n) = 3n^2 - 100n + 6 = O(n)$ ??

Using Calculus: $f(n) = O(g(n))$ iff $0 \leq \lim_{n \to \infty} \left| \frac{f(x)}{g(x)} \right| < \infty$

Formal Definitions – Big Omega

- $f(n) = \Omega(g(n))$ if there are positive constants $c$ and $n_0$ such that to the right of $n_0$, the value of $f(n)$ always lies on or above $c \cdot g(n)$.

- Examples
  - $f(n) = 4n + 3$, $f(n) = \Omega( )$
  - $f(n) = 4n + 3$, $f(n) = \Omega(n^2)$ ?

Using Calculus: $f(n) = \Omega(g(n))$ iff $0 < \lim_{n \to \infty} \left| \frac{f(x)}{g(x)} \right| \leq \infty$
Formal Definitions – Big Theta (Avg.)

\( f(n) = \theta(g(n)) \) if there are positive constants \( c_1, c_2 \) and \( n_0 \) such that to the right of \( n_0 \), the value of \( f(n) \) always lies on or below \( c_1 \cdot g(n) \) and on or above \( c_2 \cdot g(n) \)

- Examples
  \( f(n) = 4n + 3, \quad f(n) = \theta(\quad) \)
  \( f(n) = 4n + 3, \quad f(n) = \theta(n^2) \)

- Using Calculus: \( f(n) = \theta(g(n)) \) iff \( 0 < \lim_{x \to \infty} \left| \frac{g(x)}{f(x)} \right| < \infty \)

Review

- Given unsorted array \( A \) of size \( n \), suppose we want to determine if a particular element is in the array.
- What are the associated time complexities of a linear search?

  Average: \( \theta(\quad) \)
  Best: \( \Omega(\quad) \)
  Worst: \( O(\quad) \)
Logarithms

- It is important to understand intuitively what logarithms are and where they come from.
- A logarithm is simply an inverse exponential function. Saying $b^x = y$ is equivalent to saying that $\log_b y = x$
i.e. $x$ is the power to which $b$ must be raised to give $y$

- Examples:

Logarithms

- Exponential functions, like the amount owed on a n year mortgage at an interest rate of c% per year, are functions which grow distressingly fast, as anyone who has tried to pay off a mortgage knows.
- Thus inverse exponential functions, i.e. logarithms, grow refreshingly slowly.
Examples of Logarithmic Functions

- Binary search is an example of an $O(\lg n)$ algorithm. After each comparison, we can throw away half the possible number of keys.
- Thus twenty comparisons suffice to find any name in the million-name Manhattan phone book!
- If you have an algorithm which runs in $O(\lg n)$ time, take it, because this is blindingly fast even on very large instances.

Properties of Logarithms

- $\log(x \cdot y) = \log(x) + \log(y)$
- $\log\left(\frac{x}{y}\right) = \log x - \log y$
- $\log(x^n) = n \log x$

- Logarithms are increasing functions, i.e. if $x < y$, where $x, y > 0$, then $\log(x) < \log(y)$

- All three are helpful when comparing runtime of algorithms (especially the third and fourth)
Asymptotic Analysis: Examples

• $2^n$ versus $n^2$

<table>
<thead>
<tr>
<th>n</th>
<th>$O(\lg n)$</th>
<th>$O(n)$</th>
<th>$O(n \lg n)$</th>
<th>$O(n^2)$</th>
<th>$O(2^n)$</th>
<th>$O(n!)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.003 µs</td>
<td>0.01 µs</td>
<td>0.033 µs</td>
<td>0.1 µs</td>
<td>1 µs</td>
<td>3.83 ms</td>
</tr>
<tr>
<td>20</td>
<td>0.004 µs</td>
<td>0.02 µs</td>
<td>0.086 µs</td>
<td>0.4 µs</td>
<td>1 ms</td>
<td>77.1 years</td>
</tr>
<tr>
<td>30</td>
<td>0.005 µs</td>
<td>0.03 µs</td>
<td>0.147 µs</td>
<td>0.9 µs</td>
<td>1 sec</td>
<td>8.4*10^15 yrs</td>
</tr>
<tr>
<td>40</td>
<td>0.005 µs</td>
<td>0.04 µs</td>
<td>0.213 µs</td>
<td>1.6 µs</td>
<td>18.3 min</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>0.006 µs</td>
<td>0.05 µs</td>
<td>0.282 µs</td>
<td>2.5 µs</td>
<td>13 days</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0.007 µs</td>
<td>0.1 µs</td>
<td>0.644 µs</td>
<td>10 µs</td>
<td>4*10^15 yrs</td>
<td></td>
</tr>
<tr>
<td>1,000</td>
<td>0.010 µs</td>
<td>1 µs</td>
<td>9.966 µs</td>
<td>1 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td>0.013 µs</td>
<td>10 µs</td>
<td>130 µs</td>
<td>100 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100,000</td>
<td>0.017 µs</td>
<td>0.1 ms</td>
<td>1.67 ms</td>
<td>10 sec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000,000</td>
<td>0.020 µs</td>
<td>1 ms</td>
<td>19.93 ms</td>
<td>16.7 min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10,000,000</td>
<td>0.023 µs</td>
<td>0.01 sec</td>
<td>0.23 sec</td>
<td>1.16 days</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100,000,000</td>
<td>0.027 µs</td>
<td>0.1 sec</td>
<td>2.66 sec</td>
<td>115.7 days</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>0.030 µs</td>
<td>0.09 sec</td>
<td>29.90 sec</td>
<td>3.7 years</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Sorting

- **Input:** A sequence of \( n \) numbers \(<a_1, a_2, ..., a_n>\)
- **Output:** A permutation (reordering) \(<a'_1, a'_2, ..., a'_n>\) of the input sequence such that \( a'_1 \leq a'_2 \leq ... \leq a'_n \)

- We seek algorithms that are *correct* and *efficient*
Insertion Sort

\[
\text{INSERTION-SORT}(A, n) \quad // \quad A[1..n] \\
1 \quad \text{for} \quad j = 2 \quad \text{to} \quad A.\text{length} \\
2 \quad \quad key \leftarrow A[j] \\
3 \quad \quad // \text{Insert } A[j] \text{ into } A[1..j-1] \\
4 \quad \quad i \leftarrow j - 1 \\
5 \quad \quad \textbf{while} \ (i > 0 \text{ and } A[i] > key) \\
6 \quad \quad \quad \textbf{do} \ A[i+1] \leftarrow A[i] \\
7 \quad \quad \quad \quad i \leftarrow i - 1 \\
8 \quad A[i+1] \leftarrow key
\]
Example

• How would insertion sort work on the following numbers?

3 1 7 4 8 2 6

Your Turn

• Problem: How would insertion sort work on the following characters to sort them alphabetically (from A to Z)? Show each step.

S O R T E D
Insertion Sort

- Is the algorithm correct?
- How efficient is the algorithm?
  - Time Complexity?
  - Space Complexity?
- How does insertion sort do on sorted permutations?
- How about unsorted permutations?

Analysis of Insertion Sort

- Best Case
Analysis of Insertion Sort

- Worst Case

For next time: