CS3000: Algorithms & Data
Jonathan Ullman

Lecture 3:
• Divide and Conquer: Mergesort
• Asymptotic Analysis

Sep 14, 2018
Asymptotic Analysis
Asymptotic Order Of Growth

• Predicting the wall-clock time of an algorithm is nigh impossible.
  • What machine will actually run the algorithm?
  • Impossible to exactly count “operations”?
Asymptotic Order Of Growth

- Do we really need to worry about this problem?
  - Mostly we want to compare algorithms, so we can select the right one for the job
  - Mostly we don’t care about small inputs, we care about how the algorithm will scale

![Graph showing the functions $y = n^2$ and $y = 10n + 50$]
Asymptotic Order Of Growth

**Asymptotic Analysis:** How does the running time grow as the size of the input grows?

\[ T(n) = \text{exact running time} \]

\[ g(n) = \text{nice function} \]
Asymptotic Order Of Growth

• **“Big-Oh” Notation**: \( f(n) = O(g(n)) \) if there exists \( c \in (0, \infty) \) and \( n_0 \in \mathbb{N} \) such that \( f(n) \leq c \cdot g(n) \) for every \( n \geq n_0 \).

• Asymptotic version of \( f(n) \leq g(n) \)

• Roughly equivalent to \( \lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty \)

---

\[ f(n) = 2n + 10 \]
\[ g(n) = n \]
\[ n_0 = 10 \]
\[ c = 3 \]
\[ 3n = c \cdot g(n) \]
\[ f(n) = 2n + 10 \]
Ask the Audience

• “Big-Oh” Notation: \( f(n) = O(g(n)) \) if there exists \( c \in (0, \infty) \) and \( n_0 \in \mathbb{N} \) such that \( f(n) \leq c \cdot g(n) \) for every \( n \geq n_0 \).

• Which of these statements are true?
  - \( 3n^2 + n = O(n^2) \) \( \checkmark \)
  - \( n^3 = O(n^2) \) \( \times \)
  - \( 10n^4 = O(n^5) \) \( \checkmark \)
  - \( \log_2 n = O(\log_{16} n) \) \( \checkmark \)

\[ \log_{16} n = \frac{\log_2 n}{4} \]

Also, \( 10n^4 = O(n^4) \)
\[ c_n: \quad 3n^2 + n = O(n^2) \quad \quad c = 4 \quad \quad n_0 = 1 \]

\[ \forall n \geq 1 \quad f(n) = 3n^2 + n \leq 4n^2 = c \cdot g(n) \]
Big-Oh Rules

• Constant factors can be ignored
  • \( \forall C > 0 \quad Cn = O(n) \)
  \( C \cdot g(n) = O(g(n)) \)

• Smaller exponents are Big-Oh of larger exponents
  • \( \forall a > b \quad n^b = O(n^a) \)

• Any logarithm is Big-Oh of any polynomial
  • \( \forall a, \epsilon > 0 \quad \log_2^a n = O(n^\epsilon) \)
  \( \log_2^{1000} n = O(n^{0.001}) \)

• Any polynomial is Big-Oh of any exponential
  • \( \forall a > 0, b > 1 \quad n^a = O(b^n) \)
  \( n^{1000} = O(1.0001^n) \)

• Lower order terms can be dropped
  • \( n^2 + n^{3/2} + n = O(n^2) \)
  \( f_1 = O(g) \quad f_2 = O(g) \quad \Rightarrow \quad f_1 + f_2 = O(g) \)
A Word of Caution

- The notation $f(n) = O(g(n))$ is weird—do not take it too literally

\[
n = O(n^2) \quad n = O(n^3)
\]

\[
n^3 \neq O(n^2)
\]

\[
n = \underbrace{1 + \ldots + 1}_{n \text{ times}} = \frac{O(1) + o(1) + \ldots + o(1)}{n \text{ times}}
\]

\[
= \underbrace{O(1) + \ldots + o(1)}_{n-1 \text{ times}}
\]

\[
= O(1)
\]
Asymptotic Order Of Growth

\( \frac{1}{3} n^2 = \Omega(n^2) \)

- **“Big-Omega” Notation:** \( f(n) = \Omega(g(n)) \) if there exists \( c \in (0, \infty) \) and \( n_0 \in \mathbb{N} \) s.t. \( f(n) \geq c \cdot g(n) \) for every \( n \geq n_0 \).
  - Asymptotic version of \( f(n) \geq g(n) \)
  - Roughly equivalent to \( \lim_{n \to \infty} \frac{f(n)}{g(n)} > 0 \)

- **“Big-Theta” Notation:** \( f(n) = \Theta(g(n)) \) if there exists \( c_1 \leq c_2 \in (0, \infty) \) and \( n_0 \in \mathbb{N} \) such that \( c_2 \cdot g(n) \geq f(n) \geq c_1 \cdot g(n) \) for every \( n \geq n_0 \).
  - Asymptotic version of \( f(n) = g(n) \)
  - Roughly equivalent to \( \lim_{n \to \infty} \frac{f(n)}{g(n)} \in (0, \infty) \)
Asymptotic Running Times

• **We usually write running time as a Big-Theta**
  - Exact time per operation doesn’t appear
  - Constant factors do not appear
  - Lower order terms do not appear

• **Examples:**
  - $30 \log_2 n + 45 = \Theta(\log n)$
  - $Cn \log_2 2n = \Theta(n \log n)$
  - $\sum_{i=1}^{n} i = \Theta(n^2)$
    
    $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} = \Theta(n^2)$

  - $n \log_2 2n = n \log_2 n + n = \Theta(n \log n)$
Asymptotic Order Of Growth

• **“Little-Oh” Notation:** $f(n) = o(g(n))$ if for every $c > 0$ there exists $n_0 \in \mathbb{N}$ s.t. $f(n) < c \cdot g(n)$ for every $n \geq n_0$.
  - Asymptotic version of $f(n) < g(n)$
  - Roughly equivalent to $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$

• **“Little-Omega” Notation:** $f(n) = \omega(g(n))$ if for every $c > 0$ there exists $n_0 \in \mathbb{N}$ such that $f(n) > c \cdot g(n)$ for every $n \geq n_0$.
  - Asymptotic version of $f(n) > g(n)$
  - Roughly equivalent to $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$

$\mathcal{O}^2 = o(n^3)$

$n^3 = \omega(n^2)$
Ask the Audience!

$$\log_2 n = O(n)$$
$$n \log_2 n = O(n^2)$$

- Rank the following functions in increasing order of growth (i.e. $f_1, f_2, f_3, f_4$ so that $f_i = O(f_{i+1})$)
  - $n \log_2 n$
  - $n^2$
  - $100n$
  - $3 \log_2 n$

$$3^{\log_2 n}$$
$$= (\log_3 3)(\log_2 n)$$
$$= 2 \cdot 1.59$$
$$= n \log_2 3 = n$$

$$100n, 3^\log_2 n, n \log_2 n, n^2$$
$$3^\log_2 n, n^2, 100n, n \log_2 n$$
$$100n, n \log_2 n, 3^\log_2 n, n^2$$

$$100n, n \log_2 n, 3^{\log_2 3} = n^{1.59}, n^2$$
Why Asymptotics Matter

<table>
<thead>
<tr>
<th></th>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$1.5^n$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 10$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>4 sec</td>
</tr>
<tr>
<td>$n = 30$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>18 min</td>
<td>$10^{25}$ years</td>
</tr>
<tr>
<td>$n = 50$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>11 min</td>
<td>36 years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>12,892 years</td>
<td>$10^{17}$ years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 10,000$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100,000$</td>
<td>&lt; 1 sec</td>
<td>2 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000,000$</td>
<td>1 sec</td>
<td>20 sec</td>
<td>12 days</td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>

- Exponential time bad / Polynomial time good
- Exponents matter
Divide and Conquer Algorithms
Divide and Conquer Algorithms

• Split your problem into smaller subproblems
• Recursively solve each subproblem
• Combine the solutions to the subproblems
• For many problems, combining is easier than solving

Divide et impera!
-Philip II of Macedon
Divide and Conquer Algorithms

• **Examples:**
  - Mergesort: sorting a list
  - Binary Search: search in a sorted list
  - Karatsuba’s Algorithm: integer multiplication
  - Fast Fourier Transform
  - ...

• **Key Tools:**
  - Correctness: proof by induction
  - Running Time Analysis: recurrences
  - Asymptotic Analysis
Given a list of $n$ numbers, put them in ascending order.
A Simple Algorithm: Insertion Sort

Scan to find largest elt

\[
\begin{array}{ccccccccc}
11 & 3 & 42 & 28 & 17 & 8 & 2 & 15 \\
\end{array}
\]

repeat \(n-1\) times

Running Time: \(n + (n-1) + (n-2) + \ldots + 2\)

\[= \left(\frac{n(n+1)}{2}\right) - 1 = \Theta(n^2)\]
### A Simple Algorithm: Insertion Sort

**Find the maximum**

1. **Initial array:**
   
   | 11 | 3 | 42 | 28 | 17 | 8 | 2 | 15 |

2. **After finding the maximum:**
   
   | 11 | 3 | 15 | 28 | 17 | 8 | 2 | 42 |

3. **After swapping into place:**
   
   | 11 | 3 | 15 | 2 | 17 | 8 | 28 | 42 |

4. **Final array:**
   
   | 2 | 3 | 8 | 11 | 15 | 17 | 28 | 42 |

**Swap it into place, repeat on the rest**

**Repeat \( n - 1 \) times.**
# A Simple Algorithm: Insertion Sort

Find the maximum

| 11 | 3 | 42 | 28 | 17 | 8 | 2 | 15 |

Swap it into place, repeat on the rest

| 11 | 3 | 15 | 28 | 17 | 8 | 2 | 42 |

**Running Time:**
Divide and Conquer: Mergesort

Split

11  3  42  28  17  8  2  15

11  3  42  28

17  8  2  15

Recursively Sort

3  11  28  42

2  8  15  17

Recursively Sort

Merge

2  3  8  11  15  17  28  42
Divide and Conquer: Mergesort

- **Key Idea:** If \( L, R \) are sorted lists of length \( n \), then we can merge them into a sorted list \( A \) of length \( 2n \) in time \( \mathcal{O}(n) \)

- Merging two sorted lists is faster than sorting from scratch

\[
\begin{array}{c}
2n \text{ elements of } A \\
\times 2 \text{ ops per elem} \\
= \mathcal{O}(n)
\end{array}
\]

\[
\begin{array}{cccc}
3 & 11 & 28 & 42 \\
\downarrow & & & \downarrow \\
2 & 8 & 15 & 17 \\
\uparrow & & & \uparrow \\
2 & 3 & 8 & \\
\end{array}
\]

\( L \quad R \quad A \)
Merge(L,R):
    Let n ← len(L) + len(R)
    Let A be an array of length n
    j ← 1, k ← 1,

    For i = 1,...,2n:
        If (j > len(L)):
            // L is empty
            A[i] ← R[k], k ← k+1
        ElseIf (k > len(R)):
            // R is empty
            A[i] ← L[j], j ← j+1
        ElseIf (L[j] <= R[k]):
            // L is smallest
            A[i] ← L[j], j ← j+1
        Else:
            // R is smallest
            A[i] ← R[k], k ← k+1

    Return A
Merging

MergeSort(A):
If (len(A) = 1): Return A // Base Case
Let \( m \leftarrow \left\lceil \frac{\text{len}(A)}{2} \right\rceil \) // Split
Let \( L \leftarrow A[1:m], R \leftarrow A[m+1:n] \)
Let \( L \leftarrow \text{MergeSort}(L) \) // Recurse
Let \( R \leftarrow \text{MergeSort}(R) \)
Let \( A \leftarrow \text{Merge}(L,R) \) // Merge
Return A
Correctness of Mergesort

• **Claim:** The algorithm **Mergesort** is correct

\[ \forall n \in \mathbb{N} \quad \forall \text{ list } A \text{ of } n \text{ numbers} \]
\[ \text{Mergesort returns the list sorted.} \]

**Inductive Hypothesis:**

\[ H(n) : \quad \forall \text{ lists } A \text{ of } n \text{ numbers, Mergesort is correct.} \]

**Base Case:** \[ H(1) \] is true, obviously
Inductive Step:

We will show that $H(1)^{\cdot} H(2)^{\cdot} \cdots H(n) \Rightarrow H(n+1)$

1. Given any input $A$ of size $n+1$, $L$ and $R$ have size $\left\lceil \frac{n+1}{2} \right\rceil \leq n$ and $\left\lfloor \frac{n+1}{2} \right\rfloor \leq n$

2. By the IH, MergeSort sorts $L, R$ correctly.

3. Since $L, R$ are sorted, $\text{Merge}(L, R)$ will be sorted.

4. Therefore MergeSort returns $A$ in sorted order.

(Edited) Depends on the problem.
Running Time of Mergesort

\[ T(n) : \text{running time on inputs of length } n \]

\[ T(n) = 2 \times T\left(\frac{n}{2}\right) + C_n \]

\[ T(1) = C \]

\[ T(n) = C n \log_2 2n = \Theta(n \log n) \]

\[
\text{MergeSort}(A):
\]
\[
1. \text{If } (n = 1): \text{ Return } A
\]
\[
1. \text{Let } m \leftarrow \lceil n/2 \rceil
2. \text{Let } L \leftarrow A[1:m]
3. \text{Let } R \leftarrow A[m+1:n]
4. \text{Let } L \leftarrow \text{MergeSort}(L)
5. \text{Let } R \leftarrow \text{MergeSort}(R)
6. \text{Let } A \leftarrow \text{Merge}(L, R)
7. \text{Return } A
\]
Mergesort Summary

• Sort a list of \( n \) numbers in \( Cn \log_2 2n \) time
  • Can actually sort anything that allows comparisons
  • No comparison based algorithm can be (much) faster

• Divide-and-conquer
  • Break the list into two halves, sort each one and merge
  • Key Fact: Merging is easier than sorting

• Proof of correctness
  • Proof by induction

• Analysis of running time
  • Recurrences