CS3000: Algorithms & Data
Jonathan Ullman

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

Sep 14, 2018

Asymptotic Analysis

Asymptotic Order Of Growth

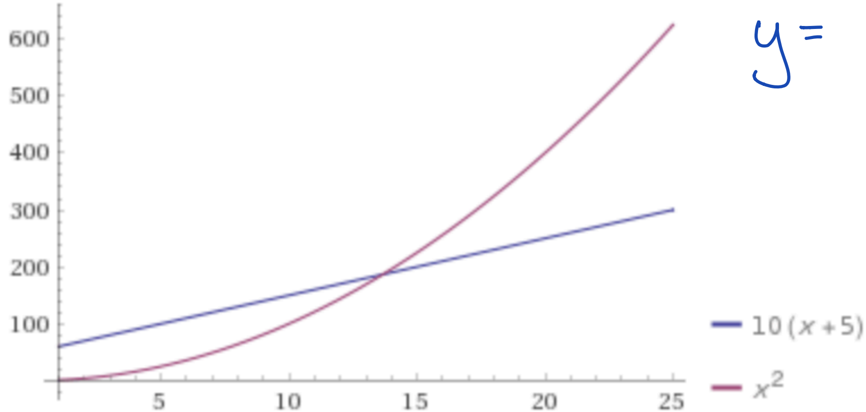
- Predicting the wall-clock time of an algorithm is nigh impossible.
  - What machine will actually run the algorithm?
  - Impossible to exactly count “operations”?

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

$y = n^2$

$y = 10n + 50$



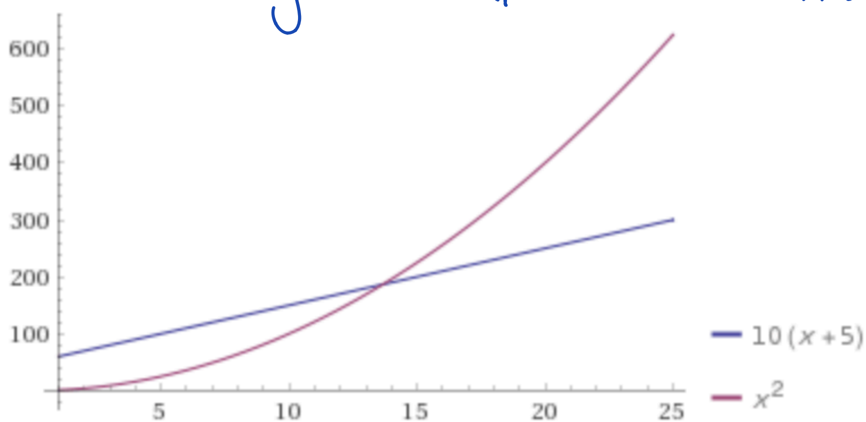
Asymptotic Order Of Growth

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

order of growth

$$f(n) \Longrightarrow g(n)$$

exact running time (messy dependent on the machine)



Asymptotic Order Of Growth

messy   nice function

- **"Big-Oh" Notation:** $f(n) = O\big(g(n)\big)$ 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 $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$

$2n = O(n)$

$f(n) = 3n^2 + n \qquad g(n) = n^2$

Clm: $f(n) = O(g(n))$

Pf: $c = 4 \qquad n_0 = 1$

$\forall\, n \geq n_0 \qquad 3n^2 + n \leq 4n^2$

$3n^2 + n \leq 3n^2 + n^2 \leq 4n^2 \leq 4n^2 \quad \square$

Ask the Audience

- **"Big-Oh" Notation:** $f(n) = O\big(g(n)\big)$ if there exists $c \in (0, \infty)$ and $n_0 \in \mathbb{N}$ such that $f(n) \le c \cdot g(n)$ for every $n \ge n_0$.

$$\lim_{n \to \infty} \frac{n^3}{n^2} = \infty$$

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

$c = 1 \quad n_0 = 10$

$\forall n \geq n_0 \quad 10n^4 \le n^5$

$$\log_{16} n = \frac{\log_2 n}{\log_2 16} = \frac{1}{4} \log_2 n$$

Big-Oh Rules

- **Constant factors can be ignored**
  - $\forall C > 0 \quad Cn = O(n)$

$f(n) = C \cdot g(n) \Rightarrow f(n) = O(g(n))$

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

$n^2 = O(n^{2.0001})$

- **Any logarithm is Big-Oh of any polynomial**
  - $\forall a, \varepsilon > 0 \quad \log_2^a n = O(n^\varepsilon)$

$\log_2^{1000} n = O(n^{.0001})$

- **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**
  - $n^2 + n^{3/2} + n = O(n^2)$

$f_1(n) + f_2(n)$ and $f_1(n) = O(g(n))$, $f_2(n) = O(g(n))$

$\Rightarrow f_1 + f_2 = O(g)$

A Word of Caution

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

$n = O(n^2)$ $\quad$ $n = O(n^3)$ $\quad$ (Not really an "=" sign)

Clm: $n = O(1)$

$$n = \sum_{i=1}^{n} 1 = \sum_{i=1}^{n} O(1)$$

$$= \sum_{i=2}^{n} O(1)$$

$$\vdots$$

$$= \sum_{i=n}^{n} O(1) = O(1)$$

Asymptotic Order Of Growth

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

- **"Big-Omega" Notation:** $f(n) = \Omega\big(g(n)\big)$ 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 $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$

$f(n) = O(g(n))$
$f(n) = \Omega(g(n))$

- **"Big-Theta" Notation:** $f(n) = \Theta\big(g(n)\big)$ 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 $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} \in (0, \infty)$

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

$= \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$

$Cn \log_2 n + Cn$

Asymptotic Order Of Growth

- **"Little-Oh" Notation:** $f(n) = o\big(g(n)\big)$ if for every $c > 0$ there exists $n_0 \in \mathbb{N}$ s.t. $f(n) < c \cdot g(n)$ for every $n \geq n_0$.

  $n^2 = o(n^3)$

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

  $n^3 = \omega(n^2)$

  - Asymptotic version of $f(n) > g(n)$
  - Roughly equivalent to $\displaystyle\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

Ask the Audience!

- 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
  - $100n$
  - $3^{\log_2 n}$

$100n, \quad n \log_2 n, \quad n^2, \quad 3^{\log_2 n}$

$3^{\log_2 n}, \quad 100n, \quad n \log_2 n, \quad n^2$

Correct Order: $100n, \quad n \log_2 n, \quad 3^{\log_2 n} \approx n^{1.59}, \quad n^2$

$100n$ vs. $n\log_2 n$

$100n = O(n\log_2 n)$ $\quad c = 100$

$n_0 = 2$

$100n \le 100n\log_2 n = O(n\log n)$

$n\log_2 n$ vs. n^2

$n \cdot \log_2 n$ vs. $n \cdot n$

$O(n) \cdot O(\log n)$ vs. $O(n) \cdot O(n)$

$2^{\log_2 n} = n$

$$3^{\log_2 n} = \left(2^{\log_2 3}\right)^{\log_2 n}$$

$$= \left(2^{\log_2 n}\right)^{\log_2 3}$$

$$= n^{\log_2 3} = n^{\approx 1.59}$$

$3^{\log_2 n} = O(n^2)$

$n\log_2 n = O(3^{\log_2 n})$

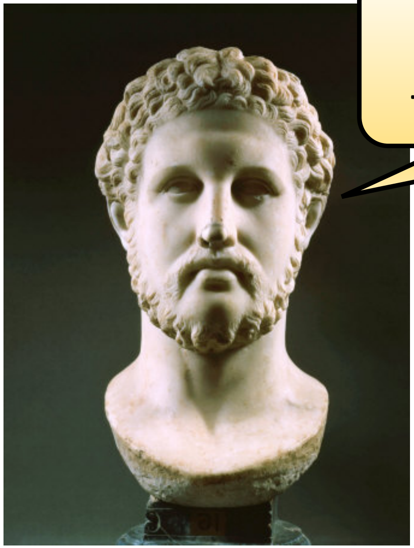
Why Asymptotics Matter

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

- polynomials good / exponentials bad
- logarithms good / polynomials bad
- different polynomials make a big difference

Divide and Conquer Algorithms

Divide and Conquer Algorithms



Divide et impera!
-Philip II of Macedon

- Split your problem into smaller subproblems
- Recursively solve each subproblem
- Combine the solutions to the subprobelms

Useful when combining solutions is easier than solving from scratch

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

Sorting

11	3	42	28	17	8	2	15

$A[1]$ $A[n]$

Given a list of n numbers,
put them in ascending order

any items that can be "compared"

2	3	8	11	15	17	28	42

A Simple Algorithm: Insertion Sort

Find the maximum

11	3	42	28	17	8	2	15

Put it at the end

11	3	15	28	17	8	2	42

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

11	3	15	2	17	8	28	42

Repeat $n - 1$ times.

2	3	8	11	15	17	28	42

A Simple Algorithm: Insertion Sort

Find the maximum

11	3	42	28	17	8	2	15

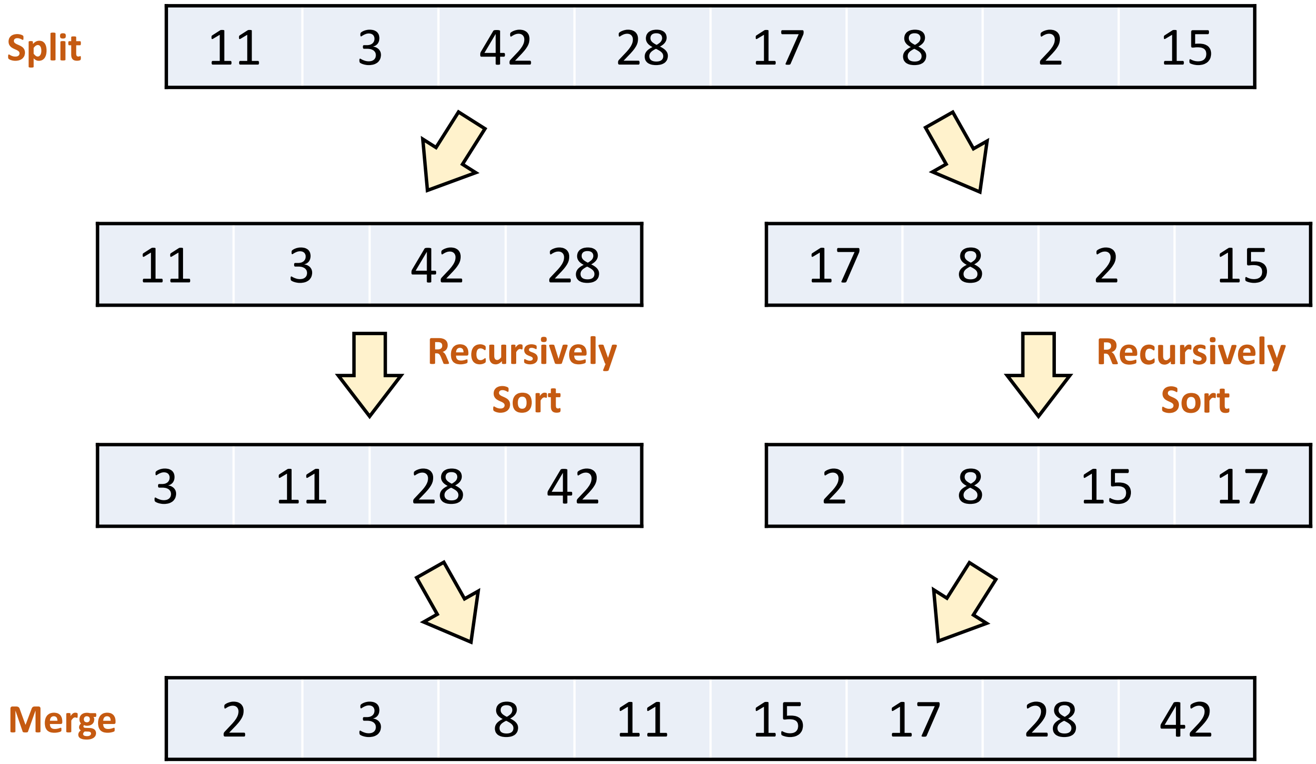
Swap it into place, repeat on the rest

11	3	15	28	17	8	2	42

Running Time: $\sum_{i=1}^{n-1} n-i+1$

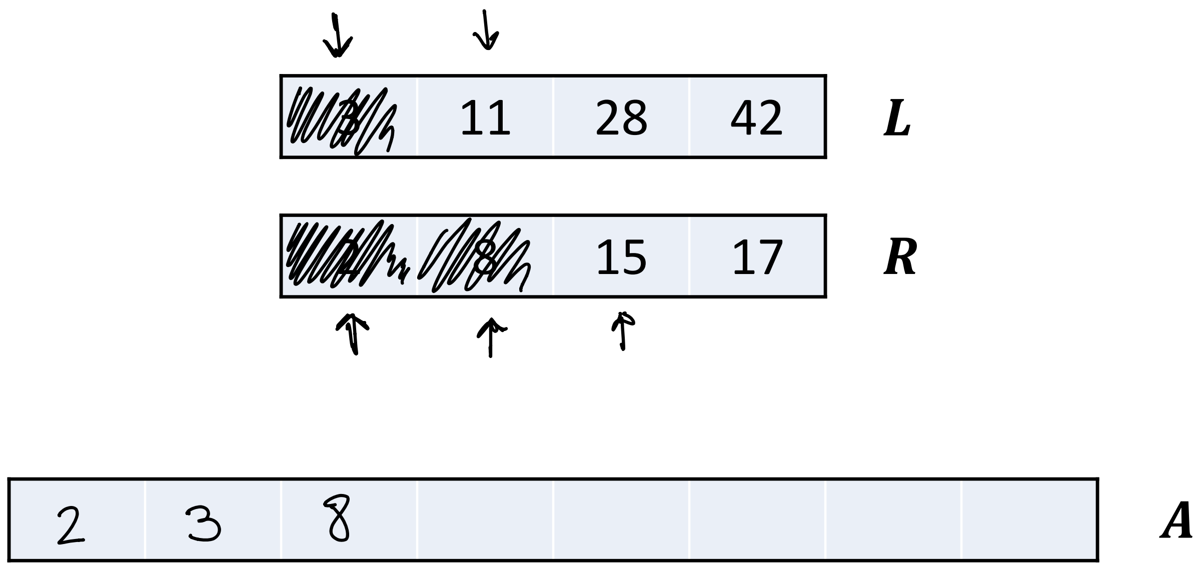
$$= \sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1 = \Theta(n^2)$$

Divide and Conquer: Mergesort



Divide and Conquer: Mergesort

- **Key Idea:** If $\boldsymbol{L}, \boldsymbol{R}$ are sorted lists of length n, then we can merge them into a sorted list $\boldsymbol{A}$ of length $2n$ in time $O(n)$
  - Merging two sorted lists is faster than sorting from scratch



Merging

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

  For i = 1,…,n:
    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 ← ⌈len(A)/2⌉            // Split
  Let L ← A[1:m], R ← A[m+1:n]

  Let L ← MergeSort(L)          // Recurse
  Let R ← MergeSort(R)

  Let A ← Merge(L,R)            // Merge

  Return A
```

Correctness of Mergesort

- **Claim:** The algorithm **`Mergesort`** is correct

$\forall n \in \mathbb{N}$ $\forall$ list A with n numbers Mergesort returns A in sorted order

Inductive Hypothesis: $H(n) = \forall$ A of size n MergeSort is correct

Base Case: $H(1)$ is true, obviously

Inductive Step: Assume $H(1), \ldots, H(n)$ are all true. We'll prove $H(n+1)$.

Correctness

~~Running Time~~ of Mergesort

Inductive Step :

Assume that MergeSort is correct for all A of size $\leq n$.

① $\lceil \frac{n+1}{2} \rceil, \lfloor \frac{n+1}{2} \rfloor \leq n$

② L, R are correctly sorted by MergeSort

③ L, R are sorted $\Rightarrow$ A is sorted

④ Mergesort is correct for lists of size n+1

```
MergeSort(A):
  If (n = 1): Return A

  Let m ← ⌈n/2⌉
  Let   L ← A[1:m]
        R ← A[m+1:n]

  Let L ← MergeSort(L)
  Let R ← MergeSort(R)
  Let A ← Merge(L,R)

  Return A
```

$$H(1) \wedge \ldots \wedge H(n)$$
$$\Downarrow$$
$$H(n+1)$$

Running Time of Mergesort

$T(n) =$ time to sort a list of size n

$$T(n) = 2 \times T\left(\frac{n}{2}\right) + cn$$

$$T(1) = c$$

$$T(n) = O(n \log n)$$

1

n

$2 \times T\left(\frac{n}{2}\right)$

cn

1

```
MergeSort(A):
  If (n = 1): Return A

  Let m ← ⌈n/2⌉
  Let   L ← A[1:m]
        R ← A[m+1:n]

  Let L ← MergeSort(L)
  Let R ← MergeSort(R)
  Let A ← Merge(L,R)

  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