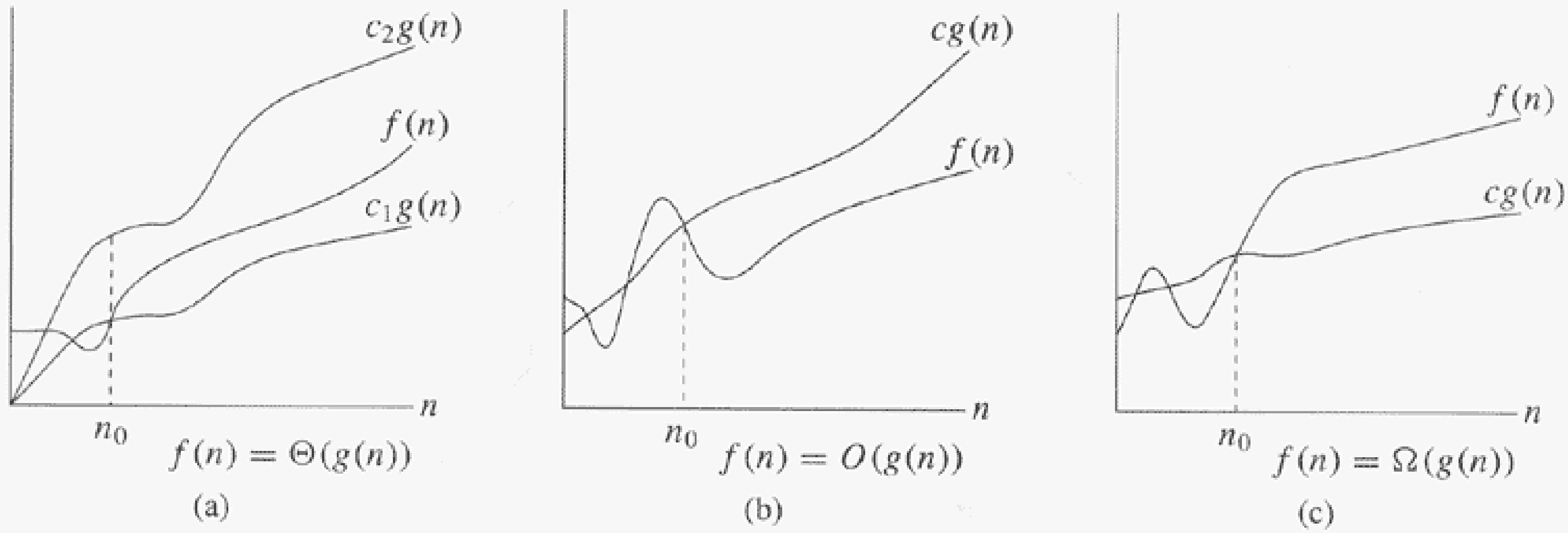


Figure 3.1 Graphic examples of the Θ, O, and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. **(a)** Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that to the right of n_0, the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive. **(b)** O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$. **(c)** Ω-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$.

Show: $f(n) = O(g(n))$ & $f(n) \neq O(g(n))$

- To show $f(n) = O(g(n))$
  - Manipulate $g(n)$ so that $f(n) <= c\, g(n)$ for some n_0.
  - Then try to show that $f(n) <= c\, g(n)$ for all $n >= n_0$.
- To show $f(n) \neq O(g(n))$
  - Assume an arbitrary choice for c.
  - Now show that no matter what n_0 is chosen, it is impossible for the following to become true as n tends to ∞: $f(n) <= cg(n)$.

Define $f(n) = \Theta(g(n))$ & $f(n) = \Omega(g(n))$

- $f(n) = \text{Omega}(g(n)) = \Omega(g(n))$
  - If and only if
    - $g(n) = O(f(n))$
- $f(n) = \text{Theta}(g(n)) = \Theta(g(n))$
  - If and only if
    - $f(n) = O(g(n))$, and
    - $g(n) = O(f(n))$

Logarithm manipulations

- Rules
  - b to the power $\log_b x$ equals x [log and exponentiation are inverse functions]
  - $\log_b b^x = x$ [log and exponentiation are inverse functions]
  - $\log(xy) = \log(x) + \log(y)$ [log of products is sum of logs]
  - $\log(x/y) = \log(x) - \log(y)$ [log of ratios is difference of logs]
  - $\log(x^y) = y \log(x)$ [
  - $a^{xy} = (a^x)^y$
  - $\log_x y = \log_b x / \log_b y$ [Change of bases is possible]

Solving Recurrences by Substitution

- Guess the form of the solution
- (Using mathematical induction) find the constants and show that the solution works

Example

$$T(n) = 2T(n/2) + n$$

Guess (#1)	$T(n) = O(n)$	
Need	**$T(n) <= cn$**	for some constant $c>0$
Assume	$T(n/2) <= cn/2$	Inductive hypothesis
Thus	$T(n) <= 2cn/2 + n = (c+1) n$	
	Our guess was wrong!!	

Solving Recurrences by Substitution: 2

$$T(n) = 2T(n/2) + n$$

Guess (#2)	$T(n) = O(n^2)$	
Need	$T(n) <= cn^2$	for some constant $c>0$
Assume	$T(n/2) <= cn^2/4$	Inductive hypothesis
Thus	$T(n) <= 2cn^2/4 + n = cn^2/2+ n$	
	Works for all n as long as c>=2 !!	
	But there is a lot of "slack"	

Solving Recurrences by Substitution: 3

$$T(n) = 2T(n/2) + n$$

Guess (#3) $T(n) = O(n\log n)$

Need $T(n) <= cn\log n$ for some constant $c>0$

Assume $T(n/2) <= c(n/2)(\log(n/2))$ Inductive hypothesis

Thus $T(n) <= 2\, c(n/2)(\log(n/2)) + n$

$<= cn\log n - cn + n <= cn\log n$

Works for all n as long as c>=1 !!

This is the correct guess. WHY?

Show $T(n) >= c'n\log n$ for some constant $c'>0$

QuickSort

$\textsc{QuickSort}(array\ A, int\ p, int\ r)$

if $(p < r)$
    then $q \leftarrow \textsc{Partition}(A, p, r)$
        $\textsc{QuickSort}(A, p, q-1)$
        $\textsc{QuickSort}(A, q+1, r)$

To sort array call $\textsc{QuickSort}(A, 1, length[A])$.

$\textsc{Partition}(array\ A, int\ p, int\ r)$

$x \leftarrow A[r]$ $\triangleright$ Choose **pivot**
$i \leftarrow p - 1$
for $j \leftarrow p$ **to** $r - 1$
    do if $(A[j] \leq x)$
        then $i \leftarrow i + 1$
            exchange $A[i] \leftrightarrow A[j]$
exchange $A[i+1] \leftrightarrow A[r]$
return $i + 1$

Page 146, CLRS

Analysis of QuickSort

- Average case
  - $T(n) <= 2T(n/2) + O(n)$
  - $T(n) = O(n \log n)$
- Worst case
  - $T(n) = T(n-1) + O(n)$
  - $T(n) = O(n^2)$

Variants of QuickSort

- Choice of Pivot
  - Random choice
  - Median of 3
- Avoiding recursion on small subarrays
  - Invoking InsertionSort for small arrays

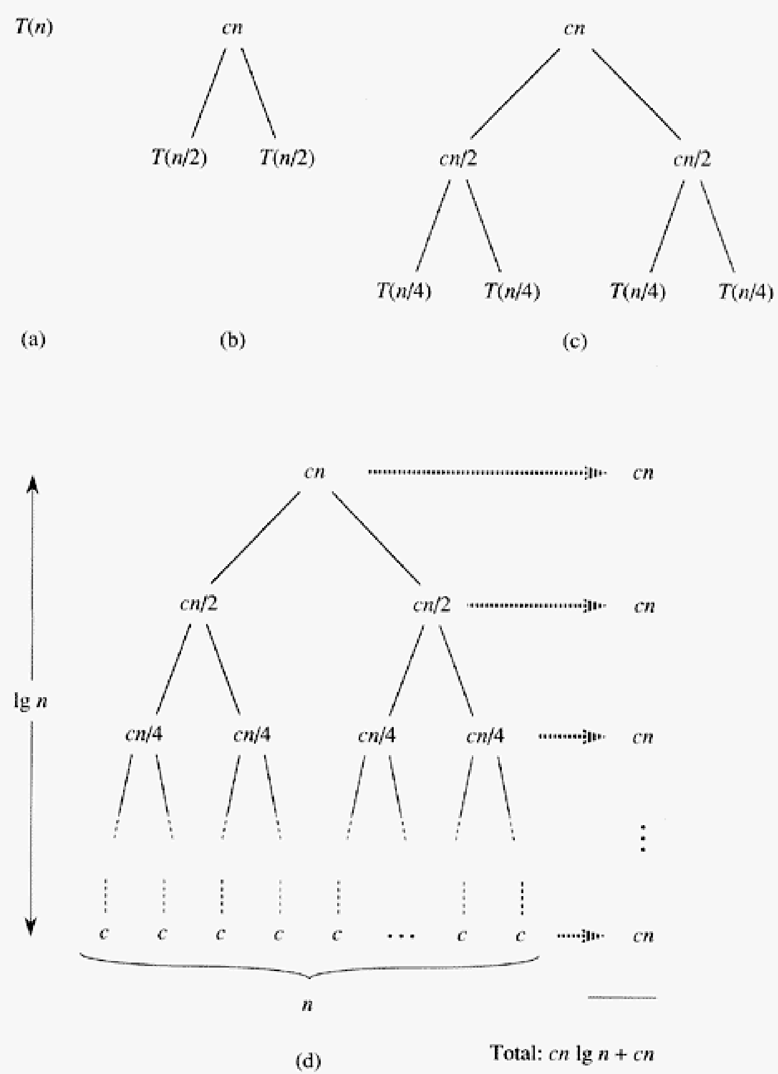


Figure 2.5 The construction of a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of cn. The total cost, therefore, is $cn \lg n + cn$, which is $\Theta(n \lg n)$.

Solving Recurrences: Recursion-tree method

- Substitution method fails when a good guess is not available
- Recursion-tree method works in those cases
  - Write down the recurrence as a tree with recursive calls as the children
  - Expand the children
  - Add up each level
  - Sum up the levels
- Useful for analyzing divide-and-conquer algorithms
- Also useful for generating good guesses to be used by substitution method

Solving Recurrences using Master Theorem

Master Theorem:

Let a,b >= 1 be constants, let f(n) be a function, and let

$$T(n) = aT(n/b) + f(n)$$

1. If f(n) = O($n^{\log_b a-e}$) for some constant e>0, then
   T(n) = Theta($n^{\log_b a}$)
2. If f(n) = Theta($n^{\log_b a}$), then
   T(n) = Theta($n^{\log_b a}$ log n)
3. If f(n) = Omega($n^{\log_b a+e}$) for some constant e>0, then
   T(n) = Theta(f(n))

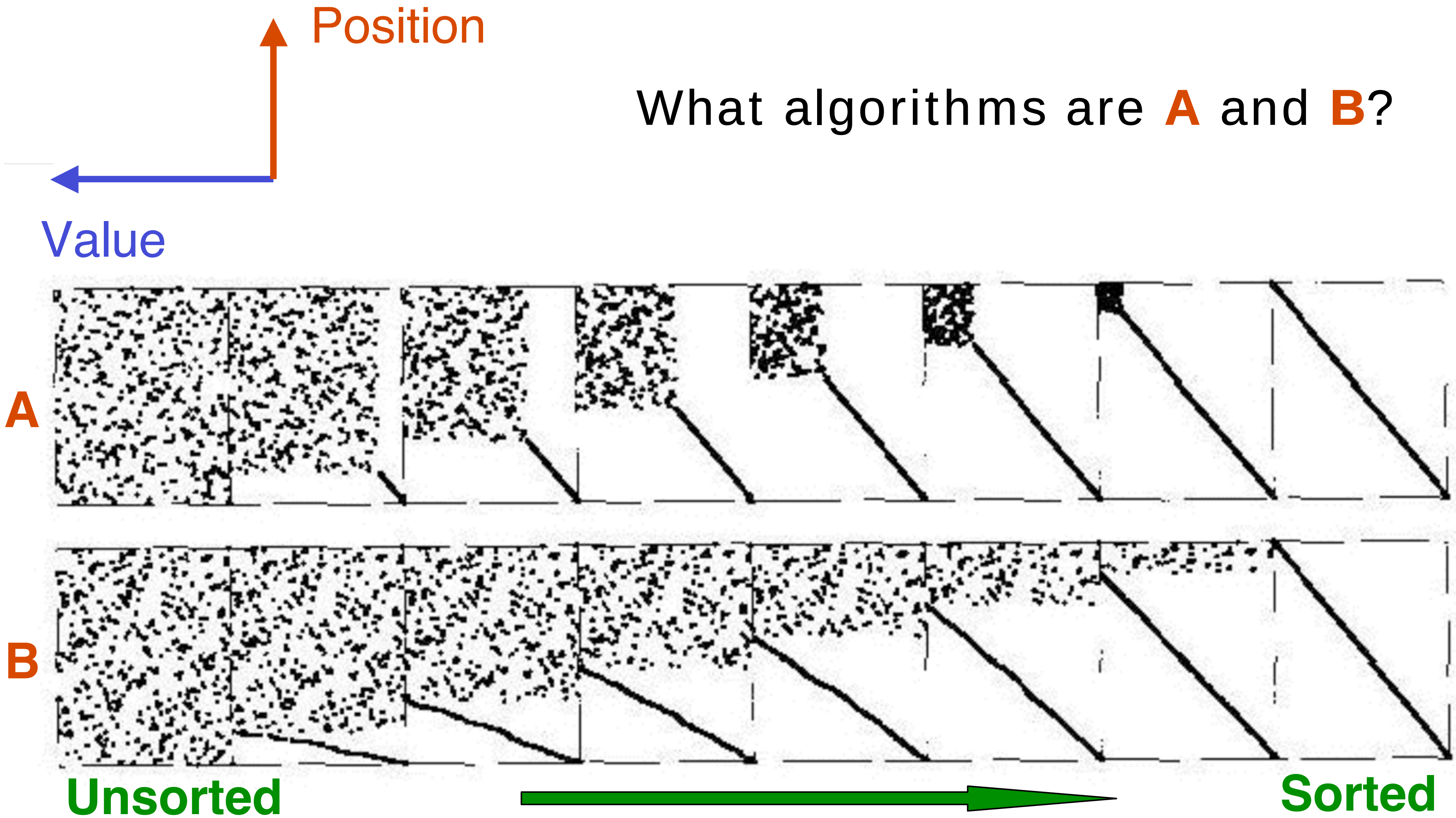
Solving Recurrence Relations

Page 62, [CLR]

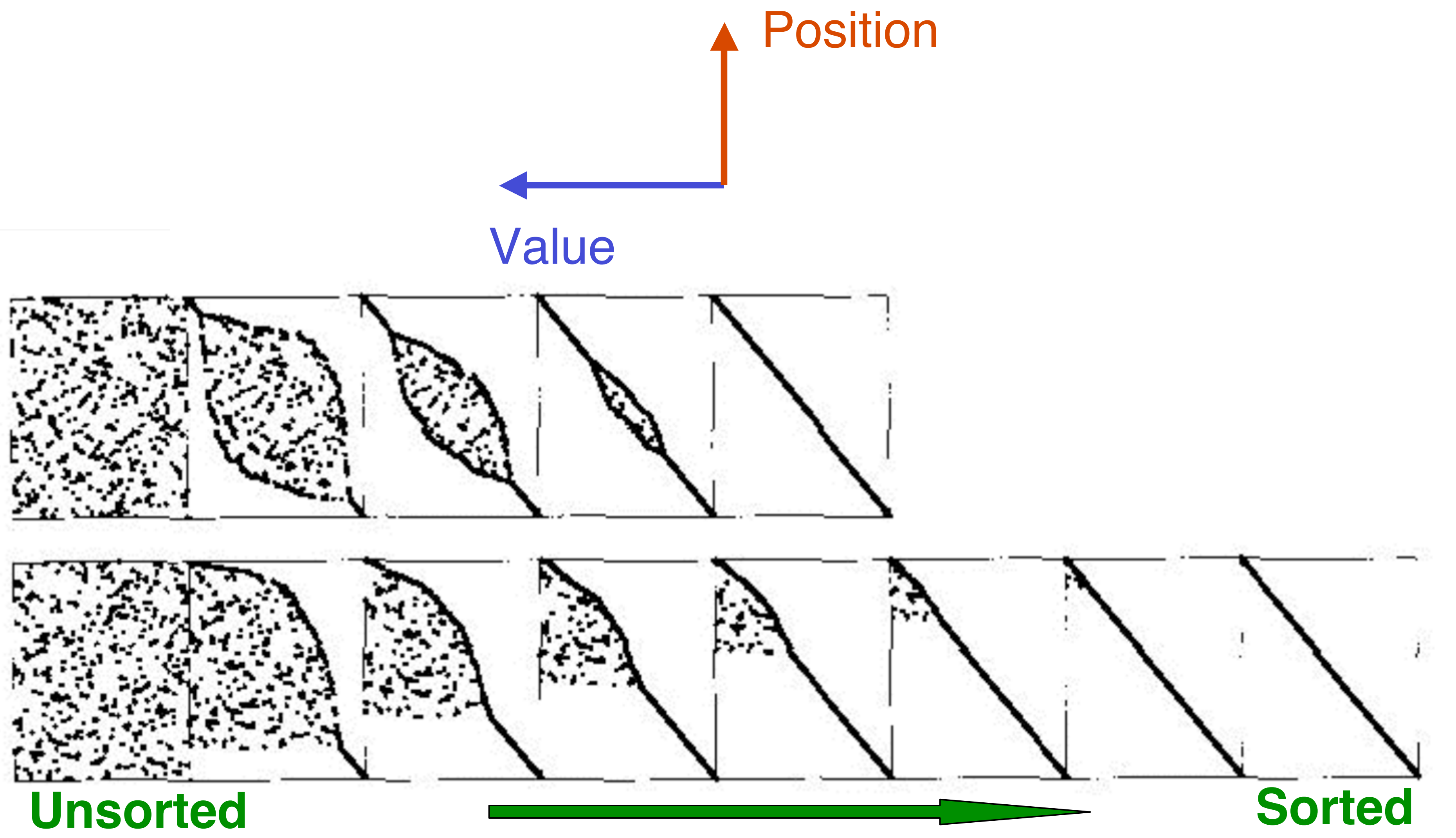
Recurrence; Cond	Solution
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$
$T(n) = T(n-c) + O(1)$	$T(n) = O(n)$
$T(n) = T(n-c) + O(n)$	$T(n) = O(n^2)$
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$
$T(n) = aT(n/b) + O(n)$; $a = b$	$T(n) = O(n \log n)$
$T(n) = aT(n/b) + O(n)$; $a < b$	$T(n) = O(n)$
$T(n) = aT(n/b) + f(n)$; $f(n) = O(n^{\log_b a - \epsilon})$	$T(n) = O(n)$
$T(n) = aT(n/b) + f(n)$; $f(n) = O(n^{\log_b a})$	$T(n) = \Theta(n^{\log_b a} \log n)$
$T(n) = aT(n/b) + f(n)$; $f(n) = \Theta(f(n))$ $af(n/b) \leq cf(n)$	$T(n) = \Omega(n^{\log_b a} \log n)$

Visualizing Algorithms 1

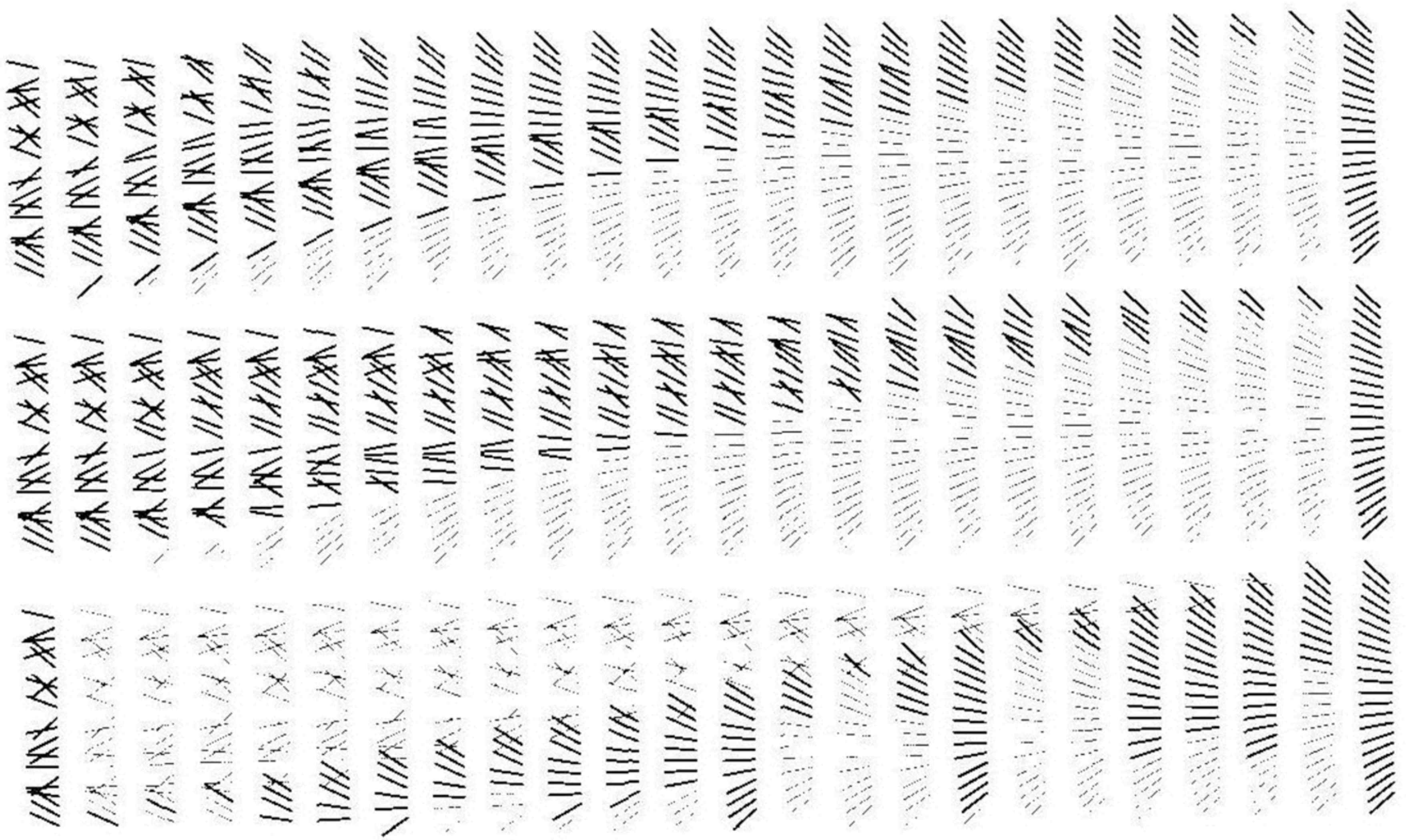
What algorithms are **A** and **B**?



Visualizing Algorithms 2



Visualizing Comparisons 3



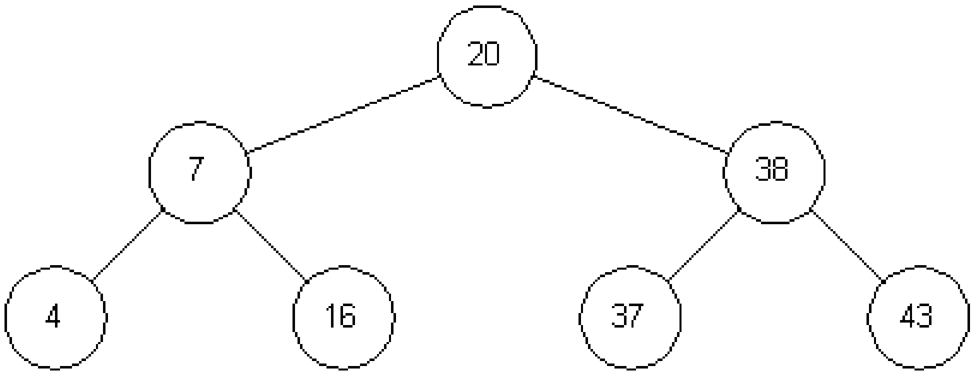
Sorting Algorithms

- SelectionSort
- InsertionSort
- BubbleSort
- ShakerSort
- QuickSort
- MergeSort
- HeapSort
- Bucket & Radix Sort
- Counting Sort

Problems to think about!

- What is the least number of comparisons you need to sort a list of 3 elements? 4 elements? 5 elements?
- How to arrange a tennis tournament in order to find the tournament champion with the least number of matches? How many tennis matches are needed?

Storing binary trees as arrays



20	7	38	4	16	37	43

Heaps (Max-Heap)

43	16	38	4	7	37	20

43	16	38	4	7	37	20	2	3	6	1	30

HEAP represents a binary tree stored as an array such that:

- Tree is filled on all levels except last
- Last level is filled from left to right
- Left & right child of i are in locations $2i$ and $2i+1$
- HEAP PROPERTY:

  Parent value is at least as large as child's value

HeapSort

- First convert array into a heap (BUILD-MAX-HEAP, p133)
- Then convert heap into sorted array (HEAPSORT, p136)

Animation Demos

http://www-cse.uta.edu/~holder/courses/cse2320/lectures/applets/sort1/heapsort.html

http://cg.scs.carleton.ca/~morin/misc/sortalg/

HeapSort: Part 1

MAX-HEAPIFY(*array* A, *int* i)

$\triangleright$ Assume subtree rooted at i is not a heap;

$\triangleright$ but subtrees rooted at children of i are heaps

$l \leftarrow \text{LEFT}[i]$

$r \leftarrow \text{RIGHT}[i]$

if $((l \leq \textit{heap-size}[A])$ *and* $(A[l] > A[i]))$

  then $\textit{largest} \leftarrow l$

  else $\textit{largest} \leftarrow i$

if $((r \leq \textit{heap-size}[A])$ *and* $(A[r] > A[\textit{largest}]))$

  then $\textit{largest} \leftarrow r$

if $(\textit{largest} \neq i)$

  then exchange $A[i] \leftrightarrow A[\textit{largest}]$

    MAX-HEAPIFY(A, *largest*)

O(height of node in location i) = O(log(size of subtree))

p130

HeapSort: Part 2

BUILD-MAX-HEAP($array\ A$)

$heap\text{-}size[A] \leftarrow length[A]$

for $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1

  do MAX-HEAPIFY(A, i)

HeapSort: Part 2

BUILD-MAX-HEAP(*array A*)

heap-size[A] $\leftarrow$ *length*[A]
for $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1
  do MAX-HEAPIFY(A, i)

HEAPSORT(*array A*)

BUILD-MAX-HEAP(A)
for $i \leftarrow length[A]$ **downto** 2
  do exchange $A[1] \leftrightarrow A[i]$
    heap-size[A] $\leftarrow$ *heap-size*[A] $- 1$
    MAX-HEAPIFY($A, 1$)

O(log n)

Total: O(nlog n)

Build-Max-Heap Analysis

For the HeapSort analysis, we need to compute:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}$$

We know from the formula for geometric series that

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

Differentiating both sides, we get

$$\sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2}$$

Multiplying both sides by x we get

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

Now replace $x = 1/2$ to show that

$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \leq \frac{1}{2}$$