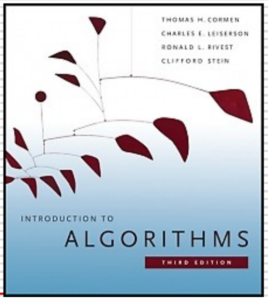
Introduction to Algorithms
(2nd edition)

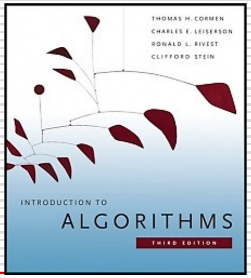


by Cormen, Leiserson, Rivest & Stein

Chapter 2: Getting Started

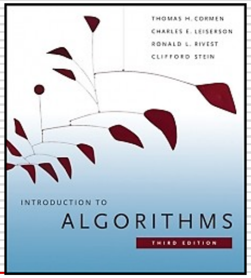
(slides enhanced by N. Adlai A. DePano)

Overview



- Aims to familiarize us with framework used throughout text
- Examines alternate solutions to the sorting problem presented in Ch. 1
- Specify algorithms to solve problem
- Argue for their correctness
- Analyze running time, introducing notation for asymptotic behavior
- Introduce divide-and-conquer algorithm technique

The Sorting Problem



Input: A sequence of n numbers $[a_1, a_2, \ldots, a_n]$.

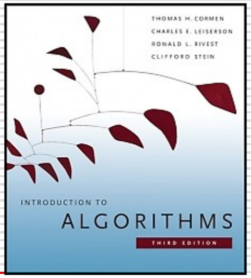
Output: A permutation or reordering $[a'_1, a'_2, \ldots, a'_n]$ of the input sequence such that $a'_1 \le a'_2 \le \ldots \le a'_n$.

An instance of the Sorting Problem:

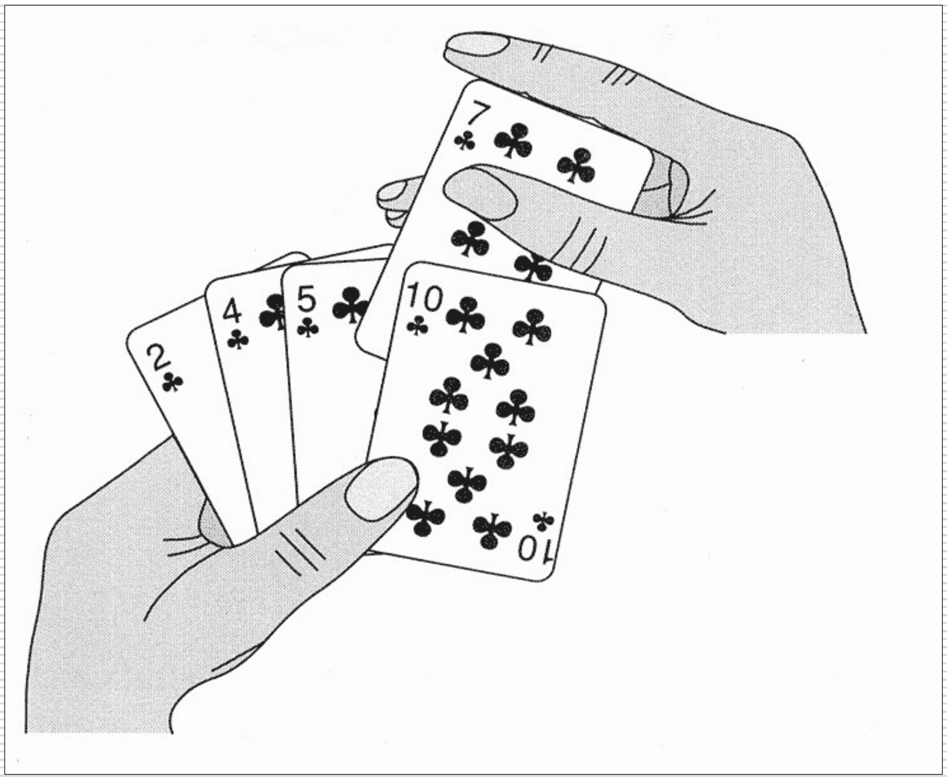
Input: A sequence of 6 number [31, 41, 59, 26, 41, 58].

Expected output for given instance:

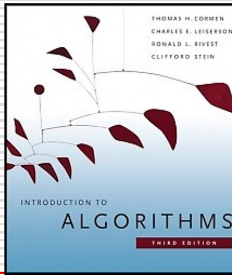
Expected Output: The permutation of the input [26, 31, 41, 41, 58 , 59].



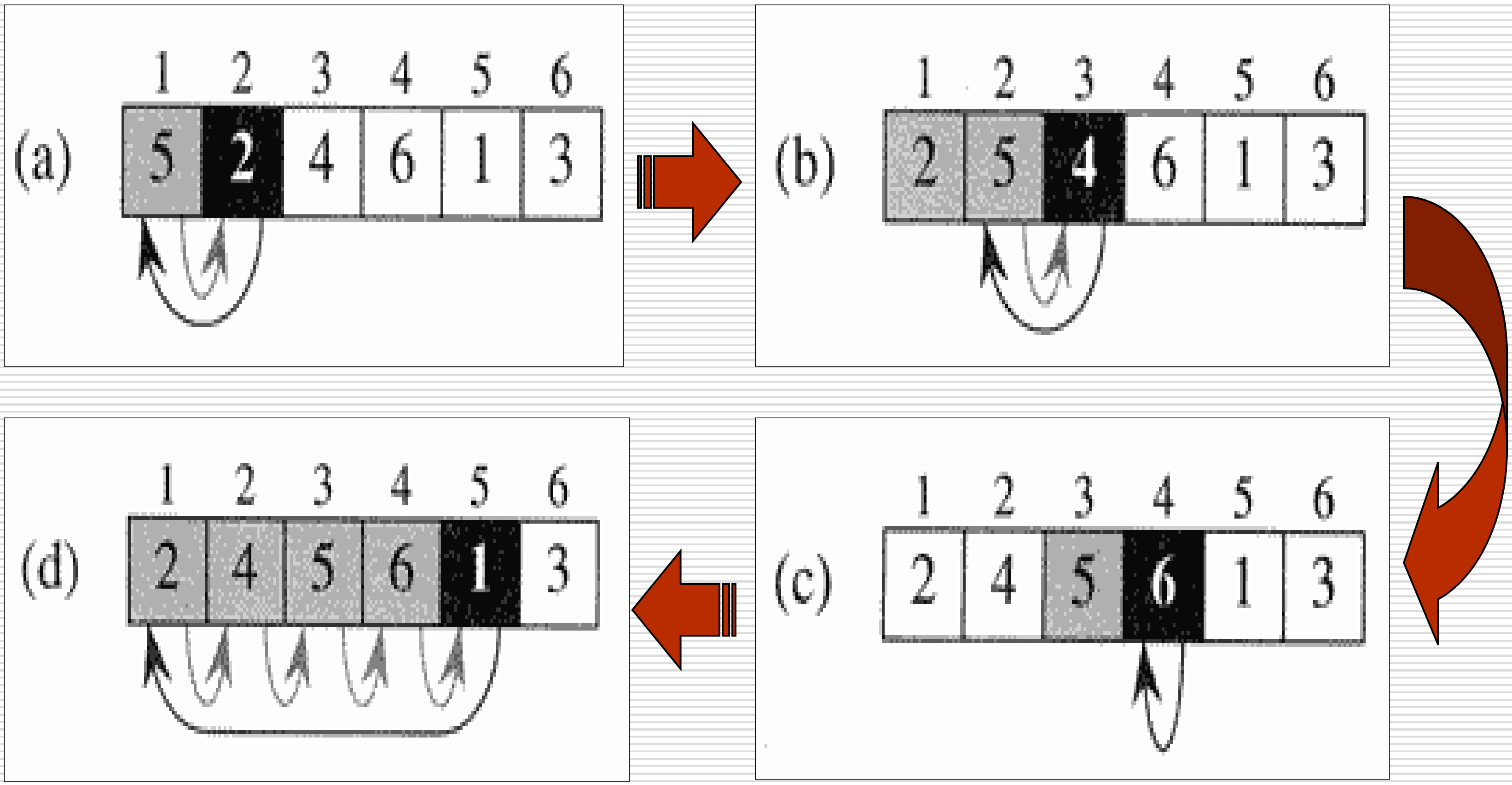
Insertion Sort

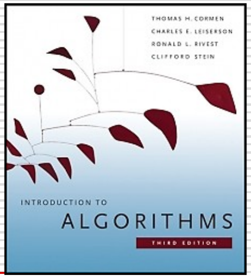


The main idea …

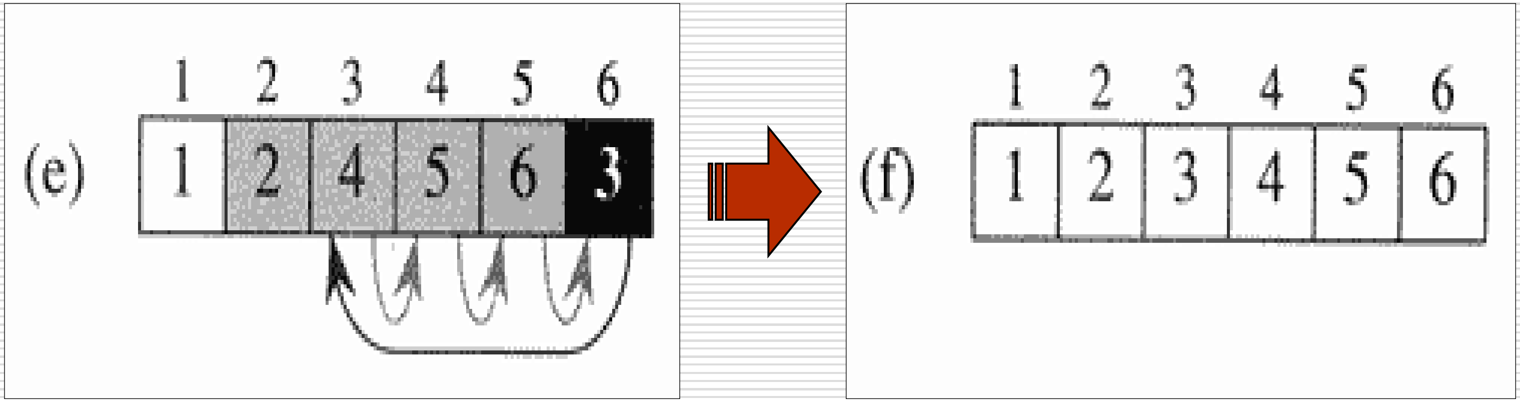


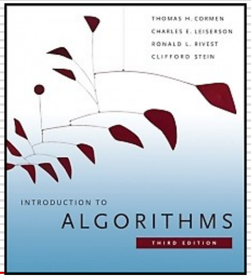
Insertion Sort (cont.)





Insertion Sort (cont.)



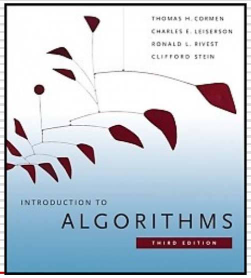


Insertion Sort (cont.)

The algorithm …

```
INSERTION-SORT(A, n)
for j = 2 to n
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

Loop Invariant

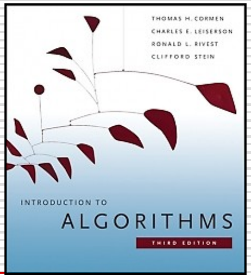


- Property of $A[1 \mathinner{..} j - 1]$

  At the start of each iteration of the **for** loop of lines 1- 8, the subarray $A[1 \mathinner{..} j - 1]$ consists of the elements originally in $A[1 \mathinner{..} j - 1]$ but in sorted order.

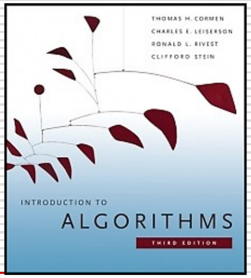
- Need to establish the following re invariant:
  - **Initialization**: true prior to first iteration
  - **Maintenance**: if true before iteration, remains true after iteration
  - **Termination**: at loop termination, invariant implies correctness of algorithm

Analyzing Algorithms



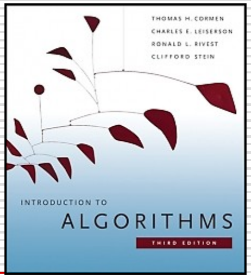
- Has come to mean predicting the resources that the algorithm requires
- Usually computational time is resource of primary importance
- Aims to identify best choice among several alternate algorithms
- Requires an agreed-upon “model” of computation
- Shall use a generic, one-processor, random-access machine (RAM) model of computation

Random-Access Machine



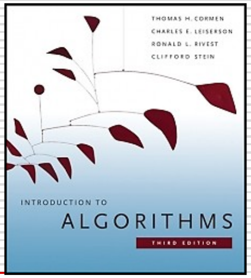
- Instructions are executed one after another (no concurrency)
- Admits commonly found instructions in “real” computers, data movement operations, control mechanism
- Uses common data types (integer and float)
- Other properties discussed as needed
- Care must be taken since model of computation has great implications on resulting analysis

Analysis of Insertion Sort



- Time resource requirement depends on ***input size***
- ***Input size*** depends on problem being studied; frequently, this is the number of items in the input
- Running time: number of primitive operations or “steps” executed for an input
- Assume constant amount of time for each line of pseudocode

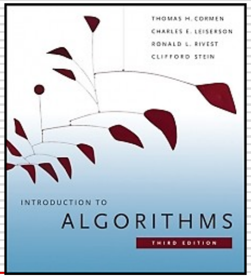
Analysis of Insertion Sort



Time efficiency analysis …

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

Best Case Analysis

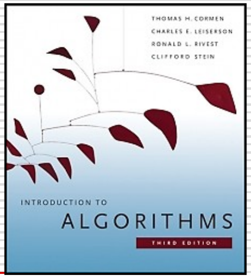


- Least amount of (time) resource ever needed by algorithm
- Achieved when incoming list is ***already sorted*** in increasing order
- Inner loop is never iterated
- Cost is given by:

$$T(n) = c_1 n + c_2\,(n-1) + c_4\,(n-1) + c_5(n-1) + c_8(n-1)$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$
$$= \;an + b$$

- Linear function of *n*

Worst Case Analysis

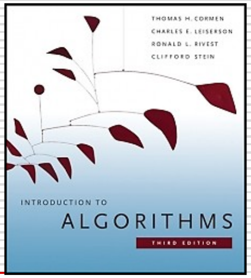


- Greatest amount of (time) resource ever needed by algorithm
- Achieved when incoming list is in ***reverse order***
- Inner loop is iterated the maximum number of times, *i.e.*, $t_j = j$
- Therefore, the cost will be:

$$
\begin{aligned}
T(n) &= c_1 n + c_2\,(n\text{-}1) + c_4\,(n\text{-}1) + c_5((n(n+1)/2)\ \text{-}1) + c_6(n(n\text{-}1)/2) \\
&\quad + c_7(n(n\text{-}1)/2) + c_8(n\text{-}1) \\
&= (\,c_5/2 + c_6/2 + c_7/2\,)\,n^2 + (c_1+c_2+c_4+c_5/2 - c_6/2 - c_7/2 + c_8)\,n \\
&\quad - (\,c_2 + c_4 + c_5 + c_8\,) \\
&= an^2 + bn + c
\end{aligned}
$$

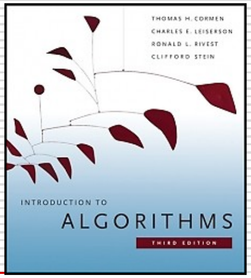
- Quadratic function of *n*

Future Analyses



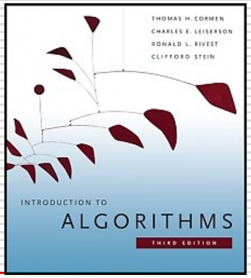
- For the most part, subsequent analyses will focus on:
  - Worst-case running time
    - Upper bound on running time for ***any*** input
  - Average-case analysis
    - Expected running time over ***all*** inputs
- Often, worst-case and average-case have the same “order of growth”

Order of Growth



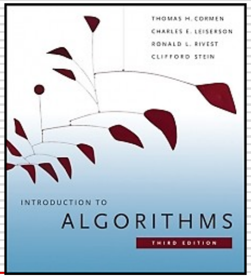
- Simplifying abstraction: interested in ***rate of growth*** or ***order of growth*** of the running time of the algorithm
- Allows us to compare algorithms without worrying about implementation performance
- Usually only highest order term without constant coefficient is taken
- Uses "theta" notation
  - Best case of insertion sort is $\Theta(n)$
  - Worst case of insertion sort is $\Theta(n^2)$

Designing Algorithms



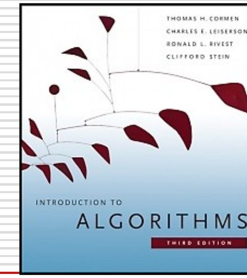
- Several techniques/patterns for designing algorithms exist
- ***Incremental approach***: builds the solution one component at a time
- ***Divide-and-conquer approach***: breaks original problem into several smaller instances of the same problem
  - Results in ***recursive*** algorithms
  - Easy to analyze complexity using proven techniques

Divide-and-Conquer

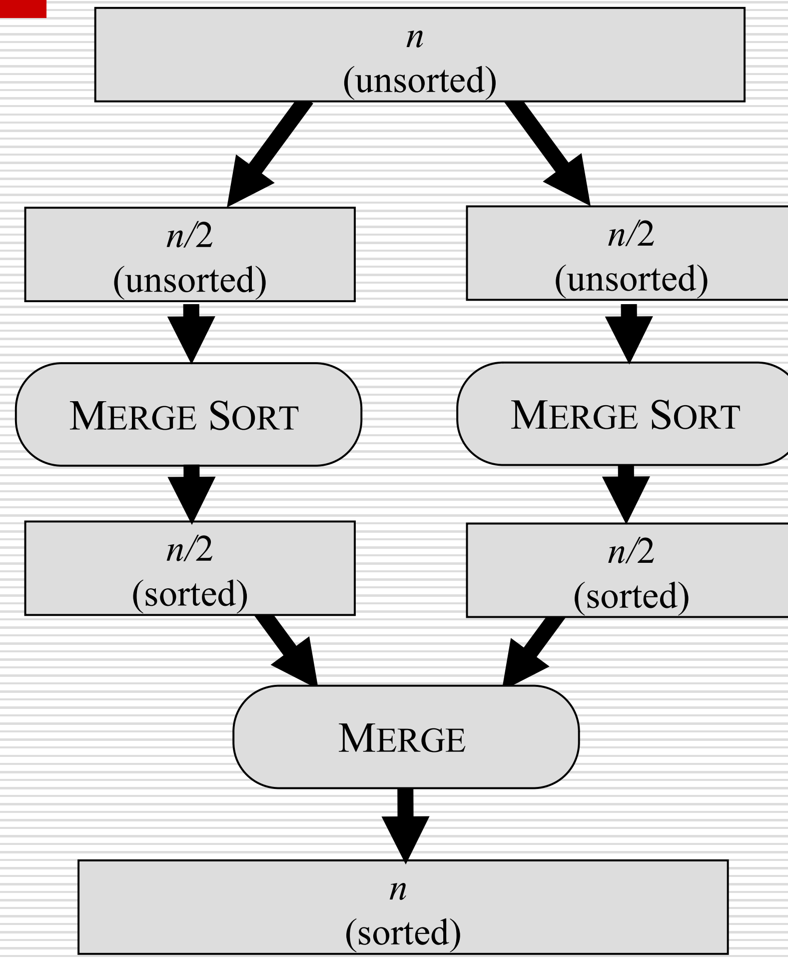


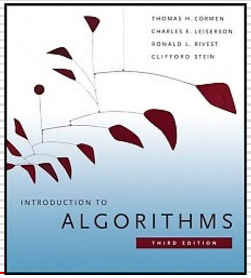
- Technique (or paradigm) involves:
  - "Divide" stage: Express problem in terms of several smaller subproblems
  - "Conquer" stage: Solve the smaller subproblems by applying solution recursively – smallest subproblems may be solved directly
  - "Combine" stage: Construct the solution to original problem from solutions of smaller subproblem

Merge Sort Strategy

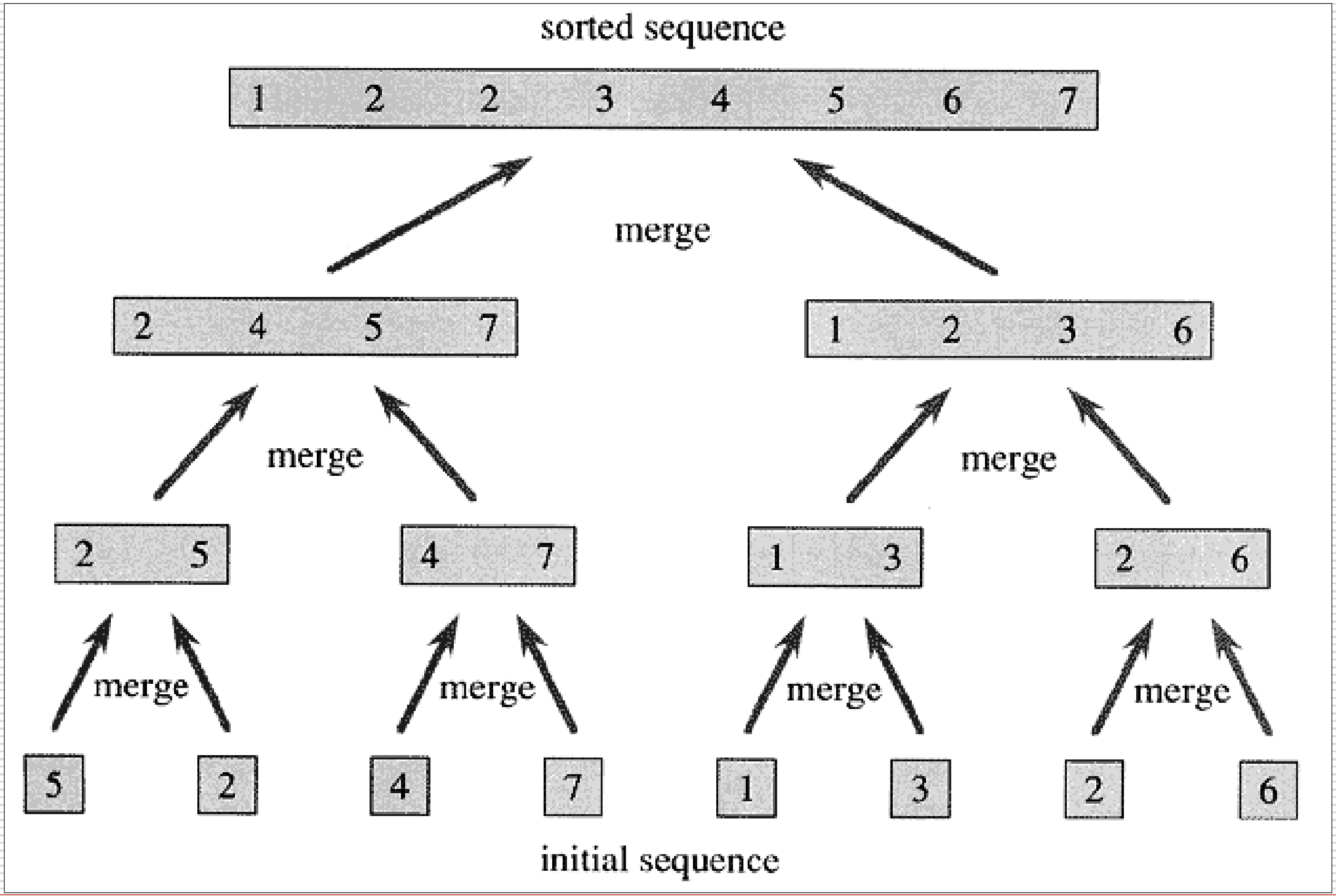


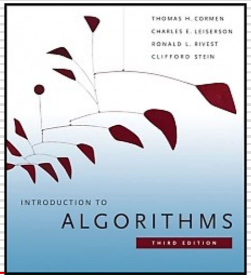
- ***Divide stage***: Split the *n*-element sequence into two subsequences of *n*/2 elements each
- ***Conquer stage***: Recursively sort the two subsequences
- ***Combine stage***: Merge the two sorted subsequences into one sorted sequence (the solution)





Merging Sorted Sequences





Merging Sorted Sequences

MERGE(A, p, q, r)

$\Theta(1)$
$n_1 = q - p + 1$
$n_2 = r - q$

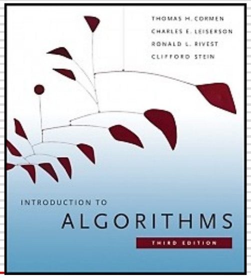
$\Theta(n)$
let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays
for $i = 1$ **to** n_1
    $L[i] = A[p + i - 1]$
for $j = 1$ **to** n_2
    $R[j] = A[q + j]$

$\Theta(1)$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$

$\Theta(n)$
for $k = p$ **to** r
    if $L[i] \leq R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
        $j = j + 1$

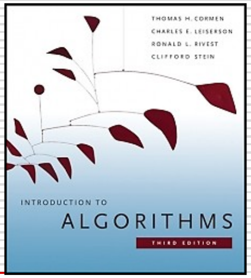
- Combines the sorted subarrays $A[p..q]$ and $A[q+1..r]$ into one sorted array $A[p..r]$
- Makes use of two working arrays L and R which initially hold copies of the two subarrays
- Makes use of sentinel value (∞) as last element to simplify logic

Merge Sort Algorithm



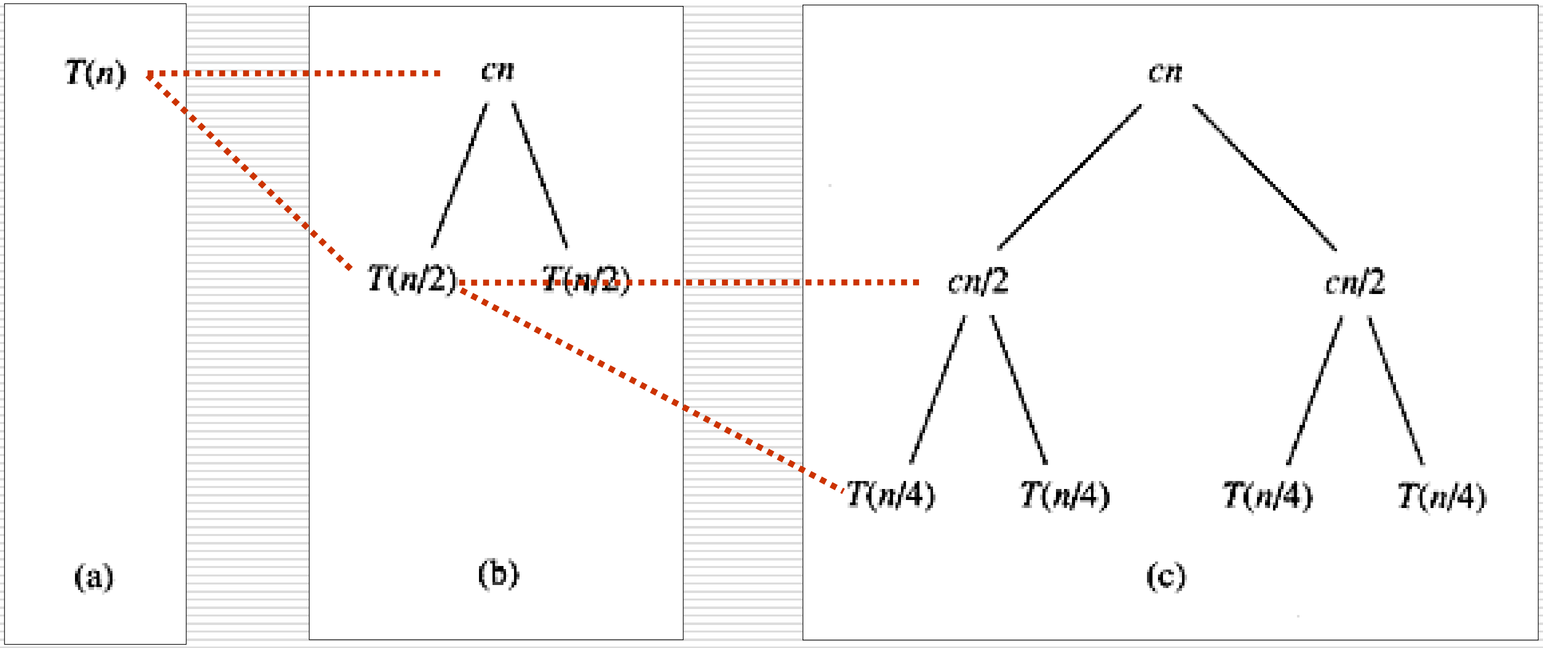
	MERGE-SORT(A, p, r)	
	if $p < r$	// check for base case
$\Theta(1)$	$\quad q = \lfloor (p + r)/2 \rfloor$	// divide
$T(n/2)$	$\quad$ MERGE-SORT(A, p, q)	// conquer
$T(n/2)$	$\quad$ MERGE-SORT$(A, q + 1, r)$	// conquer
$\Theta(n)$	$\quad$ MERGE(A, p, q, r)	// combine

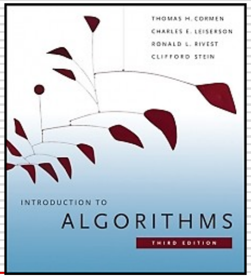
$$T(n) = 2T(n/2) + \Theta(n)$$



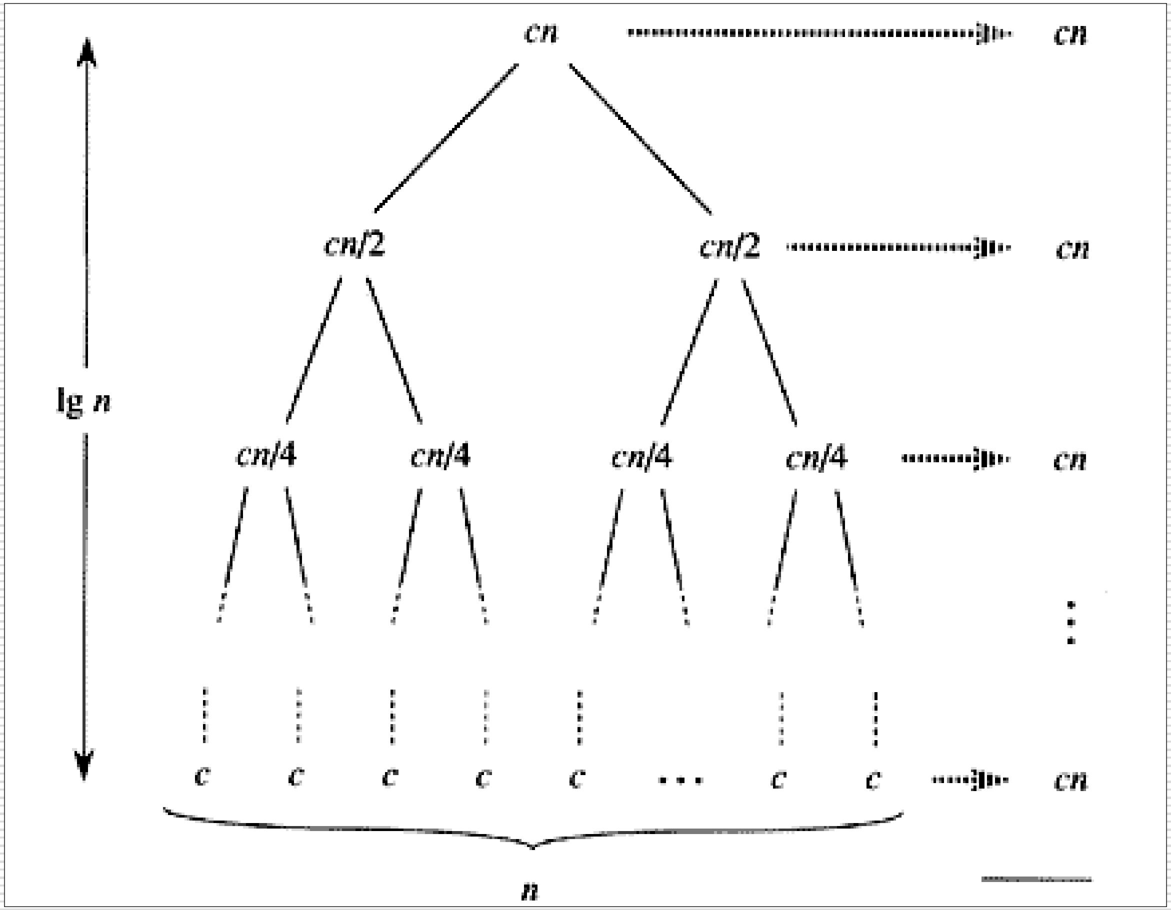
Analysis of Merge Sort

Analysis of recursive calls …





Analysis of Merge Sort



$$T(n) = cn(\lg n + 1)$$
$$= cn\lg n + cn$$

$T(n)$ is $\Theta(n \lg n)$