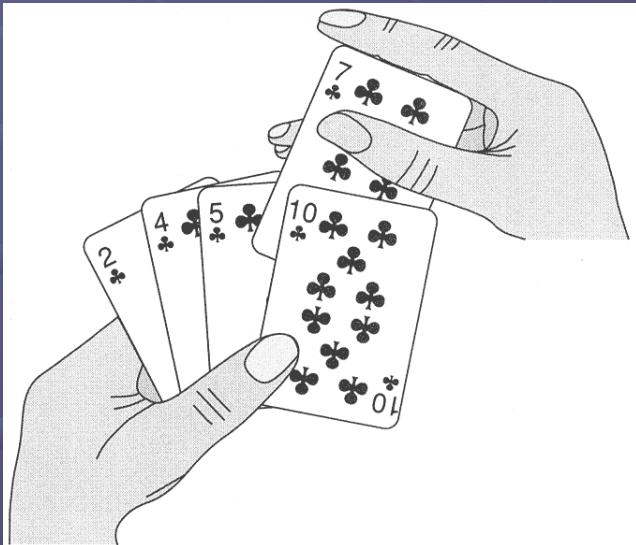


Getting Started

Insertion Sort

☞ Sorting problem

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



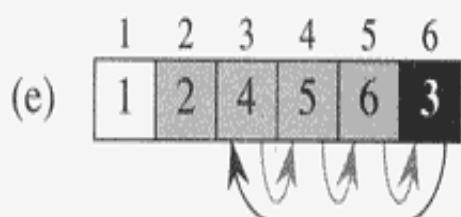
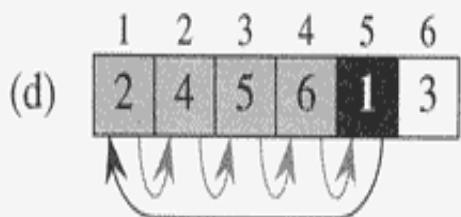
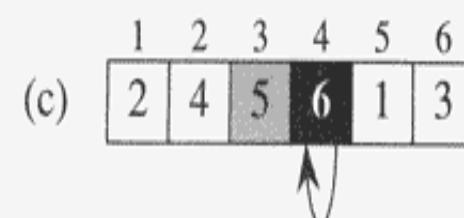
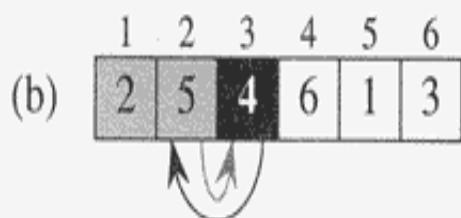
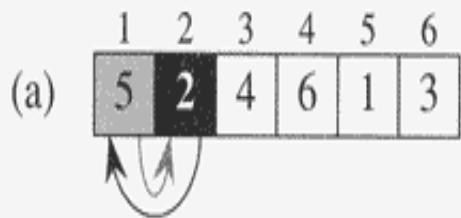
☞ **Keys**: The numbers that we wish to sort.

Pseudo Code

Insertion-Sort(A)

```
1 for  $j = 2$  to  $A.length$ 
2   key =  $A[j]$ 
3   // Insert  $A[j]$  into the sorted sequence  $A[1 .. j-1]$ 
4    $i = j - 1$ 
5   while  $i > 0$  and  $A[i] > key$ 
6      $A[i+1] = A[i]$ 
7      $i = i - 1$ 
8    $A[i+1] = key$ 
```

Insertion Sort



Loop Invariant and the Correctness

☞ Sorted in place

- The numbers are rearranged within the array A , with at most a constant number of them sorted outside the array at any time.

☞ Loop Invariant

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

☞ Loop invariant is used to help us understand why an algorithm is correct.

Loop Invariant and the Correctness

- ☞ We must show three things about a loop invariant:
 - **Initialization:** It is true prior to the first iteration of the loop.
 - **Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.
 - **Termination:** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Analyzing Algorithms

☞ Analyzing algorithms

- Predicting the resources that the algorithm requires.

☞ Resources

- Memory, communication, bandwidth, logic gate, time.

☞ Assumption

- One processor, **random-access machine** (RAM) model

☞ The best notion for *input size* depends on the problem being studied.

- *Number of items in the input* : Sorting.

- *Total number of bits* : Multiplying two integers.

Analysis of Insertion Sort

- ***Two numbers to represent the size*** : Graph algorithms.
- ☞ The ***running time*** of an algorithm
 - The number of primitive operations or “steps” executed.
 - ☞ Time to execute each line of the pseudo code: c_i .
 - ☞ The number of times to execute the while loop test: t_j .
 - ☞ Even for inputs of a given size, an algorithm’s running time may depend on which input of that size is given.
- The ***best case v.s. the worst case*** running time.

Analysis of Insertion Sort

INSERTION-SORT(A)

		<i>cost</i>	<i>times</i>
1	for $j = 2$ to $A.length$	c_1	n
2	$key = A[j]$	c_2	$n - 1$
3	// Insert $A[j]$ into the sorted sequence $A[1 \dots j - 1]$.	0	$n - 1$
4	$i = j - 1$	c_4	$n - 1$
5	while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^n t_j$
6	$A[i + 1] = A[i]$	c_6	$\sum_{j=2}^n (t_j - 1)$
7	$i = i - 1$	c_7	$\sum_{j=2}^n (t_j - 1)$
8	$A[i + 1] = key$	c_8	$n - 1$

Analysis of Insertion Sort

☞ Running time

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j +$$

$$c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)$$

☞ The best case running time: Sorted, $t_j = 1$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

Analysis of Insertion Sort

☞ The worst case: Reversed ordered, $t_j = j$.

$$\begin{aligned} T(n) &= c_1n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + \\ &\quad c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5 + c_6 + c_7}{2}\right)n^2 - \left(c_1 + c_2 + c_4 + \frac{c_5 - c_6 - c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

$$T(n) = an^2 + bn + c$$

Analyzing Algorithms

☞ We usually concentrate on the worst case. Why?

- The worst case running time of an algorithm is an upper bound on the running for any input.
- For some algorithms, the worst case occurs fairly often.
- The “average case” is often roughly as bad as the worst case.

☞ *Order of growth (rate of growth)*

- A simplifying abstraction
- Only consider the leading term of a formula.
- Also ignores the leading term's constant coefficient

$$T(n) = an^2 + bn + c = \Theta(n^2)$$

Designing Algorithms

☞ Many ways to design algorithms

■ **Incremental approach**: insertion sort.

■ **Divide-and-Conquer**: merge sort.

☞ **Divide-and-Conquer approach**

■ Paradigm

- **Divide**: Subdivide the problem into smaller subproblems.
- **Conquer**: If subproblem size small enough, solve it straightforward.
- **Combine**: Integrate solutions of subproblems into the solution of the original problem.

■ Recursive structure

Merge Sort

☞ Follow the divide-and-conquer paradigm:

- Divide: Divide the n -element sequence to be sorted into two subsequences of $n/2$ elements each.
- Conquer: Sort the two subsequences recursively using merge sort.
- Combine: Merge the two sorted subsequences to produce a sorted sequence.

☞ Key operation

- The merging of two sorted sequences.

☞ *Sentinel card*

Merge

MERGE(A, p, q, r)

```
1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  let  $L[1..n_1 + 1]$  and  $R[1..n_2 + 1]$  be new arrays
4  for  $i = 1$  to  $n_1$ 
5       $L[i] = A[p + i - 1]$ 
6  for  $j = 1$  to  $n_2$ 
7       $R[j] = A[q + j]$ 
8   $L[n_1 + 1] = \infty$ 
9   $R[n_2 + 1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13     if  $L[i] \leq R[j]$ 
14          $A[k] = L[i]$ 
15          $i = i + 1$ 
16     else  $A[k] = R[j]$ 
17          $j = j + 1$ 
```

Merge

	8	9	10	11	12	13	14	15	16	17	
A	...	2	4	5	7	1	2	3	6	...	
	k										

L	1	2	3	4	5	
	2	4	5	7	∞	
	i					

(a)

	8	9	10	11	12	13	14	15	16	17	
A	...	1	4	5	7	1	2	3	6	...	
	k										

L	1	2	3	4	5	
	2	4	5	7	∞	
	i					

(b)

	8	9	10	11	12	13	14	15	16	17	
A	...	1	2	5	7	1	2	3	6	...	
	k										

L	1	2	3	4	5	
	2	4	5	7	∞	
	i					

(c)

	8	9	10	11	12	13	14	15	16	17	
A	...	1	2	2	7	1	2	3	6	...	
	k										

L	1	2	3	4	5	
	2	4	5	7	∞	
	i					

(d)

Merge

A	8	9	10	11	12	13	14	15	16	17	\dots
	...	1	2	2	3	1	2	3	6	...	
											k
L	1	2	3	4	5	∞					
	2	4	5	7	∞						
	i					j					

(e)

A	8	9	10	11	12	13	14	15	16	17	\dots
	...	1	2	2	3	4	2	3	6	...	
											k
L	1	2	3	4	5	∞					
	2	4	5	7	∞						
	i					j					

(f)

A	8	9	10	11	12	13	14	15	16	17	\dots
	...	1	2	2	3	4	5	3	6	...	
											k
L	1	2	3	4	5	∞					
	2	4	5	7	∞						
	i					j					

(g)

A	8	9	10	11	12	13	14	15	16	17	\dots
	...	1	2	2	3	4	5	6	6	...	
											k
L	1	2	3	4	5	∞					
	2	4	5	7	∞						
	i					j					

(h)

A	8	9	10	11	12	13	14	15	16	17	\dots
	...	1	2	2	3	4	5	6	7	...	
											k
L	1	2	3	4	5	∞					
	2	4	5	7	∞						
	i					j					

(i)

Merge

☞ Loop invariant

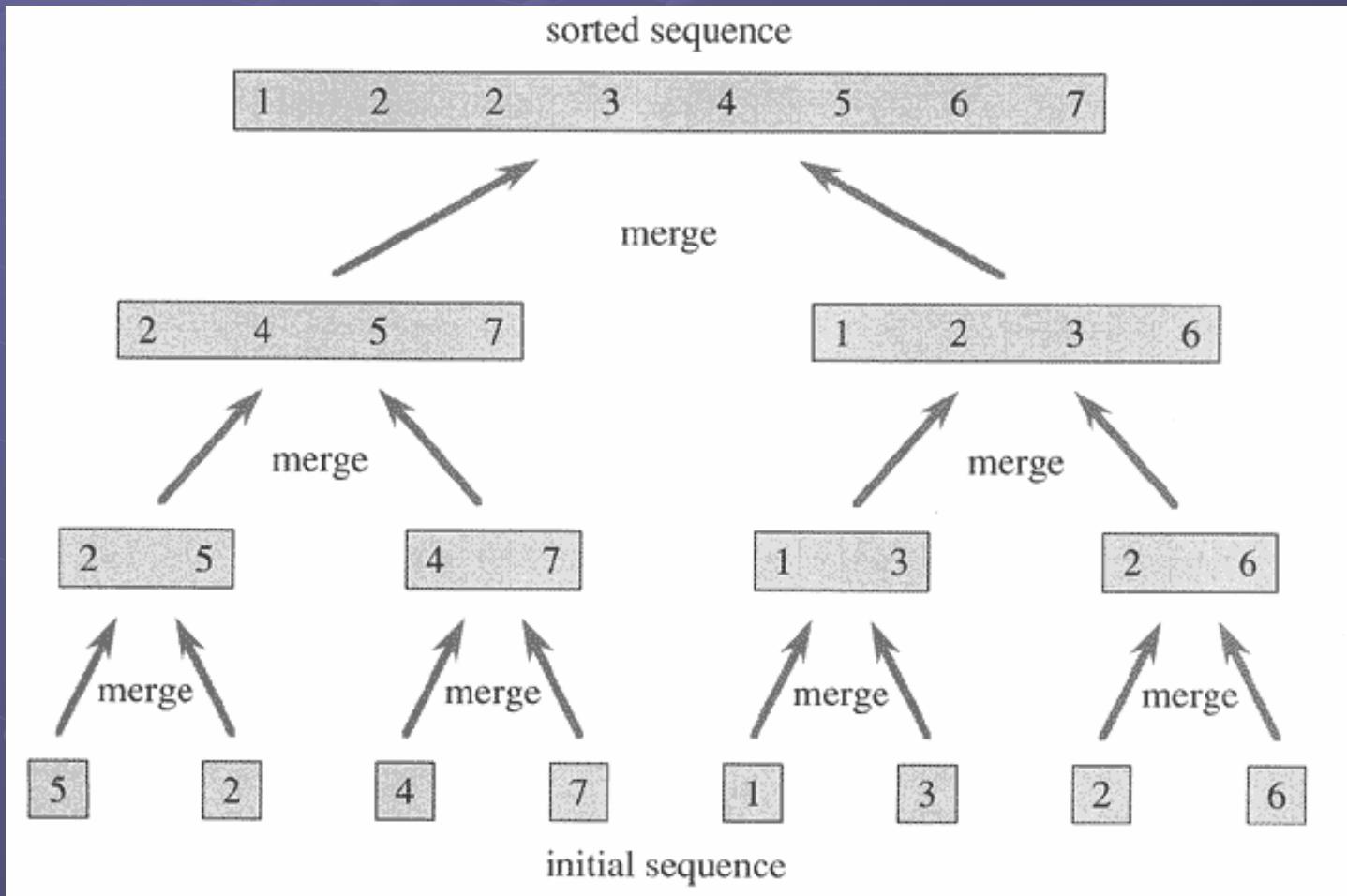
- At the start of each iteration of the **for** loop of lines 12-17, the subarray $A[p \dots k-1]$ contains the $k - p$ smallest elements of $L[1 \dots n_1+1]$ and $R[1 \dots n_2+1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A .
- Initialization
- Maintenance
- Termination

Merge Sort

```
MERGE-SORT( $A, p, r$ )
```

```
1   if  $p < r$ 
2        $q = \lfloor (p + r)/2 \rfloor$ 
3       MERGE-SORT( $A, p, q$ )
4       MERGE-SORT( $A, q + 1, r$ )
5       MERGE( $A, p, q, r$ )
```

Merge Sort



Analyzing Divide-and-Conquer Algorithms

☞ Recurrence equation (or recurrence)

- When an algorithm contains a recursive call to itself, its running time can often be described by recurrence.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Analysis of Merge Sort

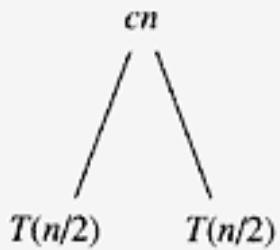
- ☞ Divide: Constant time $D(n) = \Theta(1)$.
- ☞ Conquer: Recursively solve two sub problems $2T(n/2)$.
- ☞ Combine: Merge into an n-element array $C(n) = \Theta(n)$.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

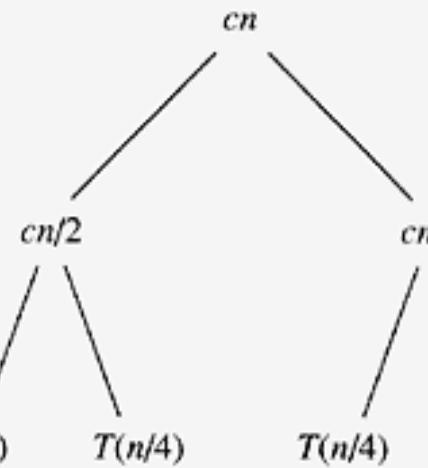
$$T(n) = \begin{cases} c & \text{if } n = 1, \\ 2T(n/2) + cn & \text{if } n > 1. \end{cases}$$

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

Analysis of Merge Sort

 $T(n)$ 

(a)



(b)

(c)

Analysis of Merge Sort

