Sorting Algorithms

Professor Yeali S. Sun

National Taiwan University

Sorting Algorithms: Two Categories

- Internal Sort
  - For data sets to be sorted that can fit entirely in main memory
- External Sort
  - For data set to be sorted that cannot fit in main memory all at once but must reside in secondary storage (e.g., disk)

Sorting Algorithms

- Selection Sort – $O(N^2)$
- Bubble Sort - $O(N^2)$
- Insertion Sort – $O(N^2)$ (worst-case)
- Mergesort - $O(N*\log_2 N)$
- Quicksort - $O(N*\log_2 N)$
- Heapsort
- Treesort

Sorting: Introduction

- Arrangement of objects according to some ordering criteria.
- Assume we have a collection of information concerning some set of objects.
- Assume this collection of information is organized in records.
- Within a record, information is structured into a number of units called fields.

Sorting: Introduction (cont'd)

- The data structure of a record depends on the application.
  - e.g.,
    - collection of objects: telephone directory
    - objects: companies/stores
    - information about an object: company name, products, phone number, address, price, etc
    - one record for one company/store
  - e.g.,
    - student data file for a university
    - objects: students
    - information about an object: name, id, address, department, etc.

There are MANY sorting algorithms!

- No single sorting technique is the "best" for all applications.
  - Size of the problem, e.q., N
  - Time complexity in search, insert, and delete
  - Space complexity – data store and auxiliary space for sorting

Formal description of the sorting problem

- Given a list of records in which each record has a key value. There exists an ordering relation on the keys (>=, =. <=)
  - Note that ordering relations are transitive.

- The sorting problems are to find a permutation such that if the ordering relation is > then Key(i-1) < Key(i).

Formal description of the sorting problem

- If values of key are not unique, then consider two cases:
  - sorted: only obeys ordering relation
  - stable: if Key (i) = Key (j), element i precedes element j then in the sorted list element i also precedes element j.

Application #1 - SEARCHING!!

- How to IDENTIFY a record (or an object)?
- Use information about a record: one or more fields, e.g.,
  - student data file: id
  - telephone directory: company name & product type or telephone number
- "Key(s)": to uniquely identify a record

"Efficiency" of searching?

- Depends on how records are arranged!
  - random, sorted, ...
- e.g., sequential search $O(n)$, binary search $O(\log_2 n)$

Application #2 -

- How to know if two sets of information are identical?
  - e.g., tax reports from employees and employer

Selection Sort

- Input: a list of records: $R_0, R_1, \ldots, R_{n-1}$
- Output: an ordered list of records:

  $$R_{k_0}, R_{k_1}, \ldots, R_{k_{N-1}} \quad , \text{ where } k_0 \le k_1 \le \ldots \le k_{N-1}$$

- Algorithm:
  - step 1: i ← 0
  - step 2: find the largest item R_j from list $R_0, \ldots, R_{N-1-i} \quad , \ 0 \le j \le N-1-i$
  - step 3: swap R_j and R_{N-1-i} to produce a sequence of ordered records $R_{N-1-i}, R_{N-i}, \ldots, R_{N-1}$
  - step 4: increment i; repeat step 2 until $i = N-1$

• In each round, select the largest one and place it to the end.

Selection Sort (cont'd)

Shaded elements are selected;
boldface elements are in order.

Initial array:	29	10	14	37	13
After 1st swap:	29	10	14	13	**37**
After 2nd swap:	13	10	14	**29**	**37**
After 3rd swap:	13	10	**14**	**29**	**37**
After 4th swap:	**10**	**13**	**14**	**29**	**37**

Figure 9-4

A selection sort of an array of five integers

$$
\begin{array}{r}
N-1 \\
N-2 \\
\vdots \\
+\quad 1 \\
\hline
\dfrac{N(N-1)}{2}
\end{array}
\Rightarrow O(N^2)
$$

Bubble Sort

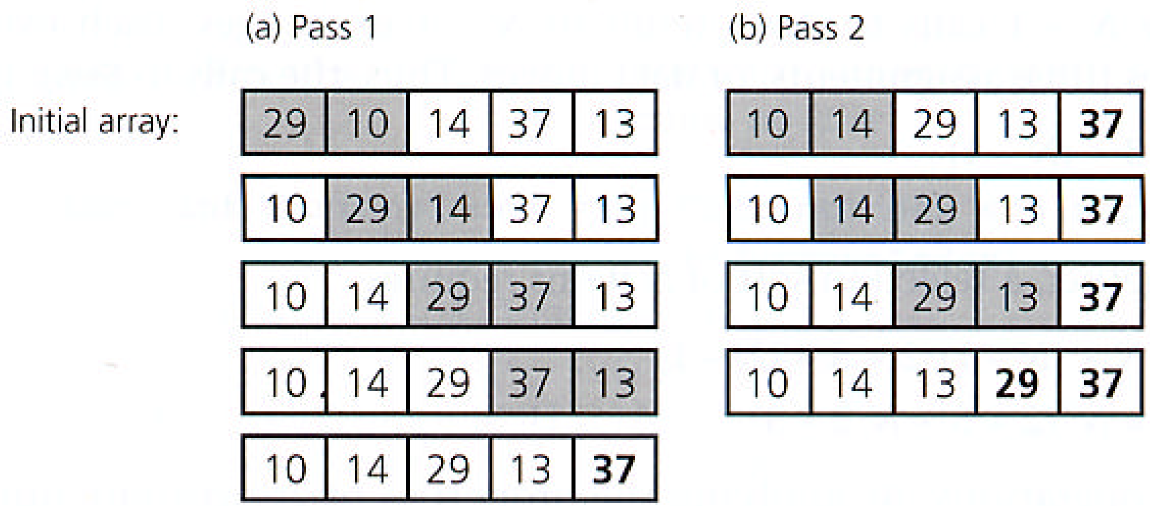


Figure 9-5

The first two passes of a bubble sort of an array of five integers: (a) pass 1; (b) pass 2

$$(N-1)+(N-2)+\ldots+1=\frac{N(N-1)}{2} \Rightarrow O(N^2)$$

$\Rightarrow$ $O(N)$ best case

Original data is sorted

Insertion Sort

- Partitioned the list into two regions: sorted (front region) and unsorted (rear region).
- At each step, takes the first item of the unsorted region and places it into its correct position in the sorted region

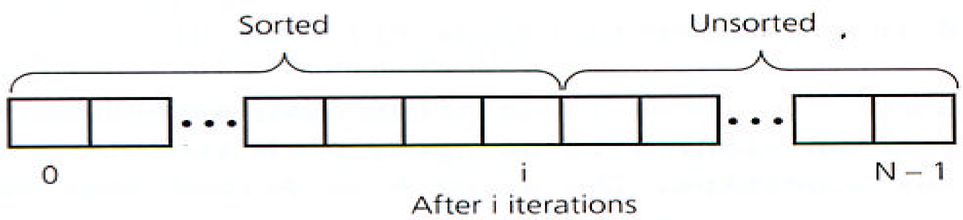


Figure 9-6
An insertion sort partitions the array into two regions

Insertion Sort (cont'd)

- Input: a list of records: $R_0, R_1, \ldots, R_{n-1}$
- Output: an ordered list of records:

$$R_{k_0}, R_{k_1}, \ldots, R_{k_{n-1}} \text{ , where } k_0 \le k_1 \le \ldots \le k_{n-1}$$

- Algorithm:
  - step 1: i ← 0
  - step 2: insert R_i into a sequence of ordered records $R_0, R_1, \ldots, R_{i-1}$ to produce a sequence of ordered records $R_0, R_1, \ldots, R_i$
  - step 3: increment i; repeat step 2 until $i \ge n$

Insertion Sort – an example

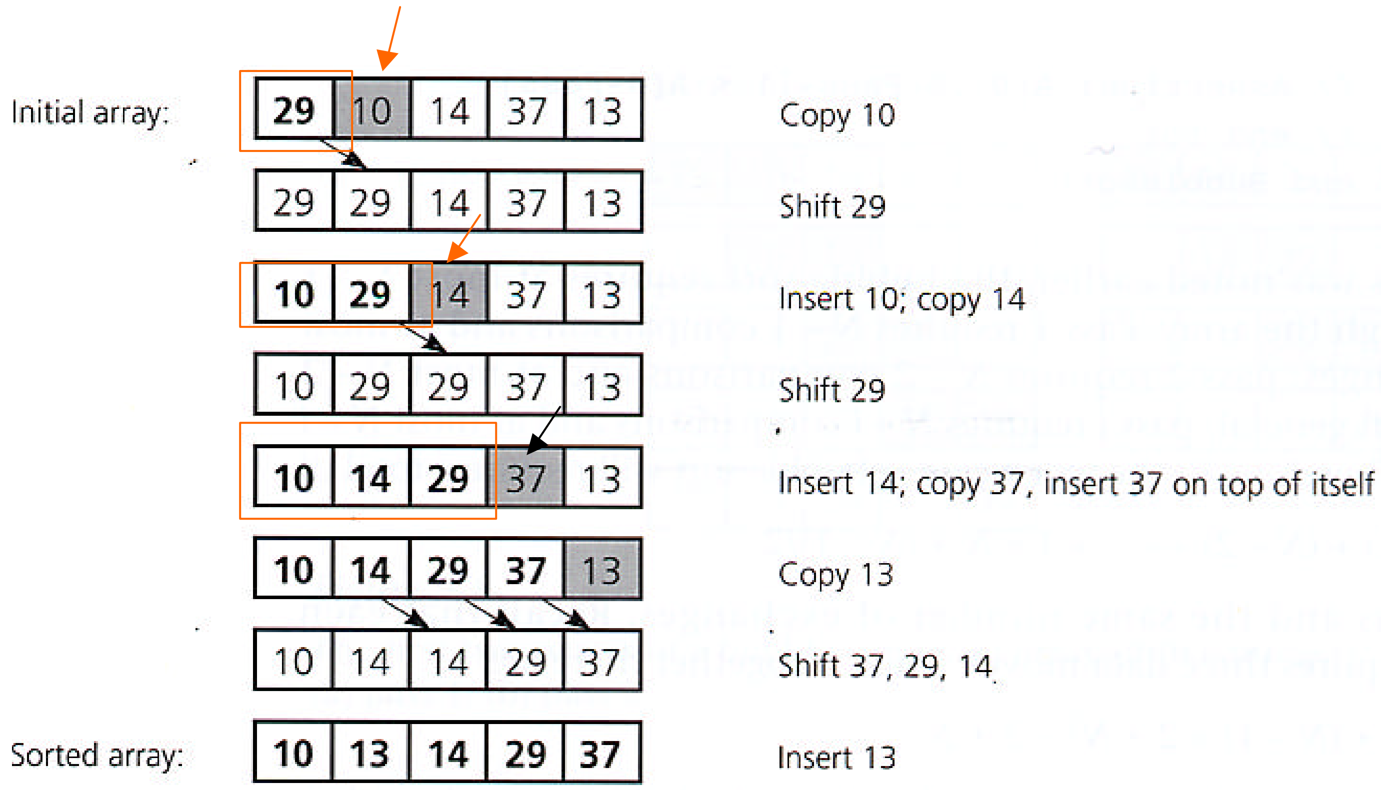


Figure 9-7

An insertion sort of an array of five integers

Insertion Sort

Example

	R_0	R_1	R_2	R_3	R_4
	5	4	3	2	1
$i=1$	4	5	3	2	1
2	3	4	5	2	1
3	2	3	4	5	1
4	1	2	3	4	5

Time Complexity

$$O\left(\sum_{i=0}^{n-1} i\right) = O(n^2)$$

Divide-and-Conquer

- Mergesort and Quicksort
  - recursive formulations
  - Efficient – $O(N*\log_2 N)$
  - Regardless the initial order of the items in the data set

Mergesort

- Algorithm
  - Divide the array into halves
  - Sort each half
  - Merge the sorted halves into one sorted list
- The merge step
  - Compare an item in one half with an item in the other half.
  - Move the smaller item to a temporary array until no more items to consider on e half.
  - Move the remaining items to the temporary array.
  - Copy the temporary array back into the original array.

Merge Sort – an example

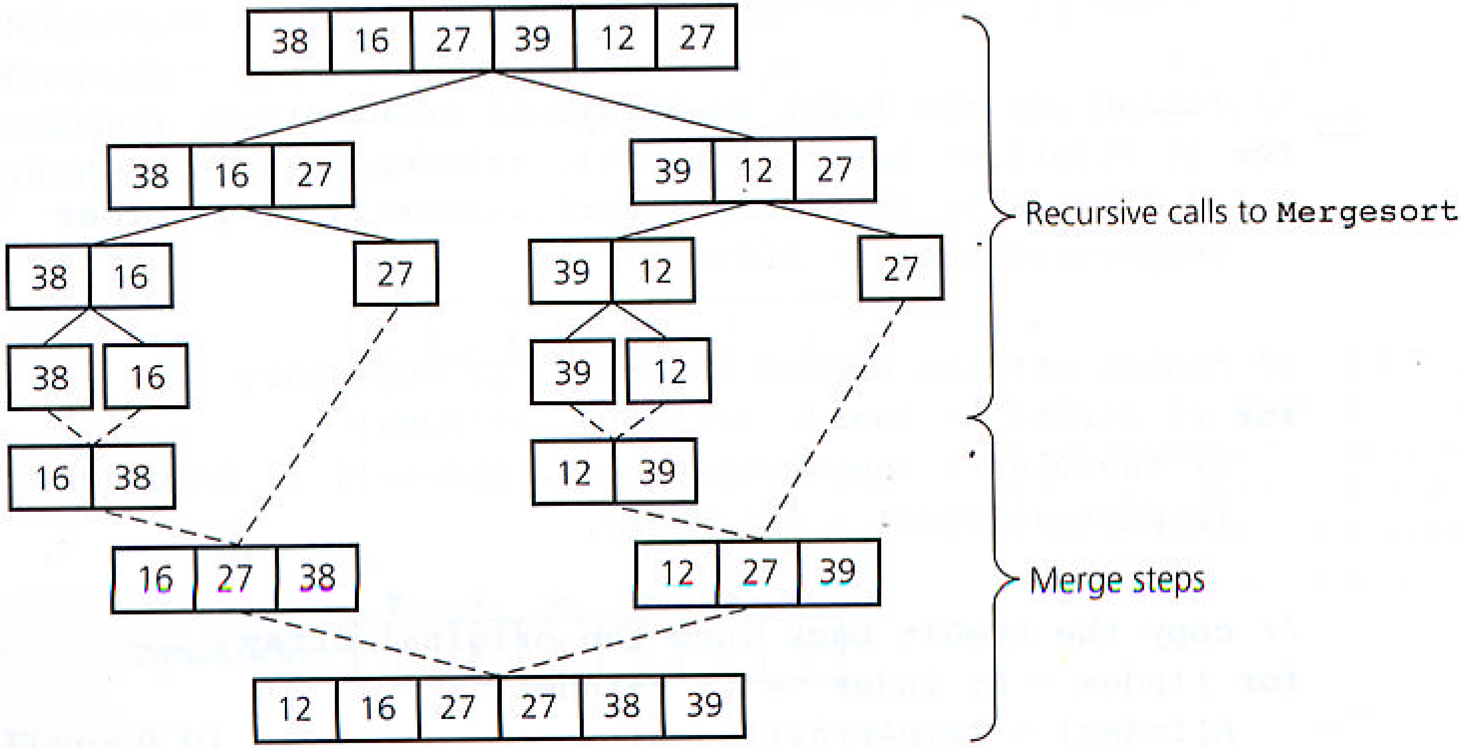


Figure 9-9

A mergesort of an array of six integers

Merge Sort - algorithm

```
mergesort(A, F, L)
// Sorts A[F..L] by
//   1. sorting the first half of the array
//   2. sorting the second half of the array
//   3. merging the two sorted halves
if (F < L)
{  Mid = (f + L)/2          // get midpoint
   mergesort(A, F, Mid)       // sort A[F..Mid]
   mergesort(A, Mid + 1, L)  // sort A[Mid+1..L]

   // merge sorted halves A[F..Mid] and A[Mid+1..L]
   merge(A, F, Mid, L)
}// end if
else quit
```

Merge Sort

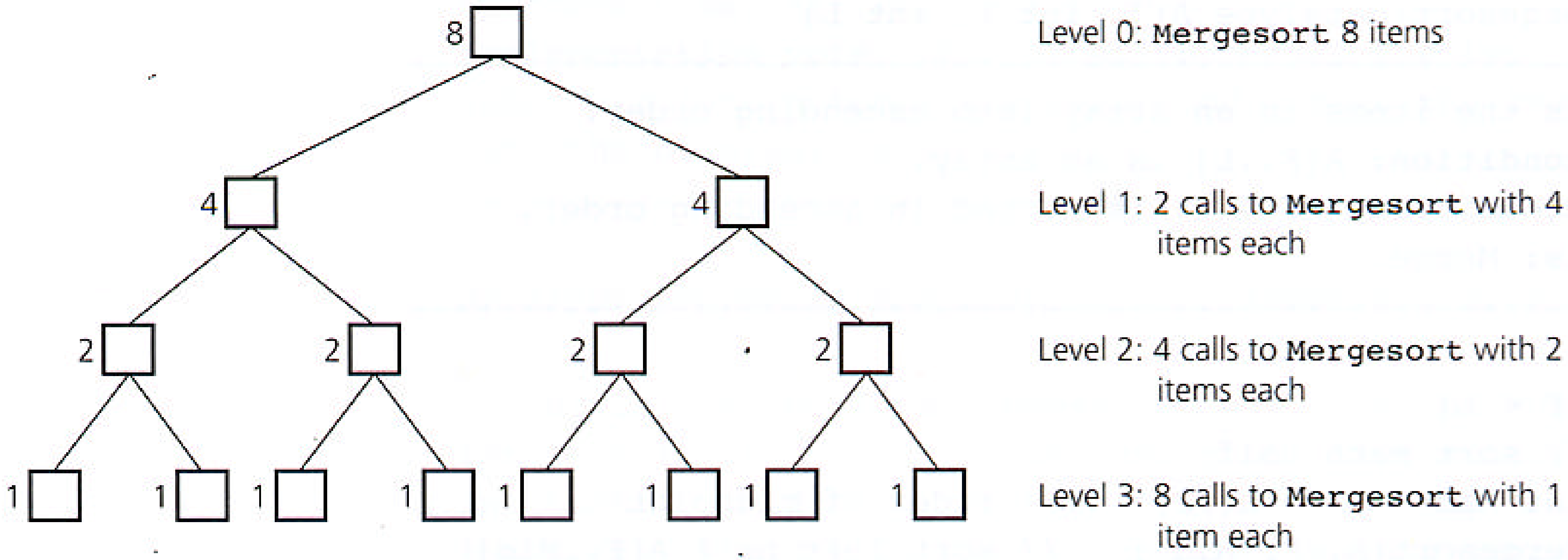


Figure 9-11

Levels of recursive calls to `Mergesort`, given an array of eight items

Merge Sort – an example (cont'd)

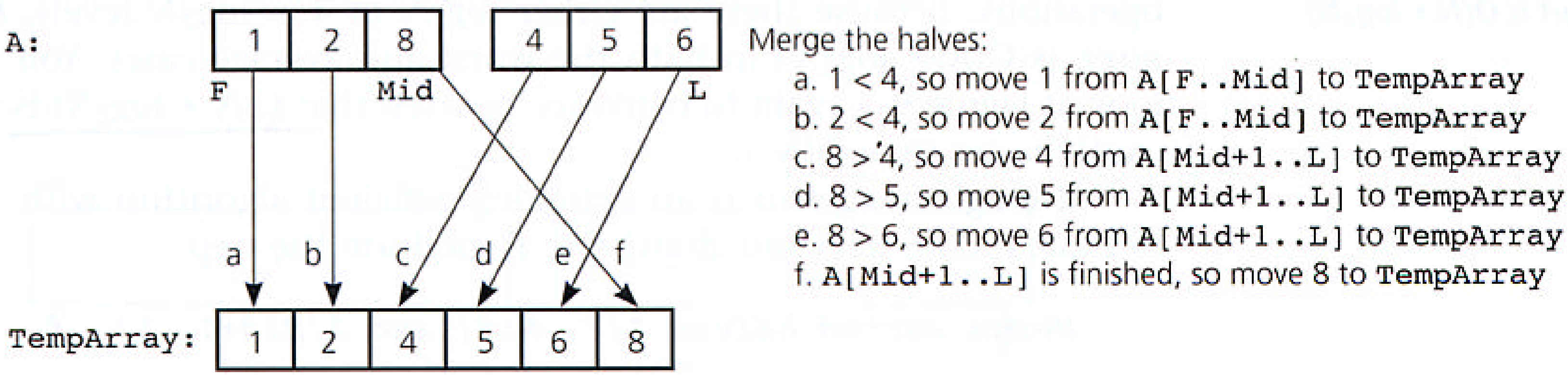


Figure 9-10

A worst-case instance of the merge step in `Mergesort`

Merge Sort – an example (cont'd)

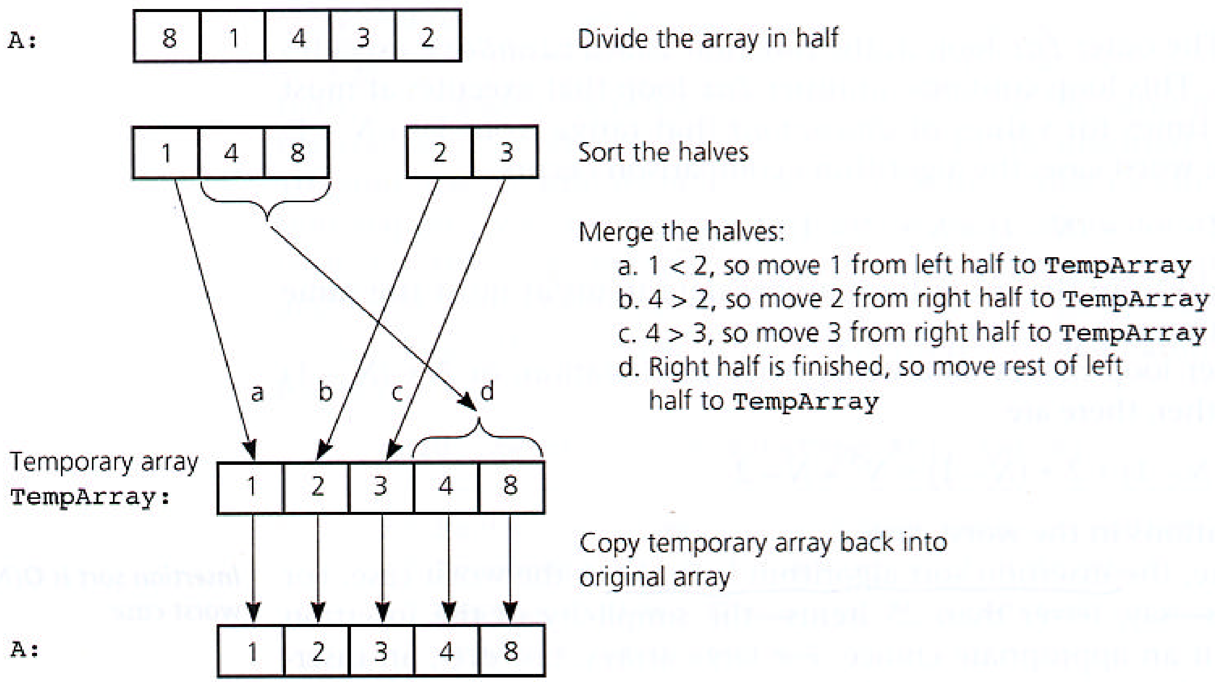


Figure 9-8

A mergesort with an auxiliary temporary array

Complexity

- Time - $O(N \log_2 N)$
- Space – twice the size of the sorted data

Quicksort

- Time complexity analysis
  - Average-case - $O(N*\log_2 N)$
  - Worse-case – $O(N^2)$

- Often used to sort large arrays
  - Usually extremely fast in practice
  - The original arrangement of data is “random”
- Has the best "average" behavior among all the sorting methods.

Quick Sort

```
quicksort(inout theArray:ItemArray, in first:integer, in
    last:integer)
// Sorts theArrray[first..last].
if (first < last)
{
  Choose a pivot item p from theArrray[first..last]
  Partition the items of theArrray[first..last] about p
  // the partition is theArrray[first..pivotIndex..last]
  quicksort(theArrray, first, pivotIndex-1)   // sort S1
  quicksort(theArray, pivotIndex+1, last)  // sort S2
}
// if  first >= last, there is nothing to do
```

Quick Sort: invariant of the partition algorithm

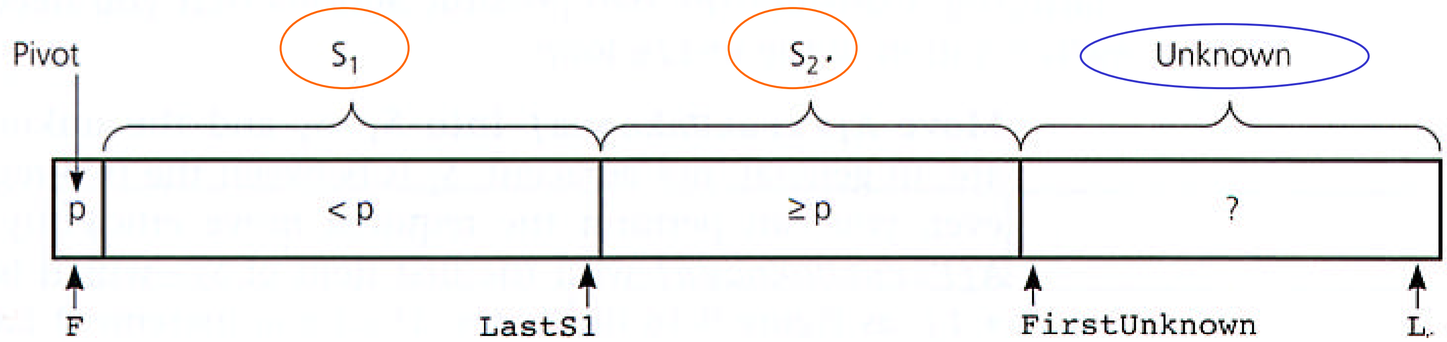


Figure 9-14

Invariant for the partition algorithm

Quick Sort: initial state of the array

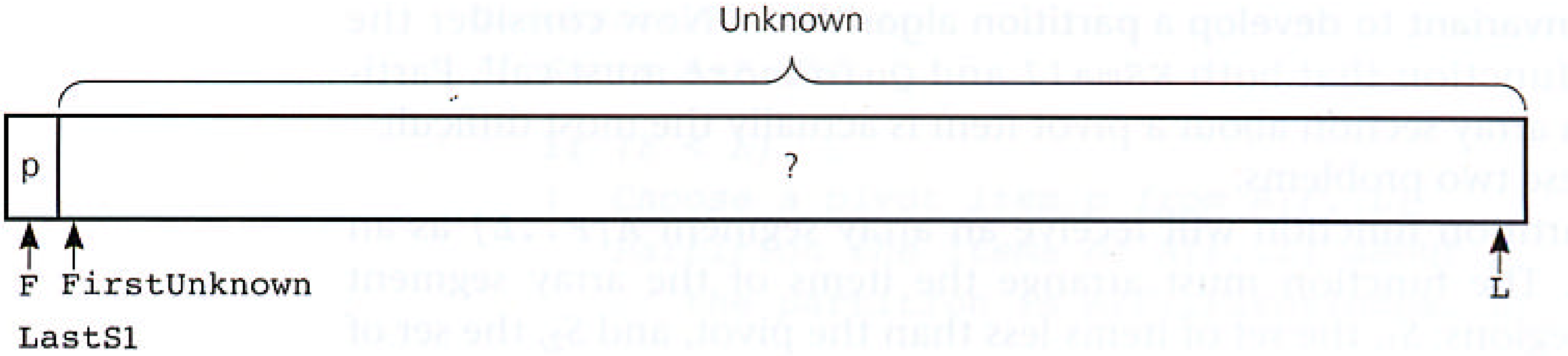


Figure 9-15

Initial state of the array

Quick Sort: swapping <

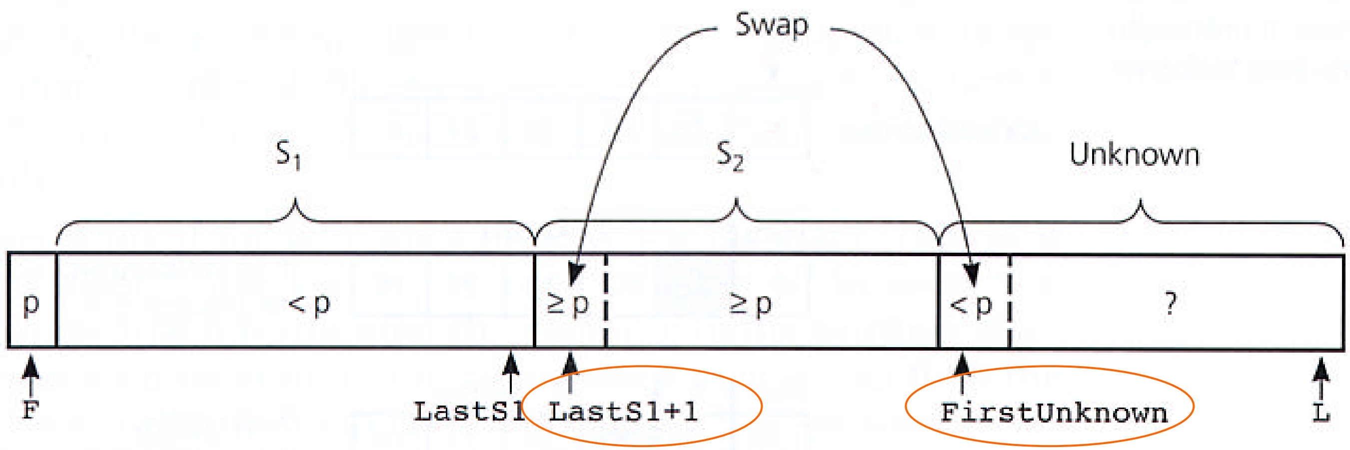


Figure 9-16

Moving `A[FirstUnknown]` into S_1 by swapping it with `A[LastS1+1]` and by incrementing both `LastS1` and `FirstUnknown`

Quick Sort: swapping >

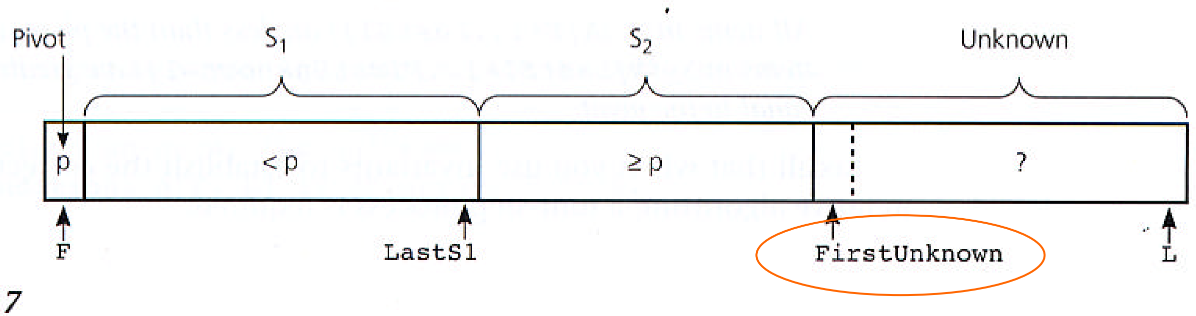


Figure 9-17

Moving **A[FirstUnknown]** into S_2 by incrementing **FirstUnknown**

Partition(cont'd)

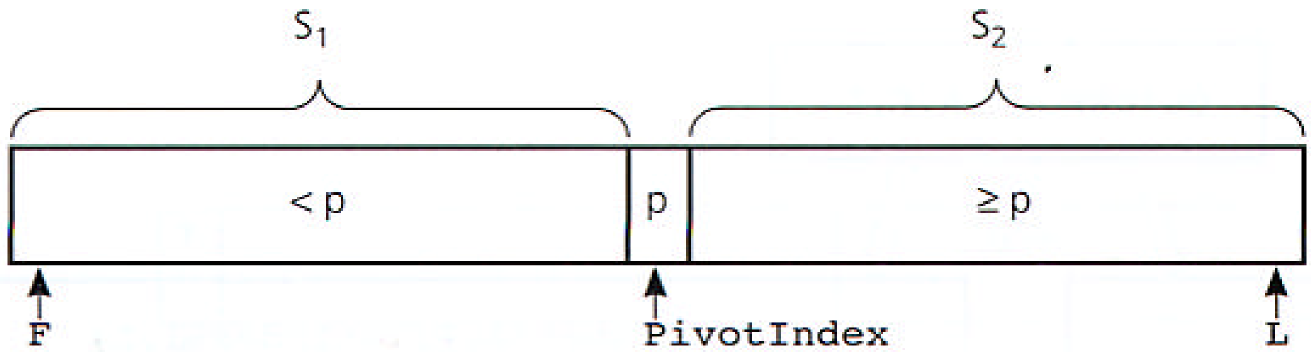


Figure 9-12

A partition about a pivot

Partition (page 467)

```
void partition(DataType theArray[], int first, int last, int
    & pivotIndex)
// ---------------------------------------------------------
// Partitions an array for quicksort.
// Precondition: theArray[first..last] is an array; first <=
    last.
// Postcondition: Partitions theArray[first..last] such that:
//    S1 = theArray[first..pivotIndex-1]  <    pivot
//             theArray[pivotIndex]            ==  pivot
//    S2 = theArray[pivotIndex+1..last] >=  Pivot
// Calls: choosePivot and swap.
// ---------------------------------------------------------
```

Partition (cont'd)

```
{  choosePivot(theArray, first, last);

   dataType pivot = theArray [first]; // copy pivot
   // index of last item in S1
   int lastS1 = first;
   // index of first item in unknown
   int firstUnknown = first+1;
```

Partition(cont'd)

```
// move one item at a time until unknown region is empty
 for (; firstUnknown <= last; ++firstUnknown)
 {  // Invariant: theArray[first..LastS1] < Pivot
    // theArray[LastS1+1..firstUnknown-1] >= Pivot
    // move item from unknown to proper region
    if (theArray[firstUnknown] < pivot)
    {  // item from unknown belongs in S1
       ++lastS1;
       swap(theArray[firstUnknown], theArray[lastS1]);
    }  // end if
    // else item from unknown belongs in S2
 }// end for
```

Partition(cont'd)

```
// place pivot in proper position and mark its
   location
   swap (theArray[first], (theArray[lastS1]);
   pivotIndex = lastS1;
}// end Partition
```

Partition - Example

lastS$_1$=0, firstUnknown=1

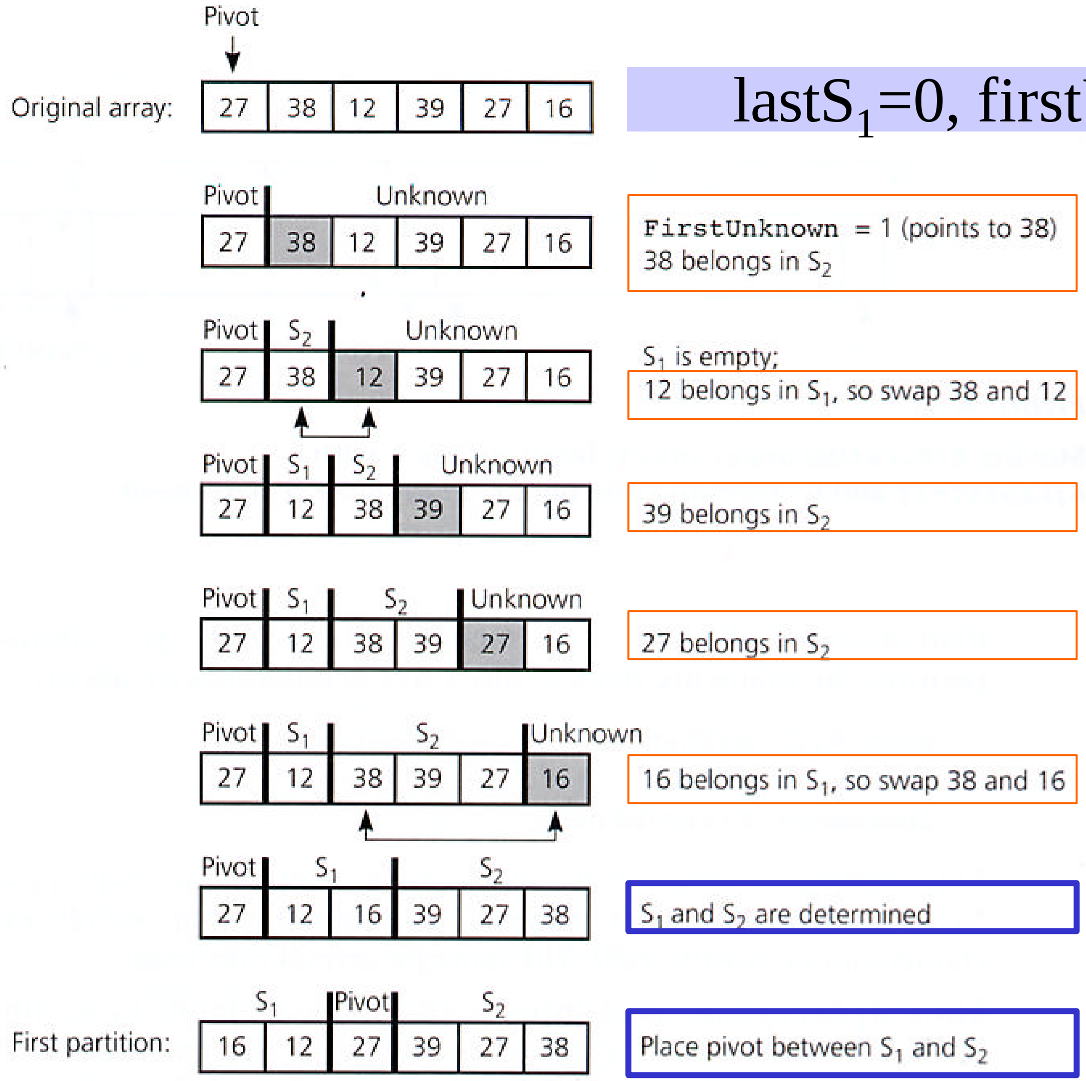


Figure 9-18

Developing the first partition of an array when the pivot is the first item

Partition - Example

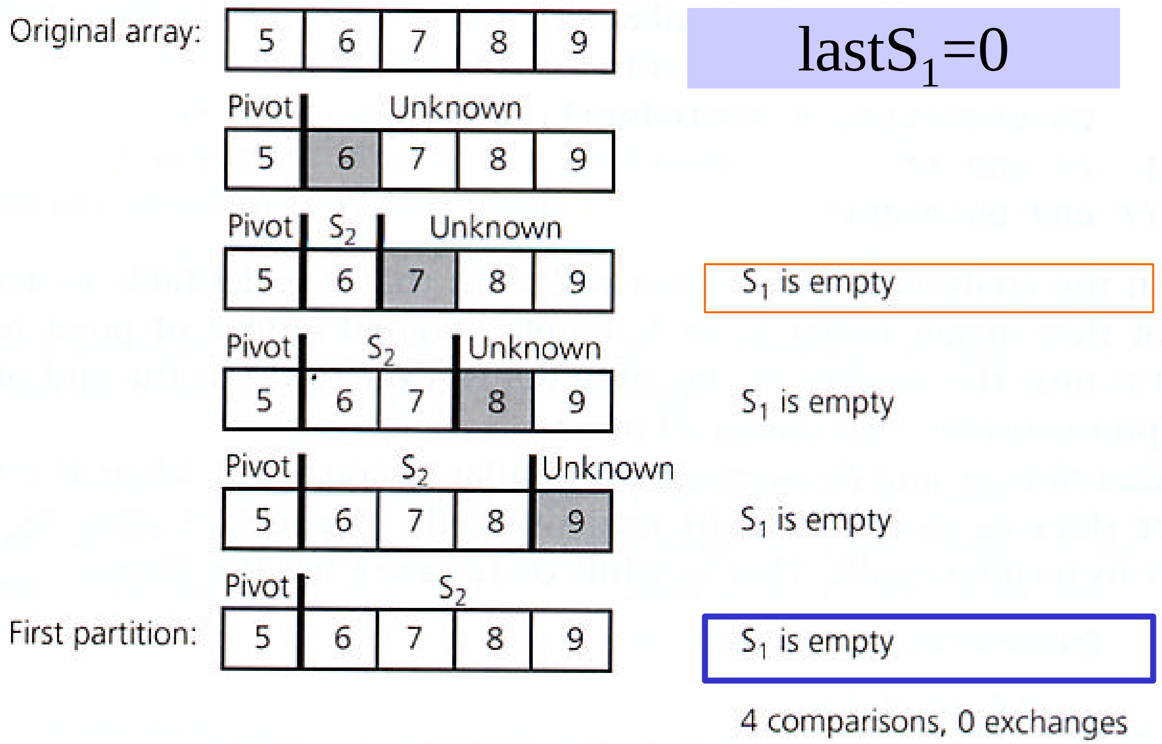


Figure 9-19

A worst-case partitioning with `Quicksort`

Quick Sort (page 468)

```
quicksort(DataType theArray, int first, int last)
{ // Sorts theArrray[first..last].
  if (first < last)
    { // create the partition: S1, pivot, S2
        partition[theArray, first, last, pivotIndex);

        // sort region S1 and S2
        quicksort(theArrray, first, pivotIndex-1);
        quicksort(theArray, pivotIndex+1, last);
     }  // end if
}  // end quicksort
```

Quick Sort - example

- Analysis
  - Worst-case: reverse sorted order $O(n^2)$
  - average case: $O(n \log_2 n)$
- Variation

QuickSort using a median of three

median (left, right, middle)

Heapsort (p.550)

- Use a heap to sort an array of items
- Algorithm
  - Transform the array into a heap - use HeapInsert to insert the items into the heap one by one

  Or,
  - Image the array as a complete binary tree
  - Transform the tree into a heap – use RebuildHeap
  - Call Rebuildheap on the leaves from right to left
  - Move up the tree
  - Until reach the root

Building a Heap from an Array of Items

```
for(Index = N-1 down to 0)
  // Assertion:  the tree rooted at Index is a semiheap
  RebuildHeap (A, Index, N)

  // Assertion: the tree rooted at Index is a heap
```

Heap Sort: Example

- Consider the array as a complete binary tree

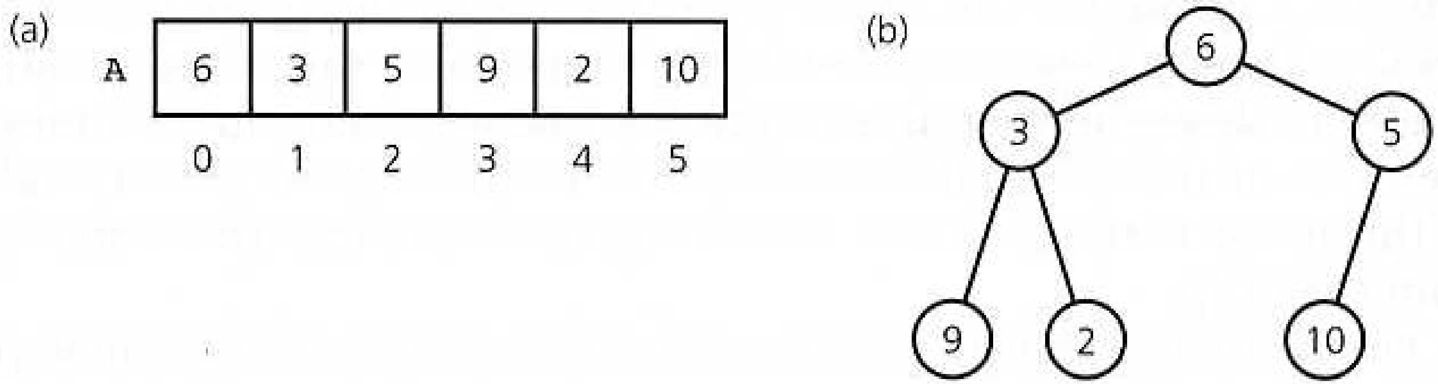


Figure 11-13

(a) The initial contents of A; (b) A's corresponding binary tree

Heap Sort: Transform an Array into a Heap

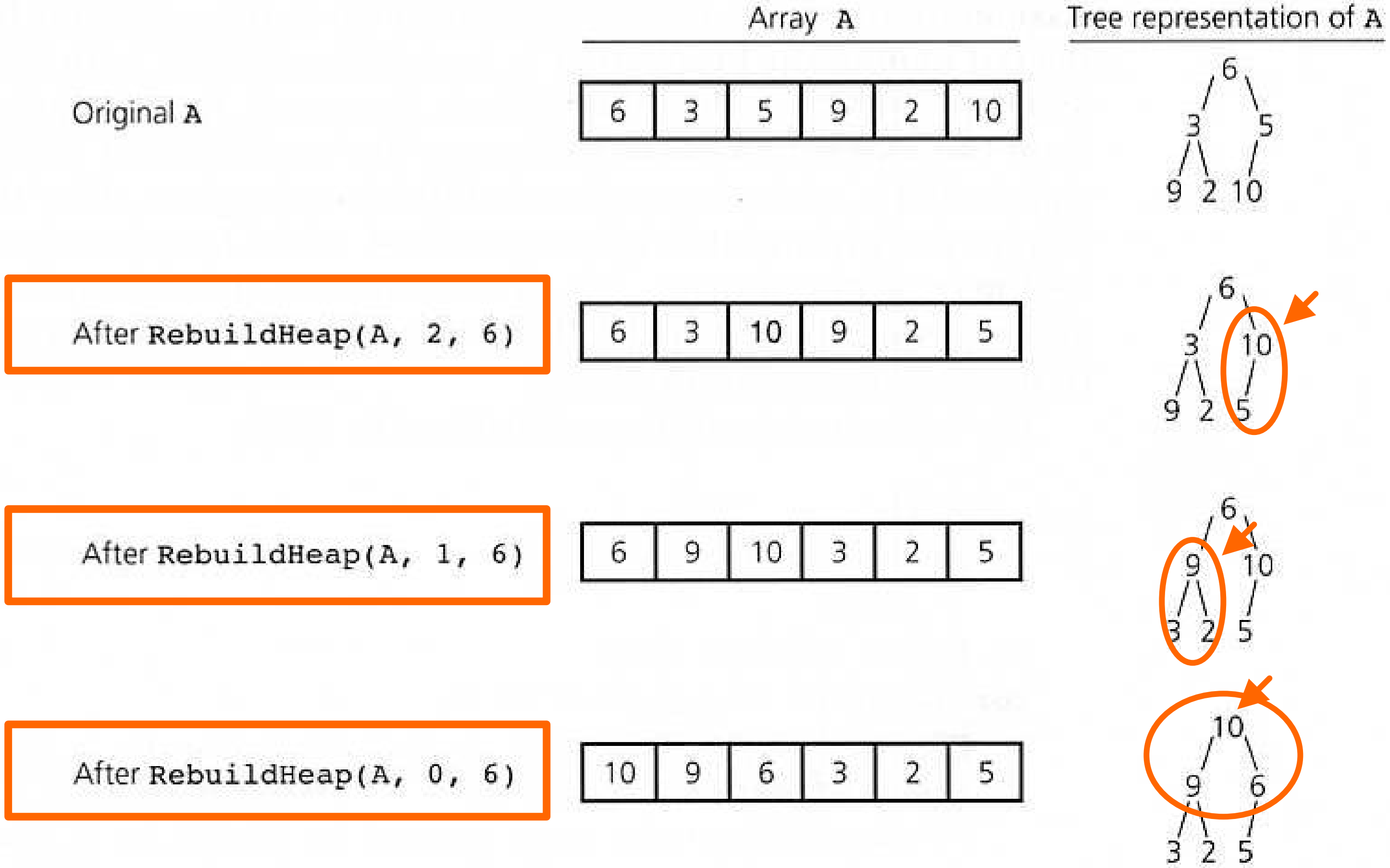


Figure 11-14

Transforming an array A into a heap

Heap Sort: Invariant

- After step k, the Sorted region contains the k largest values in A in sorted order, i.e.,
  - A[N-1] is the largest, A[N-2] is the second largest, and so on.
- The items in the Heap region form a heap.

Heap Sort: Partition an Array into Two Regions

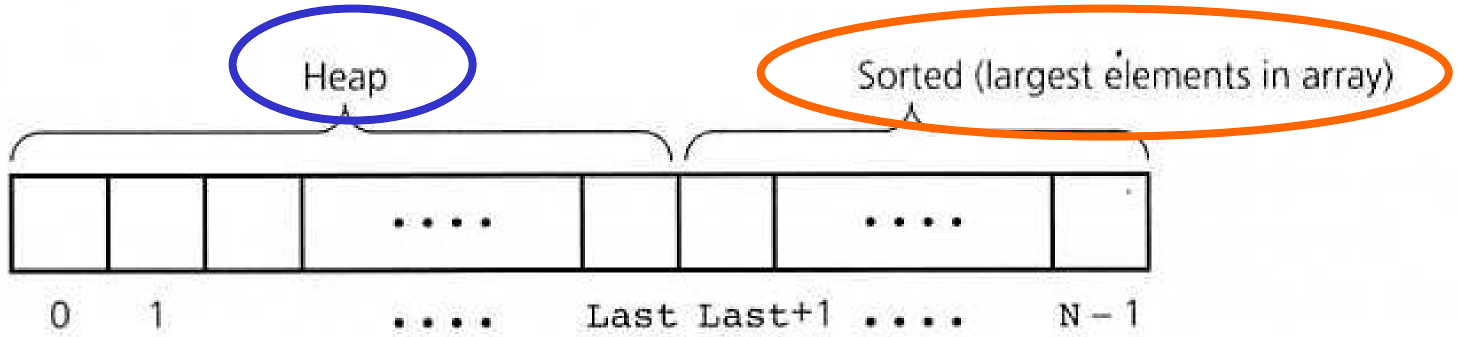
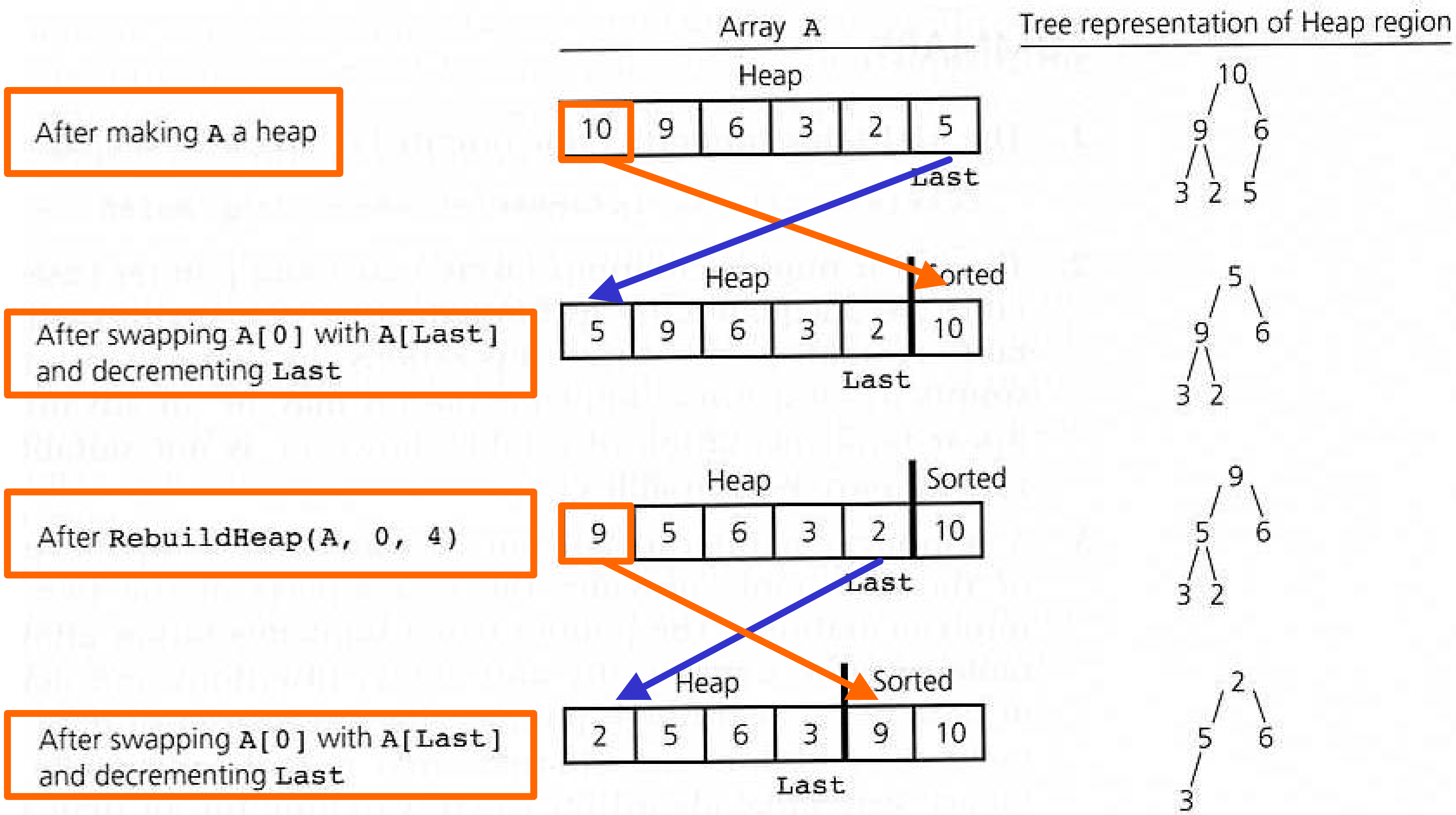


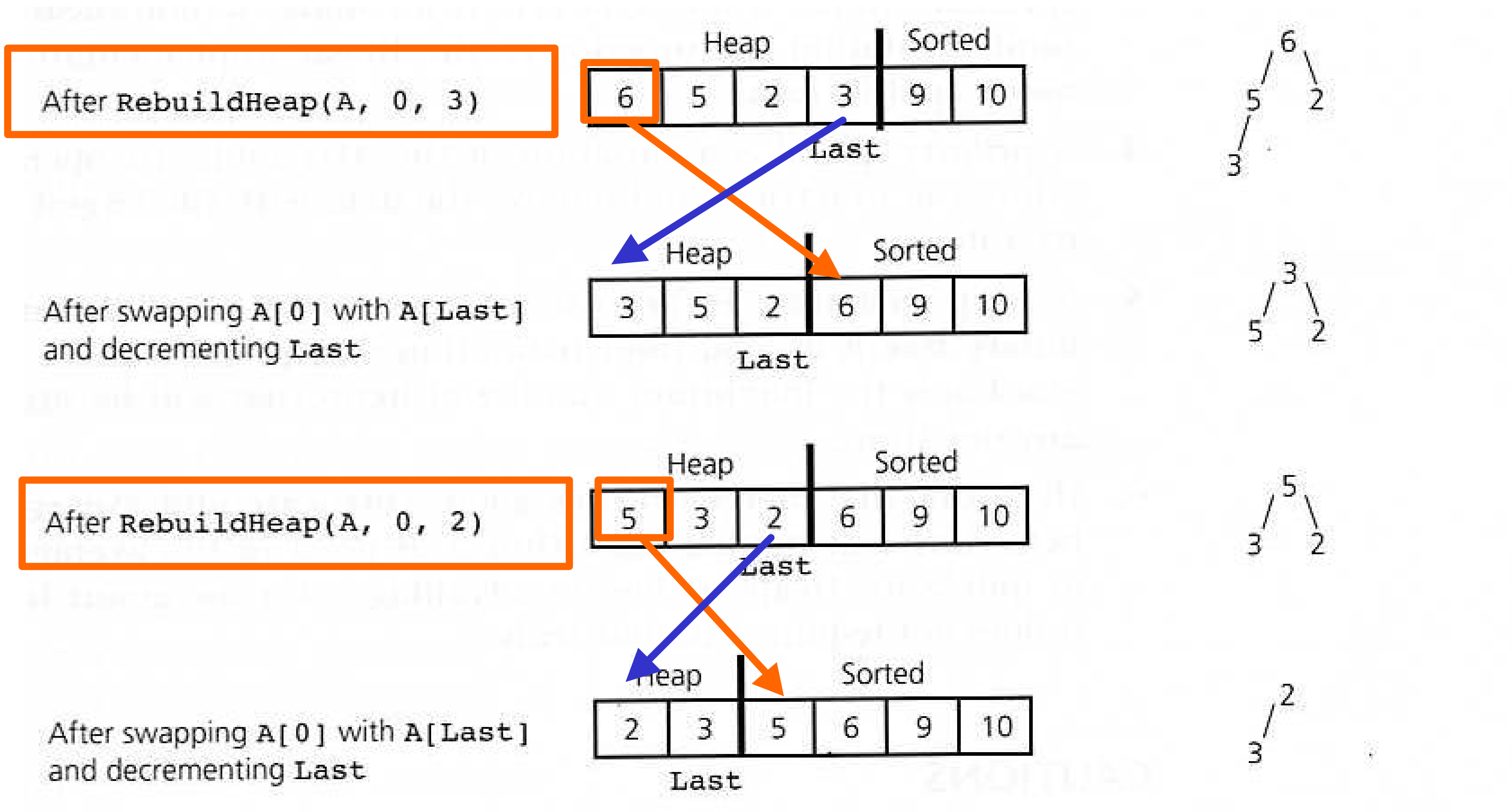
Figure 11-15

Heapsort partitions an array into two regions

Heap Sort: Example



Heap Sort: Example (cont’d)



Heap Sort: Example (cont'd)

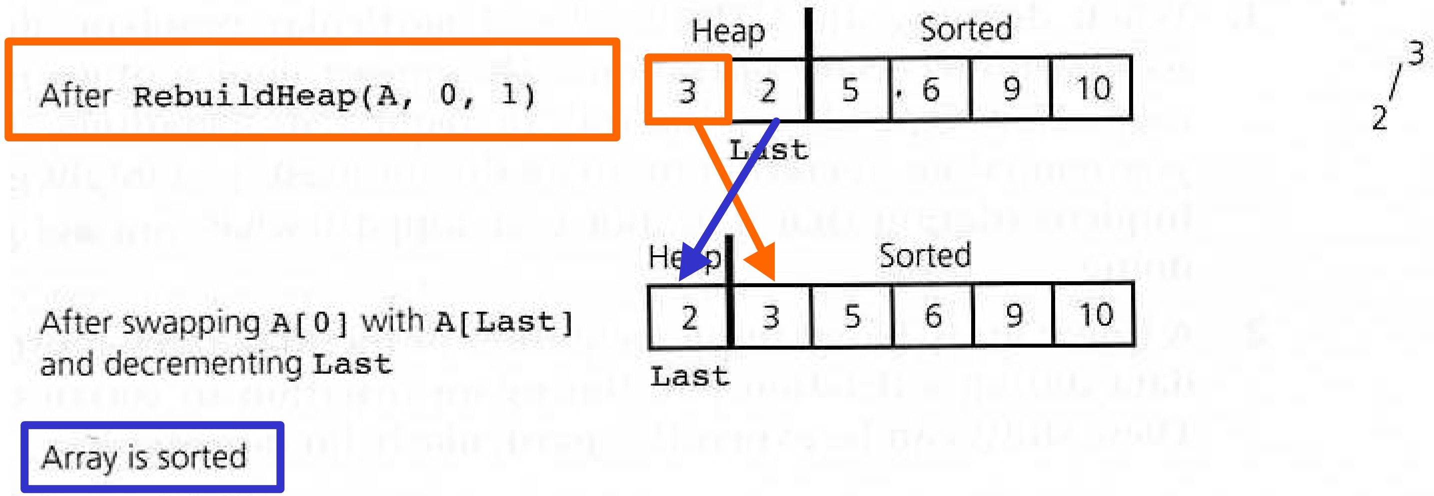


Figure 11-16
A trace of heapsort, beginning with the heap in Figure 11-14

Heap Sort: Algorithm

```
HeapSort(A, N)
  // Sorts A[0..N-1]
  // build initial heap
  for (index = N - 1 down to 0)
  {  // Invariant: the tree rooted at Index is a semiheap
     RebuildHeap(A, index, N)
     // Assertion: the tree rooted at index is a heap
  }
  // Assertion: A[0] is largest element in heap A[0..N-1]

  // initialize the regions
  Last = N - 1
```

Heap Sort: Algorithm (cont'd)

```
// Invariant: A[0..Last] is a heap, A[Last+1..N-1] is
// sorted and contains the largest elements of A

for (Step = 1 through N)
{  // move the largest item A[0] in the Heap region
   //  to the beginning of the Sorted region by swapping items
   Swap A[0] and A[Last]

   // expand the Sorted region, shrink the Heap region
   --Last

   // make the Heap region a heap again
   RebuildHeap(A, 0, Last)
}  // end for
```

Heap Sort

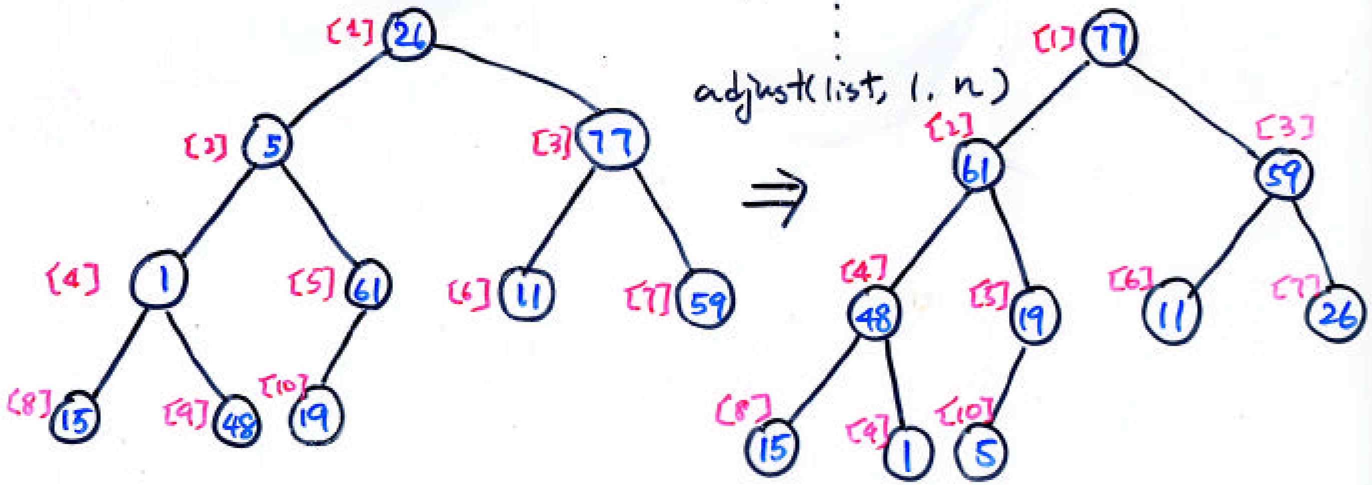
- Example

Input: a list stored in a one-dimensional array

1	2	3	4	5	6	7	8	9	10
26	5	77	1	61	11	59	15	48	19

Corresponding Binary Tree

adjust(list, n/2, n)
adjust(list, n/2 −1, n)
⋮
adjust(list, 1, n)



Treesort

- Algorithm
  - Use a binary search tree to sort an array of data items into search–key order
  - Traverse the tree “inorder”
- Analysis
  - Each item’s insertion takes $O(\log_2 N)$ or $\log(N)$
  - N times - $O(N*\log_2 N)$ or $O(N^2)$
  - Traversal $O(N)$

Treesort

treesort(A, N)

// Sorts the N integers in an array A into ascending order.

Insert A's elements into a binary search tree T

Traverse T inorder. As you visit T's nodes, copy their data portions into successive locations of A

Treesort uses a binary search tree

Treesort.

Average case: $O(N \log N)$

worst case: $O(N^2)$

Homeworks

Quick Sort Algorithm(cont'd)

- Developed by *C.A.R. Hoar*
- Has the best average behavior among all the sorting methods
- Input: a list of records: $R_{left}, R_{(left+1)}, \ldots, R_{right}$, left < right
- Assumptions:
  - $K_{left} \le K_{(right+1)}$
  - let pivot key be R_{right}
- Output of each step:

$$\forall \mathrm{j}, \mathrm{left} \le \mathrm{j} \le \mathrm{i}-1,\ k_j < k_i$$

$$\forall \mathrm{j}, \mathrm{i}+1 \le \mathrm{j} \le \mathrm{right},\ k_j \ge k_i$$

$$k_i = k_{pivot\_key} \quad (\text{i.e. initial } k_{left})$$

Quick Sort Algorithm

- Algorithm
  - step 1:
  - step 2: search from *i* and stop at a record whose key
    value is greater than or equal to ;
    let the position pointed by *i*
  - step 3: search from j and stop at a record whose key
    value is less than ;
    let the position pointed by *j*
  - step 4: If $i < j$ SWAP and and go to step 2
  - step 5: otherwise SWAP and
  - step 6: quicksort (list, left, j-1) /* all records with
    keys */
  - step 7: quicksort (list, j+1, right) /* all records with
    keys */

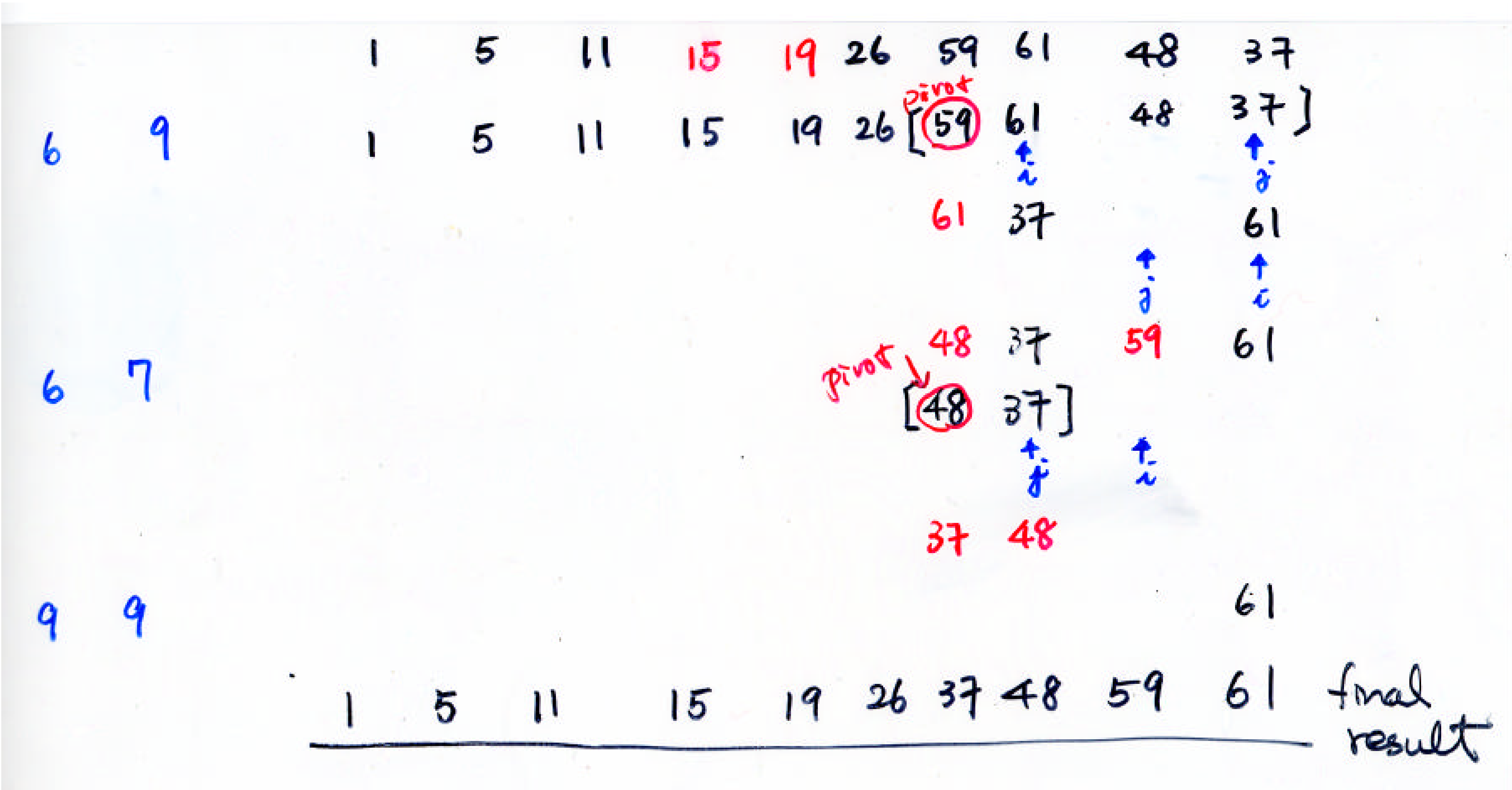
Quick Sort Algorithm

1. start from the first element and let it be the pivot element
2. scan from left to right and find an element greater than the pivot element
3. scan from right to left and find an element smaller than the pivot element
4. swap them
5. continue until two scans "cross“
6. swap pivot and the "cross" element
7. The resulting list has the left sublist smaller than the cross element(i.e. pivot element) and the right sublist greater than the cross element
8. repeat the QuickSort for left- and right-sublist

Quick Sort

Example

left	right	pivot	R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
		pivot	[26	5	37	1	61	11	59	15	48	19]
					i							j
0	9				19							37
							i			j		
							15			61		
								j	i			
			11	5	19	1	15	26	59	61	48	37
0	4	pivot	[11				]					
					i	j						
					1	19						
					j			i				
			1	5	11	19	15					
0	1	pivot	[1	5]	11	19	15	26	59	61	48	37
			j		i							
3	4	pivot	1	5	11	[19	15]	26	59	61	48	37
							j	i				


1 5 11 15 19 26 59 61 48 37
pivot
6 9 1 5 11 15 19 26 [59 61 48 37]
i j
61 37 61
j i
48 37 59 61
pivot
6 7 [48 37]
j i
37 48
9 9 61
1 5 11 15 19 26 37 48 59 61 final result

Quicksort Analysis

1. Partition

at level 0 $\quad N-1$ comparisons for N items

at level 1 $\quad N-3 \quad \left(= \left(\frac{N}{2}-1\right) + \left(\frac{N}{2}-1-1\right)\right) \qquad \left(\frac{N}{2}, \frac{N}{2}-1\right)$ items

at level m $\quad$ - 2^m calls to quicksort

- each $\left(\frac{N}{2^m}-1\right)$ comparisons
- total $2^m \cdot \left(\frac{N}{2^m}-1\right) - 1 = N - 2^m - 1$

$\Rightarrow$ Each level requires $O(N)$ operations.
There are either $\log_2 N$ or $1 + \log_2 N$ leve

$\Rightarrow O(N * \log_2 N)$

Quick Sort

- average time complexity $O(n \log_e n)$, $n \geq 2$

- Lemma 7.1: Let $T_{avg}(n)$ be the expected time for quicksort to sort a file with n records. Then there exists a constant k, s.t. $T_{avg}(n) \leq k\, n \log_e n$, $n \geq 2$

Proof:

1. In the call to quick(0, n-1),
   K_0 gets placed at position j; generates two sublists
   $(0, j-1)$ & $(j+1, n-1)$

   Their expected time: $T_{avg}(j) + T_{avg}(n-j-1)$

$$T_{avg}(n) \leq cn + \frac{1}{n} \sum_{j=0}^{n-1} \left[T_{avg}(j) + T_{avg}(n-j-1) \right]$$

$$= cn + \frac{2}{n} \sum_{j=0}^{n-1} T_{avg}(j) \quad , \quad n \geq 2 \qquad (7.1)$$

2. Assume $T_{avg}(0) \le b$

$T_{avg}(1) \le b$ for some constant b

Want to show $T_{avg}(n) \le k \cdot n \log_e n$ for $n \ge 2$

where $k = 2(b+c)$

proof by induction on n

(A) $n=2$ $\quad T_{avg}(2) \le 2c + 2b \le 2(b+c) \cdot 2 \log_e 2$

(B) assume $\quad T_{avg}(n) \le k \cdot n \log_e n$, for $1 \le n < m$ (B)

(C) $\quad T_{avg}(m) \le \overset{\downarrow (7.1)}{c} \cdot m + \frac{2}{m} \sum_{j=0}^{m-1} T_{avg}(j)$

$$\le c \cdot m + \frac{2}{m}(b+b) \overset{\downarrow (2)}{+} \frac{2}{m} \sum_{j=2}^{m-1} T_{avg}(j)$$

$$\le c \cdot m + \frac{4b}{m} + \frac{2}{m} \cdot k \cdot \sum_{j=2}^{m-1} j \log_e \overset{\downarrow (B)}{j}$$

$$\le c \cdot m + \frac{4b}{m} + \frac{2k}{m} \int_2^m x \log_e x \, dx$$

$$m\left(c+\frac{4b}{m^2}-\frac{k}{2}\right)\leq 0 \quad (*)$$

$k \geq 2c + 8b/m^2$

$k = 2(b+c)$

$2b = 8b/m^2$

$m=2$

for $m>2$, $*$ always holds

$$= c\cdot m + \frac{4b}{m} + \frac{2k}{m}\left[\frac{m^2\log_e m}{2} - \frac{m^2}{4}\right]$$

$$= c\cdot m + \frac{4b}{m} + k\cdot m\log_e m - \frac{k\cdot m}{2}$$

$$\leq k\cdot m\log_e m$$

Quick Sort - example

- Example: 26, 5, 37, 1, 61, 11, 59, 15, 48, 19
- Analysis
  - worse case: reverse sorted order $O(n^2)$
  - average case: $O(n \log_2 n)$
- Variation

  QuickSort using a median of three

  median (left, right, middle)