Sort Algorithms

Humayun Kabir
Professor, CS, Vancouver Island University, BC, Canada

Sorting

- ***Sorting*** is a process that organizes a collection of data into either ascending or descending order.
- An ***internal sort*** requires that the collection of data fit entirely in the computer’s main memory.
- We can use an ***external sort*** when the collection of data cannot fit in the computer’s main memory all at once but must reside in secondary storage such as on a disk.
- We will analyze only internal sorting algorithms.

Sorting

- Any significant amount of computer output is generally arranged in some sorted order so that it can be interpreted.
- Sorting also has indirect uses. An initial sort of the data can significantly enhance the performance of an algorithm.
- Majority of programming projects use a sort somewhere, and in many cases, the sorting cost determines the running time.
- A comparison-based sorting algorithm makes ordering decisions only on the basis of comparisons.

Sorting Algorithms

- There are many comparison based sorting algorithms, such as:
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Merge Sort
  - Quick Sort

Bubble Sort

- The list is divided into two sublists: sorted and unsorted.
- Starting from the bottom of the list, the smallest element is bubbled up from the unsorted list and moved to the sorted sublist.
- After that, the wall moves one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
- Each time an element moves from the unsorted part to the sorted part one sort pass is completed.
- Given a list of n elements, bubble sort requires up to n-1 passes to sort the data.

Bubble Sort

23	78	45	8	32	56	Original List
8	23	78	45	32	56	After pass 1
8	23	32	78	45	56	After pass 2
8	23	32	45	78	56	After pass 3
8	23	32	45	56	78	After pass 4

Bubble Sort Algorithm

```
void swap( int &lhs, int &rhs );

void bubleSort(int a[], int n) {
   bool sorted = false;
   int last = n-1;
   for (int i = 0; (i < last) && !sorted; i++){
      sorted = true;
      for (int j=last; j > i; j--)
         if (a[j-1] > a[j]{
            swap(a[j],a[j-1]);
            sorted = false; // signal exchange
         }
    }
}

void swap( int &lhs, int &rhs ){
   int tmp = lhs;
   lhs = rhs;
   rhs = tmp;
}
```

Bubble Sort – Analysis

- In general, we compare keys and move items (or exchange items) in a sorting algorithm (which uses key comparisons).

➔ **So, to analyze a sorting algorithm we should count the number of key comparisons and the number of moves.**

  - Ignoring other operations does not affect our final result.

Bubble Sort – Analysis

- ***Best-case:*** ➔ **O(n)**
  - Array is already sorted in ascending order.
  - Outer loop executes 1 time and inner loop n-1 times.
  - The number of moves: 0 ➔ O(1)
  - The number of key comparisons: (n-1) ➔ O(n)
- ***Worst-case:*** ➔ **$O(n^2)$**
  - Array is in reverse order:
  - Outer loop is executed n-1 times and inner loop executes (n-1-i) times,
  - The number of moves: 3*((n-1)+(n-2)+...+3+2+1) = 3 * n*(n-1)/2 ➔ $O(n^2)$
  - The number of key comparisons: ((n-1)+(n-2)+...+3+2+1) = n*(n-1)/2 ➔ $O(n^2)$
- **Average-case**: ➔ **$O(n^2)$**
  - We have to look at all possible initial data organizations.
- **So, Bubble Sort is $O(n^2)$**

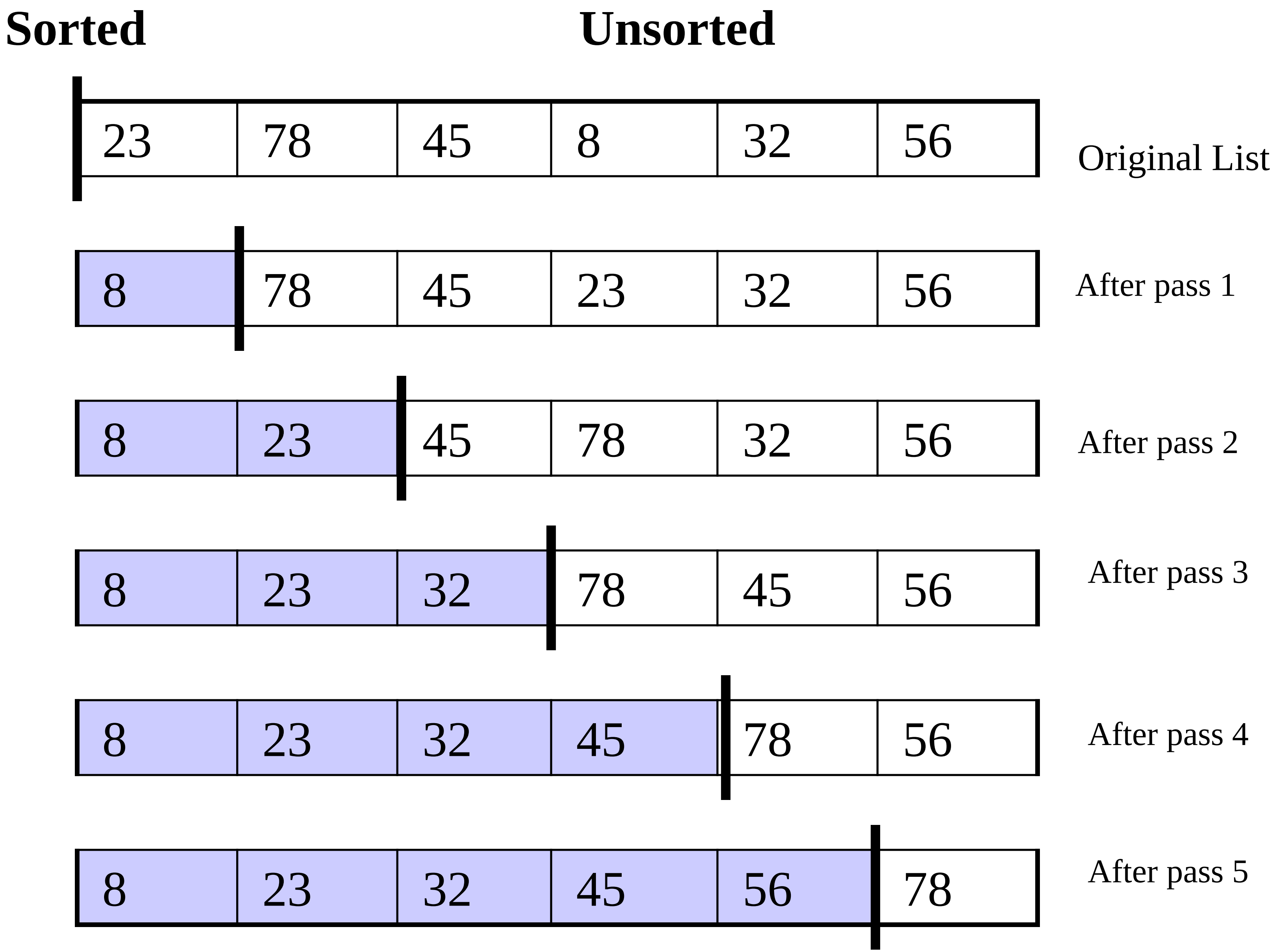
Comparison of *N*, *logN* and *N²*

N	O(LogN)	O(N^2)
16	4	256
64	6	4K
256	8	64K
1,024	10	1M
16,384	14	256M
131,072	17	16G
262,144	18	6.87E+10
524,288	19	2.74E+11
1,048,576	20	1.09E+12
1,073,741,824	30	1.15E+18

Selection Sort

- The list is divided into two sublists, *sorted* and *unsorted*, which are divided by an imaginary wall.
- We find the smallest element from the unsorted sublist and swap it with the element at the beginning of the unsorted data.
- After each selection and swapping, the imaginary wall between the two sublists move one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
- Each time we move one element from the unsorted sublist to the sorted sublist, we say that we have completed a sort pass.
- A list of *n* elements requires *n-1* passes to completely rearrange the data.

Selection Sort



Selection Sort

```
void swap( int &lhs, int &rhs );

void selectionSort( int a[], int n) {
  for (int i = 0; i < n-1; i++) {
    int min = i;
    for (int j = i+1; j < n; j++){
        if (a[j] < a[min]) min = j;
    }
    swap(a[i], a[min]);
  }
}
```

Selection Sort -- Analysis

- In selectionSort function, the outer for loop executes n-1 times.
- We invoke swap function once at each iteration.

  ➔ Total Swaps: n-1

  ➔ Total Moves: 3*(n-1) (Each swap has three moves)

Selection Sort – Analysis (cont.)

- The inner for loop executes the size of the unsorted part minus 1 (n-1-i), and in each iteration we make one key comparison.

  ➔ # of key comparisons = ((n-1)+(n-2)+...+3+2+1) = n*(n-1)/2

  ➔ **So, Selection sort is O(n^2)**

- The best case, the worst case, and the average case of the selection sort algorithm are same. ➔ all of them are **O(n^2)**
  - This means that the behavior of the selection sort algorithm does not depend on the initial organization of data.
  - Since O(n^2) grows so rapidly, the selection sort algorithm is appropriate only for small n.
  - Although the selection sort algorithm requires O(n^2) key comparisons, it only requires O(n) moves.
  - A selection sort could be a good choice if data moves are costly but key comparisons are not costly (short keys, long records).

Insertion Sort

- Insertion sort is a simple sorting algorithm that is appropriate for small inputs.
  - Most common sorting technique used by card players.
- The list is divided into two parts: sorted and unsorted.
- In each pass, the first element of the unsorted part is picked up, transferred to the sorted sublist, and inserted at the appropriate place.
- A list of *n* elements will take at most *n-1* passes to sort the data.

Insertion Sort

Sorted			Unsorted			
23	78	45	8	32	56	Original List
23	78	45	8	32	56	After pass 1
23	45	78	8	32	56	After pass 2
8	23	45	78	32	56	After pass 3
8	23	32	45	78	56	After pass 4
8	23	32	45	56	78	After pass 5

Insertion Sort Algorithm

```
void insertionSort(int a[], int n) {
    for (int i = 1; i < n; i++){
        int tmp = a[i];
        int j = i;
        for (; j>0 && tmp < a[j-1]; j--){
            a[j] = a[j-1];
        }
        a[j] = tmp;
    }
}
```

Insertion Sort – Analysis

- Running time depends on not only the size of the array but also the contents of the array.
- ***Best-case:*** ➔ **O(n)**
  - Array is already sorted in ascending order.
  - Inner loop will not be executed.
  - The number of moves: 2*(n-1) ➔ O(n)
  - The number of key comparisons: (n-1) ➔ O(n)
- ***Worst-case:*** ➔ **$O(n^2)$**
  - Array is in reverse order:
  - Inner loop is executed i-1 times, for i = 2,3, …, n
  - The number of moves: 2*(n-1)+(1+2+...+n-1)= 2*(n-1)+ n*(n-1)/2 ➔ $O(n^2)$
  - The number of key comparisons: (1+2+...+n-1)= n*(n-1)/2 ➔ $O(n^2)$
- Average-case: ➔ **$O(n^2)$**
  - We have to look at all possible initial data organizations.
- **So, Insertion Sort is $O(n^2)$**

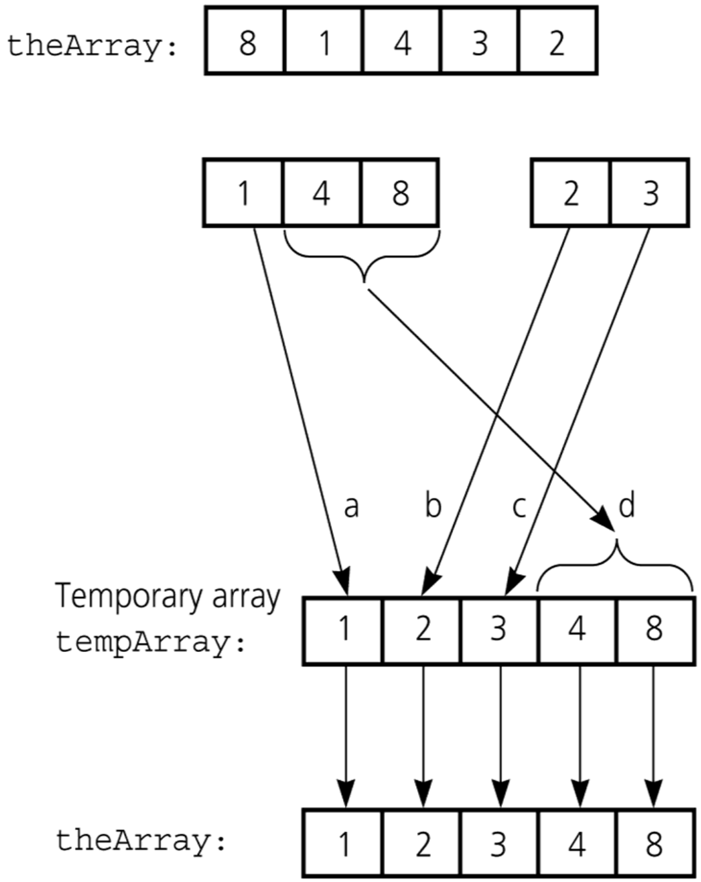
Analysis of Insertion sort

- Which running time will be used to characterize this algorithm?
  - Best, worst or average?
- Worst:
  - Longest running time (this is the upper limit for the algorithm)
  - It is guaranteed that the algorithm will not be worse than this.
- Sometimes we are interested in average case. But there are some problems with the average case.
  - It is difficult to figure out the average case. i.e. what is average input?
  - Are we going to assume all possible inputs are equally likely?
  - In fact for most algorithms average case is same as the worst case.

Mergesort

- Mergesort algorithm is one of two important divide-and-conquer sorting algorithms (the other one is quicksort).
- It is a recursive algorithm.
  - Divides the list into halves,
  - Sort each halve separately, and
  - Then merge the sorted halves into one sorted array.

Mergesort - Example



Divide the array in half

Sort the halves

Merge the halves:

a. $1 < 2$, so move 1 from left half to `tempArray`
b. $4 > 2$, so move 2 from right half to `tempArray`
c. $4 > 3$, so move 3 from right half to `tempArray`
d. Right half is finished, so move rest of left half to `tempArray`

Copy temporary array back into original array

Merge Sort

```
void merge(int theArray[], int first, int mid, int last) {
   int tempArray[last+1];  // temporary array
   int first1 = first;     // beginning of first subarray
   int last1 = mid;        // end of first subarray
   int first2 = mid + 1;   // beginning of second subarray
   int last2 = last;       // end of second subarray
   int index = first1; // next available location in tempArray
   for ( ; (first1 <= last1) && (first2 <= last2); ++index) {
      if (theArray[first1] < theArray[first2]) {
         tempArray[index] = theArray[first1];
         ++first1;
      }
      else {
          tempArray[index] = theArray[first2];
          ++first2;
      }
  }
```

//Continued to the next page…………..

Merge Sort (cont.)

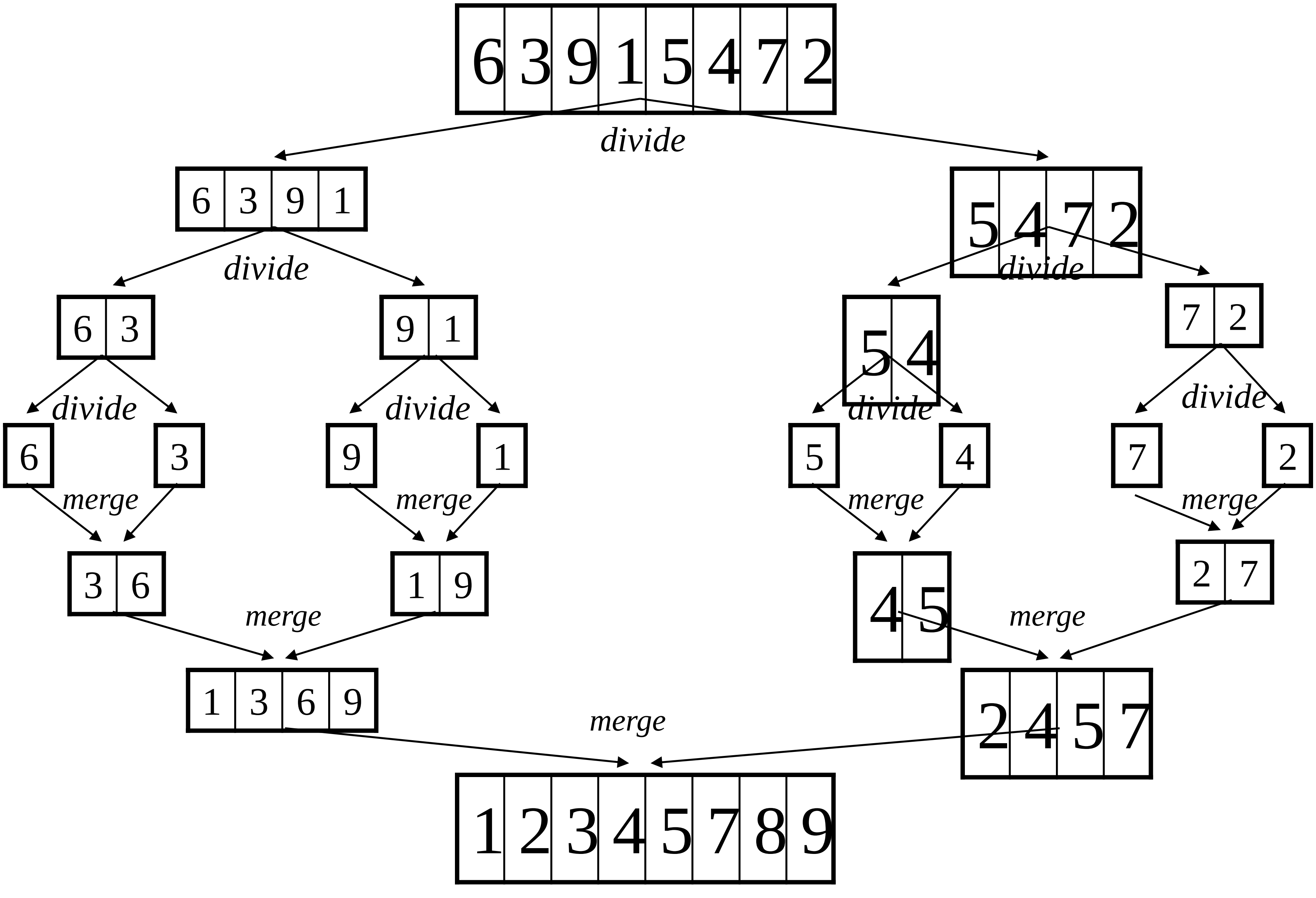
//Continued from the previous page…………

```
// finish off the first subarray, if necessary
for (; first1 <= last1; ++first1, ++index)
   tempArray[index] = theArray[first1];

// finish off the second subarray, if necessary
for (; first2 <= last2; ++first2, ++index)
   tempArray[index] = theArray[first2];

// copy the result back into the original array
for (index = first; index <= last; ++index)
   theArray[index] = tempArray[index];
}
```

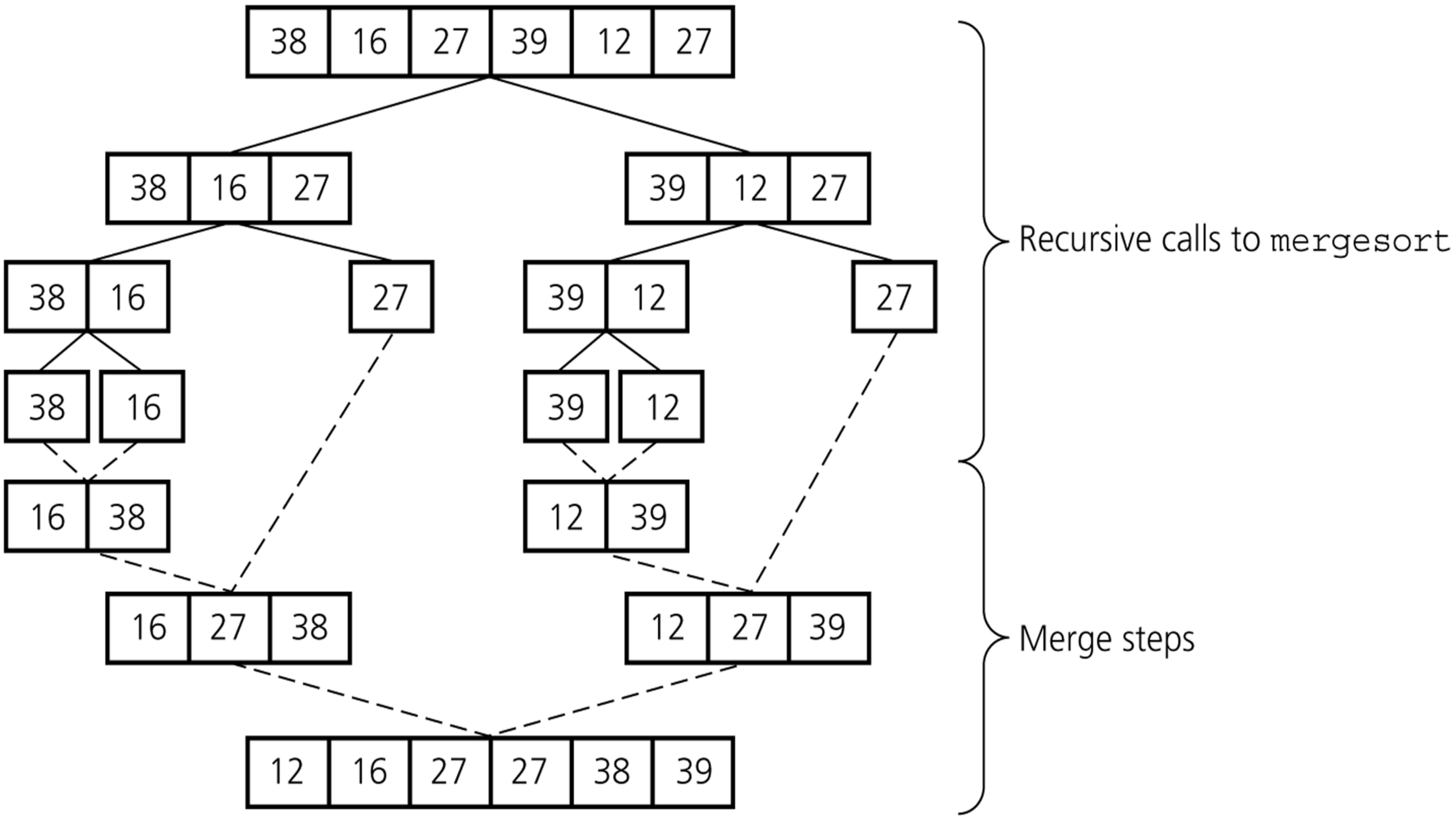
Merge Sort

```
void mergesort(int theArray[], int first, int last) {
   if (first < last) {
      int mid = (first + last)/2; // index of midpoint

      // dived into two halves at the middle
      mergesort(theArray, first, mid);
      mergesort(theArray, mid+1, last);

      // merge the two halves
      merge(theArray, first, mid, last);
   }
}
```

Merge Sort - Example



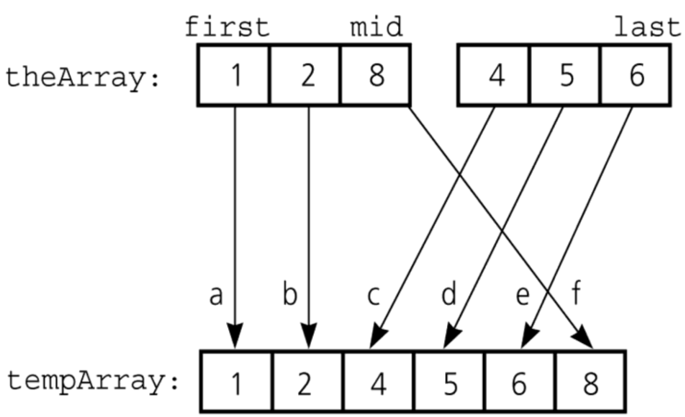
Mergesort – Example2



Mergesort – Analysis of Merge

A worst-case instance of the merge step in *mergesort*

Some elements in the first array are smaller and some elements are larger than all the elements in the second array

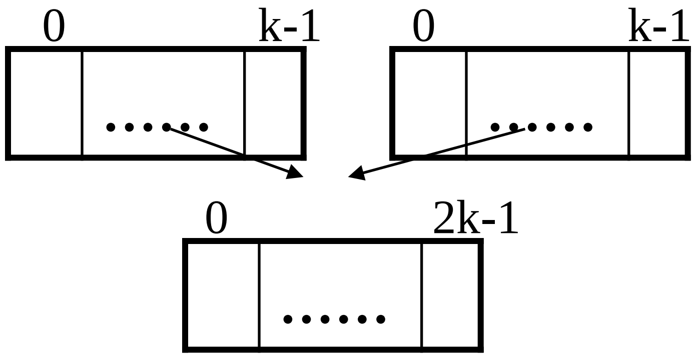


Merge the halves:

a. 1 < 4, so move 1 from `theArray[first..mid]` to `tempArray`
b. 2 < 4, so move 2 from `theArray[first..mid]` to `tempArray`
c. 8 > 4, so move 4 from `theArray[mid+1..last]` to `tempArray`
d. 8 > 5, so move 5 from `theArray[mid+1..last]` to `tempArray`
e. 8 > 6, so move 6 from `theArray[mid+1..last]` to `tempArray`
f. `theArray[mid+1..last]` is finished, so move 8 to `tempArray`

Mergesort – Analysis of Merge (cont.)

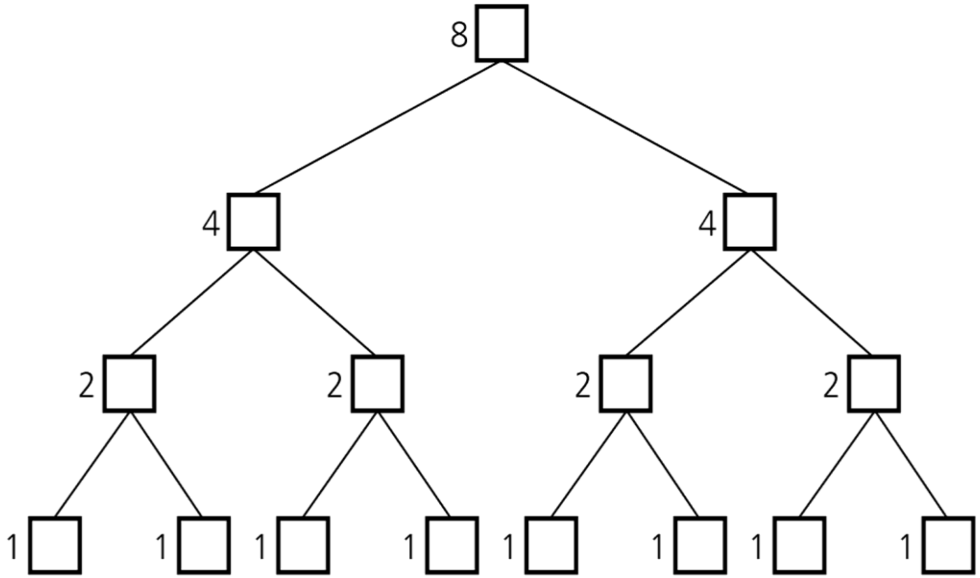
Merging two sorted arrays of size ***k***



- ***Best-case:***
  - All the elements in the first array are smaller (or larger) than all the elements in the second array.
  - The number of moves: 2k + 2k
  - The number of key comparisons: k
- ***Worst-case:***
  - The number of moves: 2k + 2k
  - The number of key comparisons: 2k-1

Mergesort - Analysis

Levels of recursive calls to *mergesort*, given an array of eight items



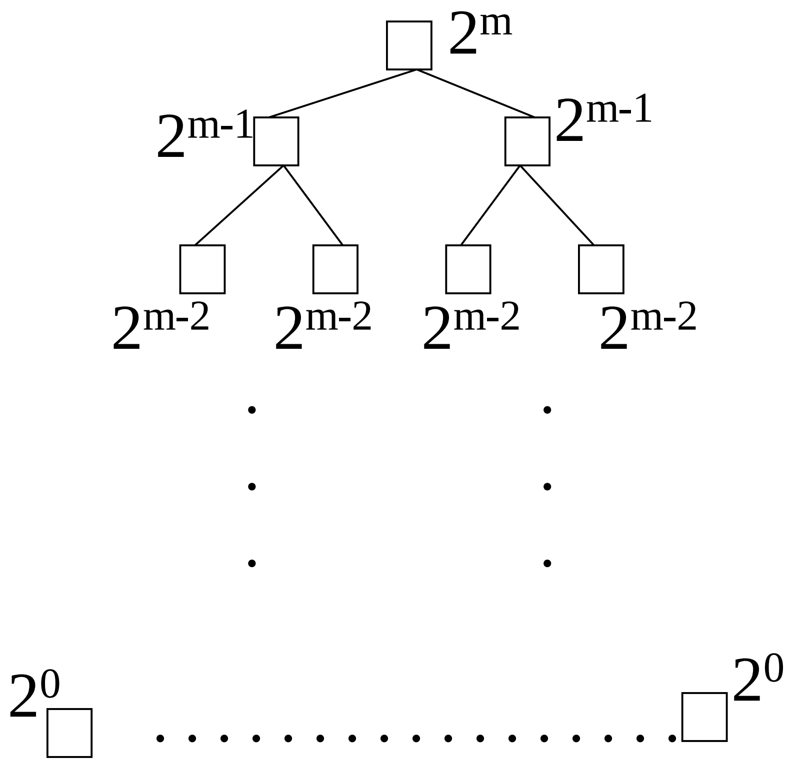
Level 0: `mergesort` 8 items

Level 1: 2 calls to `mergesort` with 4 items each

Level 2: 4 calls to `mergesort` with 2 items each

Level 3: 8 calls to `mergesort` with 1 item each

Mergesort - Analysis



level 0 : 1 merge (size 2^{m-1})

level 1 : 2 merges (size 2^{m-2})

level 2 : 4 merges (size 2^{m-3})

level i : 2^i merges (size 2^{m-i-1})

level m-1 : 2^{m-1} merges (size 2^0)

Mergesort - Analysis

- ***Worst-case –***

The number of key comparisons:

$= 2^0*(2*2^{m-1}-1) + 2^1*(2*2^{m-2}-1) + \ldots + 2^{m-1}*(2*2^0-1)$

$= (2^m - 2^0) + (2^m - 2^1) + \ldots + (2^m - 2^{m-1})$ (m terms)

$= m2^m - (2^0 + 2^1 + \ldots\ldots + 2^{m-1})$

$= m*2^m - \sum_{i=0}^{m-1} 2^i$

$= m*2^m - 2^m - 1$

Using m = log n

$\mathbf{= n * \log_2 n - n - 1}$

➔ $\mathbf{O(n * \log_2 n)}$

Mergesort – Analysis

- Mergesort is extremely efficient algorithm with respect to time.
  - Both worst case and average cases are **O (n * $\log_2$n)**

- But, mergesort requires an extra array whose size equals to the size of the original array.

Quicksort

- Like mergesort, Quicksort is also based on the *divide-and-conquer* paradigm.
- But it uses this technique in a somewhat opposite manner, as all the hard work is done *before* the recursive calls.
- It works as follows:

1. First, it partitions an array into two parts with respect to a pivot,
2. Then, it sorts the parts independently,
3. Finally, it combines the sorted subsequences by a simple concatenation.

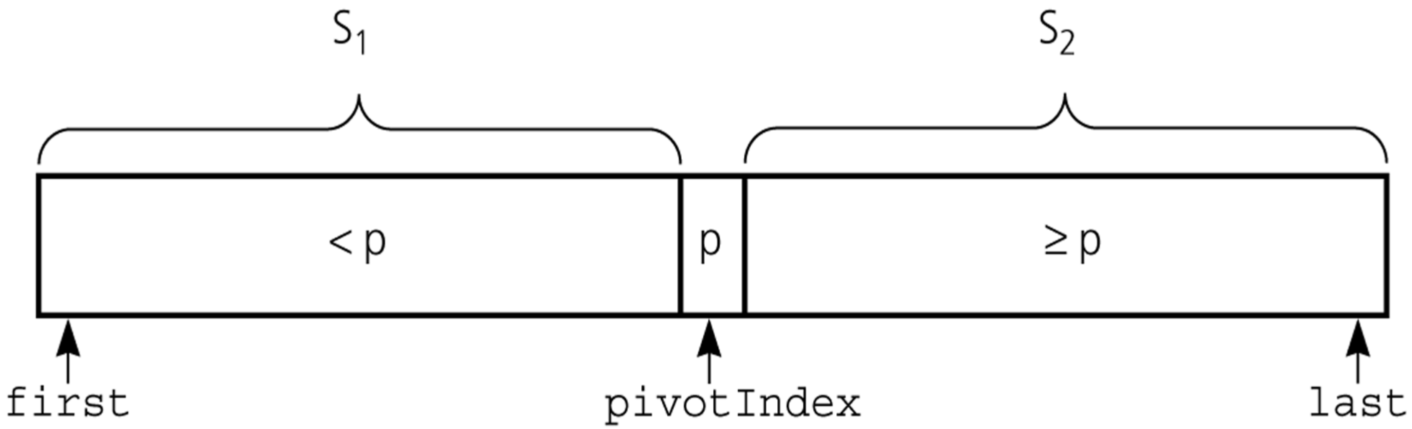
Quicksort (cont.)

The quick-sort algorithm consists of the following three steps:

1. ***Divide***: Partition the list.
   - To partition the list, we first choose some element from the list for which we hope about half the elements will come before and half after. Call this element the ***pivot***.
   - Then we partition the elements so that all those with values less than the pivot come in one sublist and all those with greater values come in another.
2. ***Recursion***: Recursively sort the sublists separately.
3. ***Conquer***: Put the sorted sublists together.

Quick Sort Partition

- Partitioning places the pivot in its correct place position within the array.




```
Partitions theArray[first..last] such that:
S1 = theArray[first..pivotIndex-1] < pivot
     theArray[pivotIndex] == pivot
S2 = theArray[pivotIndex+1..last] >= pivot
```

Quick Sort Partition

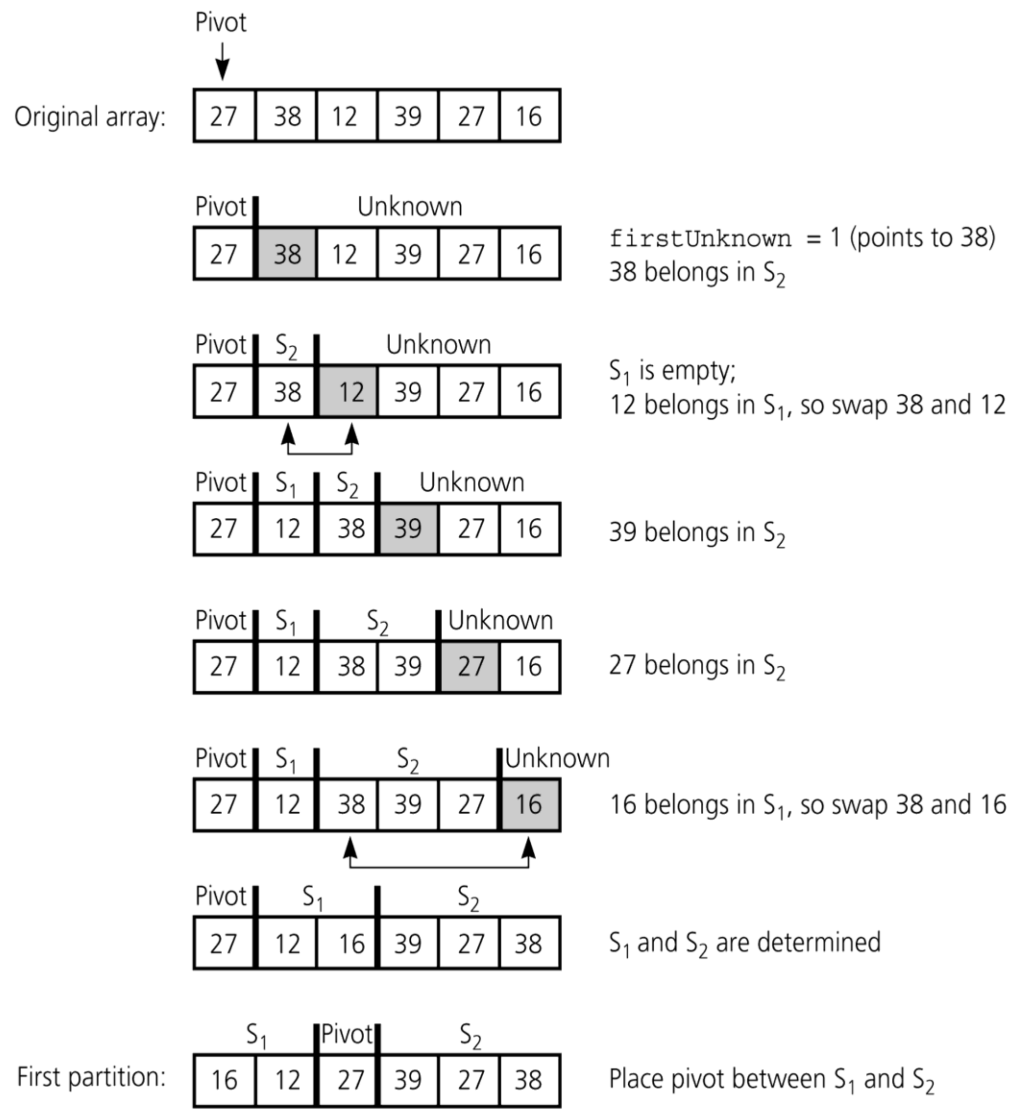
- Generates two smaller sorting problems.
  - Sort the left section of the array
  - Sort the right section of the array
  - Two smaller sorting problems are solved recursively to solve bigger sorting problem.

Quick Sort Partition: Choosing Pivot

- Which array item should be selected as pivot?
  - Somehow we have to select a pivot, and we hope that we will get a good partitioning.
  - If the items in the array arranged randomly, we choose a pivot randomly.
  - We can choose the first or last element as a pivot (it may not give a good partitioning).
  - We can use different techniques to select the pivot.
- Put this pivot into the first location of the array before partitioning

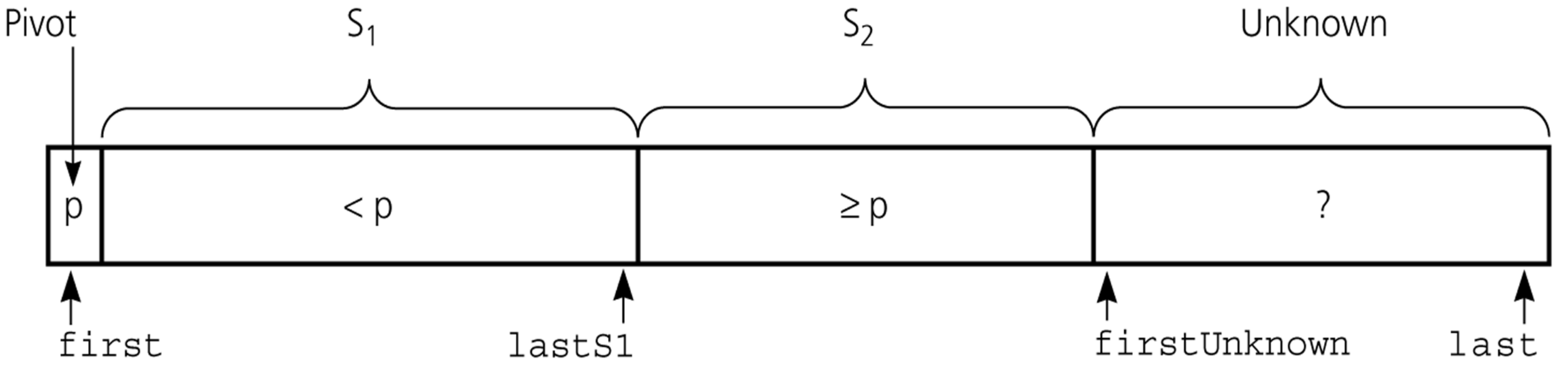
Partition (cont.)

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



Partition (cont.)

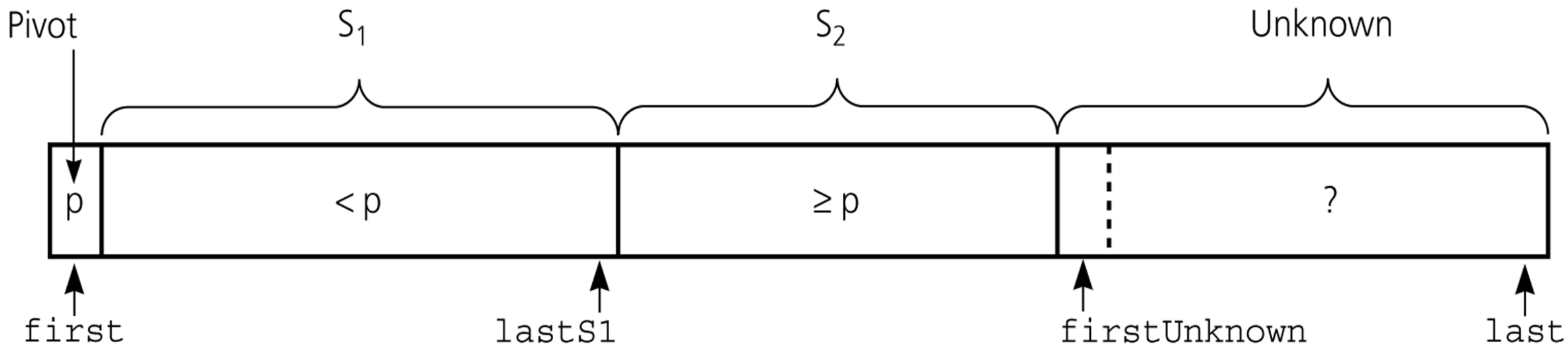
Invariant for the partition algorithm




```
S1: theArray[first+1..lastS1] < pivot
S2: theArray[lastS1+1..firstUnknown-1] >= pivot
```

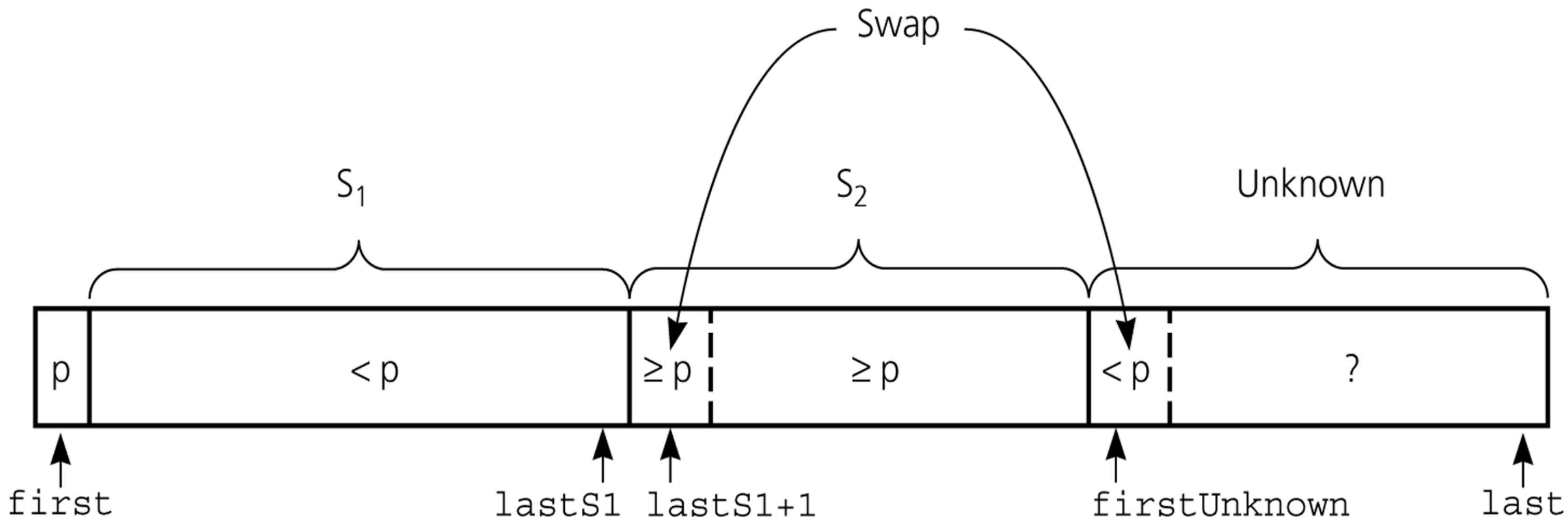
Partition (cont.)

When ***theArray[firstUnknown]*** >= ***pivot***
Move ***theArray[firstUnknown]*** into $\boldsymbol{S_2}$ by incrementing **firstUnknown**.



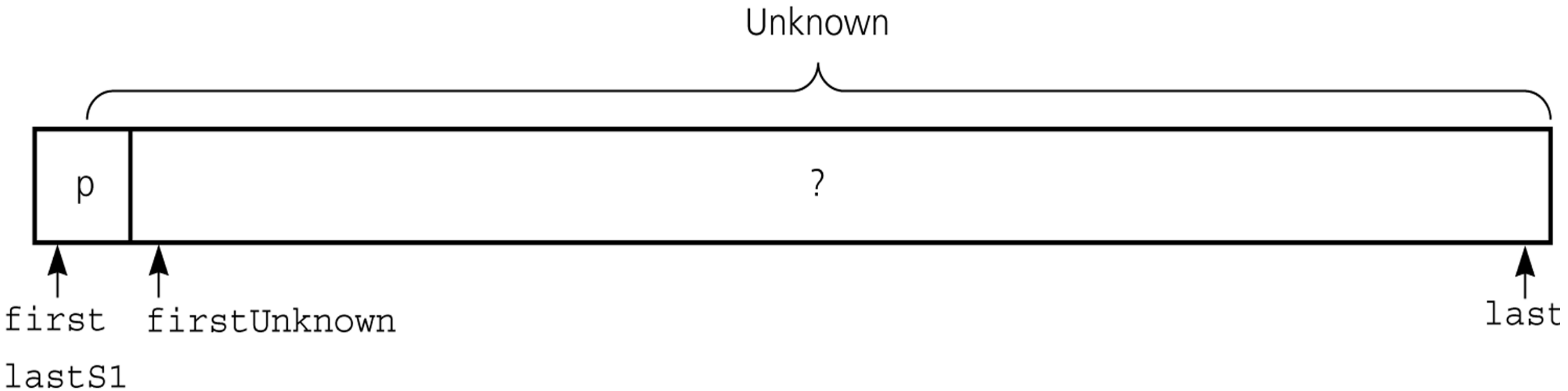
Partition (cont.)

When ***theArray[firstUnknown]*** < ***pivot***
Move ***theArray[firstUnknown]*** into $\mathbf{S_1}$ by swapping ***theArray[firstUnknown]*** with ***theArray[lastS1+1]*** and incrementing both ***lastS1*** and ***firstUnknown***.



Partition (cont.)

Initial state of the array



lastS1 = first
firstUnknown = first + 1
S1: theArray[first+1..lastS1]: **Empty**
S2: theArray[lastS1+1..firstUnknown-1]: **Empty**

Partition Function

```
void swap( int &lhs, int &rhs );

void partition(int theArray[], int first, int last,
            int &pivotIndex) {

   // Choose and place pivot in theArray[first]
   choosePivot(theArray, first, last);

   // Initialize
   int pivot = theArray[first];
   int lastS1 = first;
   int firstUnknown = first + 1;
```

//Continued to the next page…………

Partition Function (cont.)

//Continued from the previous page…………….

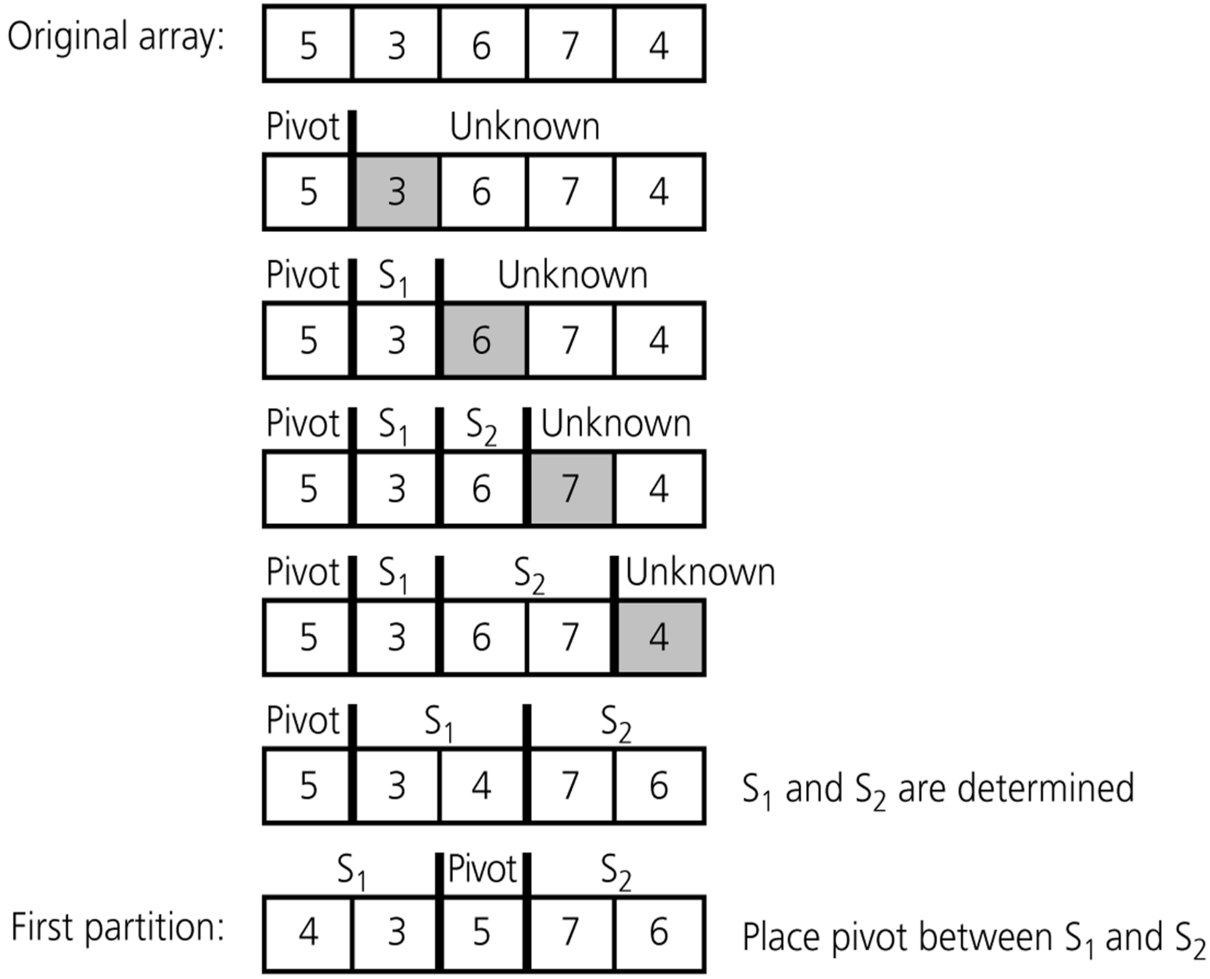
```
 // Move one item at a time until unknown region is empty
for (; firstUnknown <= last; ++firstUnknown) {
   if (theArray[firstUnknown] < pivot) {  // Belongs to S1
       ++lastS1; // Expands S1 by incrementing lastS1
       // Swap firstUnknown with lastS1
       swap(theArray[firstUnknown], theArray[lastS1]);
   }
    // else belongs to S2, ++firstUnknown in the loop
    // places it to S2
}
// Place pivot in proper position and mark its location
swap(theArray[first], theArray[lastS1]);
pivotIndex = lastS1;
}
```

Quicksort Function

```
void quicksort(int theArray[], int first, int last) {
    int pivotIndex;
   if (first < last) {
      // create the partition: S1, pivot, S2
      partition(theArray, first, last, pivotIndex);
      // sort regions S1 and S2
      quicksort(theArray, first, pivotIndex-1);
      quicksort(theArray, pivotIndex+1, last);
   }
}
```

Quicksort – Analysis

An average-case partitioning with quicksort

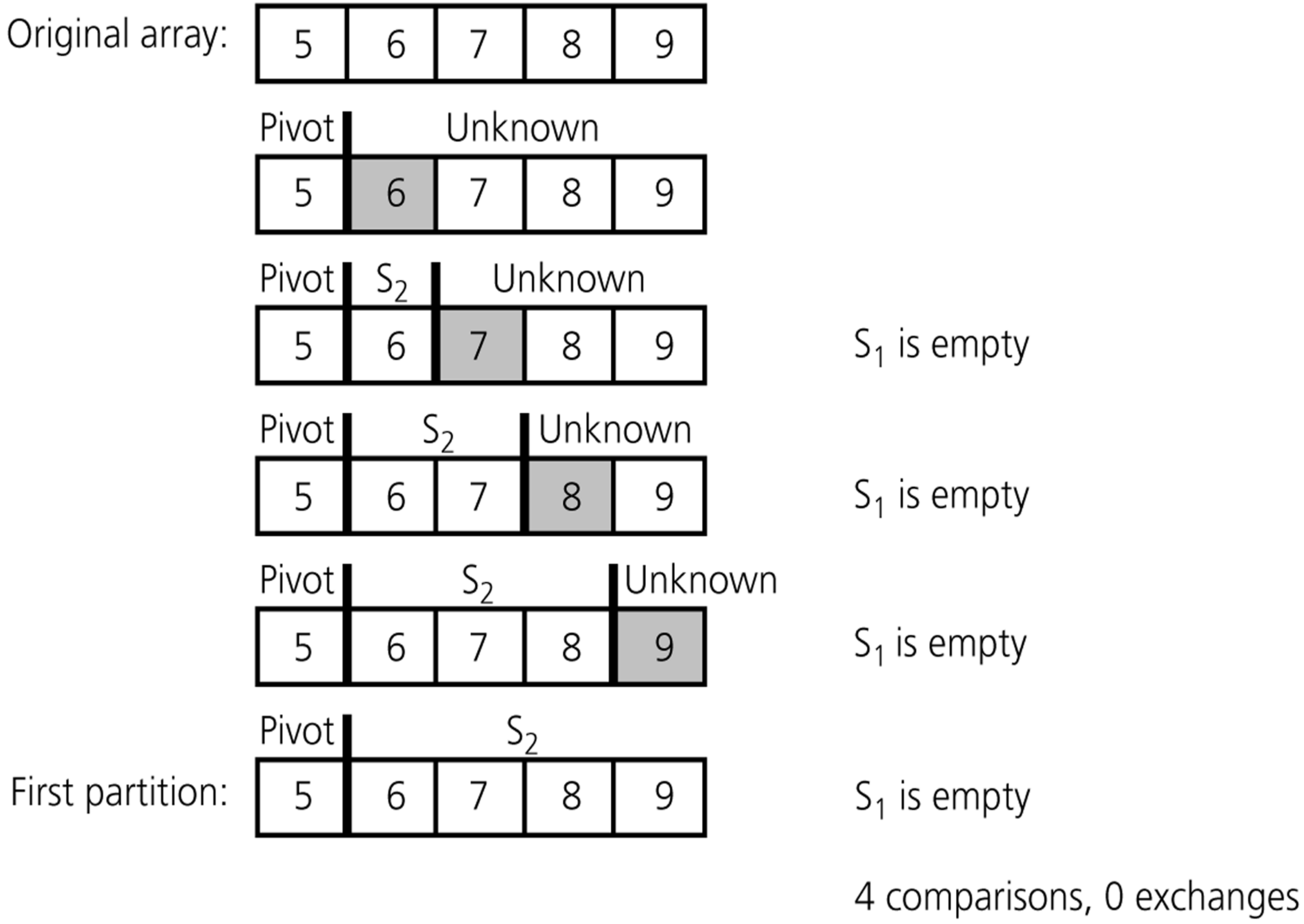


Quicksort – Analysis

- Quicksort is **$O(n*\log_2 n)$** in the best case and average case.
- Quicksort is slow when the array is sorted and we choose the first element as the pivot.
- Although the worst case behavior is not so good, its average case behavior is much better than its worst case.
  - So, Quicksort is one of best sorting algorithms using key comparisons.

Quicksort – Analysis

A worst-case partitioning with quicksort



Quicksort – Analysis

Worst Case: (assume that we are selecting the first element as pivot)

– The pivot divides the list of size n into two sublists of sizes 0 and n-1.

– The number of key comparisons

$= n-1 + n-2 + ... + 1$

$= n(n-1)/2$

$= \mathbf{n^2/2 - n/2}$ ➔ $\mathbf{O(n^2)}$

– The number of swaps =

swaps inside of the for loop — swaps outside of the for loop

$= (n-1 + n-2 + ... + 1) \quad + \quad (n-1)$

$= (n-1) + n(n-1)/2$

$= \mathbf{n^2/2 + n/2 - 1}$ ➔ $\mathbf{O(n^2)}$

- So, Quicksort is $\mathbf{O(n^2)}$ in worst case

Comparison of Sorting Algorithms

	Worst case	Average case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Mergesort	n * log n	n * log n
Quicksort	n^2	n * log n
Radix sort	n	n
Treesort	n^2	n * log n
Heapsort	n * log n	n * log n