Complexity Analysis of Sorting Algorithms

Sorting Algorithms

- There are many sorting algorithms. We shall see three of them:
  - Insertion Sort
  - Merge Sort
  - Quick Sort

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.

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

```
template <class Item>
void insertionSort(Item a[], int n)
{
   for (int i = 1; i < n; i++)
   {
      Item tmp = a[i];

      for (int j=i; 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 main steps, comparison, et by two loops = 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
  - Number of main steps, comparison, etc for two loops = (1+2+...+n-1)= n*(n-1)/2 ➔ $O(n^2)$
- *Average-case:* (not easy) ➔ **$O(n^2)$**
  - We have to look at all possible initial data organizations.
- **So, Insertion Sort is $O(n^2)$**

Insertion Sort: Best Case (already sorted)

8	23	32	45	56	78	Original List
8	23	32	45	56	78	pass 1, no move
8	23	32	45	56	78	pass 2, no move
8	23	32	45	56	78	pass 3, no move
8	23	32	45	56	78	pass 4, no move
8	23	32	45	56	78	pass 5, no move

Insertion Sort: Worst Case (sorted in reverse order)

78	56	45	32	23	12	Original List
56	78	45	32	23	12	pass 1, 1 move
45	56	78	32	23	12	pass 2, 2 moves
32	45	56	78	23	12	pass 3, 3 moves
23	32	45	56	78	12	pass 4, 4 moves
12	23	32	45	56	78	pass 5, 5 moves

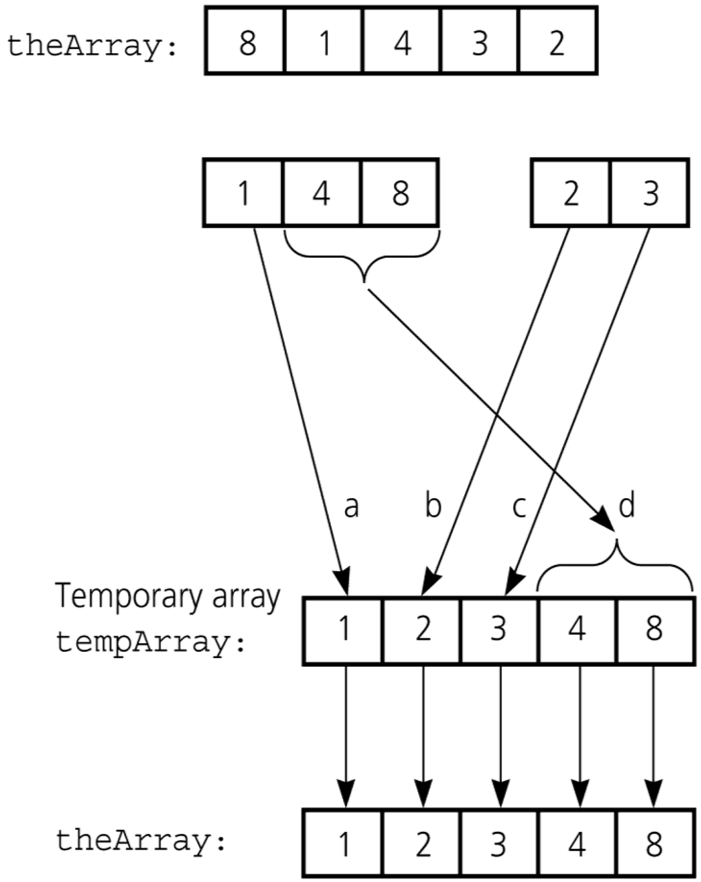
Analysis of insertion sort

- 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

```
const int MAX_SIZE = maximum-number-of-items-in-array;
void merge(DataType theArray[], int first, int mid, int last)
   {
   DataType tempArray[MAX_SIZE];  // 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;
      } }
```

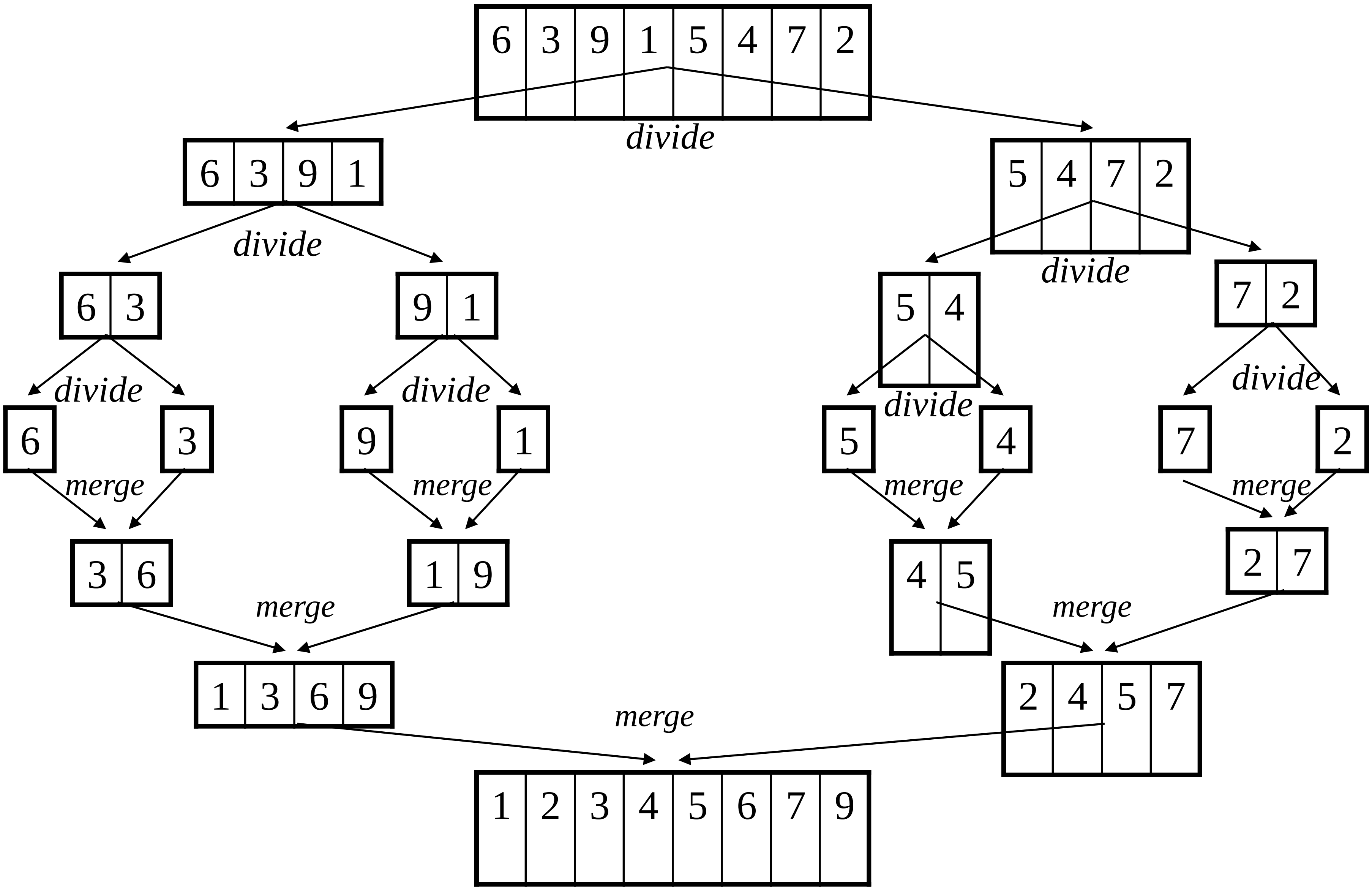
Merge (cont.)

```
    // 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];
}   // end merge
```

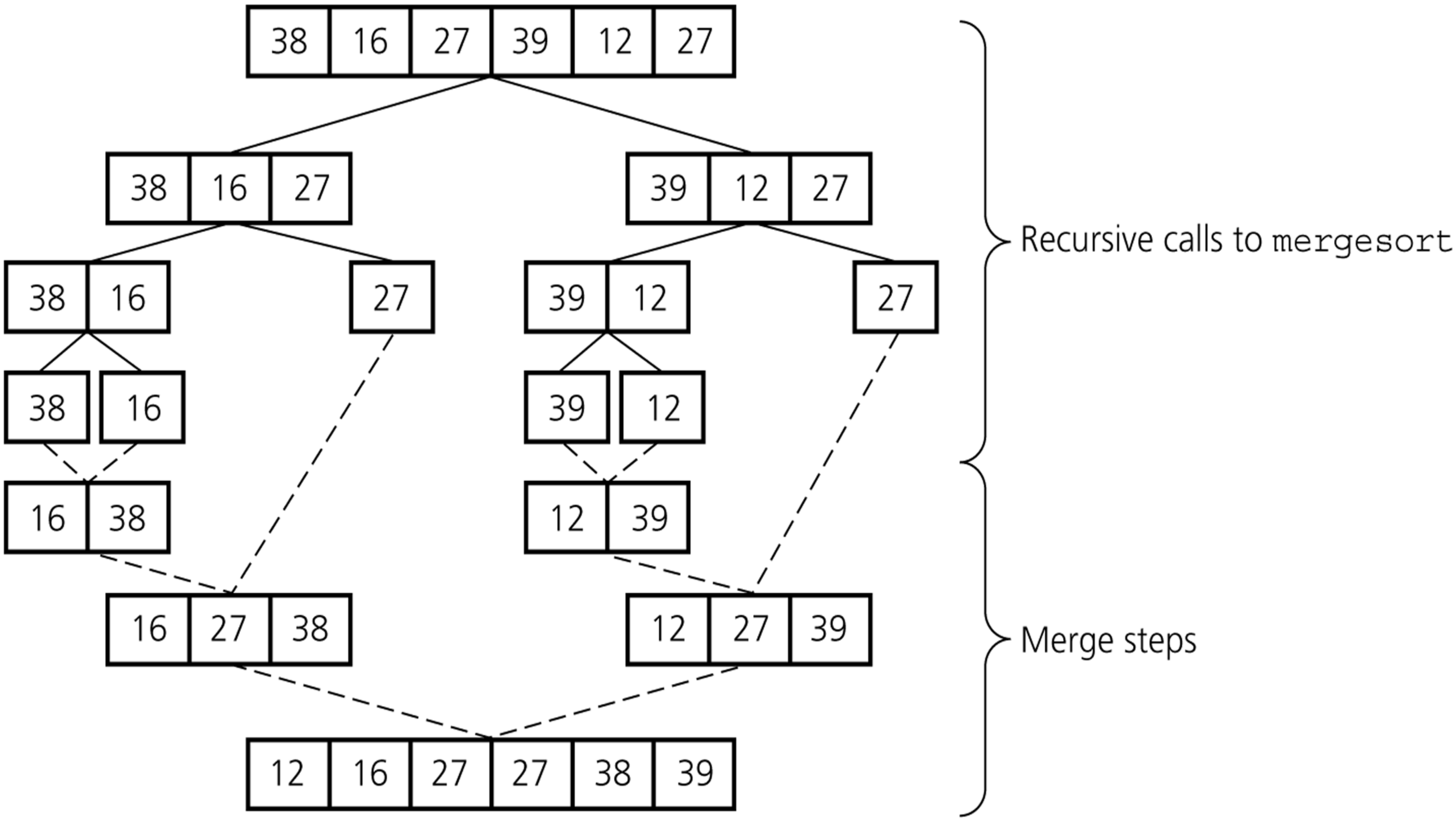
Mergesort

```
void mergesort(DataType theArray[], int first, int last) {
   if (first < last) {
      int mid = (first + last)/2;       // index of midpoint
      mergesort(theArray, first, mid);
      mergesort(theArray, mid+1, last);

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

Mergesort - Example



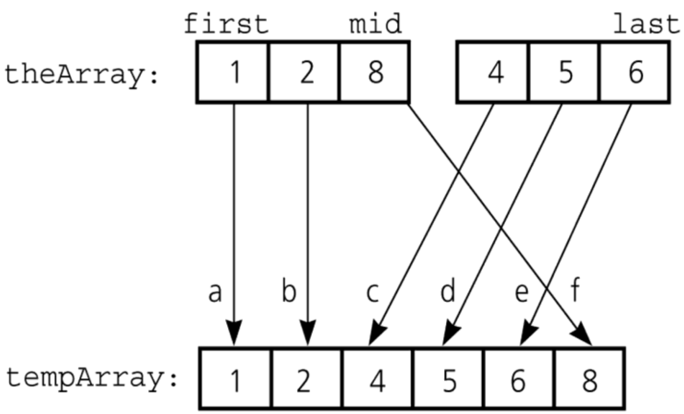
Mergesort – Example2



Mergesort – Analysis of Merge

A worst-case instance of the merge step in `mergesort`:

`Maximum possible key comparisons are there.`

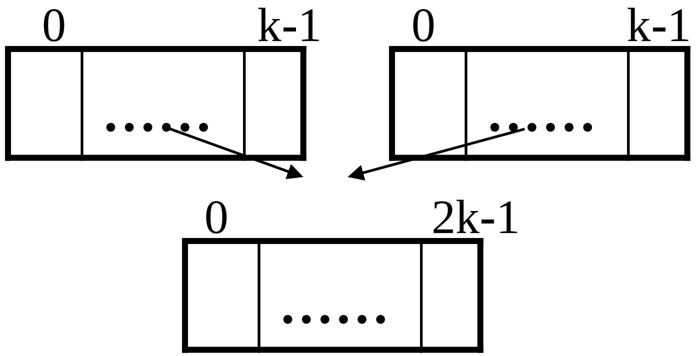


Merge the halves:

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.)

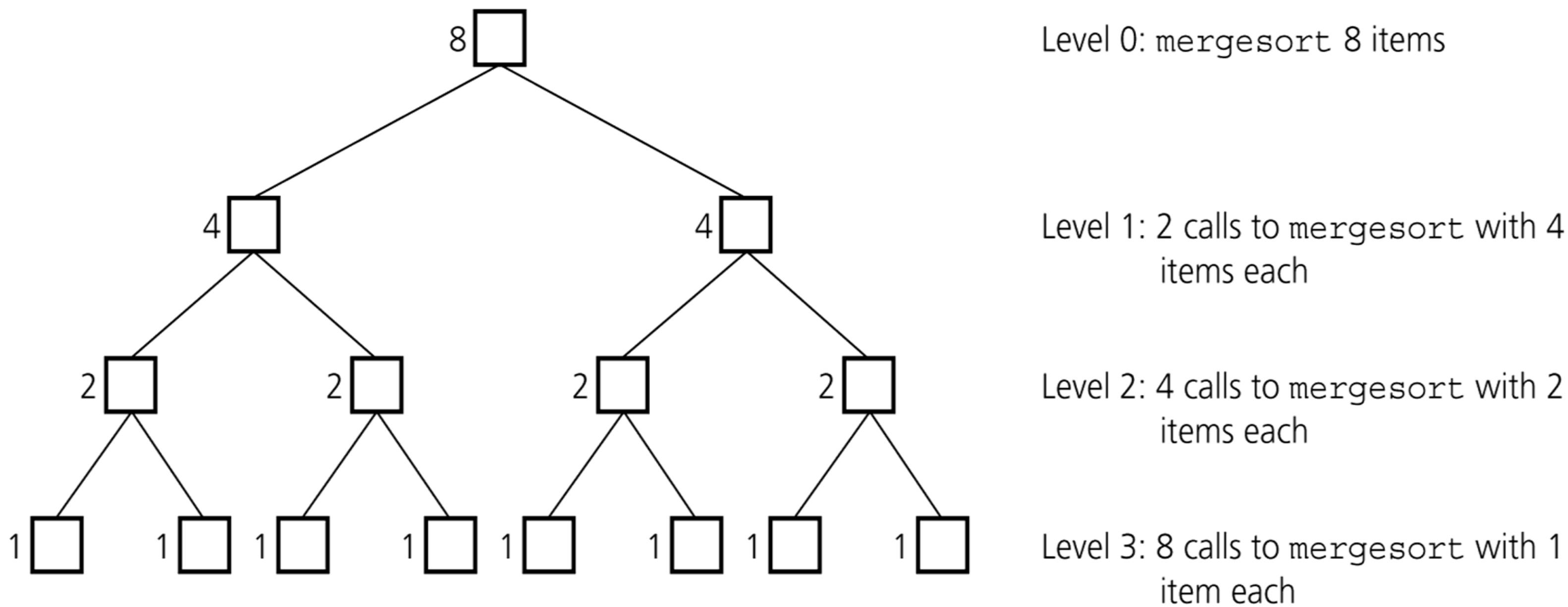
Merging two sorted arrays of size ***k***



- ***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, comparisons, etc: 2k + 2k
  - In general, main steps: ck
- ***Worst-case:***
  - The number of moves, comparisons, etc: 2k + 2k
  - In general, main steps: ck

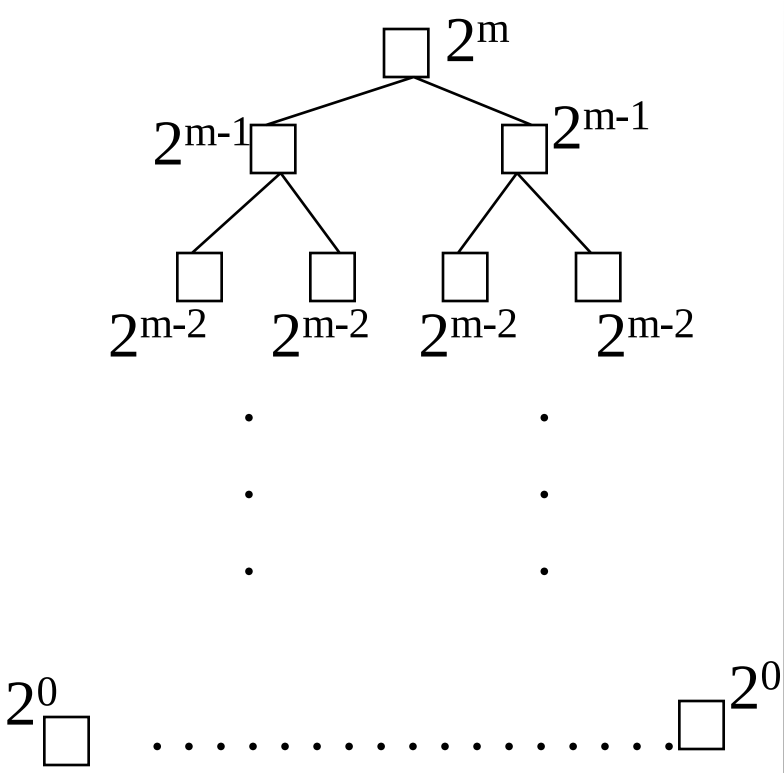
Mergesort - Analysis

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



Mergesort – Analysis

Assume, $n = 2^m$, it will be easier for the analysis.



level 0 : 1 merge (size $2*2^{m-1}$) = $c2^m$

level 1 : 2 merges (size $2*2^{m-2}$) = $2^2c2^{m-2} = c2^m$

level 2 : 4 merges (size $2*2^{m-3}$) = $2^3c2^{m-3} = c2^m$

level m-1 : 2^{m-1} merges (size $2*2^0$)=$2^mc2^0 = c2^m$

level m: no merger = 0

Mergesort - Analysis

- ***Worst-case –***

The number of key comparisons, moves, copy, etc:

$= c2^m + c2^m + \ldots + c2^m$ (m terms)

$= cm*2^m$

Remember, $n = 2^m$. So, $m = \log_2 n$

$= c \log_2 n\ n$

➔ $O(n \log_2 n)$

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.

- If we use a linked list, we do not need an extra array
  - But, we need space for the links
  - And, it will be difficult to divide the list into half ($O(n)$)

Comparison of N^2 and $NlogN$

N	O(NLogN)	O(N^2)
16	64	256
64	384	4096 = 4K
256	2048	65536=64K
1,024	10240	1048576 = 1M
16,384	229376	268435456 = 256M

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,
  2. Then, it sorts the parts independently,
  3. Finally, it combines the sorted subsequences by a simple concatenation.

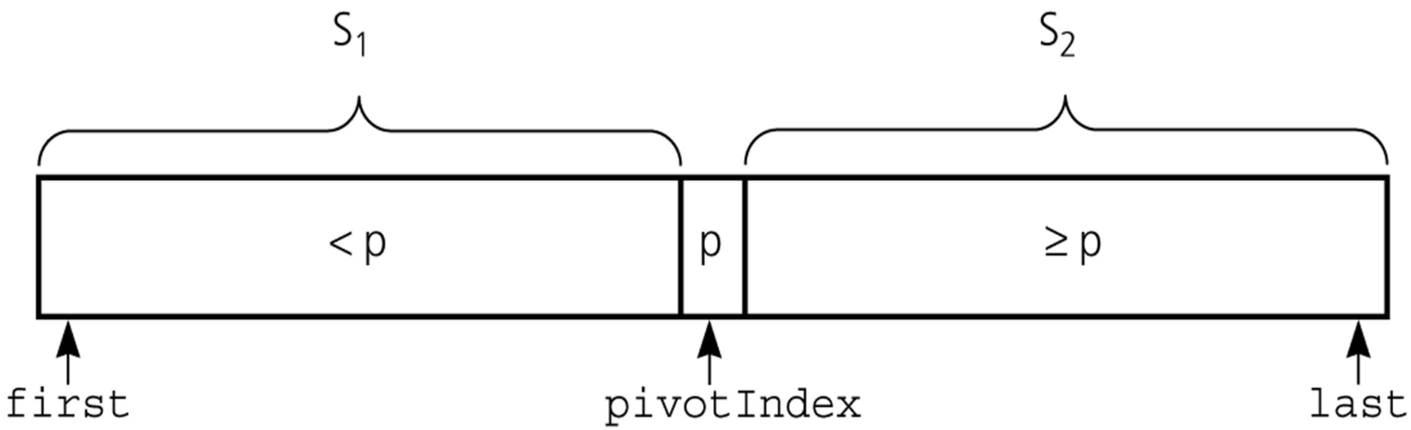
Quicksort (cont.)

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.

Partition

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



- Arranging the array elements around the pivot p generates two smaller sorting problems.
  - sort the left section of the array, and sort the right section of the array.
  - when these two smaller sorting problems are solved recursively, our bigger sorting problem is solved.

Partition – Choosing the pivot

- First, we have to select a pivot element among the elements of the given array, and we put this pivot into the first location of the array before partitioning.
- 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.
  - 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. For example, select a pivot randomly from the elements.

Partition Function

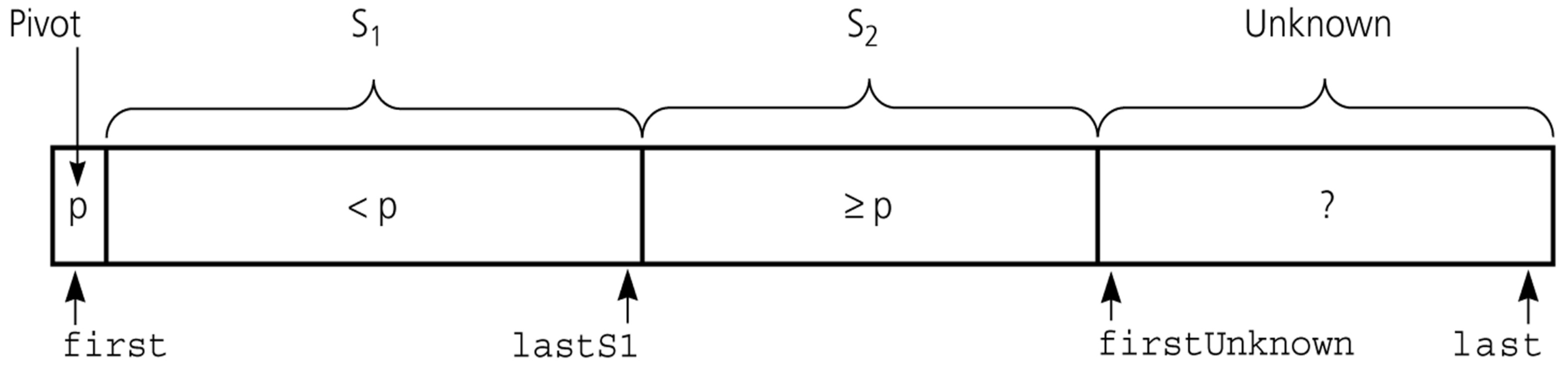
```
template <class DataType>
void partition(DataType theArray[], int first, int last,
            int &pivotIndex) {
// Partitions an array for quicksort.
// Precondition: first <= last.
// Postcondition: Partitions theArray[first..last] such that:
//   S1 = theArray[first..pivotIndex-1] < pivot
//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

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

   DataType pivot = theArray[first]; // copy pivot
```

Partition Function (cont.)

```
   // initially, everything but pivot is in unknown
   int lastS1 = first;           // index of last item in S1
   int firstUnknown = first + 1; //index of 1st item in unknown
   // move one item at a time until unknown region is empty
   for (; firstUnknown <= last; ++firstUnknown) {
       // Invariant: theArray[first+1..lastS1] < pivot
      //            theArray[lastS1+1..firstUnknown-1] >= pivot
      // move item from unknown to proper region
      if (theArray[firstUnknown] < pivot) {     // belongs to S1
           ++lastS1;
          swap(theArray[firstUnknown], theArray[lastS1]);
       }      // else belongs to S2
   }
   // place pivot in proper position and mark its location
   swap(theArray[first], theArray[lastS1]);
   pivotIndex = lastS1;
} // end partition
```

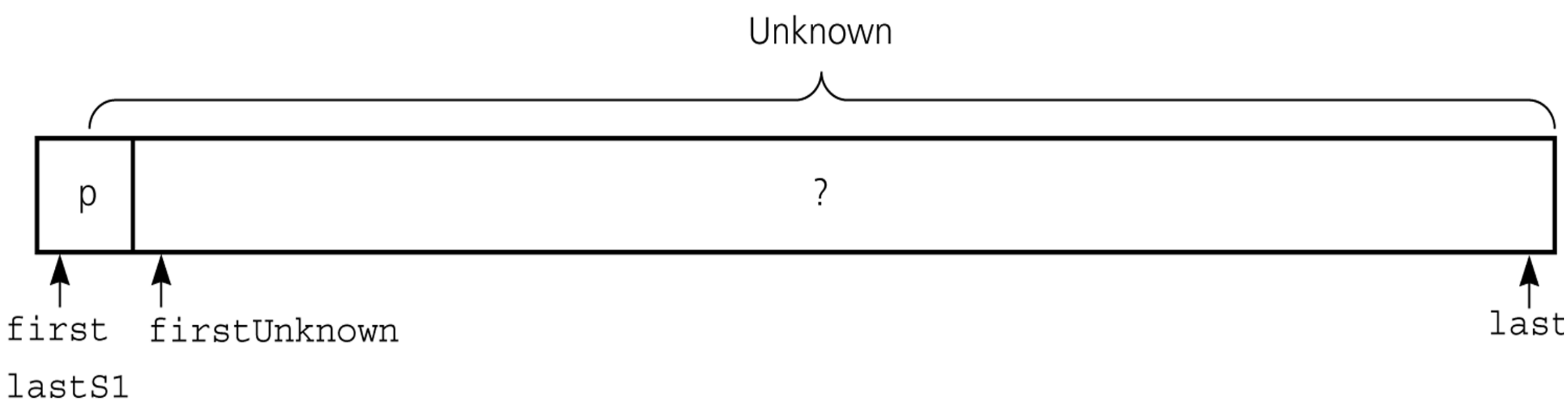
Partition Function (cont.)

Invariant for the partition algorithm



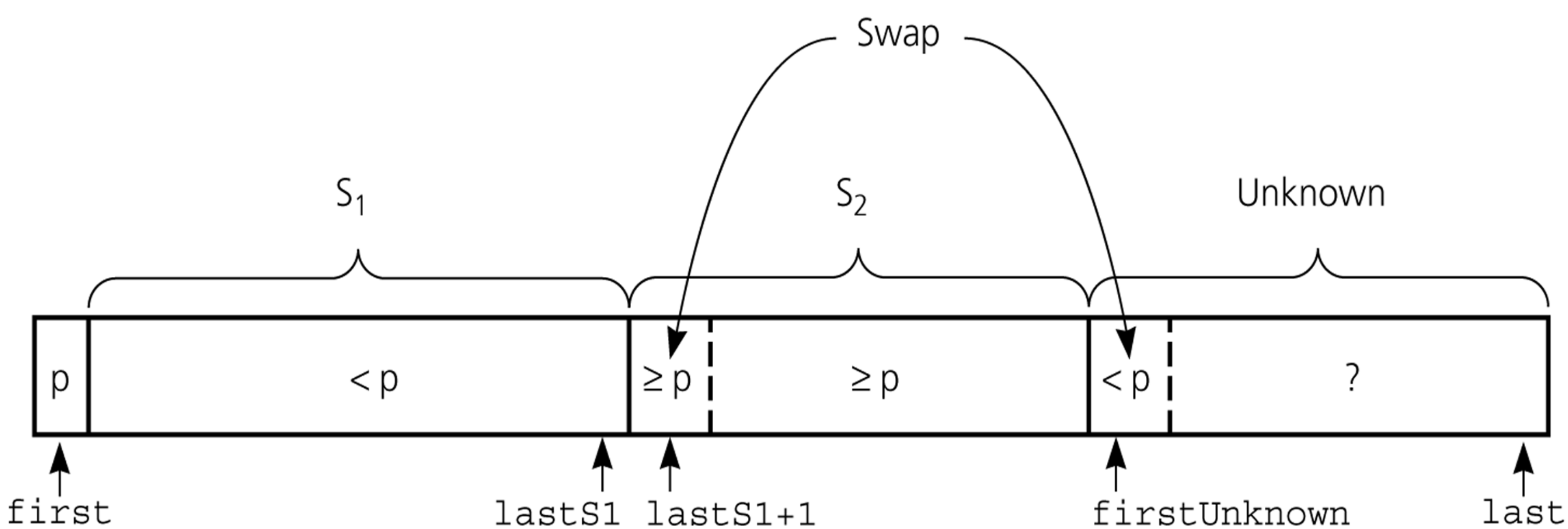
Partition Function (cont.)

Initial state of the array



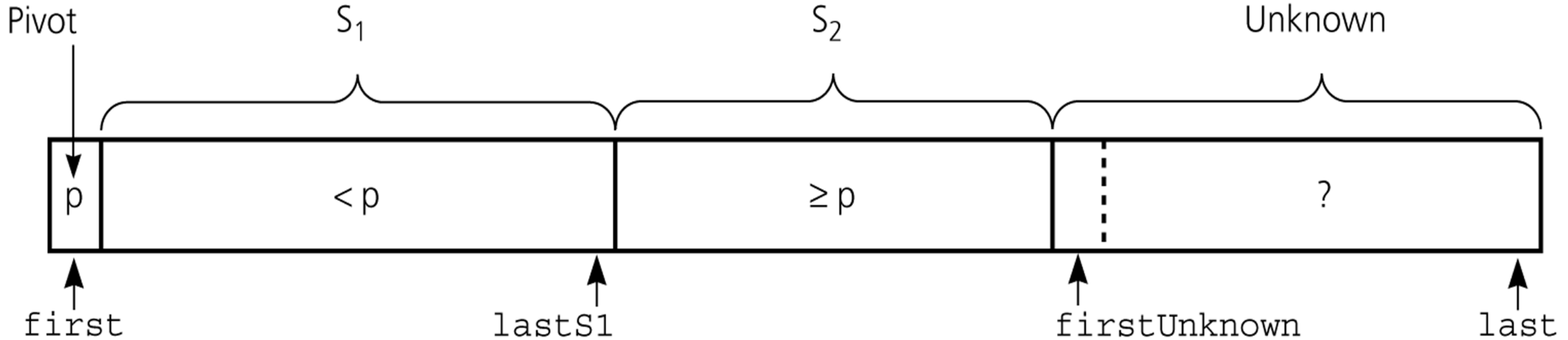
Partition Function (cont.)

Moving *theArray[firstUnknown]* ***into S_1 by swapping it with*** *theArray[lastS1+1]* ***and by incrementing both*** *lastS1* ***and*** *firstUnknown****.***



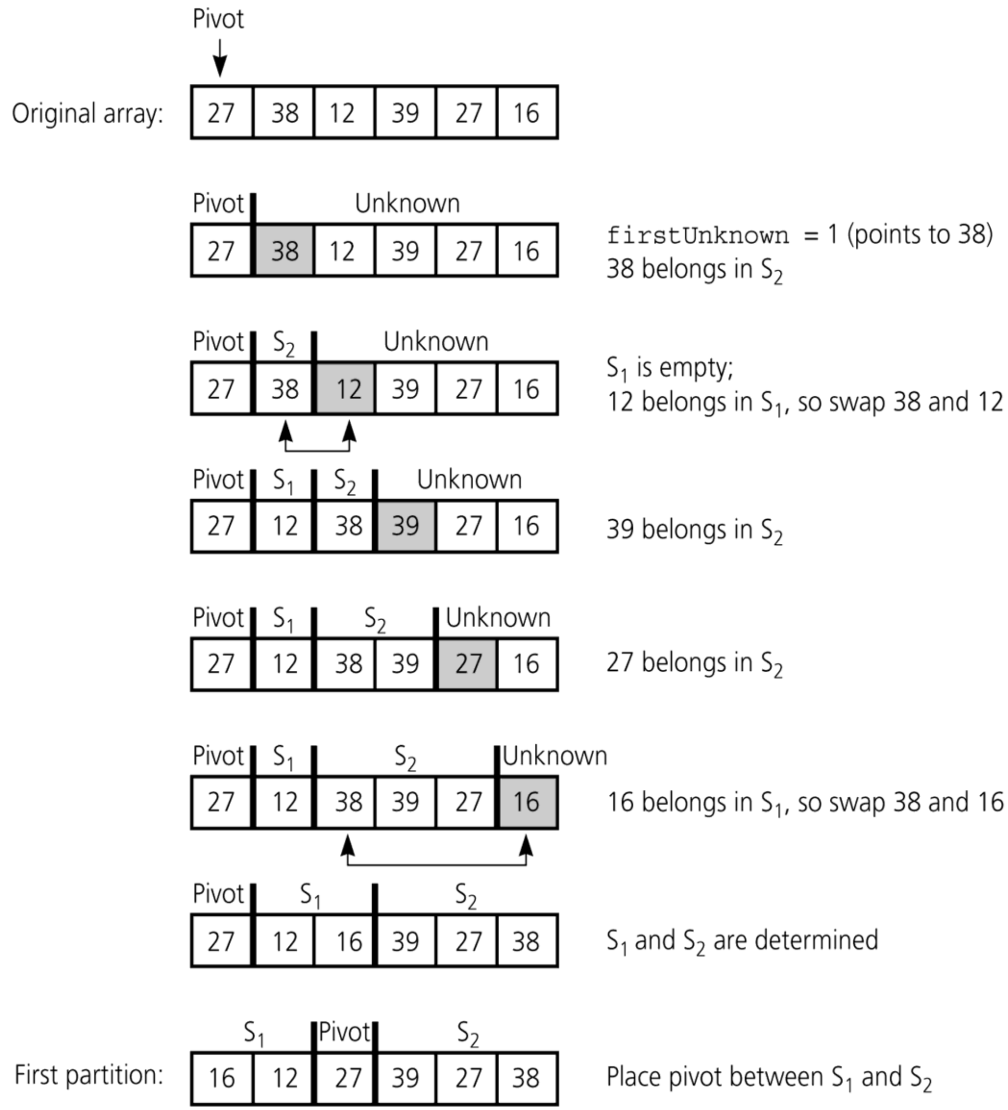
Partition Function (cont.)

Moving *theArray[firstUnknown]* ***into S_2 by incrementing*** *firstUnknown*.



Partition Function (cont.)

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



Quicksort Function

```
void quicksort(DataType theArray[], int first, int last) {
// Sorts the items in an array into ascending order.
// Precondition: theArray[first..last] is an array.
// Postcondition: theArray[first..last] is sorted.
// Calls: partition.
   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

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

- The pivot **always** divides the list of size n into two sublists of sizes 0 and n-1.
- The number of key comparisons, moves, swaps, etc

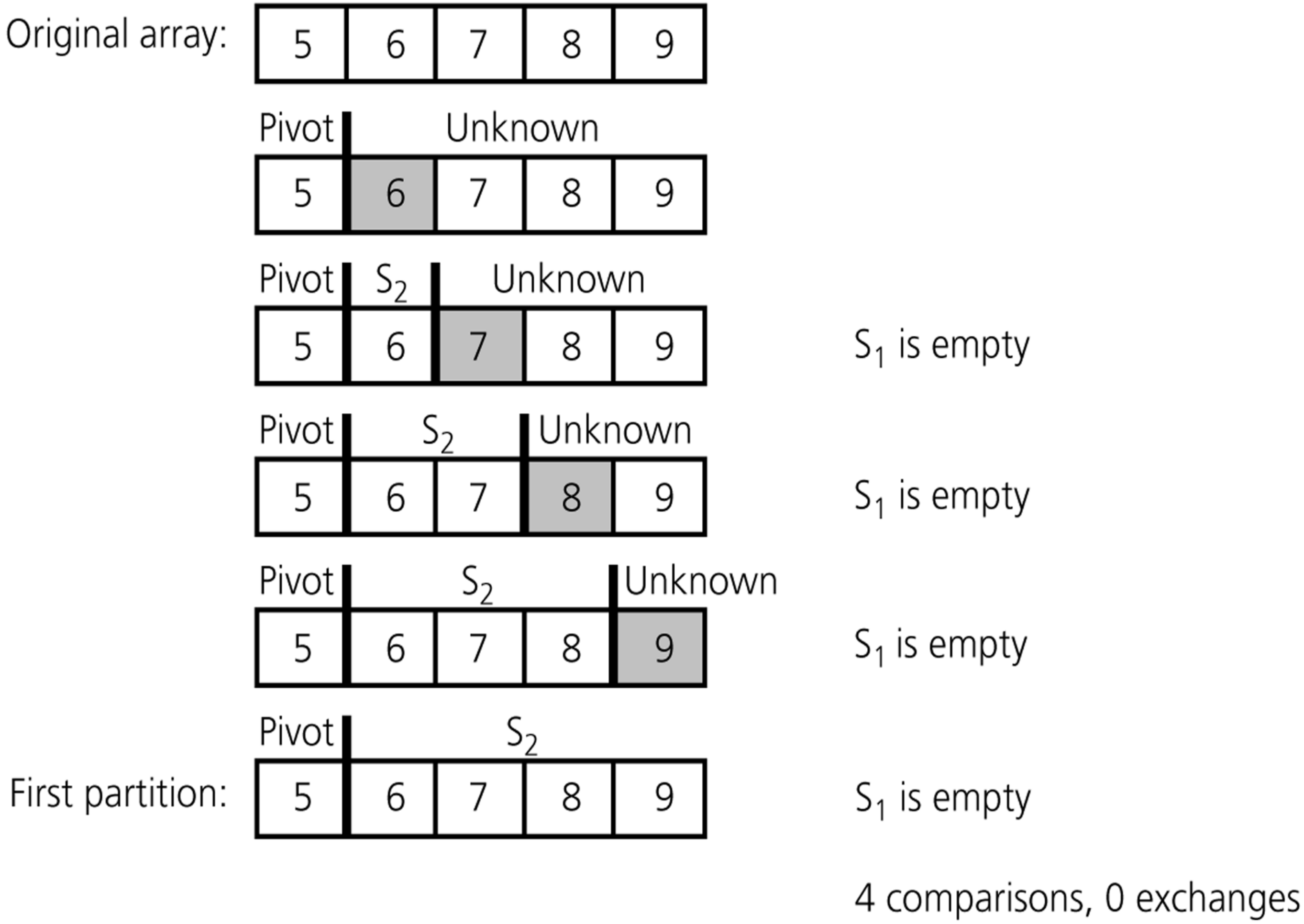
  $= n-1 + n-2 + ... + 1$

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

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

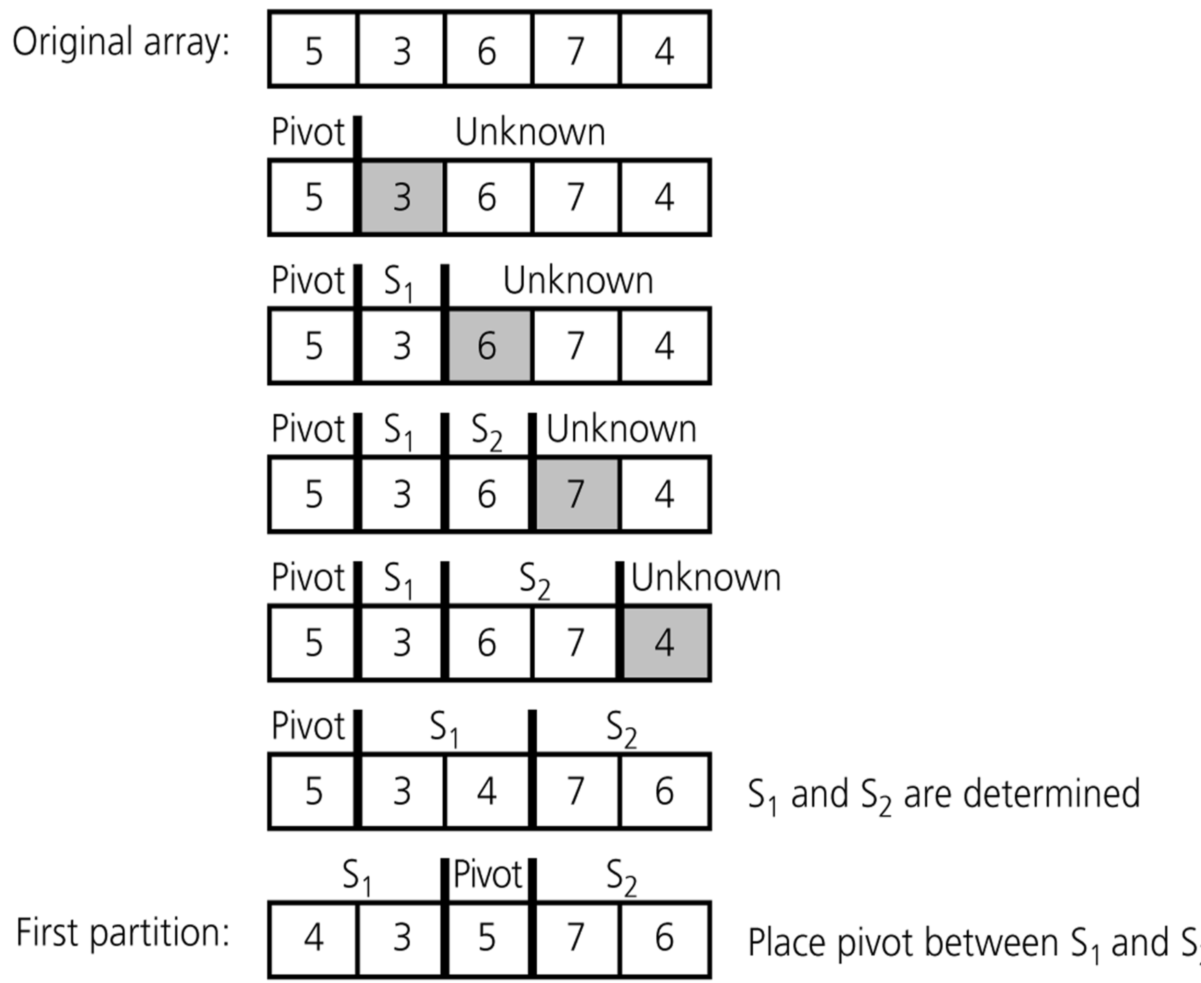
Quicksort – Analysis

An example of worst-case partitioning with `quicksort`



Quicksort – Analysis

An example of average-case partitioning with `quicksort`



Quicksort – Analysis

- Quicksort is $\mathbf{O(n*log_2n)}$ 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.

Quick Sort - Analysis

- We shall see more analysis on Quick Sort when in coming chapters