

Sorting Algorithms

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.
- 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 sorting algorithms, such as:
 - Selection Sort
 - Insertion Sort
 - Bubble Sort
 - Merge Sort
 - Quick Sort
- The first three are the foundations for faster and more efficient algorithms.

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.

Sorted

Unsorted

23	78	45	8	32	56
----	----	----	---	----	----

Original List

8	78	45	23	32	56
---	----	----	----	----	----

After pass 1

8	23	45	78	32	56
---	----	----	----	----	----

After pass 2

8	23	32	78	45	56
---	----	----	----	----	----

After pass 3

8	23	32	45	78	56
---	----	----	----	----	----

After pass 4

8	23	32	45	56	78
---	----	----	----	----	----

After pass 5

Selection Sort (cont.)

```
template <class Item>
void selectionSort( Item 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]);
    }
}
```

```
template < class Object>
void swap( Object &lhs, Object &rhs )
{
    Object tmp = lhs;
    lhs = rhs;
    rhs = tmp;
}
```

Selection 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.
- 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 (from 1 to $n-1$), and in each iteration we make one key comparison.
 - ➔ # of key comparisons = $1+2+\dots+n-1 = n*(n-1)/2$
 - ➔ 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).

Comparison of N , $\log N$ and N^2

<u>N</u>	<u>O(LogN)</u>	<u>O(N²)</u>
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

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:** $\rightarrow O(n)$
 - Array is already sorted in ascending order.
 - Inner loop will not be executed.
 - The number of moves: $2*(n-1)$ $\rightarrow O(n)$
 - The number of key comparisons: $(n-1)$ $\rightarrow O(n)$
- **Worst-case:** $\rightarrow O(n^2)$
 - Array is in reverse order:
 - Inner loop is executed $i-1$ times, for $i = 2, 3, \dots, n$
 - The number of moves: $2*(n-1) + (1+2+\dots+n-1) = 2*(n-1) + n*(n-1)/2$ $\rightarrow O(n^2)$
 - The number of key comparisons: $(1+2+\dots+n-1) = n*(n-1)/2$ $\rightarrow O(n^2)$
- **Average-case:** $\rightarrow O(n^2)$
 - 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.

Bubble Sort

- The list is divided into two sublists: sorted and unsorted.
- The smallest element is bubbled 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

```
template <class Item>
void bubbleSort(Item 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
            }
        }
    }
}
```

Bubble Sort – Analysis

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

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 half separately, and
 - Then merge the sorted halves into one sorted array.

Mergesort - Example

theArray:

8	1	4	3	2
---	---	---	---	---

Divide the array in half

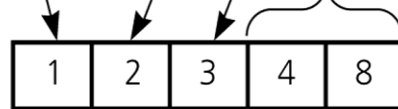


Sort the halves

Merge the halves:

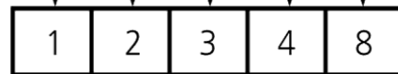
- a. $1 < 2$, so move 1 from left half to tempArray
- b. $4 > 2$, so move 2 from right half to tempArray
- c. $4 > 3$, so move 3 from right half to tempArray
- d. Right half is finished, so move rest of left half to tempArray

Temporary array
tempArray:



Copy temporary array back into
original array

theArray:



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];

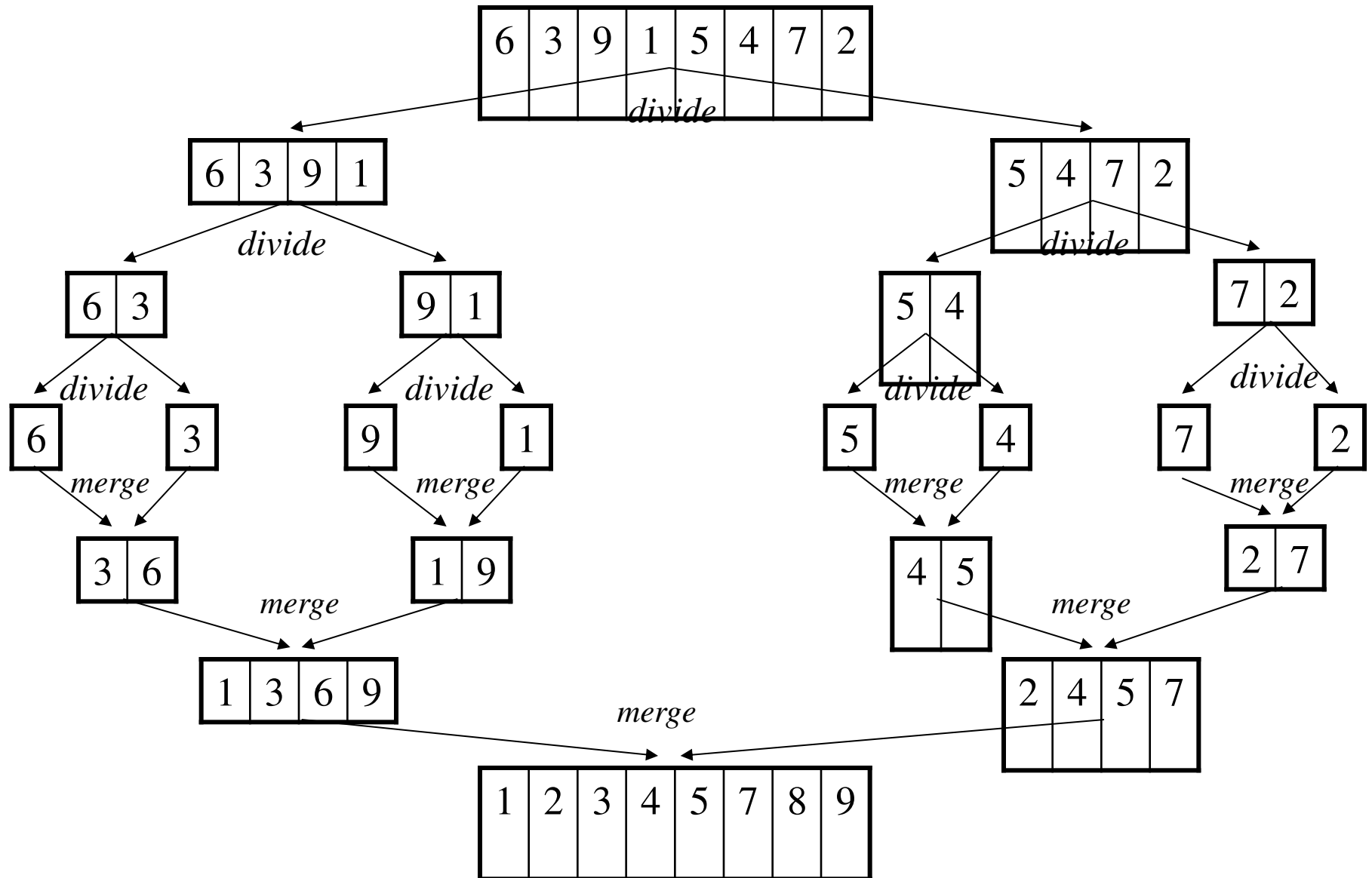
// finish off the second subarray, if necessary
for (; first2 <= last2; ++first2, ++index)
    tempArray[index] = theArray[first2];

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

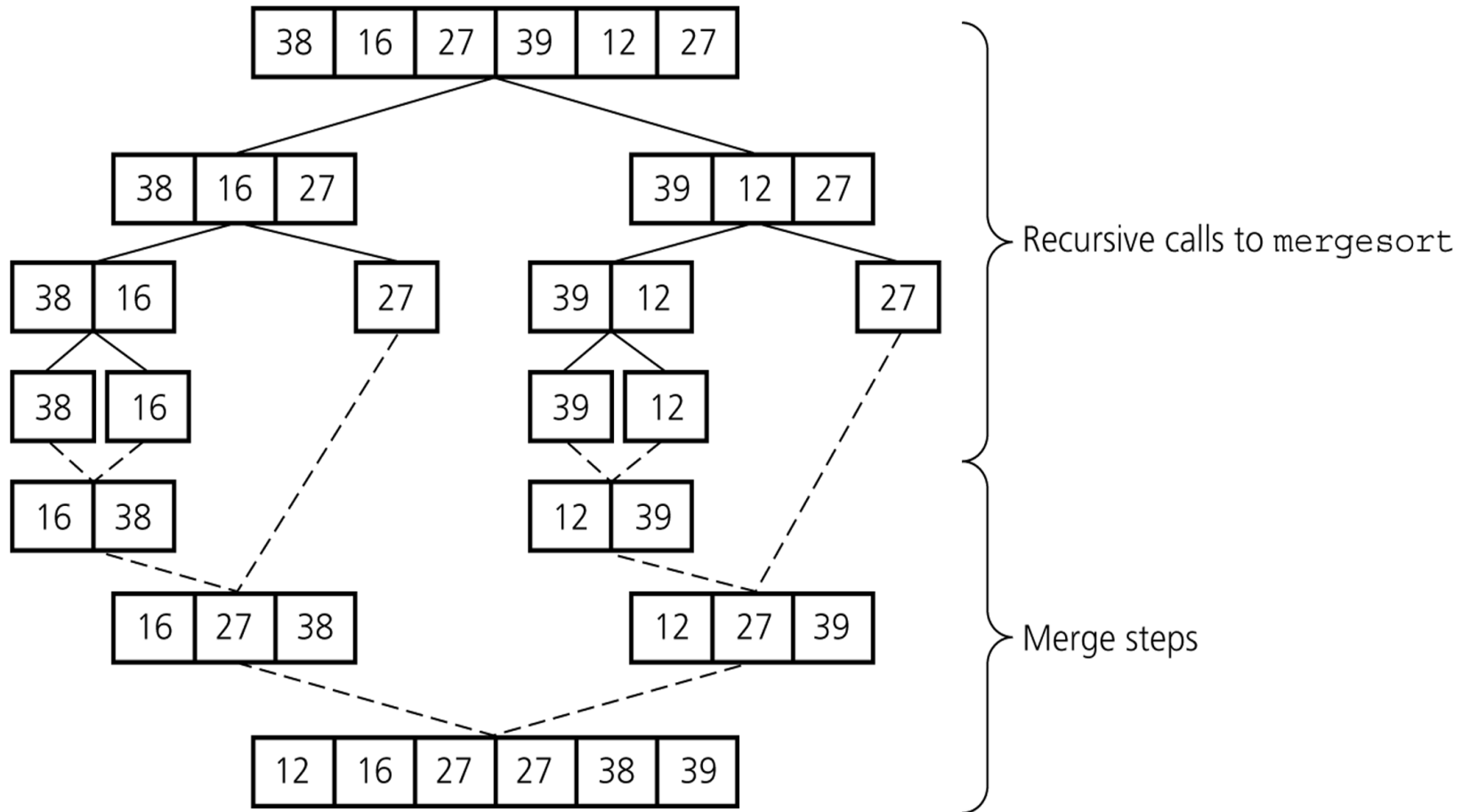
Mergesort

```
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

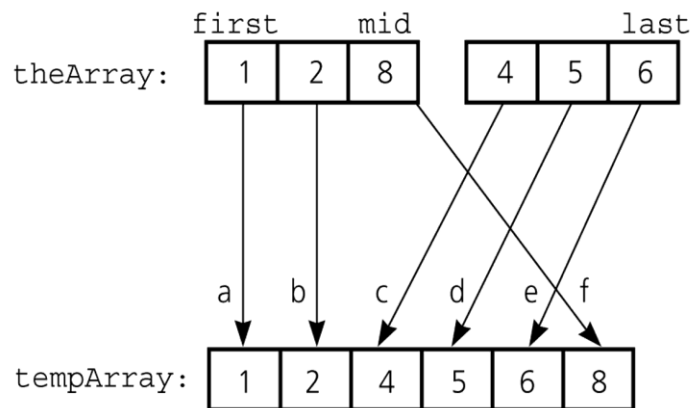


Mergesort – Example2



Mergesort – Analysis of Merge

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

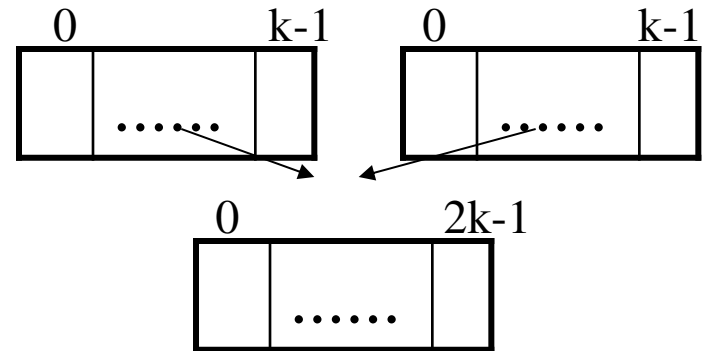


Merge the halves:

- 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:**

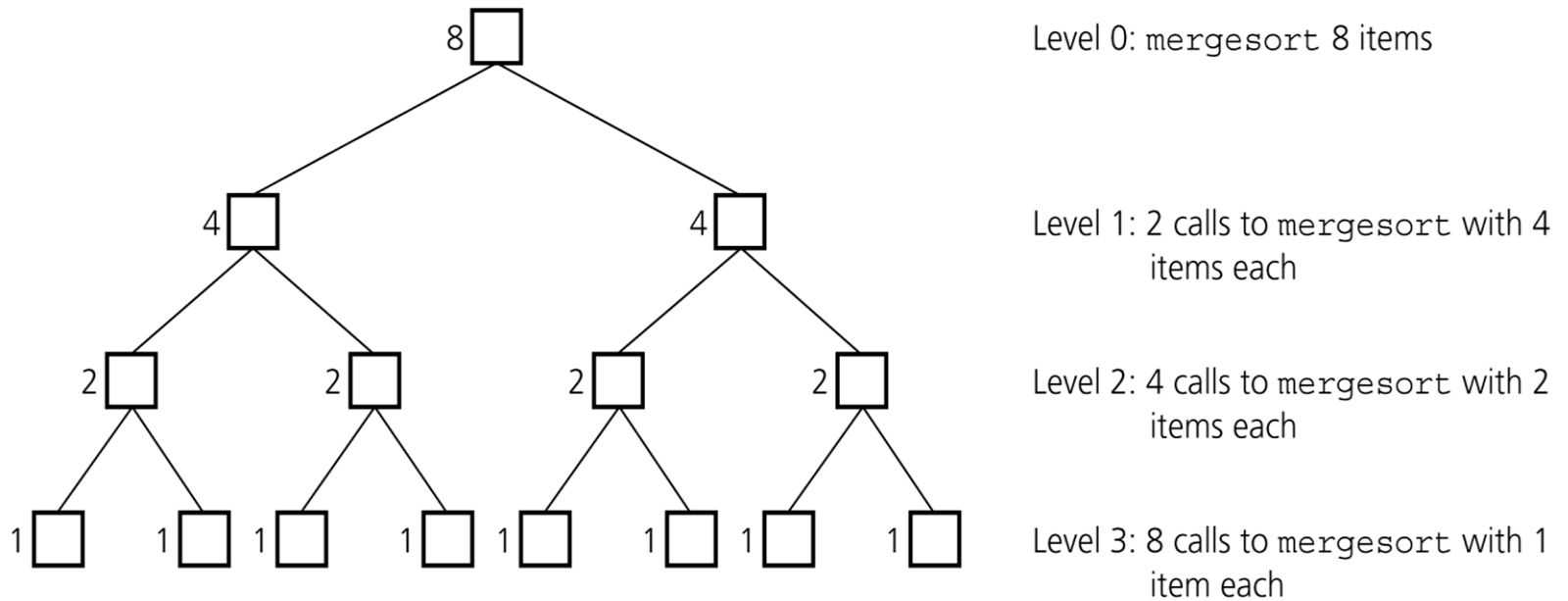
- All the elements in the first array are smaller (or larger) than all the elements in the second array.
- The number of moves: $2k + 2k$
- The number of key comparisons: k

- **Worst-case:**

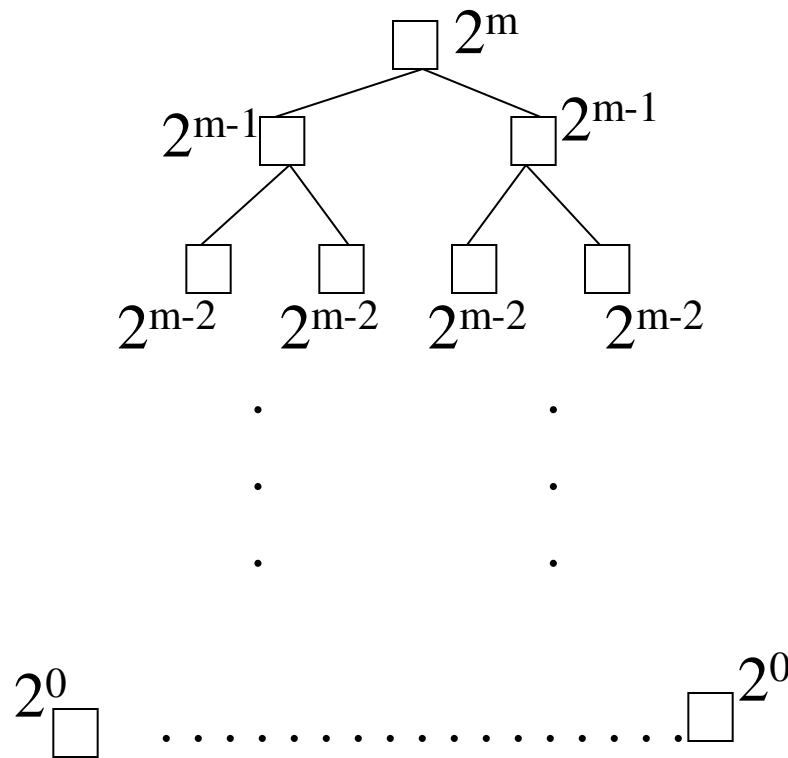
- The number of moves: $2k + 2k$
- The number of key comparisons: $2k-1$

Mergesort - Analysis

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



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

level m

Mergesort - Analysis

- *Worst-case* –

The number of key comparisons:

$$\begin{aligned} &= 2^0 * (2 * 2^{m-1} - 1) + 2^1 * (2 * 2^{m-2} - 1) + \dots + 2^{m-1} * (2 * 2^0 - 1) \\ &= (2^m - 1) + (2^m - 2) + \dots + (2^m - 2^{m-1}) \quad (\text{ m terms }) \end{aligned}$$

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

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

Using $m = \log n$

$$= n * \log_2 n - n - 1$$

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

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 ($O(n)$)

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.

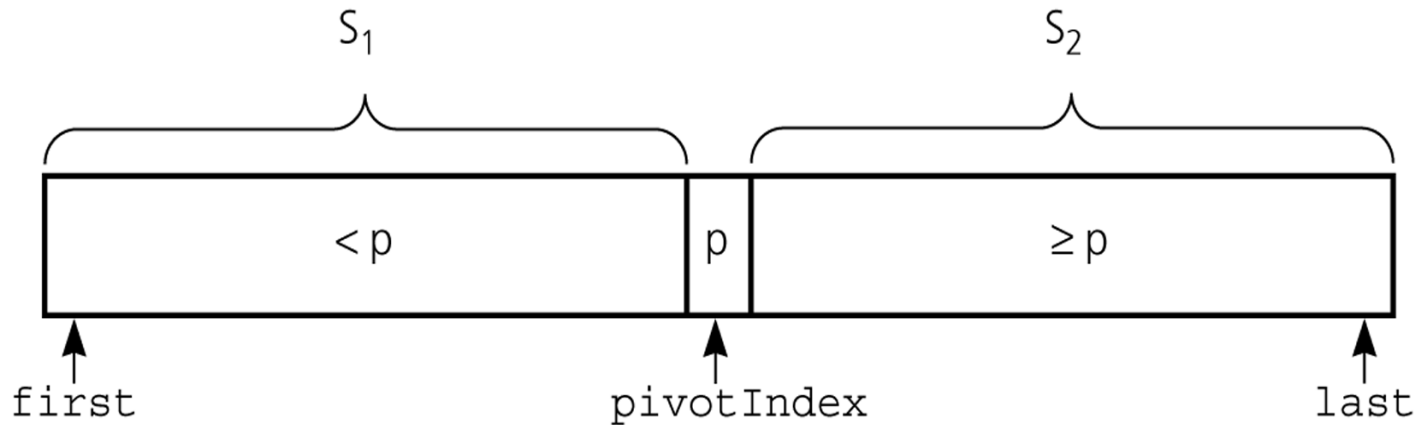
- To partition the list, we first choose some element from the list for which we hope about half the elements will come before and half after. Call this element the *pivot*.
- Then we partition the elements so that all those with values less than the pivot come in one sublist and all those with greater 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.
 - 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.

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
```

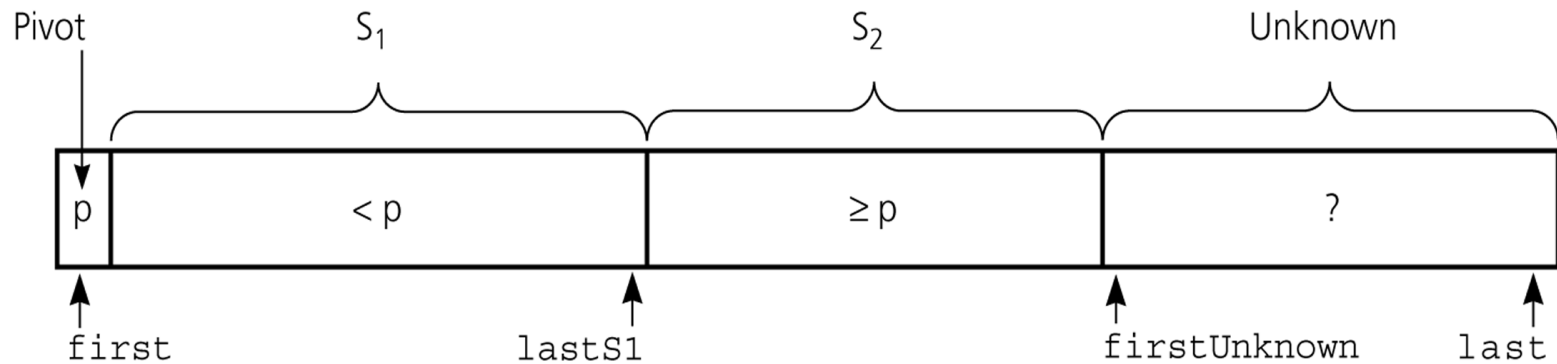
Partition Function (cont.)

```
// 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
```

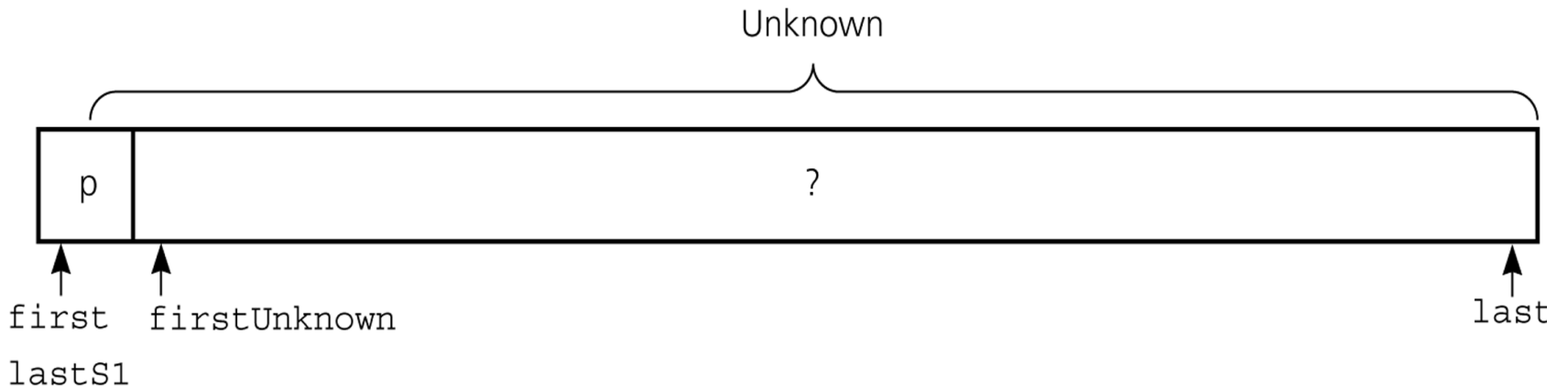
Partition Function (cont.)

Invariant for the partition algorithm



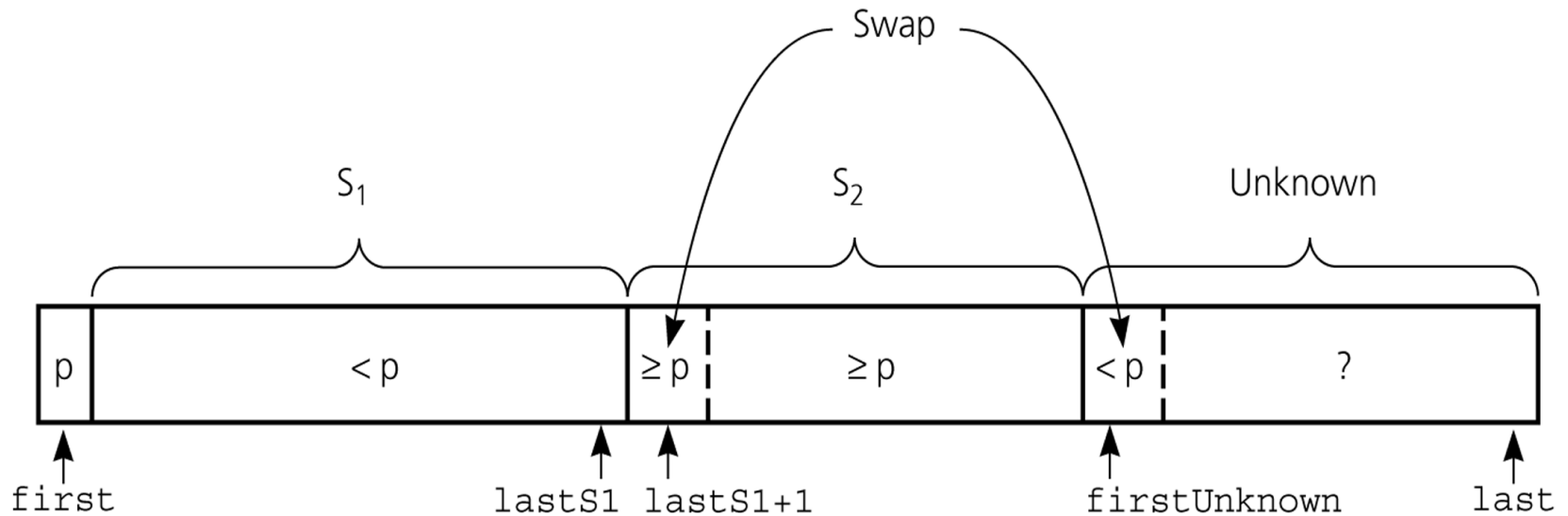
Partition Function (cont.)

Initial state of the array



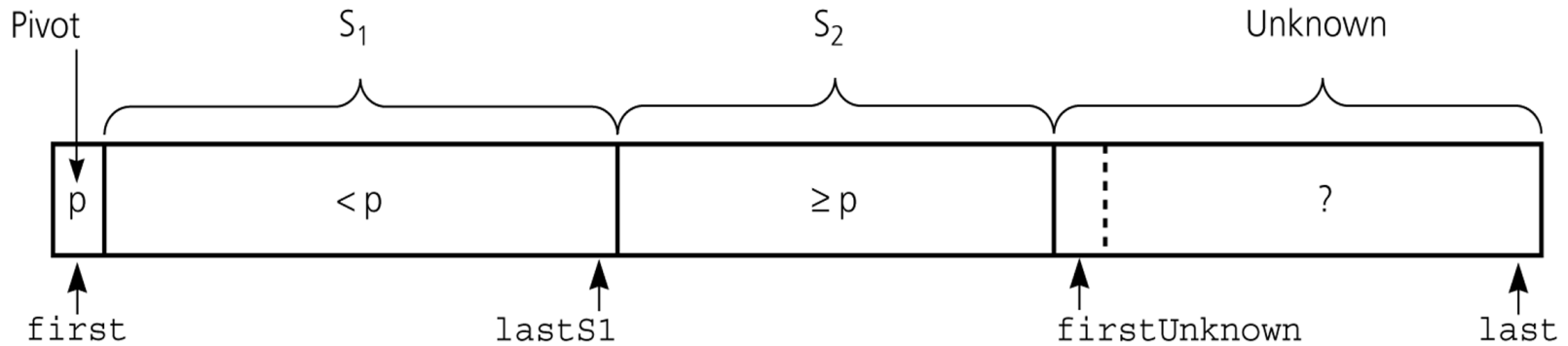
Partition Function (cont.)

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



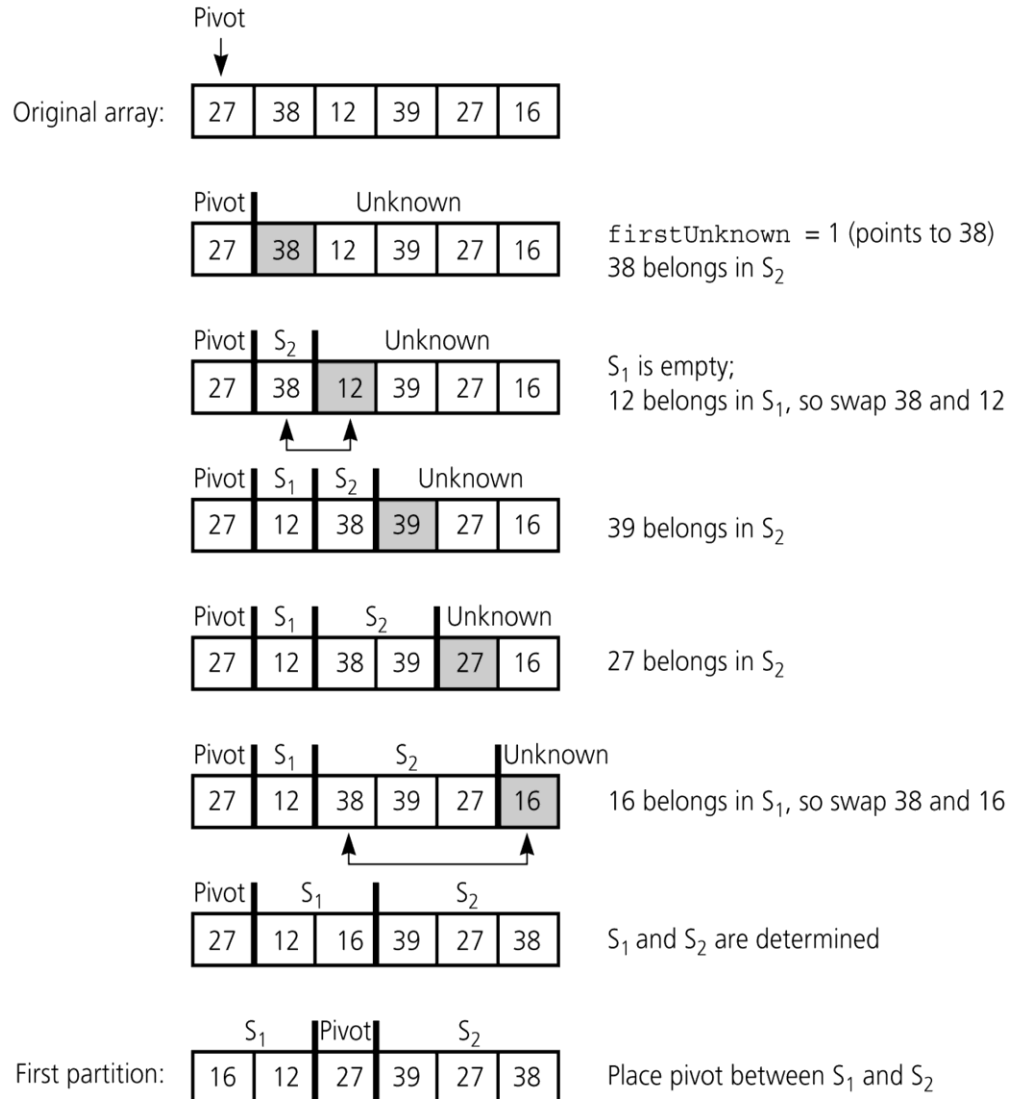
Partition Function (cont.)

Moving the `Array[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 divides the list of size n into two sublists of sizes 0 and $n-1$.
- The number of key comparisons

$$= n-1 + n-2 + \dots + 1$$

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

- The number of swaps =

$$= n-1 \quad + \quad n-1 + n-2 + \dots + 1$$

swaps outside of the for loop

swaps inside of the for loop

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

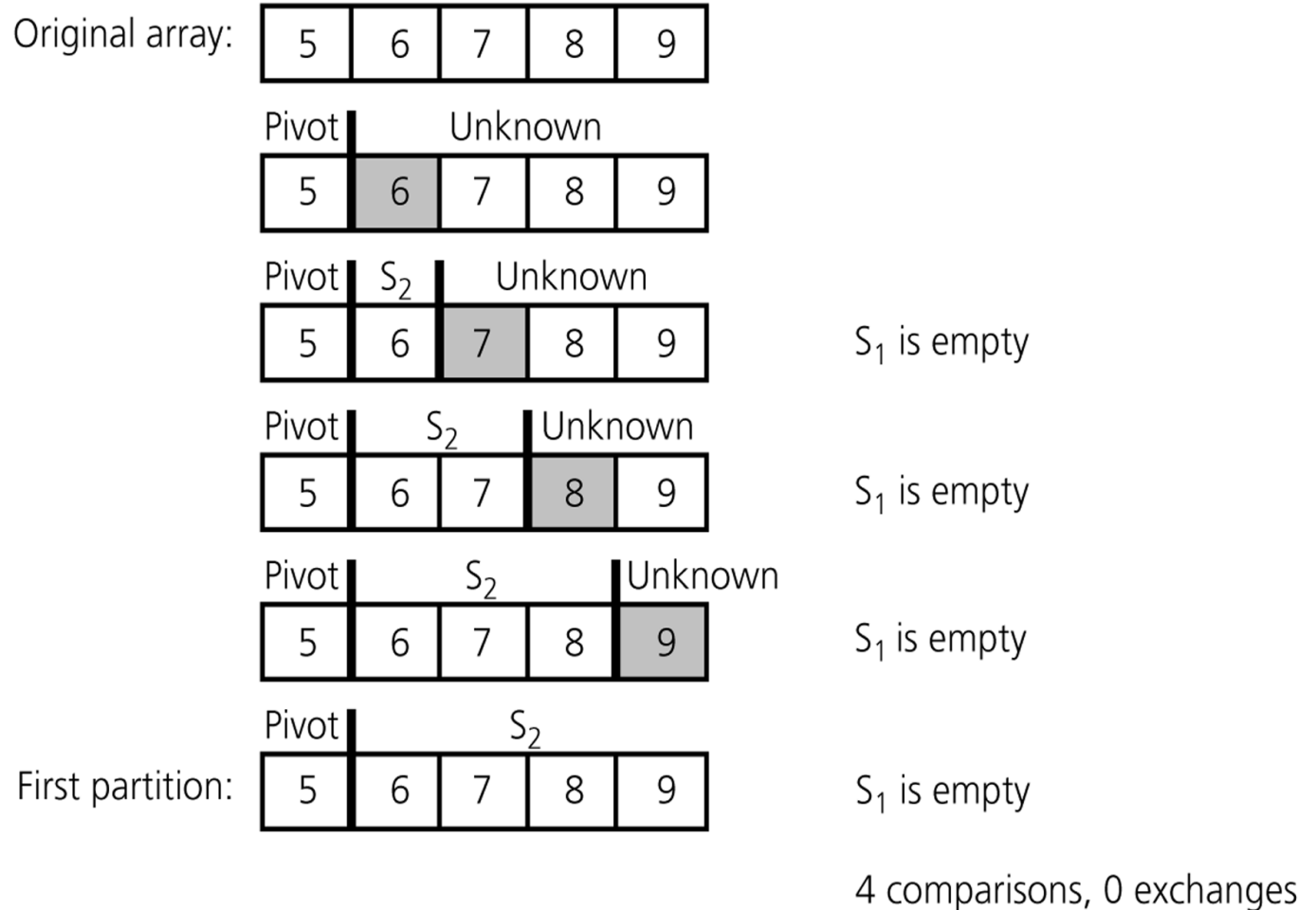
- So, Quicksort is **$O(n^2)$** in worst case

Quicksort – Analysis

- Quicksort is $O(n \cdot \log_2 n)$ 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

An average-case partitioning with quicksort

Original array:

5	3	6	7	4
---	---	---	---	---

Pivot | Unknown

5	3	6	7	4
---	---	---	---	---

Pivot | S_1 | Unknown

5	3	6	7	4
---	---	---	---	---

Pivot | S_1 | S_2 | Unknown

5	3	6	7	4
---	---	---	---	---

Pivot | S_1 | S_2 | Unknown

5	3	6	7	4
---	---	---	---	---

Pivot | S_1 | S_2

5	3	4	7	6
---	---	---	---	---

S_1 and S_2 are determined

First partition:

S_1		Pivot	S_2	
4	3	5	7	6

Place pivot between S_1 and S_2

Radix Sort

- Radix sort algorithm different than other sorting algorithms that we talked.
 - It does not use key comparisons to sort an array.
- The radix sort :
 - Treats each data item as a character string.
 - First it groups data items according to their rightmost character, and put these groups into order w.r.t. this rightmost character.
 - Then, combine these groups.
 - We, repeat these grouping and combining operations for all other character positions in the data items from the rightmost to the leftmost character position.
 - At the end, the sort operation will be completed.

Radix Sort – Example

mom, dad, god, fat, bad, cat, mad, pat, bar, him

original list

(dad, god, bad, mad) (mom, him) (bar) (fat, cat, pat)

group strings by rightmost letter

dad, god, bad, mad, mom, him, bar, fat, cat, pat

combine groups

(dad, bad, mad, bar, fat, cat, pat) (him) (god, mom)

group strings by middle letter

dad, bad, mad, bar, fat, cat, pat, him, god, mom

combine groups

(bad, bar) (cat) (dad) (fat) (god) (him) (mad, mom) (pat) group strings by first letter

bad, bar, cat, dad, fat, god, him, mad, mom, par

combine groups (SORTED)

Radix Sort – Example

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150

Original integers

(156**0**, 215**0**) (106**1**) (022**2**) (012**3**, 028**3**) (215**4**, 000**4**)

Grouped by fourth digit

1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004

Combined

(000**4**) (022**2**, 012**3**) (215**0**, 215**4**) (156**0**, 106**1**) (028**3**)

Grouped by third digit

0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283

Combined

(000**4**, 106**1**) (012**3**, 215**0**, 215**4**) (022**2**, 028**3**) (156**0**)

Grouped by second digit

0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560

Combined

(000**4**, 012**3**, 022**2**, 028**3**) (106**1**, 156**0**) (215**0**, 215**4**)

Grouped by first digit

0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154

Combined (sorted)

Radix Sort - Algorithm

```
radixSort(inout theArray:ItemArray, in n:integer, in d:integer)
// sort n d-digit integers in the array theArray
  for (j=d down to 1) {
    Initialize 10 groups to empty
    Initialize a counter for each group to 0
    for (i=0 through n-1) {
      k = jth digit of theArray[i]
      Place theArray[i] at the end of group k
      Increase kth counter by 1
    }
    Replace the items in theArray with all the items in group 0,
    followed by all the items in group 1, and so on.
  }
```

Radix Sort -- Analysis

- The radix sort algorithm requires $2*n*d$ moves to sort n strings of d characters each.
→ So, Radix Sort is $O(n)$
- Although the radix sort is $O(n)$, it is not appropriate as a general-purpose sorting algorithm.
 - Its memory requirement is $d * \textit{original size of data}$ (because each group should be big enough to hold the original data collection.)
 - For example, we need 27 groups to sort string of uppercase letters.
 - The radix sort is more appropriate for a linked list than an array. (we will not need the huge memory in this case)

Comparison of Sorting Algorithms

	<u>Worst case</u>	<u>Average case</u>
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	$n * \log n$	$n * \log n$