Sorting and Searching

Initially prepared by Dr. İlyas Çiçekli; improved by various Bilkent CS202 instructors.

Problem of the Day

True or False?

1. $2n^2 + 1 = O(n^2)$
2. $\sqrt{n} = O(\log n)$
3. $\log n = O(\sqrt{n})$
4. $n^2(1 + \sqrt{n}) = O(n^2 \log n)$
5. $3n^2 + \sqrt{n} = O(n^2)$
6. $\sqrt{n} \log n = O(n)$
7. $\log n = O(n^{-1/2})$

T
F
T
F
T
T
F(?)

Sequential Search

```
int sequentialSearch( const int a[], int item, int n){
   for (int i = 0; i < n && a[i]!= item; i++);
   if (i == n)
        return -1;
   return i;
}
```

Unsuccessful Search: ➔ O(n)

Successful Search:

Best-Case: *item* is in the first location of the array ➔ O(1)

Worst-Case: *item* is in the last location of the array ➔ O(n)

Average-Case: The number of key comparisons 1, 2, ..., n

$$\frac{\sum_{i=1}^{n} i}{n} = \frac{(n^2+n)/2}{n}$$ ➔ O(n)

Binary Search

```
int binarySearch( int a[], int size, int x) {
   int low =0;
   int high = size -1;
   int mid;              // mid will be the index of
                         // target when it's found.
   while (low <= high) {
     mid = (low + high)/2;
     if (a[mid] < x)
        low = mid + 1;
     else if (a[mid] > x)
         high  = mid - 1;
     else
         return mid;
   }
   return -1;
}
```

Binary Search – Analysis

- For an unsuccessful search:
  - The number of iterations in the loop is $\lfloor \log_2 n \rfloor + 1$ ➔ $O(\log_2 n)$

- For a successful search:
  - ***Best-Case:*** The number of iterations is 1 ➔ $O(1)$
  - ***Worst-Case:*** The number of iterations is $\lfloor \log_2 n \rfloor + 1$ ➔ $O(\log_2 n)$
  - ***Average-Case:*** The avg. # of iterations < $\log_2 n$ ➔ $O(\log_2 n)$

0	1	2	3	4	5	6	7	← an array with size 8
3	2	3	1	3	2	3	4	← # of iterations

The average # of iterations = $21/8 < \log_2 8$

How much better is $O(log_2 n)$?

n	$O(log_2 n)$
16	4
64	6
256	8
1024 (1KB)	10
16,384	14
131,072	17
262,144	18
524,288	19
1,048,576 (1MB)	20
1,073,741,824 (1GB)	30

Sorting

Importance of Sorting

Why don't CS profs ever stop talking about sorting?

1. Computers spend more time sorting than anything else, historically 25% on mainframes.

2. Sorting is the best studied problem in computer science, with a variety of different algorithms known.

3. Most of the interesting ideas we will encounter in the course can be taught in the context of sorting, such as divide-and-conquer, randomized algorithms, and lower bounds.

(slide by Steven Skiena)

Sorting

- Organize data into ascending / descending order
  - Useful in many applications
  - Any examples can you think of?

- Internal sort vs. external sort
  - We will analyze only internal sorting algorithms

- Sorting also has other uses. It can make an algorithm faster.
  - e.g., find the intersection of two sets

Efficiency of Sorting

- Sorting is important because once a set of items is sorted, many other problems become easy.
- Furthermore, using O(n log n) sorting algorithms leads naturally to sub-quadratic algorithms for these problems.
- Large-scale data processing would be impossible if sorting took $O(n^2)$ time.

n	$n^2/4$	$n \lg n$
10	*25*	33
100	2,500	*664*
1,000	250,000	*9,965*
10,000	25,000,000	*132,877*
100,000	2,500,000,000	*1,660,960*

(slide by Steven Skiena)

Applications of Sorting

- **Closest Pair:** Given n numbers, find the pair which are closest to each other.
  - Once the numbers are sorted, the closest pair will be next to each other in sorted order, so an O(n) linear scan completes the job. – Complexity of this process: O(??)

- **Element Uniqueness:** Given a set of n items, are they all unique or are there any duplicates?
  - Sort them and do a linear scan to check all adjacent pairs.
  - This is a special case of closest pair above.
  - Complexity?

- **Mode:** Given a set of n items, which element occurs the largest number of times? More generally, compute the frequency distribution.
  - How would you solve it?

Sorting Algorithms

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

- First three sorting algorithms are not so efficient, but last two are efficient sorting algorithms.

Selection Sort

Selection Sort

- List divided into two sublists, *sorted* and *unsorted*.

- Find the biggest element from the *unsorted* sublist. Swap it with the element at the end of the unsorted data.

- After each selection and swapping, imaginary wall between the two sublists move one element back.

- Sort pass: 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 sort data.

Selection Sort (cont.)

Shaded elements are selected;
boldface elements are in order.

Unsorted | Sorted

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**

Selection Sort (cont.)

```
typedef type-of-array-item DataType;

void selectionSort( DataType theArray[], int n) {
  for (int last = n-1; last >= 1; --last) {
    int largest = indexOfLargest(theArray, last+1);
    swap(theArray[largest], theArray[last]);
  }
}
```

Selection Sort (cont.)

```
int indexOfLargest(const DataType theArray[], int size) {
  int indexSoFar = 0;
  for (int currentIndex=1; currentIndex<size;++currentIndex)
  {
    if (theArray[currentIndex] > theArray[indexSoFar])
         indexSoFar = currentIndex;
  }
  return indexSoFar;
}
--------------------------------------------------------
void swap(DataType &x, DataType &y) {
   DataType temp = x;
   x = y;
   y = temp;
}
```

Selection Sort -- Analysis

- To analyze sorting, count simple operations

- For sorting, important simple operations: key comparisons and number of moves

- In **selectionSort()** function, the **for** loop executes n-1 times.

- In **selectionSort()** function, we invoke swap() once at each iteration.

  ➔ Total Swaps: n-1

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

Selection Sort – Analysis (cont.)

- In **indexOfLargest()** function, the for loop executes (from n-1 to 1), and each iteration we make one key comparison.

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

  ➔ So, Selection sort is $O(n^2)$

- The best case, worst case, and average case are the same ➔ all **$O(n^2)$**
  - Meaning: behavior of selection sort does not depend on initial organization of data.
  - Since $O(n^2)$ grows so rapidly, the selection sort algorithm is appropriate only for small n.

- Although selection sort requires $O(n^2)$ key comparisons, it only requires $O(n)$ moves.
  - Selection sort is good choice if data moves are costly but key comparisons are not costly (short keys, long records).

Insertion Sort

Insertion Sort

- *Insertion sort* is a simple sorting algorithm appropriate for small inputs.
  - Most common sorting technique used by card players.

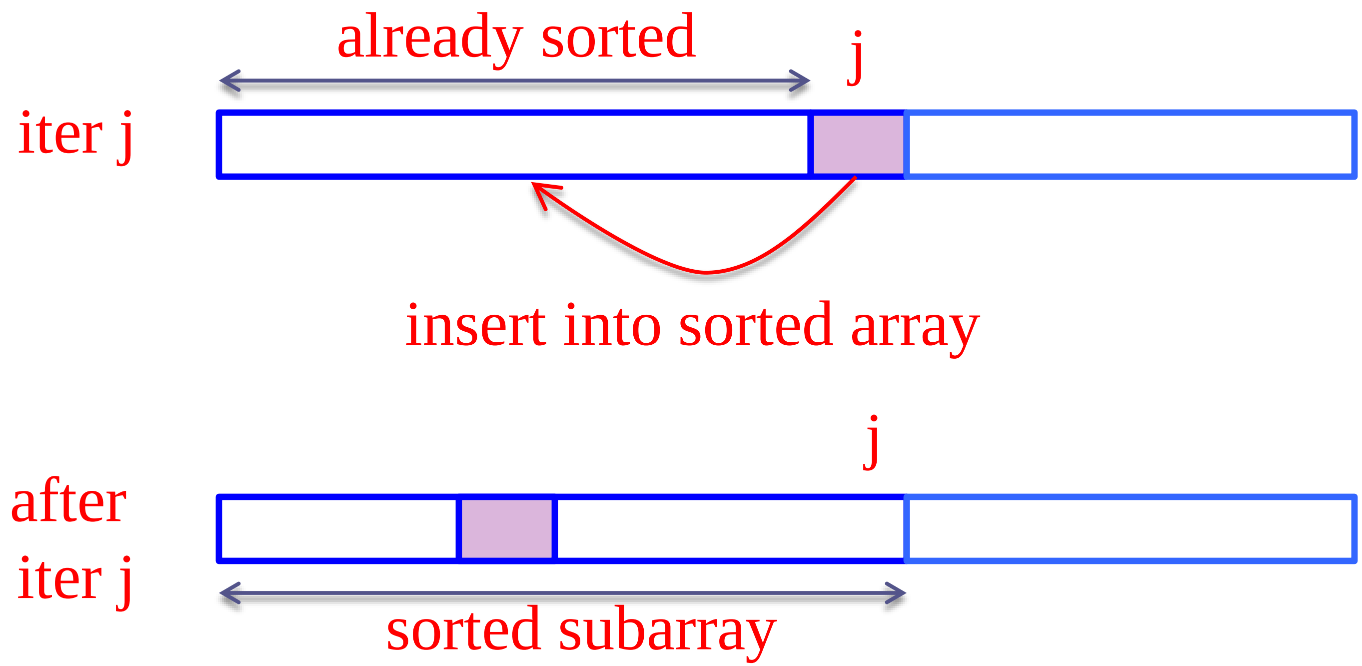
- List divided into two parts: *sorted* and *unsorted*.

- In each pass, the first element of the unsorted part is picked up, transferred to the sorted sublist, and inserted in place.

- List of *n* elements will take at most *n-1* passes to sort data.

Insertion Sort: Basic Idea

- Assume input array: A[1..n]
- Iterate j from 2 to n



Algorithm: Insertion Sort

<u>Insertion-Sort</u> (A)

```
1. for j ← 2 to n do
2.      key ← A[j];
3.      i ← j - 1;
4.      while i > 0 and A[i] > key
        do
5.          A[i+1] ← A[i];
6.          i ← i - 1;
        endwhile
7.      A[i+1] ← key;
   endfor
```

Algorithm: Insertion Sort

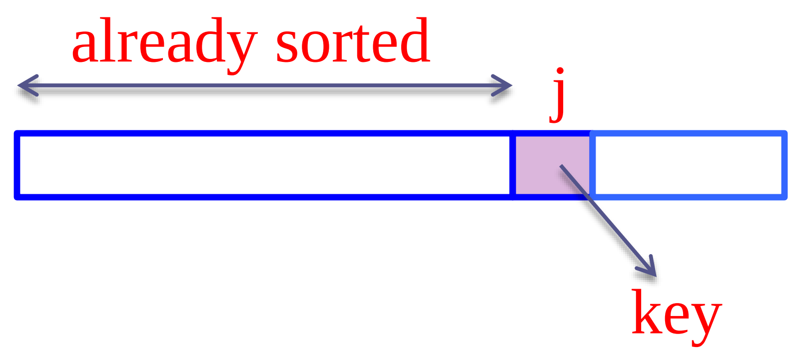
Insertion-Sort (A)

```
1. for j ← 2 to n do
2.      key ← A[j];
3.      i ← j - 1;
4.      while i > 0 and A[i] > key
        do
5.           A[i+1] ← A[i];
6.           i ← i - 1;
        endwhile
7.      A[i+1] ← key;
    endfor
```

Iterate over array elts j

Loop invariant:

The subarray A[1..j-1] is always sorted

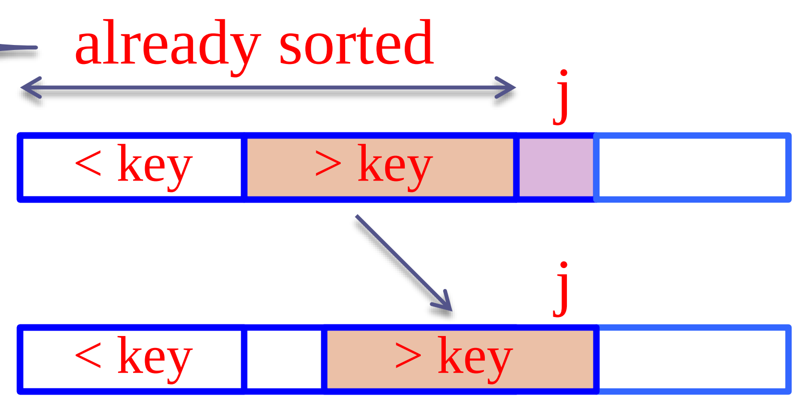


Algorithm: Insertion Sort

Insertion-Sort (A)

```
1. for j ← 2 to n do
2.      key ← A[j];
3.      i ← j - 1;
4.      while i > 0 and A[i] > key
        do
5.           A[i+1] ← A[i];
6.           i ← i - 1;
        endwhile
7.      A[i+1] ← key;
   endfor
```

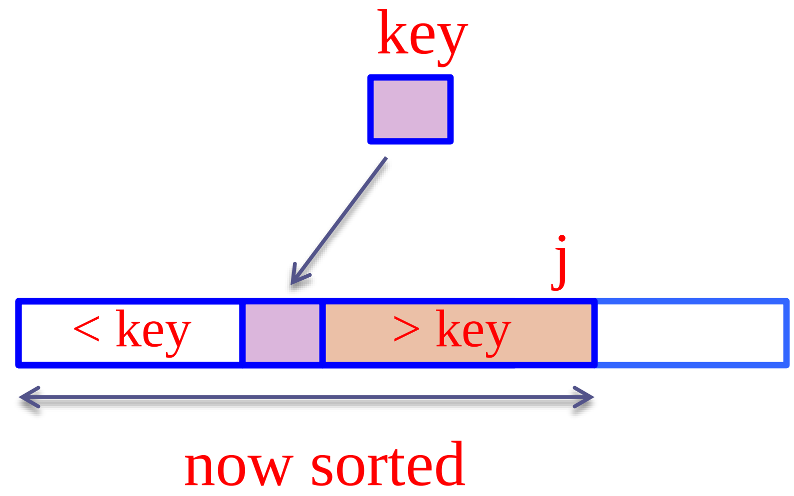
Shift right the entries in A[1..j-1] that are > key



Algorithm: Insertion Sort

Insertion-Sort (A)

```
1. for j ← 2 to n do
2.       key ← A[j];
3.       i ← j - 1;
4.       while i > 0 and A[i] > key
         do
5.            A[i+1] ← A[i];
6.            i ← i - 1;
         endwhile
7.       A[i+1] ← key;
   endfor
```



Insert key to the correct location

End of iter j: A[1..j] is sorted

Insertion Sort - Example

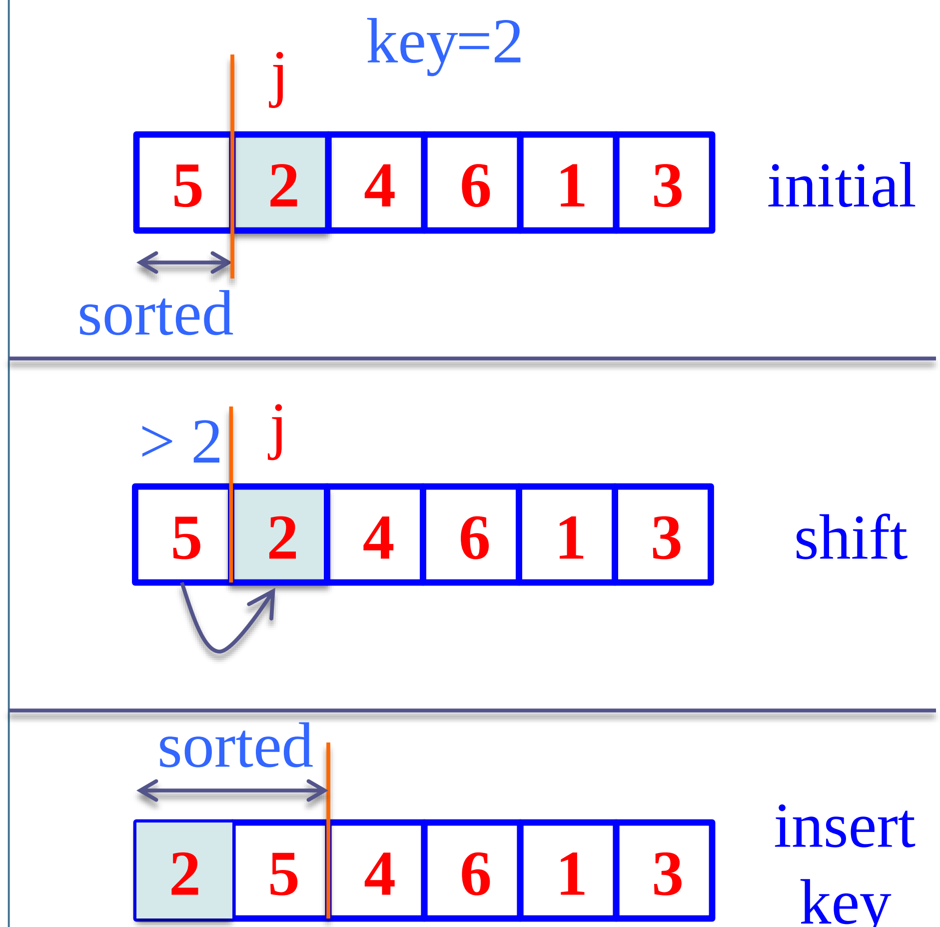
<u>Insertion-Sort</u> (A)

```
1.  for j ← 2 to n do
2.        key ← A[j];
3.        i ← j - 1;
4.        while i > 0 and A[i] > key
          do
5.             A[i+1] ← A[i];
6.             i ← i - 1;
          endwhile
7.        A[i+1] ← key;
      endfor
```

5	2	4	6	1	3

Insertion Sort - Example: Iteration j=2

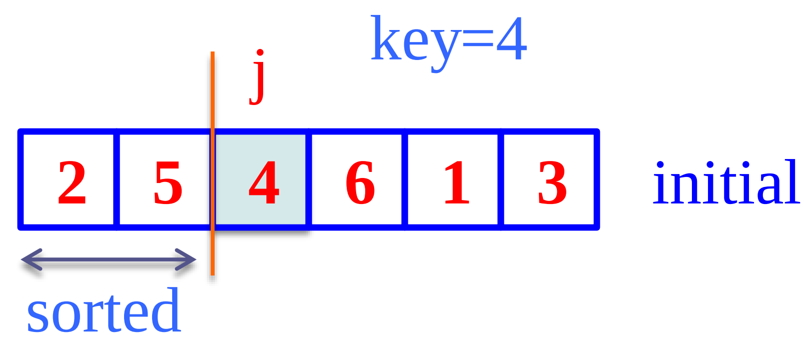
```
Insertion-Sort (A)
1.  for j ← 2 to n do
2.        key ← A[j];
3.        i ← j - 1;
4.        while i > 0 and A[i] > key
          do
5.            A[i+1] ← A[i];
6.            i ← i - 1;
          endwhile
7.        A[i+1] ← key;
      endfor
```



Insertion Sort - Example: Iteration j=3

Insertion-Sort (A)

```
1.  for j ← 2 to n do
2.      key ← A[j];
3.      i ← j - 1;
4.      while i > 0 and A[i] > key
        do
5.          A[i+1] ← A[i];
6.          i ← i - 1;
        endwhile
7.      A[i+1] ← key;
    endfor
```



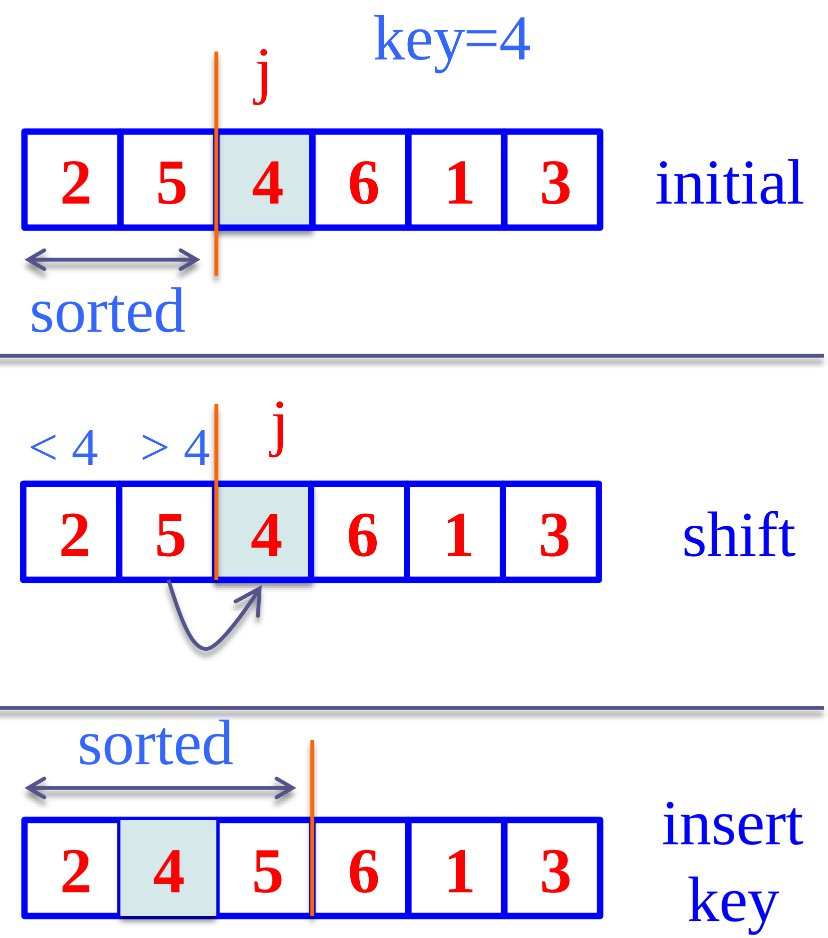
What are the entries at the end of iteration j=3?

?	?	?	?	?	?

Insertion Sort - Example: Iteration j=3

Insertion-Sort (A)

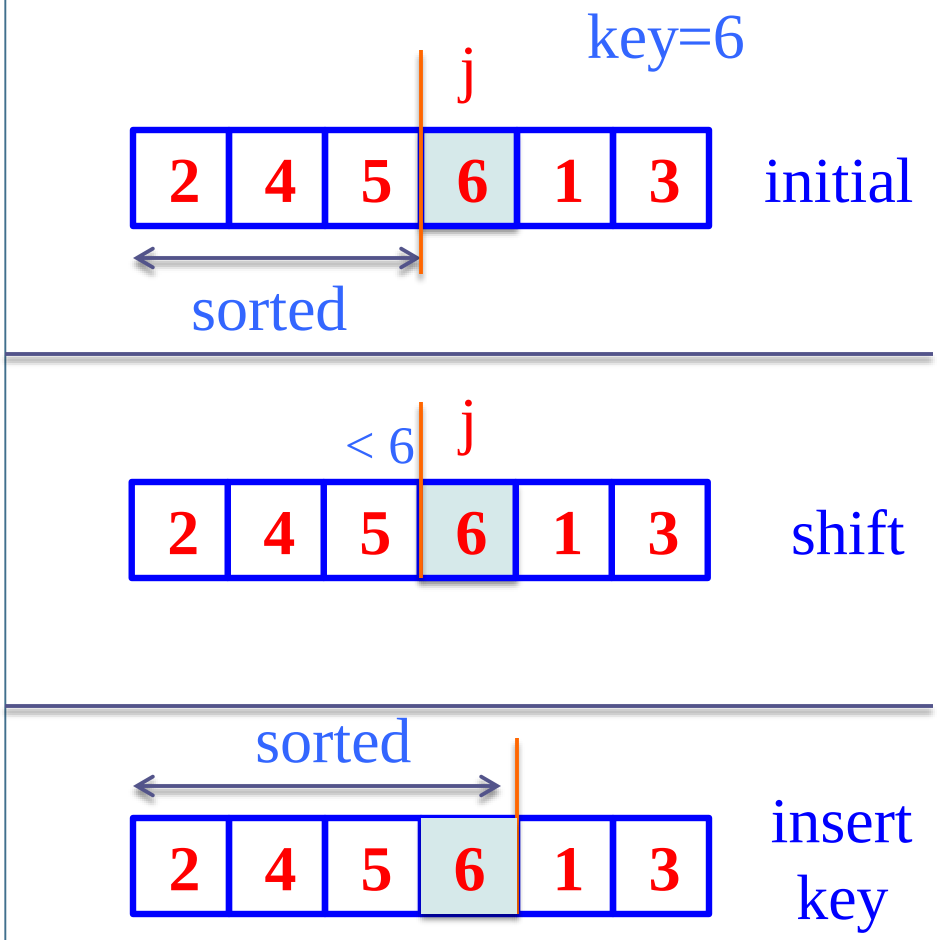
```
1.  for j ← 2 to n do
2.      key ← A[j];
3.      i ← j - 1;
4.      while i > 0 and A[i] > key
        do
5.          A[i+1] ← A[i];
6.          i ← i - 1;
        endwhile
7.      A[i+1] ← key;
    endfor
```



Insertion Sort - Example: Iteration j=4

Insertion-Sort (A)

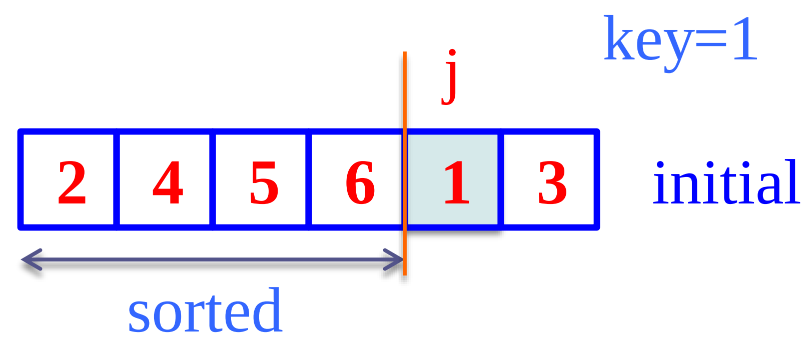
```
1.  for j ← 2 to n do
2.      key ← A[j];
3.      i ← j - 1;
4.      while i > 0 and A[i] > key
        do
5.          A[i+1] ← A[i];
6.          i ← i - 1;
        endwhile
7.      A[i+1] ← key;
    endfor
```



Insertion Sort - Example: Iteration j=5

Insertion-Sort (A)

```
1. for j ← 2 to n do
2.     key ← A[j];
3.     i ← j - 1;
4.     while i > 0 and A[i] > key
       do
5.         A[i+1] ← A[i];
6.         i ← i - 1;
       endwhile
7.     A[i+1] ← key;
   endfor
```



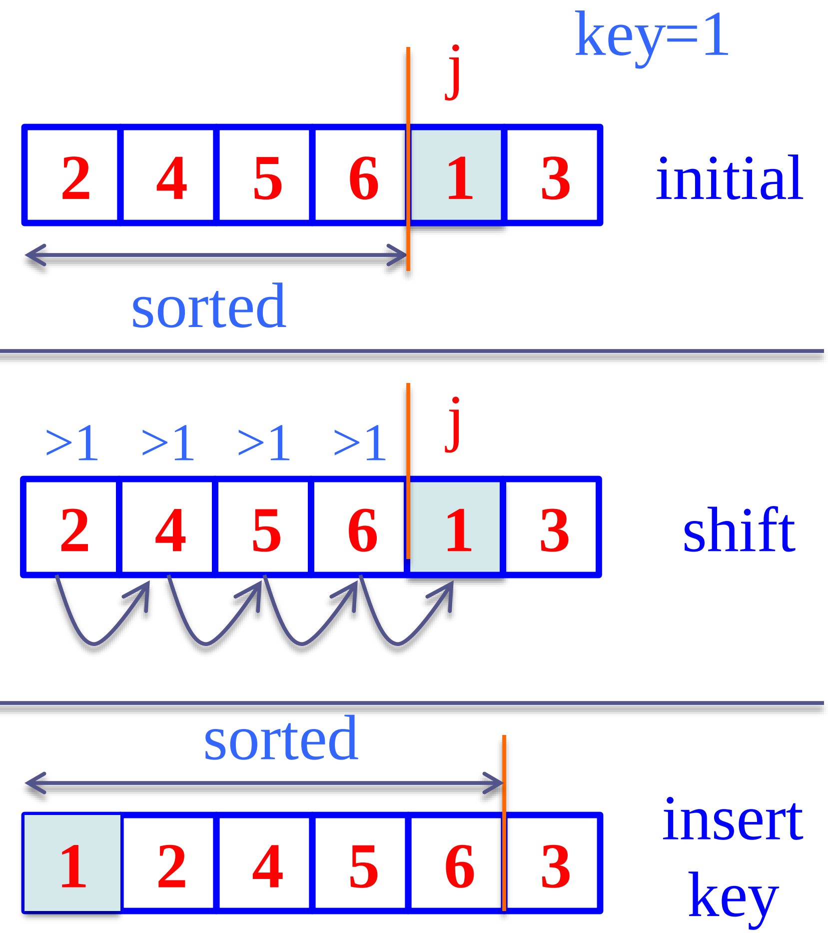
What are the entries at the end of iteration j=5?

?	?	?	?	?	?

Insertion Sort - Example: Iteration j=5

<u>Insertion-Sort</u> (A)

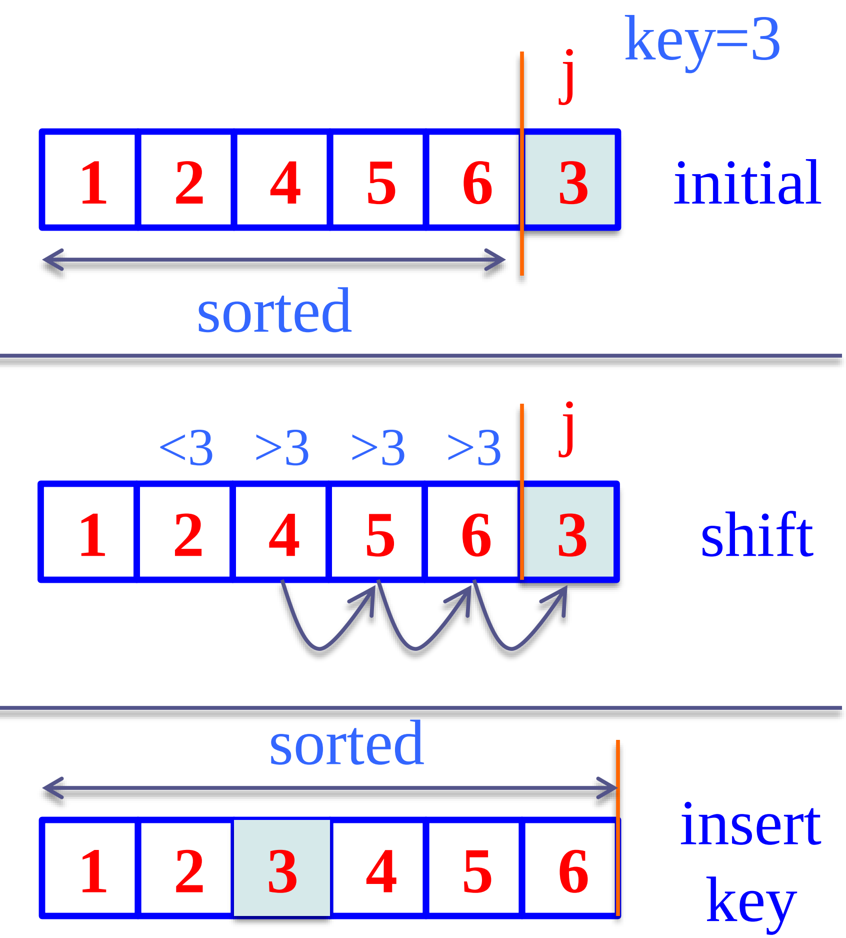
```
1.  for j ← 2 to n do
2.       key ← A[j];
3.       i ← j - 1;
4.       while i > 0 and A[i] > key
         do
5.            A[i+1] ← A[i];
6.            i ← i - 1;
         endwhile
7.       A[i+1] ← key;
    endfor
```



Insertion Sort - Example: Iteration j=6

Insertion-Sort (A)

```
1. for j ← 2 to n do
2.       key ← A[j];
3.       i ← j - 1;
4.       while i > 0 and A[i] > key
         do
5.             A[i+1] ← A[i];
6.             i ← i - 1;
         endwhile
7.       A[i+1] ← key;
     endfor
```



Insertion Sort Algorithm - Notes

- Items sorted in-place
  - Elements rearranged within array
  - At most constant number of items stored outside the array at any time (e.g. the variable *key*)
  - Input array A contains sorted output sequence when the algorithm ends

- Incremental approach
  - Having sorted A[1..j-1], place A[j] correctly so that A[1..j] is sorted

Insertion Sort (in C++)

```
void insertionSort(DataType theArray[], int n) {

  for (int unsorted = 1; unsorted < n; ++unsorted) {

    DataType nextItem = theArray[unsorted];
    int loc = unsorted;

    for (  ;(loc > 0) && (theArray[loc-1] > nextItem); --loc)
       theArray[loc] = theArray[loc-1];

    theArray[loc] = nextItem;
  }
}
```

Insertion Sort (cont.)

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 – Analysis

- What is the complexity of insertion sort? ➔
- ***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 p-1 times, for p = 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)**

Insertion Sort – Analysis

- Which running time will be used to characterize this algorithm?
  - Best, worst or average?

➔ **Worst case:**

  - 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 problems:
  - Difficult to figure out average case. i.e. what is the average input?
  - Are we going to assume all possible inputs are equally likely?
  - In fact, for most algorithms average case is the same as the worst case.

Bubble Sort

Bubble Sort

- List divided into two sublists: *sorted* and *unsorted*.

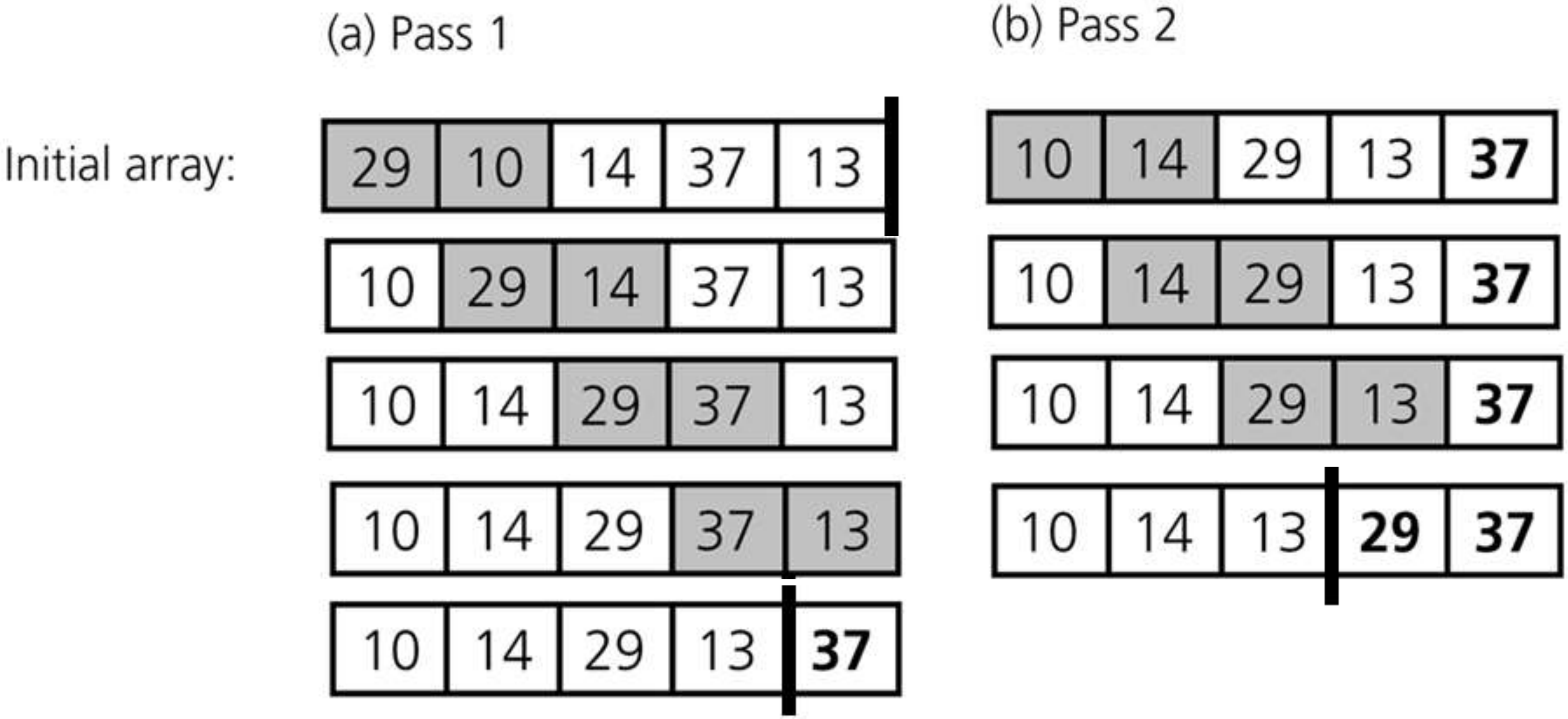
- The largest element is bubbled from the unsorted list and moved to the sorted sublist.

- After that, the wall moves one element back, increasing the number of sorted elements and decreasing the number of unsorted ones.

- One sort pass: each time an element moves from the unsorted part to the sorted part.

- Given a list of *n* elements, bubble sort requires up to n-1 passes (maximum passes) to sort data.

Bubble Sort (cont.)



Bubble Sort (cont.)

```
void bubbleSort( DataType theArray[], int n) {
   bool sorted = false;

   for (int pass = 1; (pass < n) && !sorted; ++pass) {
      sorted = true;
      for (int index = 0; index < n-pass; ++index) {
         int nextIndex = index + 1;
         if (theArray[index] > theArray[nextIndex]) {
            swap(theArray[index], theArray[nextIndex]);
            sorted = false; // signal exchange
         }
      }
   }
}
```

Bubble Sort – Analysis

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

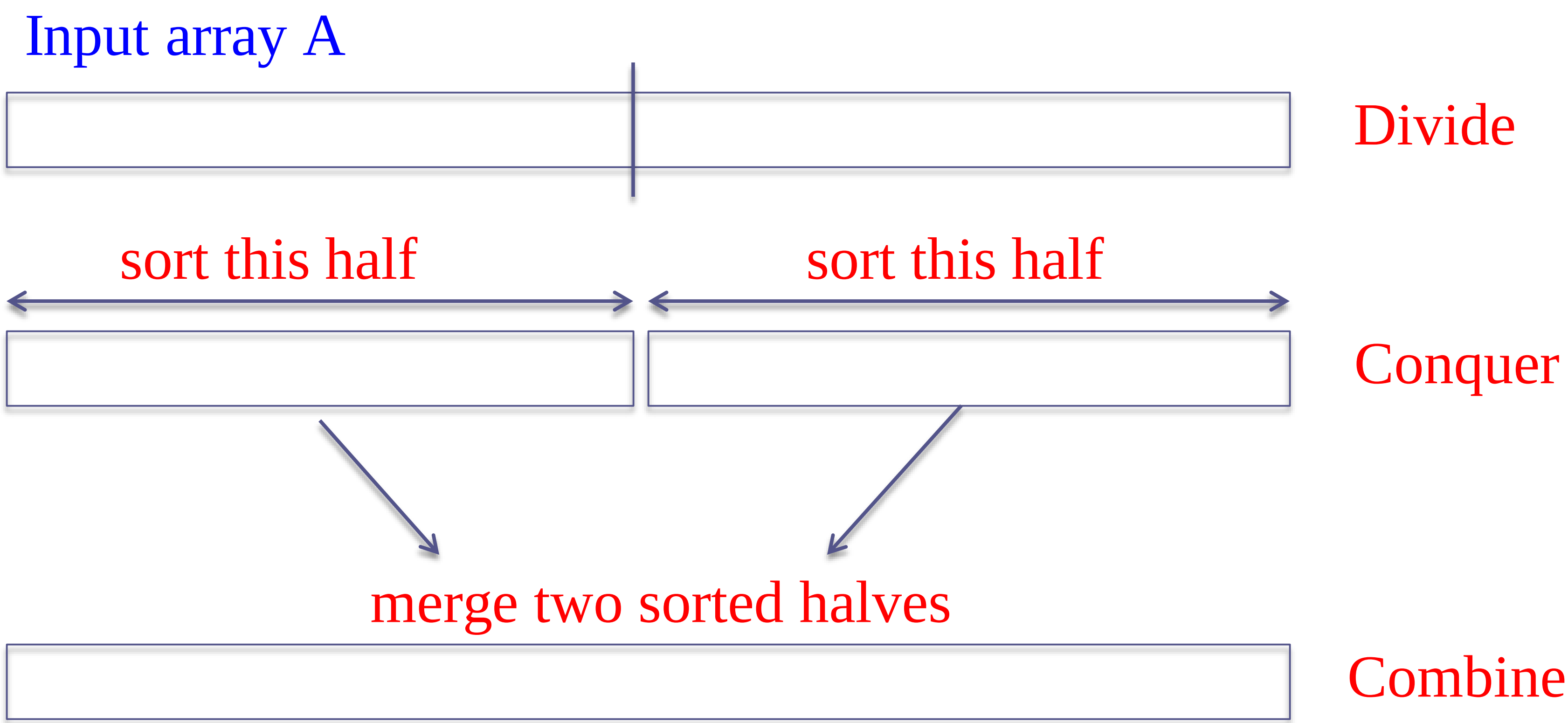
Merge Sort

Mergesort

- One of two important **divide-and-conquer sorting algorithms**
  - Other one is Quicksort

- It is a recursive algorithm.
  - Divide the list into halves,
  - Sort each half separately, and
  - Then merge the sorted halves into one sorted array.

Merge Sort: Basic Idea



Merge-Sort (A, p, r)

if p = r **then return;**

else

q ← $\lfloor (p+r)/2 \rfloor$; *(Divide)*

Merge-Sort (A, p, q); *(Conquer)*

Merge-Sort (A, q+1, r); *(Conquer)*

Merge (A, p, q, r); *(Combine)*

endif

- Call Merge-Sort(A,1,n) to sort A[1..n]
- Recursion bottoms out when subsequences have length 1

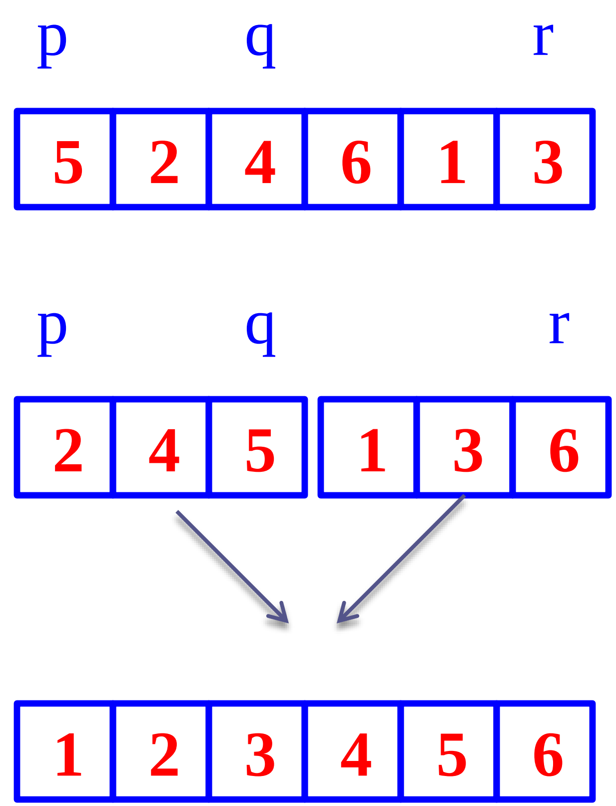
Merge Sort: Example

Merge-Sort (A, p, r)

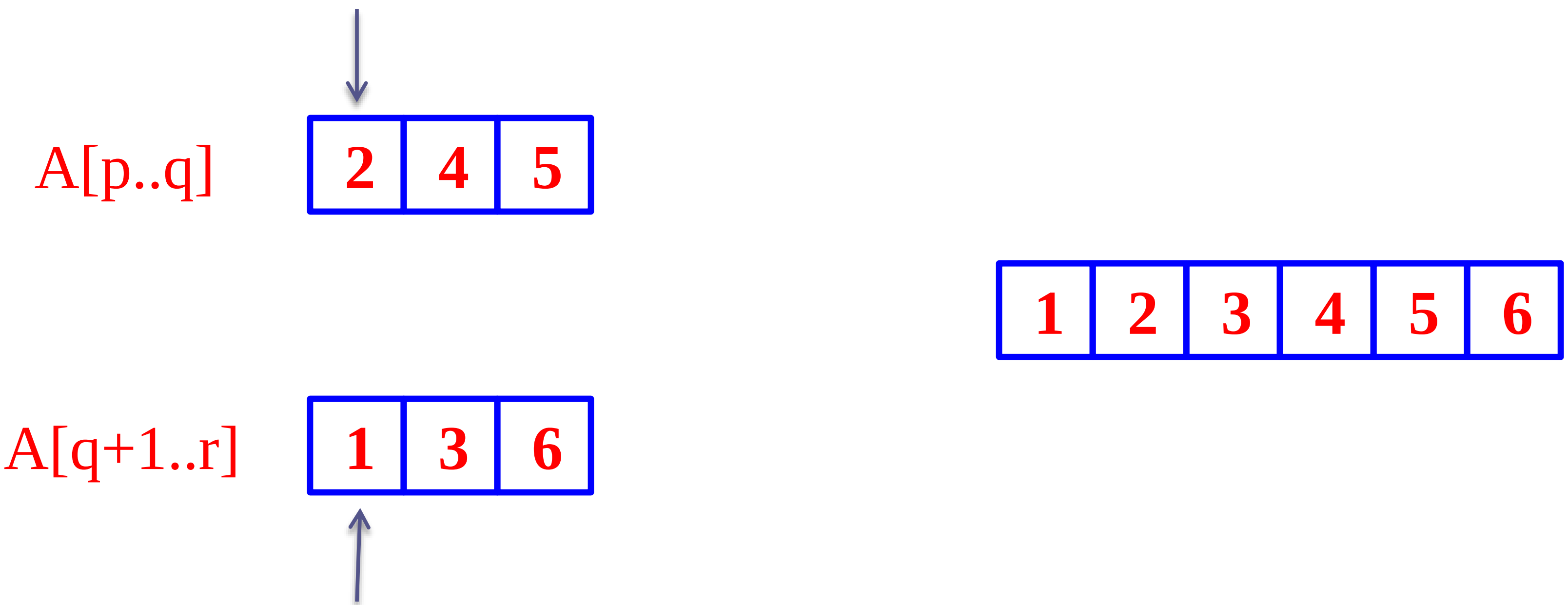
if p = r **then**
    return
else
    q ← ⌊ (p+r)/2⌋

    Merge-Sort (A, p, q)
    Merge-Sort (A, q+1, r)

    Merge(A, p, q, r)
endif



How to merge 2 sorted subarrays?



- What is the complexity of this step? $\Theta(n)$

Merge Sort: Correctness

Merge-Sort (A, p, r)

if p = r **then**
  return
else
  q ← ⌊ (p+r)/2⌋

  Merge-Sort (A, p, q)
  Merge-Sort (A, q+1, r)

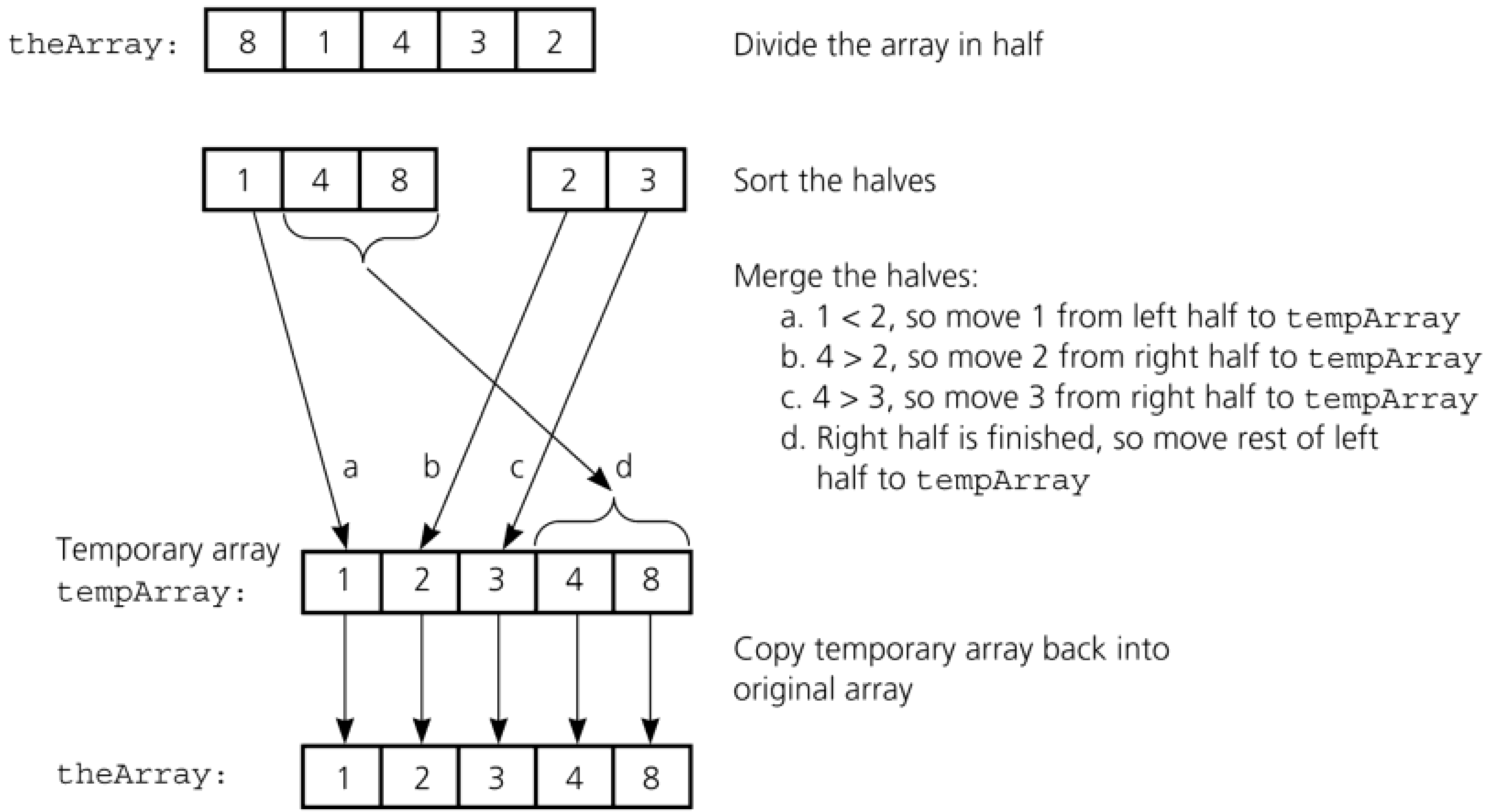
  Merge(A, p, q, r)
endif

Base case: p = r
→ Trivially correct

Inductive hypothesis: MERGE-SORT is correct for any subarray that is a ***strict*** (smaller) ***subset*** of A[p, q].

General Case: MERGE-SORT is correct for A[p, q].
→ From inductive hypothesis and correctness of *Merge*.

Mergesort – Another Example



Mergesort (in C++)

```
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
```

first recursion is done before second recursion starts (left side of the merge sort is done first)

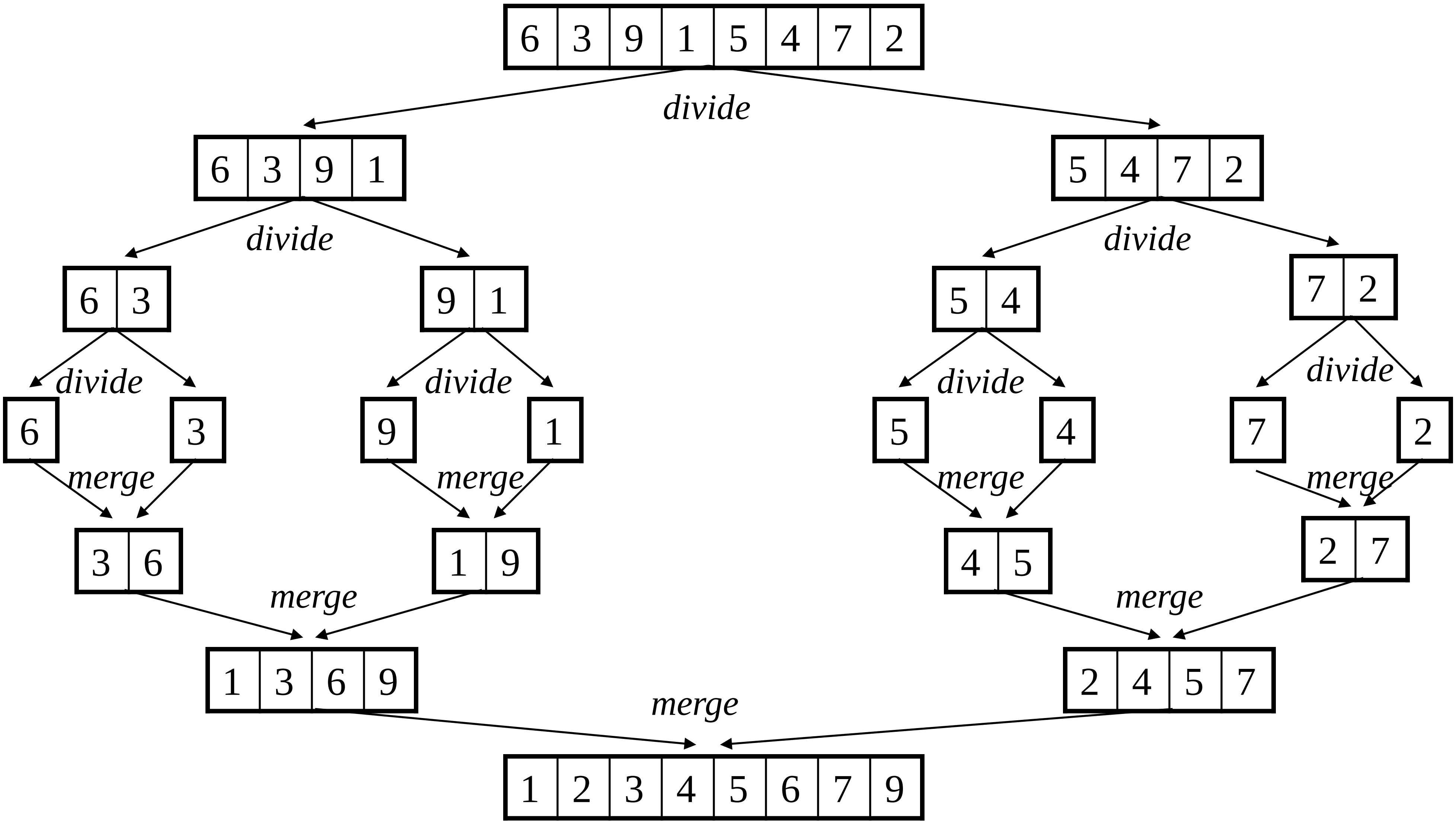
Merge

```
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;
      }
   } …
```

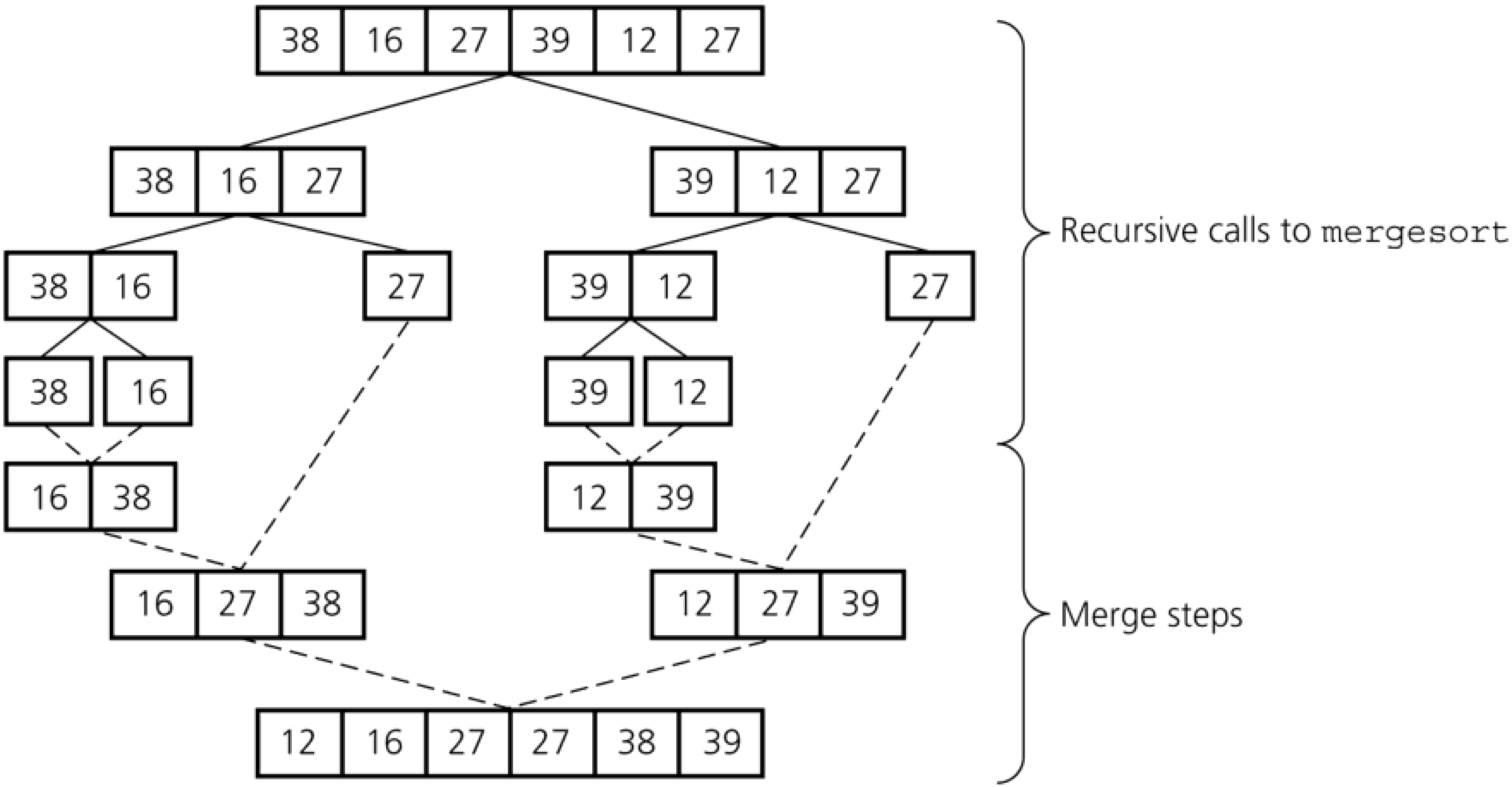
Merge (cont.)

```
   …
   // 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 - Example

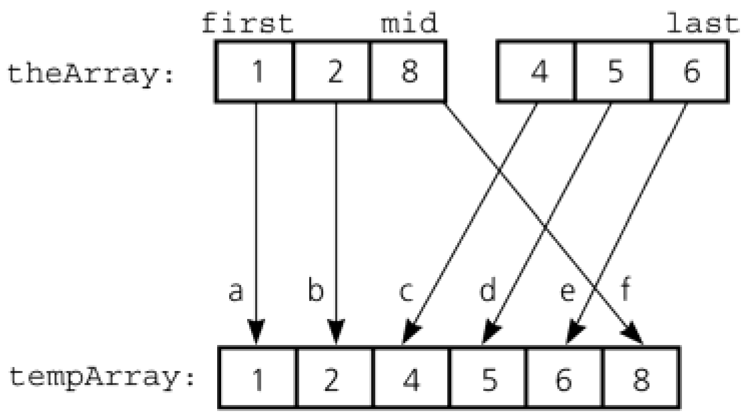


Mergesort – Example2



Mergesort – Analysis of Merge

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

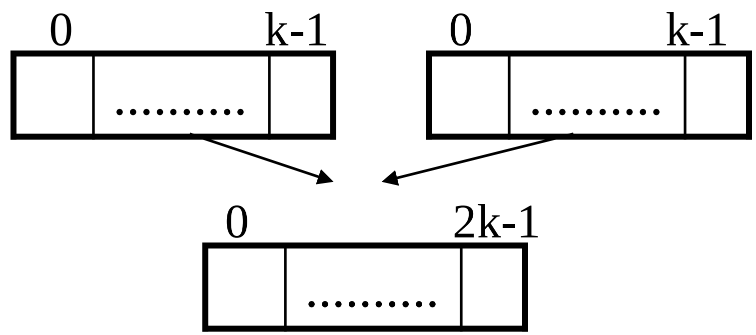


Merge the halves:

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

    2k since copy array to orig array (not inplace)
  - The number of key comparisons: k

    one array part remains same until other one is copied all

- ***Worst-case:***
  - The number of moves: 2k + 2k
  - The number of key comparisons: 2k-1

Merge Sort: Complexity

Merge-Sort (A, p, r)	$\longrightarrow$ T(n)
if p = r **then** **return**	$\longrightarrow$ $\Theta(1)$
else	
q $\leftarrow \lfloor (p+r)/2 \rfloor$	$\longrightarrow$ $\Theta(1)$
Merge-Sort (A, p, q)	$\longrightarrow$ T(n/2)
Merge-Sort (A, q+1, r)	$\longrightarrow$ T(n/2)
Merge(A, p, q, r)	$\longrightarrow$ $\Theta(n)$
endif	

Merge Sort – Recurrence

- Describe a function recursively in terms of itself
- To analyze the performance of recursive algorithms

- For merge sort:

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

How to solve for T(n)?

$$T(n) = \begin{cases} \Theta(1) & \textit{if } n=1 \\ 2T(n/2) + \Theta(n) & \textit{otherwise} \end{cases}$$

- Generally, we will assume $T(n) = \Theta(1)$ for sufficiently small n
- The recurrence above can be rewritten as:

$$T(n) = 2\,T(n/2) + \Theta(n)$$

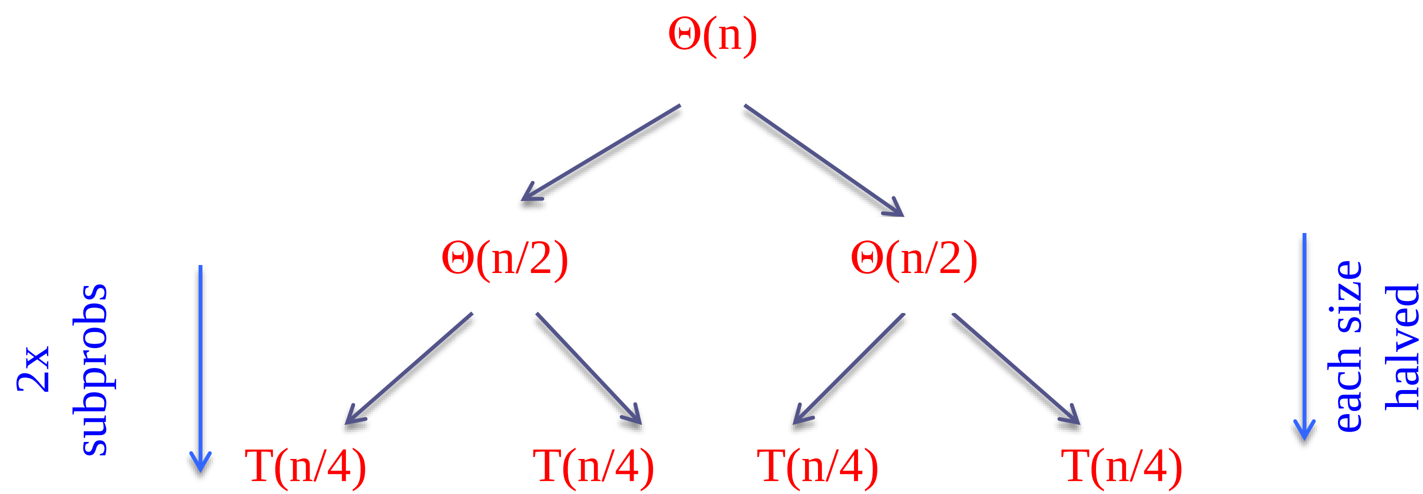
- How to solve this recurrence?

Solve Recurrence: $T(n) = 2T(n/2) + \Theta(n)$

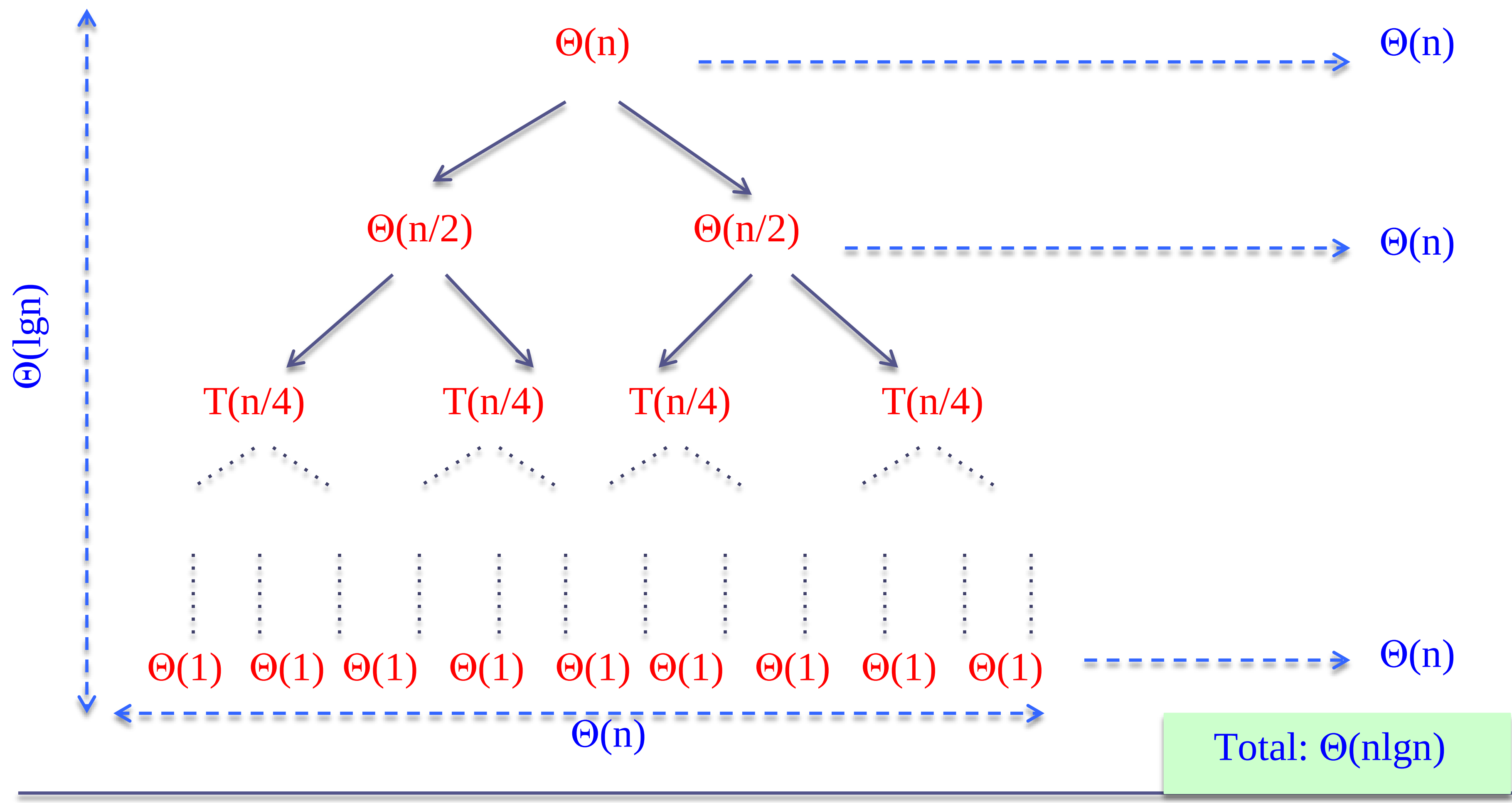
$\Theta(n)$

$T(n/2)$ $T(n/2)$

Solve Recurrence: $T(n) = 2T(n/2) + \Theta(n)$

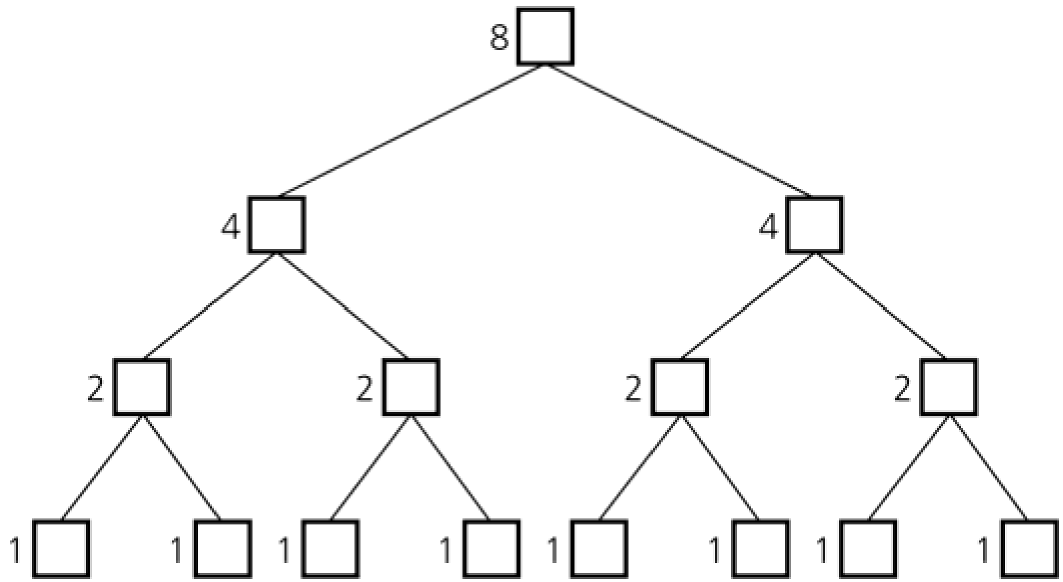


Solve Recurrence: $T(n) = 2T(n/2) + \Theta(n)$



Mergesort - Analysis

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



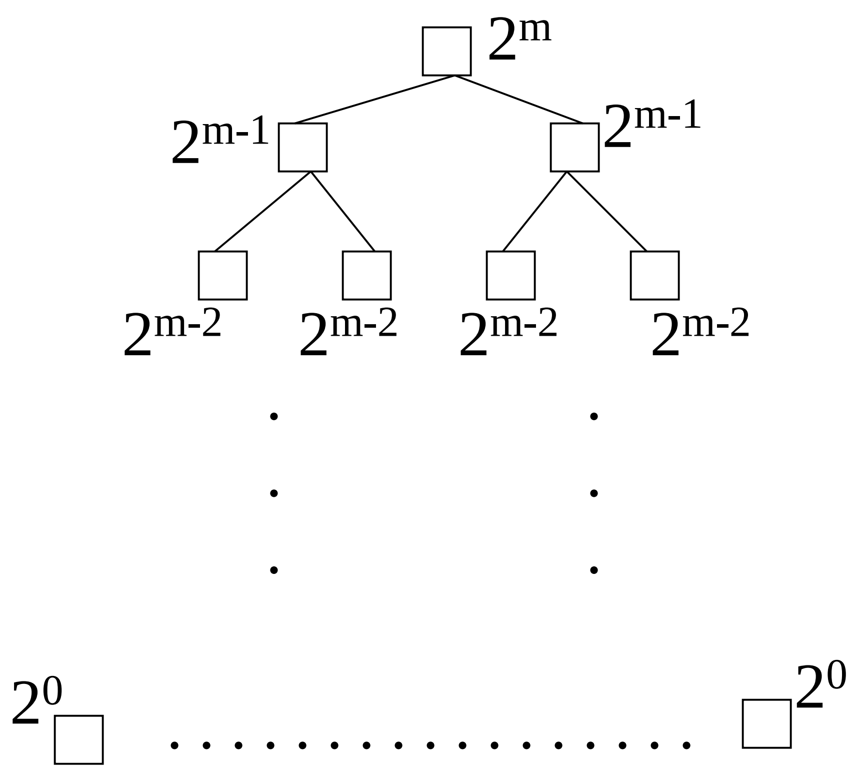
Level 0: `mergesort` 8 items

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

level m

Mergesort - Analysis

- ***Worst-case –***

The number of key comparisons:

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

$= (2^m - 1) + (2^m - 2) + \dots + (2^m – 2^{m-1})$ (m terms)

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

$= m*2^m – (2^m – 1)$

$\mathbf{= n * \log_2 n – n + 1}$

➔ **O (n * $\log_2$n)**

Mergesort – Average Case

- There are $\binom{2k}{k}$ possibilities when merging two sorted lists of size k.
- k=2 ➔ $\binom{4}{2} = \frac{4!}{2!*2!} = 6$ different cases

# of key comparisons = ((2*2)+(4*3)) / 6 = 16/6 = 2 + 2/3

Average # of key comparisons in mergesort is

$n * \log_2 n – 1.25*n – O(1)$

➔ **$O(n * \log_2 n)$**

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

Quicksort

- Like Mergesort, Quicksort is based on *divide-and-conquer* paradigm.

- But somewhat opposite to Mergesort
  - Mergesort: Hard work done *after* recursive call
  - Quicksort: Hard work done *before* recursive call

- **Algorithm**

1. First, partition an array into two parts,
2. Then, sort each part independently,
3. Finally, combine sorted parts by a simple concatenation.

Quicksort (cont.)

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

1. Divide: Partition the list.

1.1 Choose some element from list. Call this element the ***pivot***.

- We hope about half the elements will come before and half after.

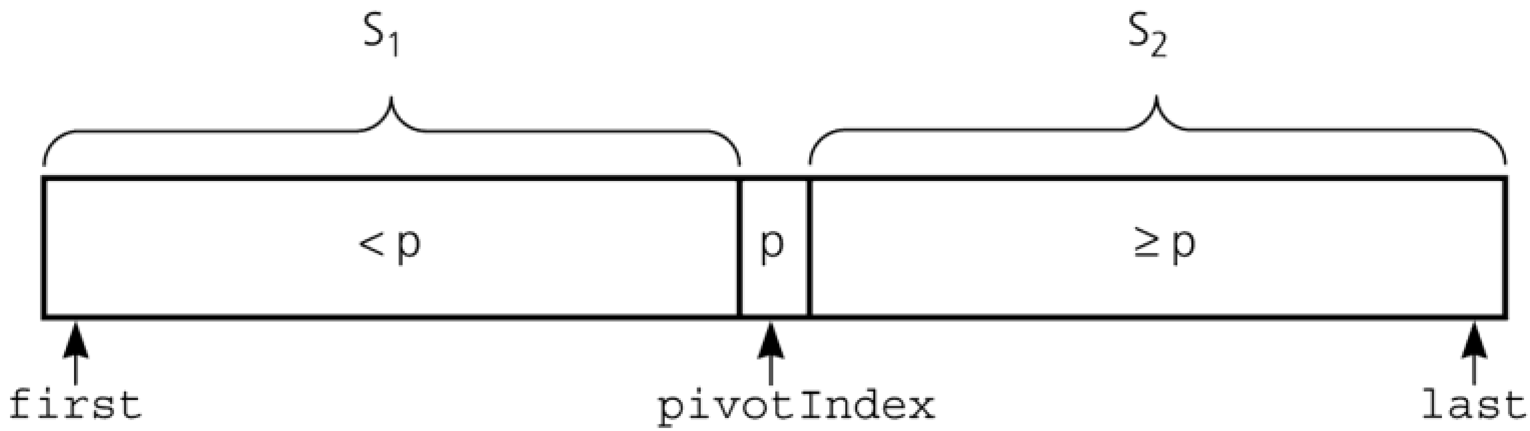
1.2 Then partition the elements so that all those with values less than the pivot come in one sublist and all those with values greater than or equal to come in another.

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

Divide: Partition the array around a pivot element

1. Choose a **pivot** element x
2. Rearrange the array such that:

   Left subarray: All elements $< x$

   Right subarray: All elements $\geq x$

Input:

5	3	2	7	4	1	3	6

e.g. $x = 5$

After partitioning:

3	3	2	1	4	5	7	6

< 5 ≥ 5

Conquer: Recursively Sort the Subarrays

Note: Everything in the left subarray < everything in the right subarray

3	3	2	1	4	5	7	6

sort recursively **sort recursively**

After conquer:

1	2	3	3	4	5	6	7

Note: Combine is trivial after conquer. Array already sorted.

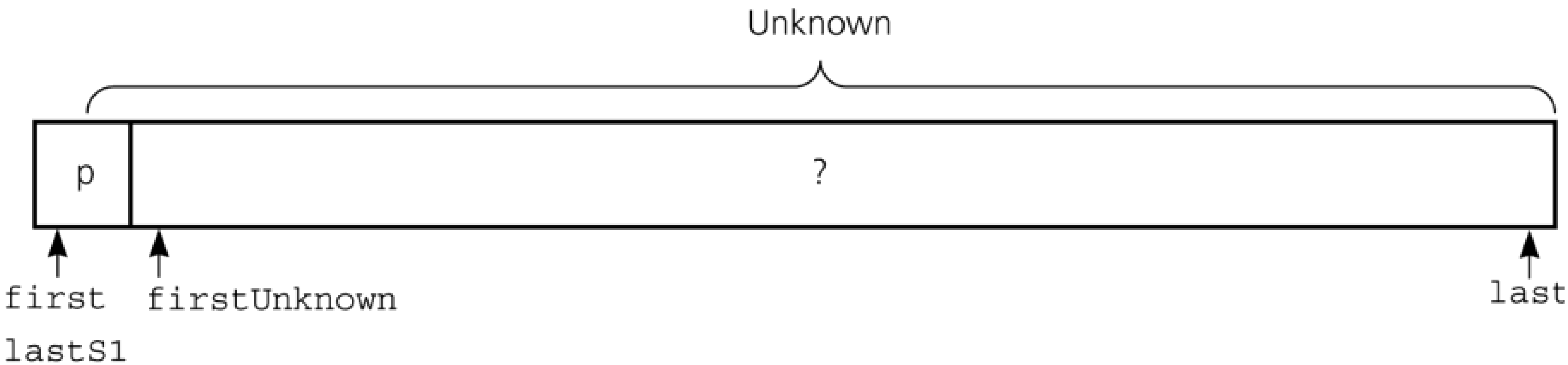
Partition – Choosing the pivot

- First, select a pivot element among the elements of the given array, and put pivot into 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.

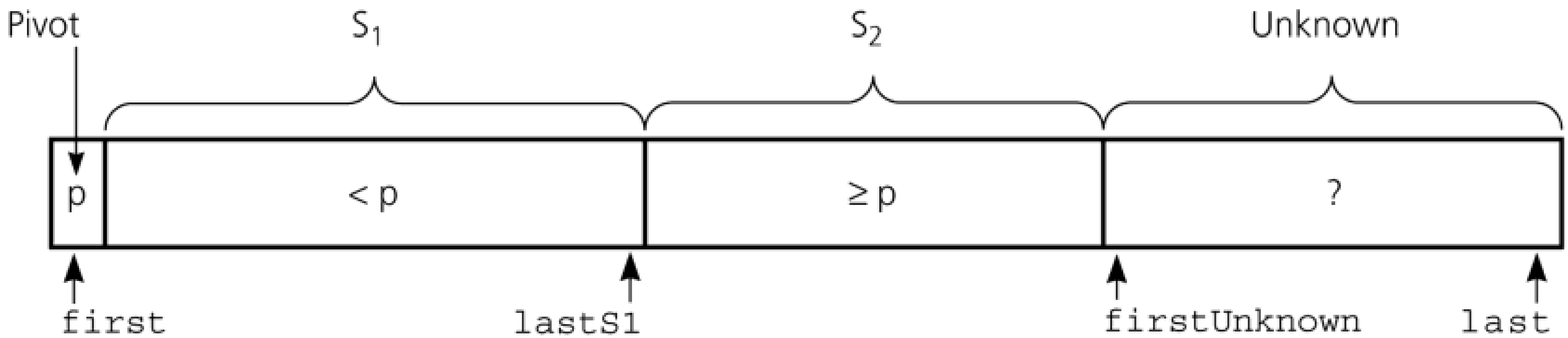
Partition Function (cont.)

Initial state of the array



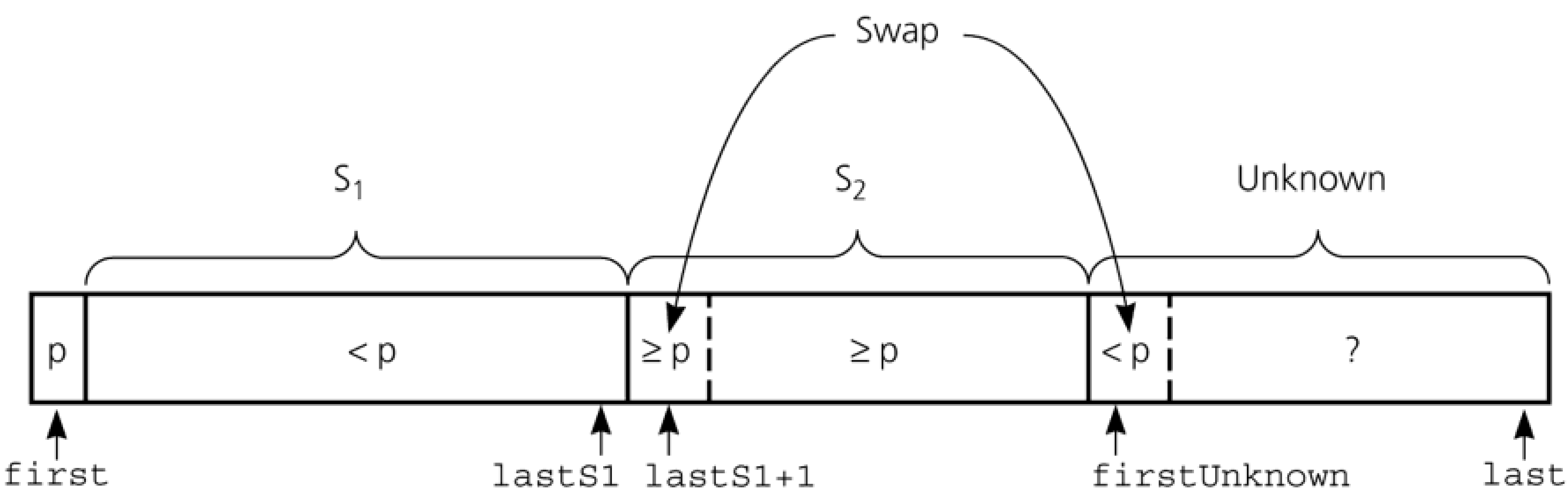
Partition Function (cont.)

Invariant for the partition algorithm



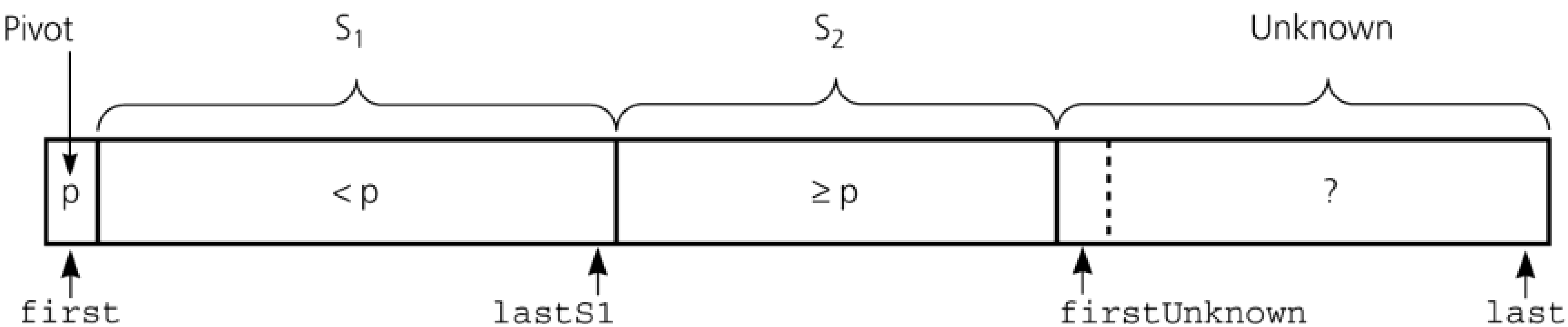
Partition Function (cont.)

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



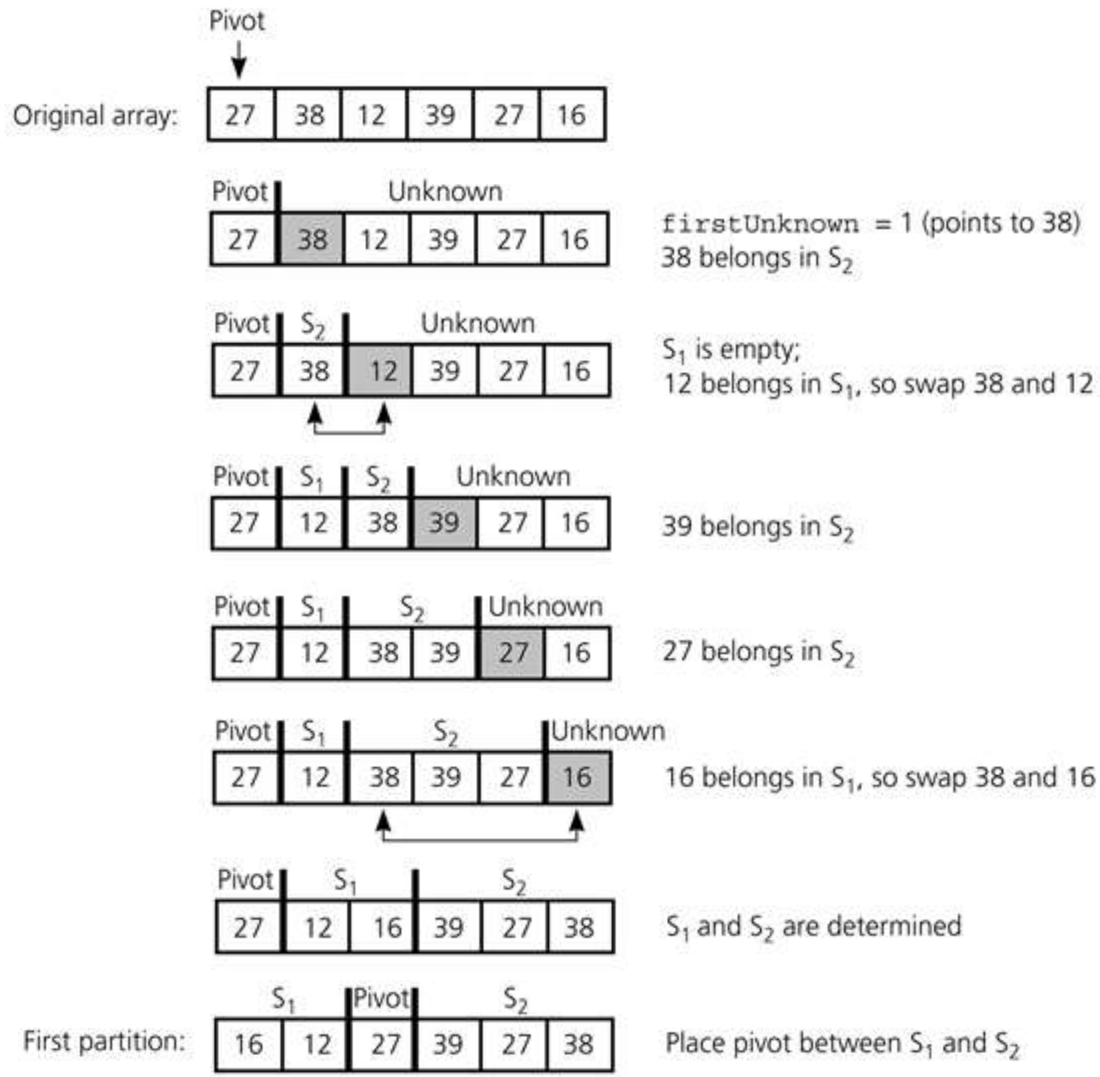
Partition Function (cont.)

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) {
// Precondition: theArray[first..last] is an array.
// Postcondition: theArray[first..last] is sorted.

   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);
    }
}
```

Partition Function

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

   // 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 first 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;
}
```

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

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

  $= \mathbf{n^2/2 - n/2}$ ➔ **O(n^2)**
- The number of swaps =

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

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

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

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

Quicksort – Analysis

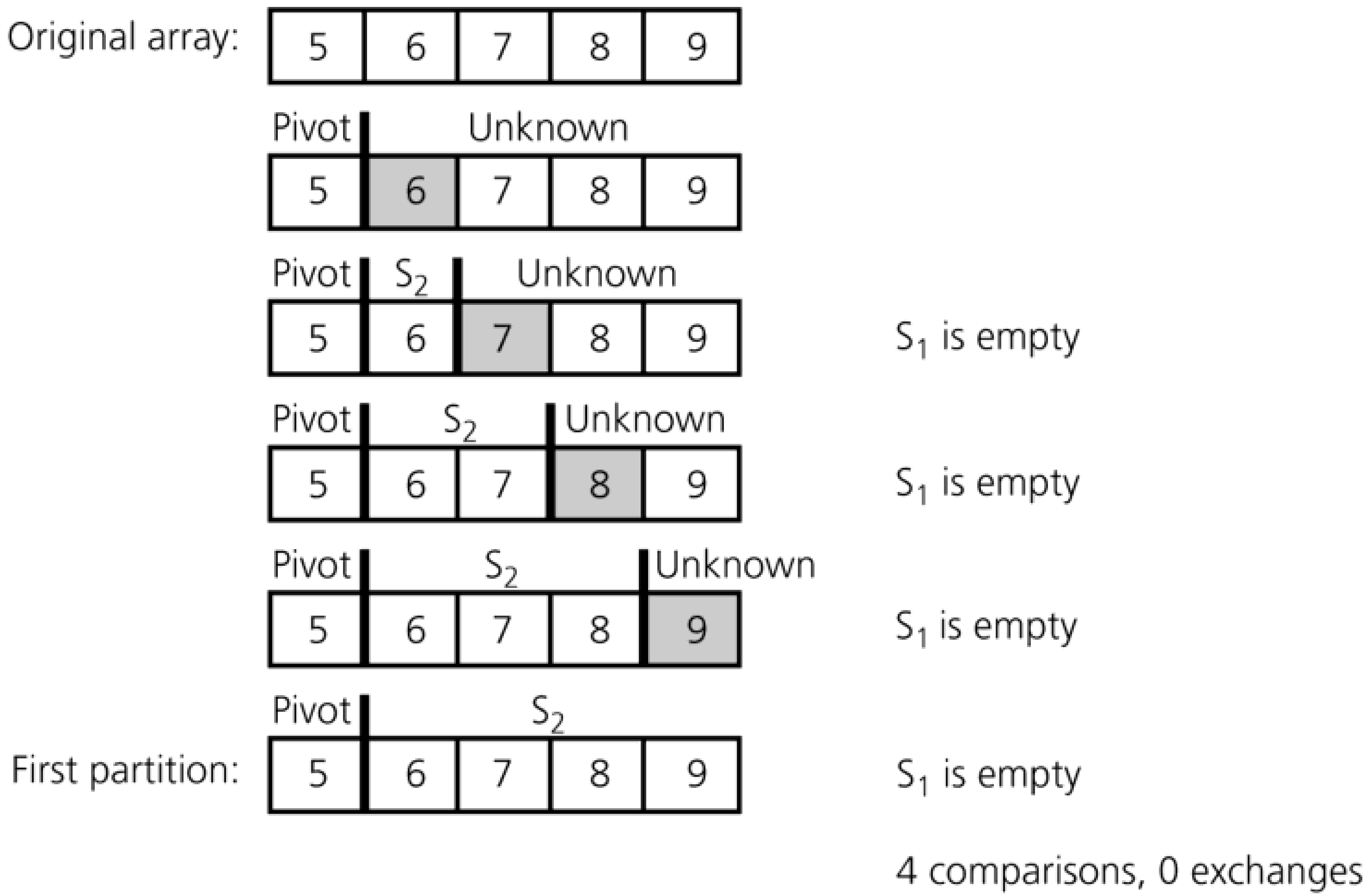
- Quicksort is **$O(n*\log_2 n)$** in the best case and average case.

- Quicksort is **slow** when the array is **already sorted** and we choose the **first element as the pivot**.

- Although the worst case behavior is not so good, and its average case behavior is much better than its worst case.
  - So, Quicksort is one of best sorting algorithms using key comparisons.

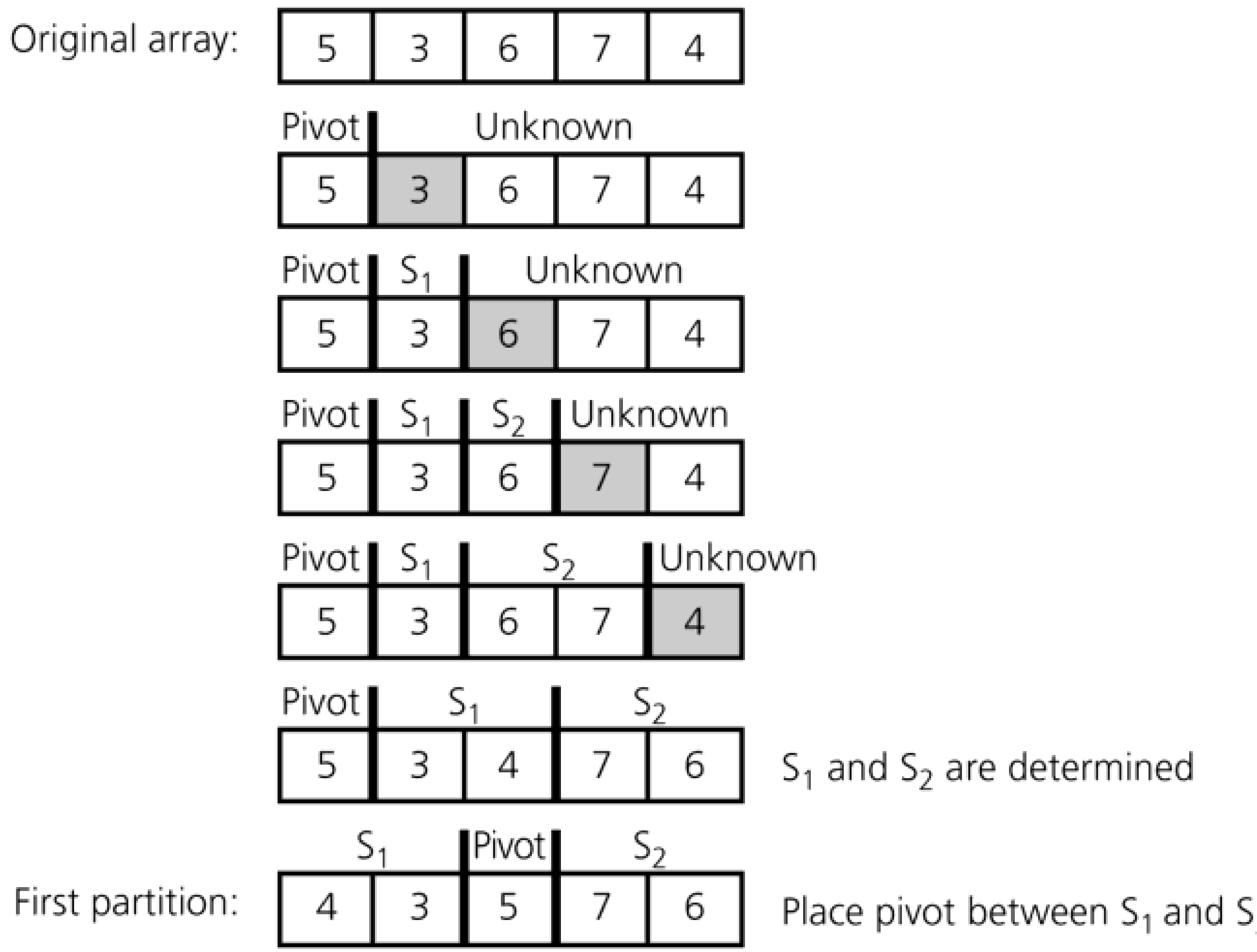
Quicksort – Analysis

A worst-case partitioning with quicksort



Quicksort – Analysis

An average-case partitioning with quicksort



Other Sorting Algorithms?

Other Sorting Algorithms?

Many! For example:

- Shell sort
- Comb sort
- Heapsort
- Counting sort
- Bucket sort
- Distribution sort
- Timsort

- e.g. Check http://en.wikipedia.org/wiki/Sorting_algorithm for a table comparing sorting algorithms.

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 group data items according to their rightmost character, and put these groups into order w.r.t. this rightmost character.
  - Then, combine these groups.
  - 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

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
(00**0**4) (02**2**2, 01**2**3) (21**5**0, 21**5**4) (15**6**0, 10**6**1) (02**8**3)	Grouped by third digit
0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283	Combined
(0**0**04, 1**0**61) (0**1**23, 2**1**50, 2**1**54) (0**2**22, 0**2**83) (1**5**60)	Grouped by second digit
0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560	Combined
(**0**004, **0**123, **0**222, **0**283) (**1**061, **1**560) (**2**150, **2**154)	Grouped by first digit
0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154	Combined (sorted)

Radix Sort – Example

mom, dad, god, fat, bad, cat, mad, pat, bar, him	original list
(da**d**,go**d**,ba**d**,ma**d**) (mo**m**,hi**m**) (ba**r**) (fa**t**,ca**t**,pa**t**)	group strings by rightmost letter
dad,god,bad,mad,mom,him,bar,fat,cat,pat	combine groups
(d**a**d,b**a**d,m**a**d,b**a**r,f**a**t,c**a**t,p**a**t) (h**i**m) (g**o**d,m**o**m)	group strings by middle letter
dad,bad,mad,bar,fat,cat,pat,him,god,mom	combine groups
(**b**ad,**b**ar) (**c**at) (**d**ad) (**f**at) (**g**od) (**h**im) (**m**ad,**m**om) (**p**at)	group strings by middle letter
bad,bar,cat,dad,fat,god,him,mad,mom,par	combine groups (SORTED)

Radix Sort - Algorithm

```
radixSort( int theArray[], 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 * original size of data*** (because each group should be big enough to hold the original data collection.)
  - For example, to sort string of uppercase letters. we need 27 groups.
  - 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

	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	$n * \log n$	$n * \log n$