# EECE 2560:
## Fundamentals of Engineering Algorithms

Sorting Algorithms


# Basic Sorting Algorithms

- Sorting:
  - Organize a collection of data into either ascending or descending order
- Internal sort
  - Collection of data fits in memory
- External sort
  - Collection of data does *not* all fit in memory
  - Must reside on secondary storage
- Example of basic sorting algorithms:
  - The Selection Sort
  - The Bubble Sort
  - The Insertion Sort


# The Selection Sort

- The idea is to select the largest item and put it in its place, select the next largest and put it in its place, and so on.

Gray elements are selected;
blue elements comprise the sorted portion of the array.

| | | | | | |
|---|---|---|---|---|---|
| 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** |

# The Selection Sort Implementation (1 of 3)

```
/** Finds the largest item in an array.
 @pre  The size of the array is >= 1.
 @post  The arguments are unchanged.
 @param theArray  The given array.
 @param size  The number of elements in theArray.
 @return  The index of the largest entry in the array. */
template<class ItemType>
int findIndexOfLargest(const ItemType theArray[], int size)
{
   int indexSoFar = 0; // Index of largest entry found so far
   for (int currentIndex = 1; currentIndex < size; currentIndex++)
   {
      // At this point, theArray[indexSoFar] >= all entries in
      // theArray[0..currentIndex - 1]
      if (theArray[currentIndex] > theArray[indexSoFar])
         indexSoFar = currentIndex;
   }  // end for
   return indexSoFar; // Index of largest entry
}  // end findIndexOfLargest
```

# The Selection Sort Implementation (2 of 3)

```
/** Sorts the items in an array into ascending order.
 @pre  None.
 @post  The array is sorted into ascending order; the size of the array is unchanged.
 @param theArray  The array to sort.
 @param n  The size of theArray. */
template<class ItemType>
void selectionSort(ItemType theArray[], int n)
{ // last = index of the last item in the subarray of items yet to be sorted;
  // largest = index of the largest item found
    for (int last = n - 1; last >= 1; last--) {
    // At this point, theArray[last+1..n-1] is sorted, and its entries are greater than those in
    // theArray[0..last]. Now, select the largest entry in theArray[0..last]
        int largest = findIndexOfLargest(theArray, last+1);

        // Swap the largest entry, theArray[largest], with theArray[last]
        if (largest != last)  //to avoid unnecessary movement
            std::swap(theArray[largest], theArray[last]);
    }  // end for
}  // end selectionSort
```

# The Selection Sort Implementation (3 of 3)

```
int main()
{
   std::string a[6] = {"Z", "X", "R", "K", "F", "B"};
   selectionSort(a, 6);
   for (int i = 0; i < 6; i++)
      std::cout << a[i] << " ";
   std::cout << std::endl;
}  // end main
```

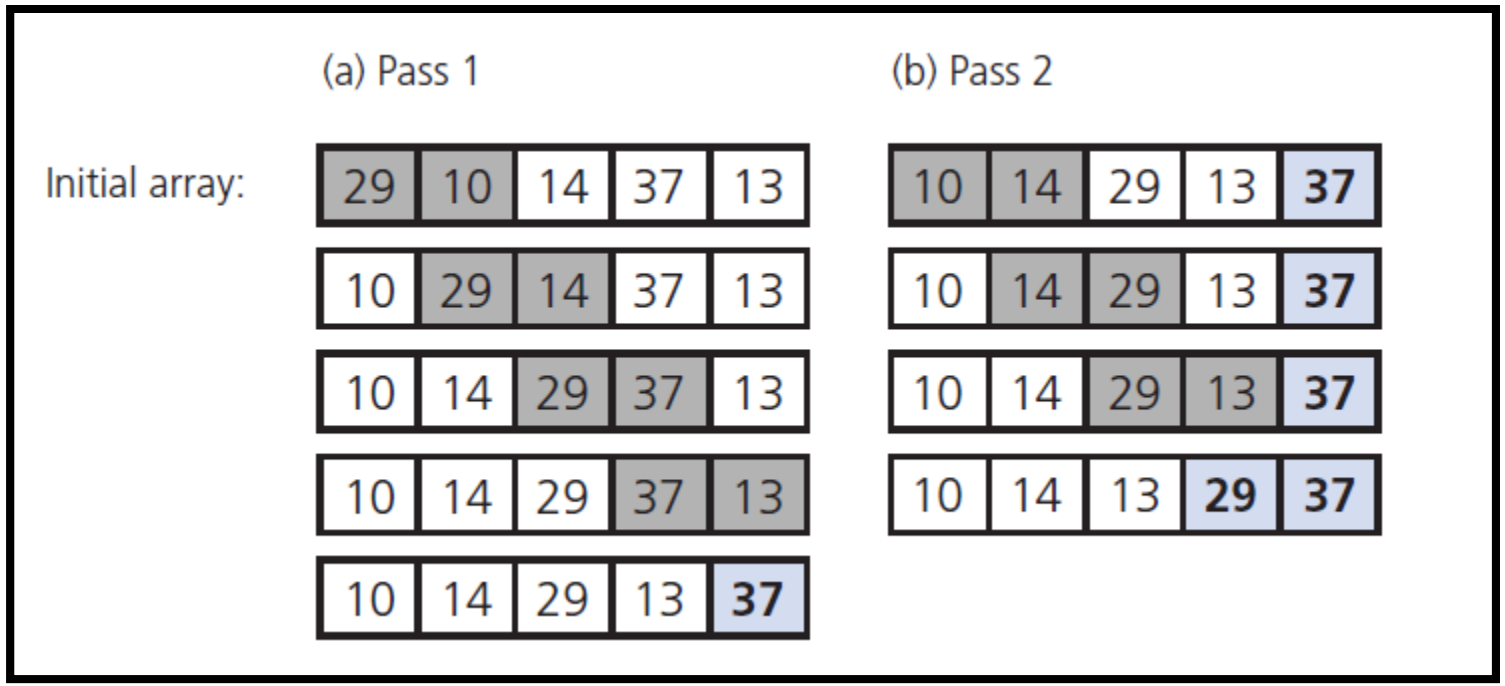
# The Selection Sort Analysis

- Analysis
  - Selection sort is $O(n^2)$
  - Appropriate only for small *n*
  - Requires less movements of data

| | | | | |
|---|---|---|---|---|
| 29 | 10 | 14 | 37 | 13 |
| 29 | 10 | 14 | 13 | **37** |
| 13 | 10 | 14 | **29** | **37** |
| 13 | 10 | **14** | **29** | **37** |
| **10** | **13** | **14** | **29** | **37** |

# The Bubble Sort

- Compares adjacent items
  - Exchanges them if out of order
  - Requires several passes over the data
- When ordering successive pairs
  - Largest item bubbles to end of the array



# The Bubble Sort Implementation (1 of 2)

```
/** Sorts the items in an array into ascending order.
 @pre  None.
 @post  theArray is sorted into ascending order; n is unchanged.
 @param theArray  The given array.
 @param n  The size of theArray. */
template<class ItemType>
void bubbleSort(ItemType theArray[], int n)
{
   bool sorted = false; // False when swaps occur
   int pass = 1;
   while (!sorted && (pass < n))
   {
    // At this point, theArray[n+1-pass..n-1] is sorted
    // and all of its entries are > the entries in theArray[0..n-pass]
      sorted = true; // Assume sorted
      for (int index = 0; index < n - pass; index++)
      {
         // At this point, all entries in theArray[0..index-1]
         // are <= theArray[index]
```

# The Bubble Sort Implementation (2 of 2)

```
            int nextIndex = index + 1;
            if (theArray[index] > theArray[nextIndex]) {
               // Exchange entries
               std::swap(theArray[index], theArray[nextIndex]);
               sorted = false; // Signal exchange
            } // end if
         }  // end for
         // Assertion: theArray[0..n-pass-1] < theArray[n-pass]
         pass++;
      }  // end while
   }  // end bubbleSort

   int main() {
      std::string a[6] = {"Z", "X", "R", "K", "F", "B"};
      bubbleSort(a, 6);
      for (int i = 0; i < 6; i++)
         std::cout << a[i] << " ";
      std::cout << std::endl;
   }  // end main
```

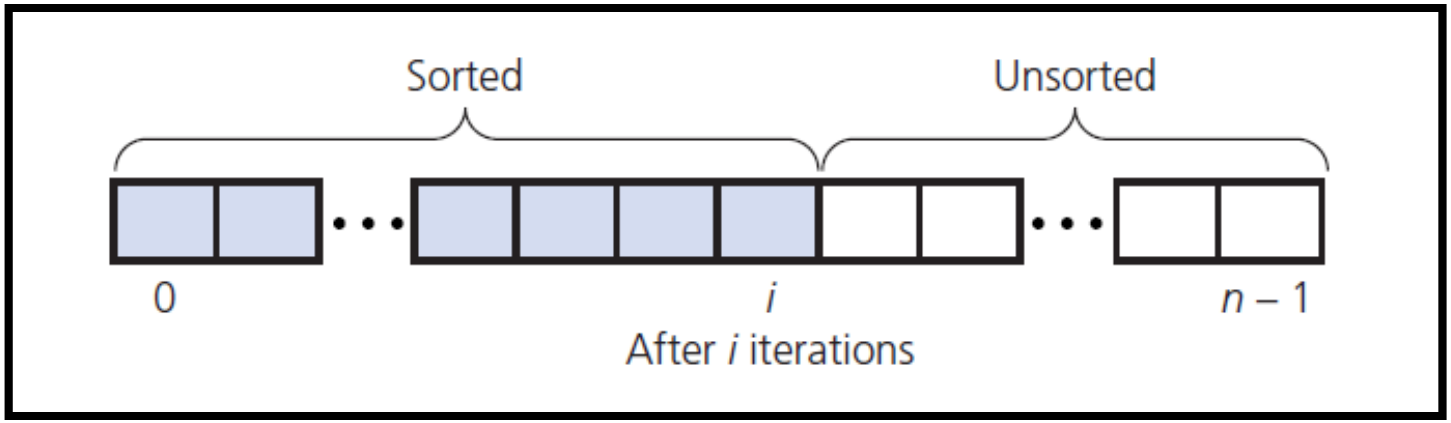

# The Bubble Sort Analysis

- Worst case $O(n^2)$
- Best case (array already in order) is O(n)

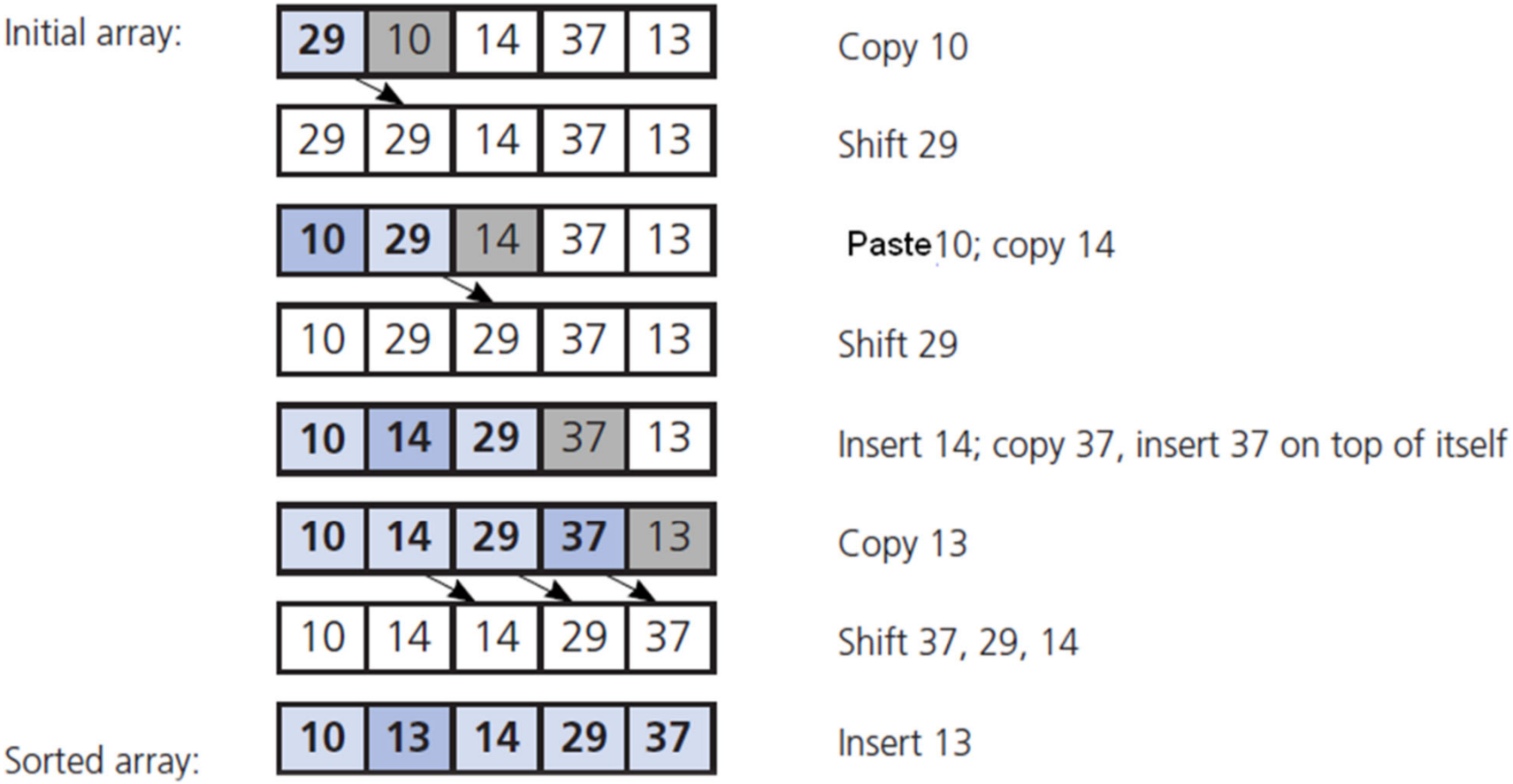
| 10 | 13 | 14 | 29 | 37 |
|---|---|---|---|---|

# The Insertion Sort (1 of 2)

- Take each item from unsorted region
  - Insert it into correct order in sorted region



# The Insertion Sort (2 of 2)



# The Insertion Sort Implementation (1 of 3)

```
/** Sorts the items in an array into ascending order.
 @pre  None.
 @post  theArray is sorted into ascending order; n is unchanged.
 @param theArray  The given array.
 @param n  The size of theArray. */
template<class ItemType>
void insertionSort(ItemType theArray[], int n)
{
   // unsorted = first index of the unsorted region,
  // loc = index of insertion in the sorted region,
  // nextItem = next item in the unsorted region.
  // Initially, sorted region is theArray[0], unsorted region is theArray[1..n-1].
  // In general, sorted region is theArray[0..unsorted-1], unsorted region theArray[unsorted..n-1]
```

# The Insertion Sort Implementation (2 of 3)

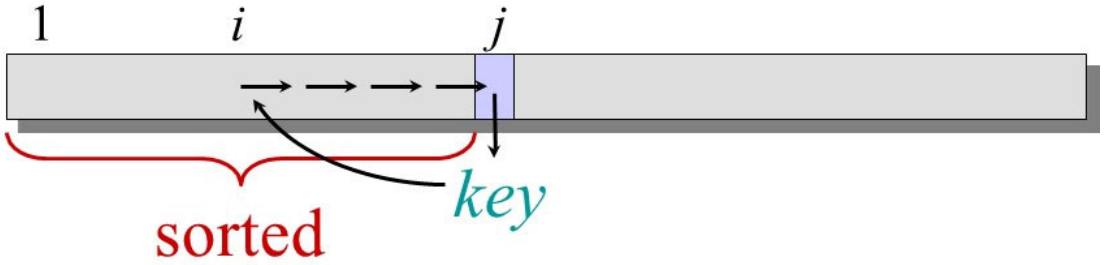
```
for (int unsorted = 1; unsorted < n; unsorted++)
  {
    // At this point, theArray[0..unsorted-1] is sorted.
    // Find the right position (loc) in theArray[0..unsorted]
    // for theArray[unsorted], which is the first entry in the
    // unsorted region; shift, if necessary, to make room
    ItemType nextItem = theArray[unsorted];
    int loc = unsorted;
    while ((loc > 0) && (theArray[loc - 1] > nextItem))
    {
      // Shift theArray[loc - 1] to the right
      theArray[loc] = theArray[loc - 1];
      loc--;
    }  // end while
    // At this point, theArray[loc] is where nextItem belongs
    if (loc != unsorted) //to avoid unnecessary movement
      theArray[loc] = nextItem; //Insert nextItem into sorted region
  }  // end for
}  // end insertionSort
```

# The Insertion Sort Implementation (3 of 3)

```
int main()
{
   std::string a[6] = {"Z", "X", "R", "K", "F", "B"};
   insertionSort(a, 6);
   for (int i = 0; i < 6; i++)
      std::cout << a[i] << " ";
   std::cout << std::endl;
}  // end main
```

# The Insertion Sort Analysis

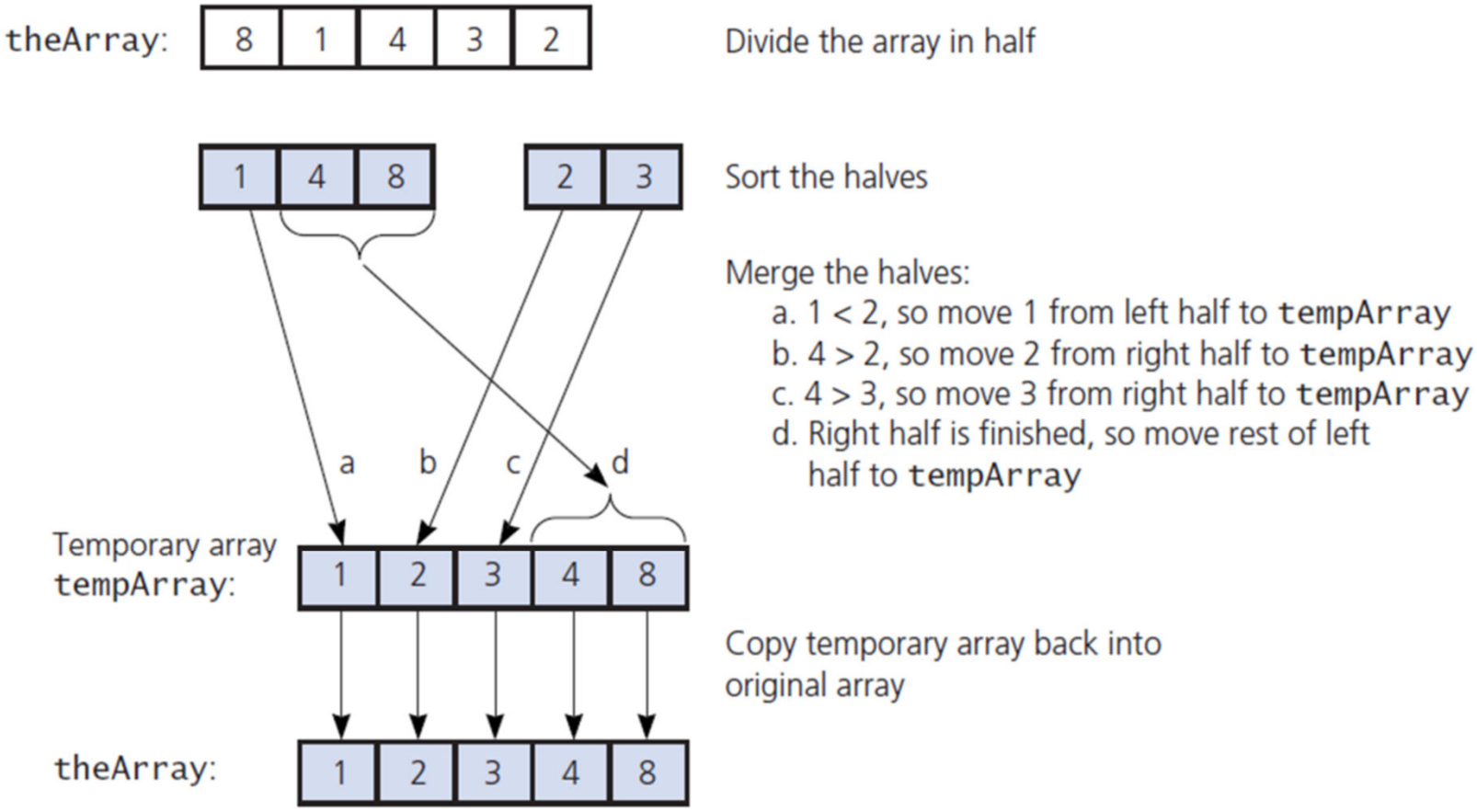
- Analysis
  - Worst case $O(n^2)$
  - Best case (array already in order) is O(n)
- Appropriate for small (n < 25) arrays
- Unsuitable for large arrays
  - Unless already sorted




# Faster Sorting Algorithms

- The Merge Sort
- The Quick Sort
- The Radix Sort

# Merge Sort



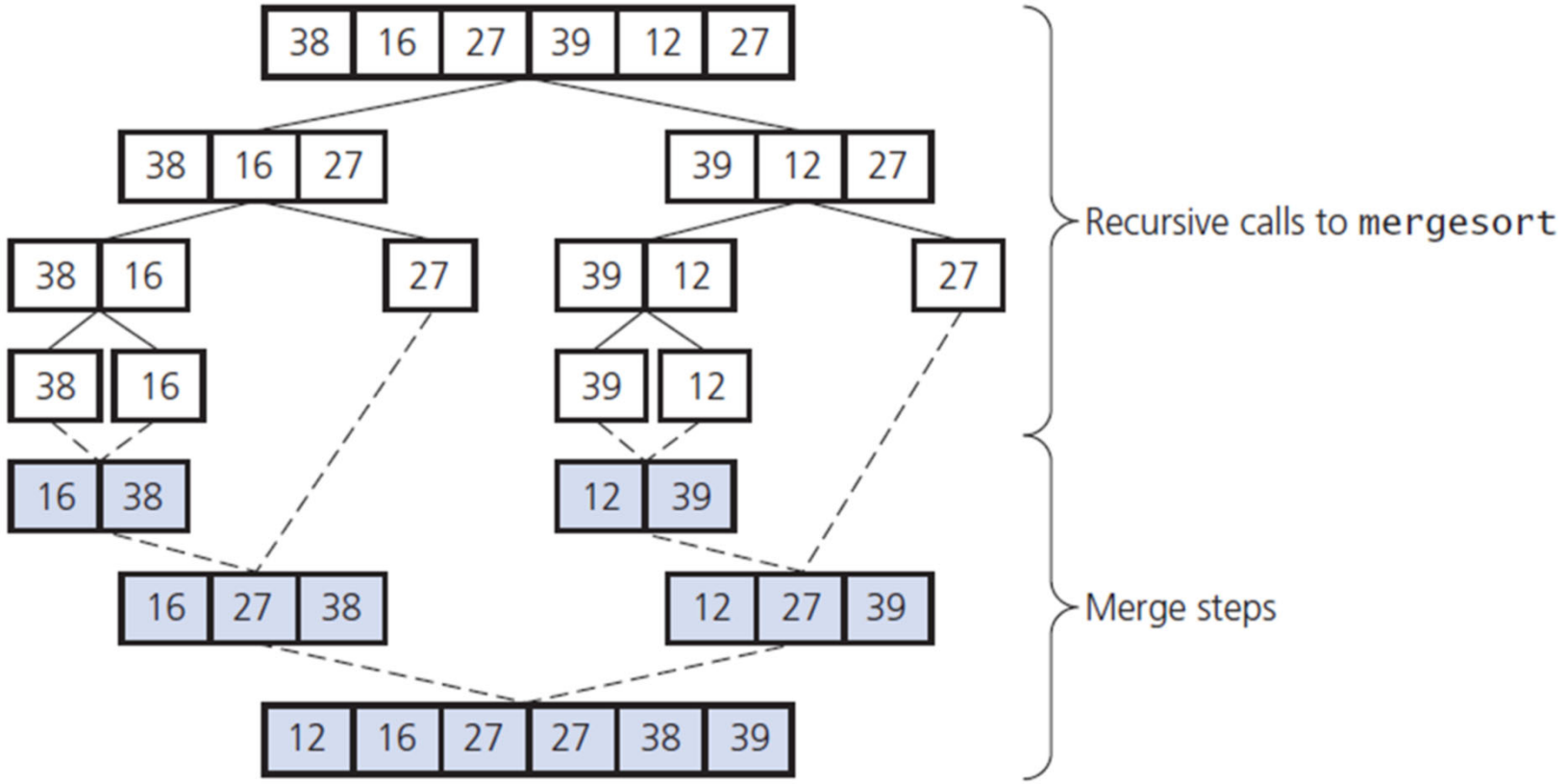

A merge sort with an auxiliary temporary array

# Merge Sort Pseudocode

```
// Sorts theArray[first..last] by
// 1. Sorting the first half of the array
// 2. Sorting the second half of the array
// 3. Merging the two sorted halves
mergeSort(theArray: ItemArray, first: integer, last: integer)
  if (first < last) {
        mid = (first + last) / 2 // Get midpoint
        // Sort theArray[first..mid]
        mergeSort(theArray, first, mid)
        // Sort theArray[mid+1..last]
        mergeSort(theArray, mid + 1, last)

        // Merge sorted halves theArray[first..mid] and theArray[mid+1..last]
        merge(theArray, first, mid, last)
}
// If first >= last, there is nothing to do
```

# Merge Sort Example



# Merge Sort Implementation (1 of 5)

```
const int MAX_SIZE = 50;
/** Merges two sorted arrays theArray[first..mid] and theArray[mid+1..last] into one sorted array.
 @pre  first <= mid <= last. The subarrays theArray[first..mid] and theArray[mid+1..last] are sorted.
 @post  theArray[first..last] is sorted.
 @param theArray  The given array.
 @param first  The index of the beginning of the first segment in theArray.
 @param mid  The index of the end of the 1st segment;  mid + 1 the beginning of the 2nd segment.
 @param last  The index of the last element in the second segment in theArray.
 @note  two subarrays are merged into a temp array. The result is copied into the original array */
template<class ItemType>
void merge(ItemType theArray[], int first, int mid, int last)
{
   ItemType tempArray[MAX_SIZE];  // Temporary array

   // Initialize the local indices to indicate the subarrays
   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
```

# Merge Sort Implementation (2 of 5)

```
// While both subarrays are not empty, copy the
//smaller item into the temporary array
int index = first1;    // Next available location in tempArray
while ((first1 <= last1) && (first2 <= last2))
{
   // At this point, tempArray[first..index-1] is in order
   if (theArray[first1] <= theArray[first2]) {
      tempArray[index] = theArray[first1];
      first1++; }
   else {
      tempArray[index] = theArray[first2];
      first2++; }  // end if
   index++;
}  // end while
```

# Merge Sort Implementation (3 of 5)

```
// Finish off the first subarray, if necessary
   while (first1 <= last1) {
      // At this point, tempArray[first..index-1] is in order
      tempArray[index] = theArray[first1];
      first1++;
      index++;
   }  // end while

// Finish off the second subarray, if necessary
   while (first2 <= last2) {
      // At this point, tempArray[first..index-1] is in order
      tempArray[index] = theArray[first2];
      first2++;
      index++;
   }  // end for

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

# Merge Sort Implementation (4 of 5)

```
/** Sorts the items in an array into ascending order.
 @pre  theArray[first..last] is an array.
 @post  theArray[first..last] is sorted in ascending order.
 @param theArray  The given array.
 @param first  The index of the first element to consider in theArray.
 @param last  The index of the last element to consider in theArray.*/
template<class ItemType>
void mergeSort(ItemType theArray[], int first, int last) {
   if (first < last) {
      // Sort each half
      int mid = first + (last - first) / 2; // Index of midpoint
      // Sort left half theArray[first..mid]
      mergeSort(theArray, first, mid);
     // Sort right half theArray[mid+1..last]
      mergeSort(theArray, mid + 1, last);
      // Merge the two halves
      merge(theArray, first, mid, last);
   }  // end if
}  // end mergeSort
```

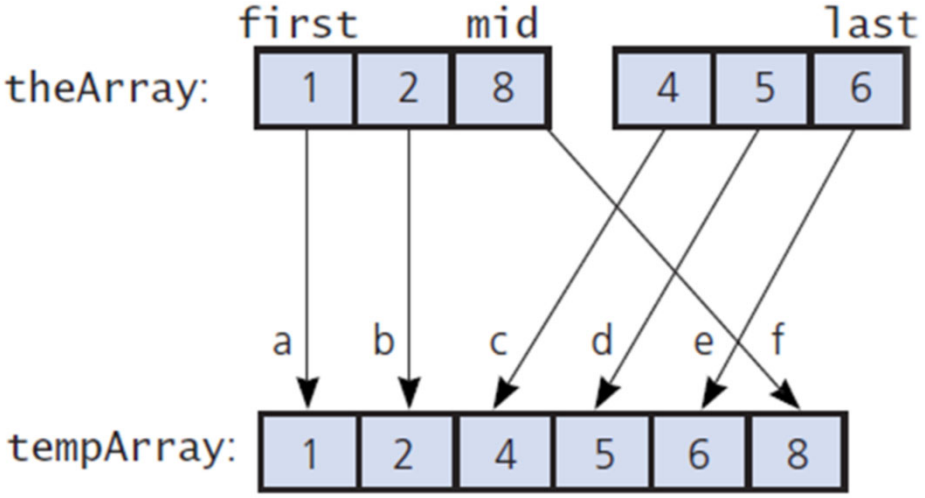
# Merge Sort Implementation (5 of 5)

```
int main()
{
   std::string a[6] = {"Z", "X", "R", "K", "F", "B"};
   mergeSort(a, 0, 5);
   for (int i = 0; i < 6; i++)
      std::cout << a[i] << " ";
   std::cout << std::endl;

}  // end main
```

# Merge Step Example

- If the total number of items in the two array segments to be merged is *n* , then merging the segments requires at most *n* – 1 comparisons and at least n/2 comparisons.

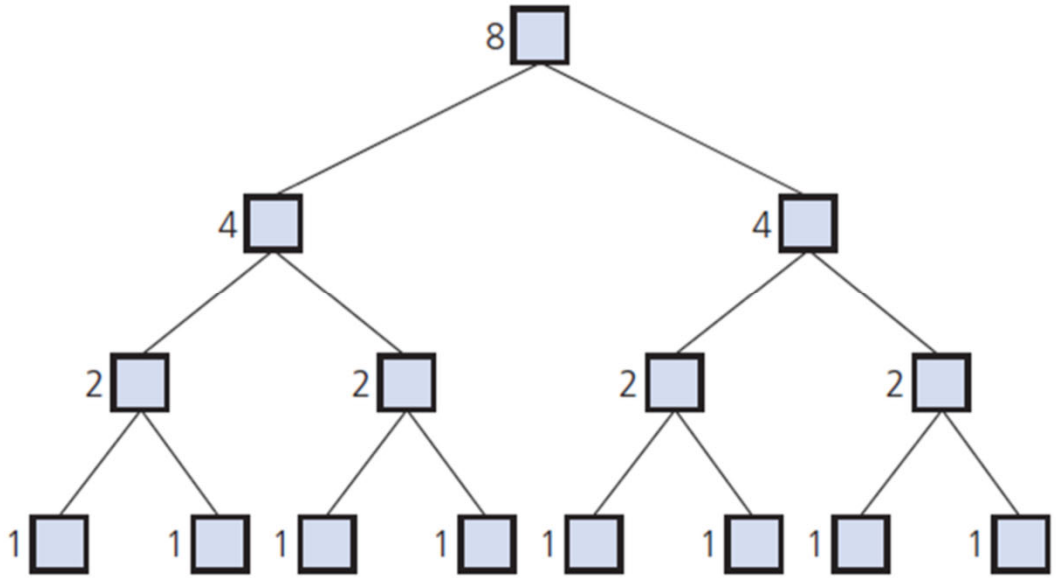



A worst-case instance of the merge step in a merge sort

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`

# Levels of Recursive Calls to Merge Sort




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

Levels of recursive calls to mergeSort, given an array of eight items


# The Merge Sort Analysis

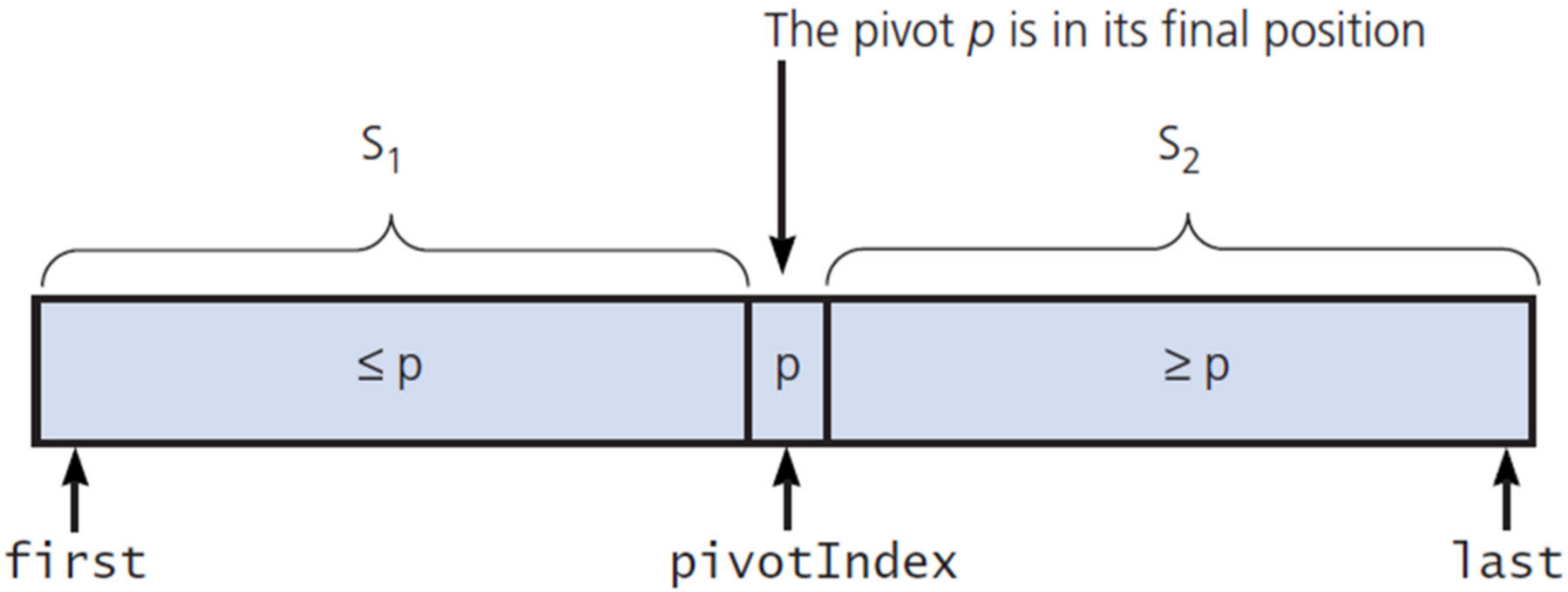
- Analysis
  - The merge sort is O($n \times \log n$ ) in both the worst and average cases
- Although the merge sort is an extremely efficient algorithm with respect to time, it does have one drawback:
  - The merge step requires an auxiliary array. This extra storage and the necessary copying of entries are disadvantages.


# The Quick Sort

- Another divide-and-conquer algorithm
- Partitions an array into items that are
  - Less than or equal to the pivot and
  - Those that are greater than or equal to the pivot

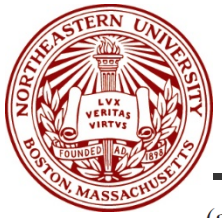
- Partitioning places pivot in its correct position within the array
  - Initially place the chosen pivot in theArray[last] before partitioning

# The Quick Sort Pivot

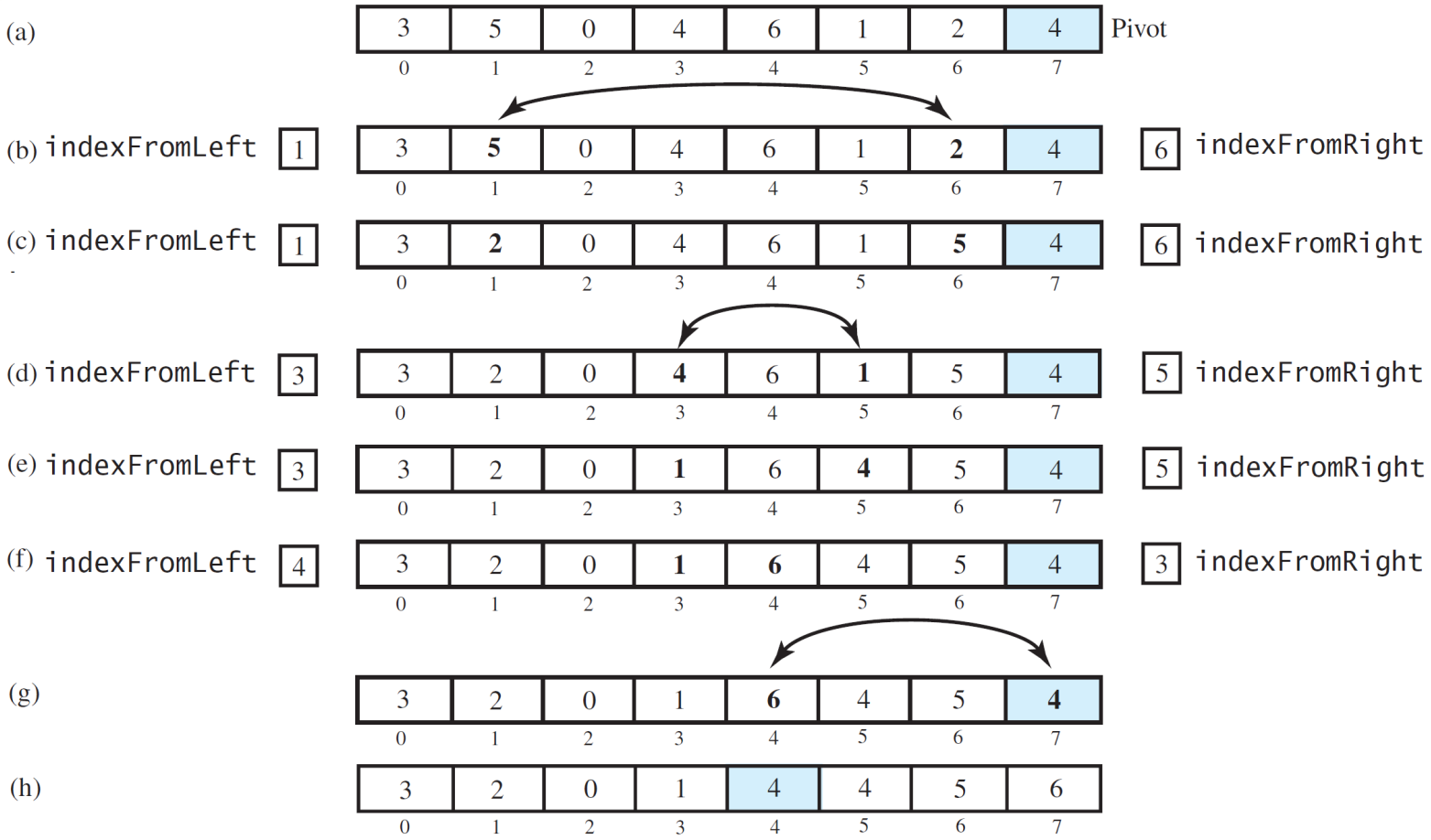


# The Quick Sort Pseudocode

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



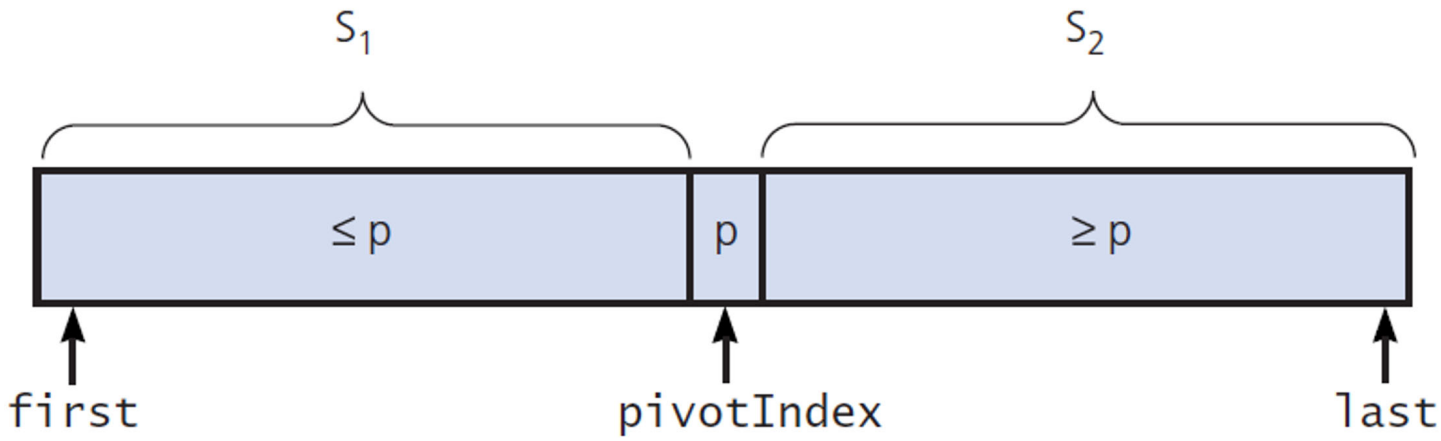
# Quick Sort Partitioning Example




1. indexFromLeft++ until it reaches an element **greater than or equal** the Pivot.
2. indexFromRight-- until it reaches an element **less than or equal** the Pivot.
3. If indexFromLeft < indexFromRight, swap the left element with the right one, indexFromLeft++, indexFromRight--, and go to step 1.
   Else swap Pivot with left element and we are done.

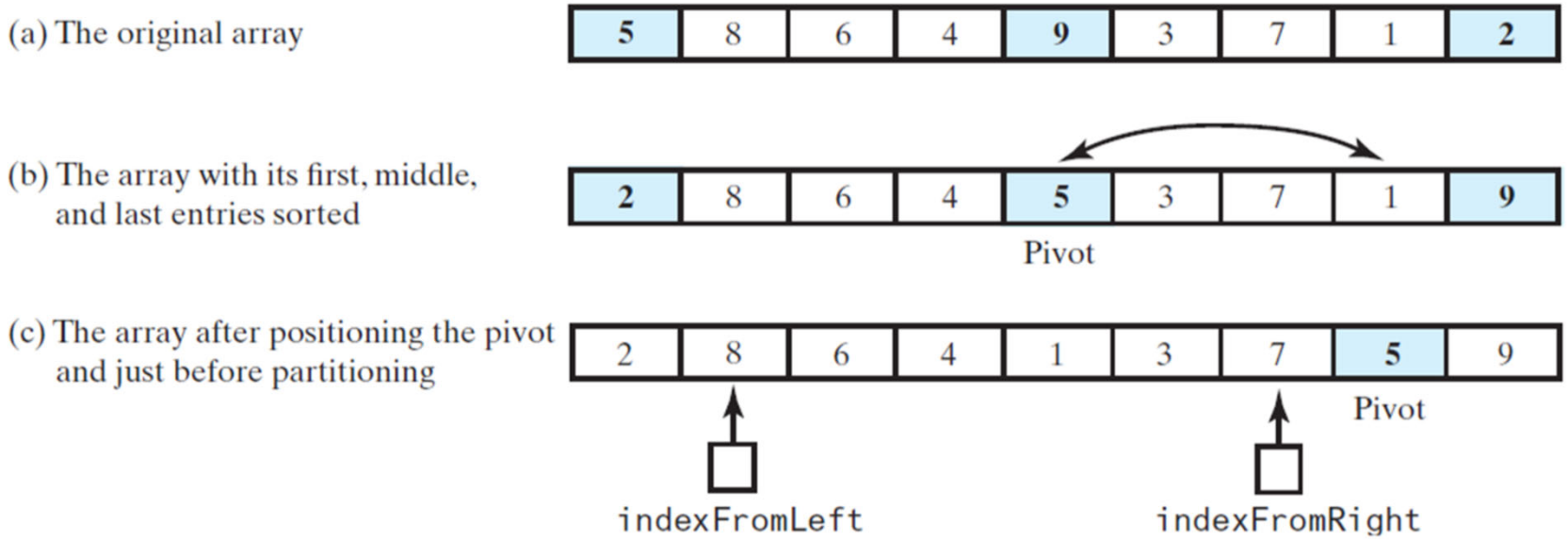
# Partitioning of Equal Elements

- To enhance the quick sort's performance, we want the subarrays S1 and S2 to be as nearly equal in size as possible that is why both of the subarrays can contain entries equal to the pivot. Therefore, both the search from the left and the search from the right stop and swap when they encounter an entry that equals the pivot.



# Median-of-Three Pivot Selection

- Ideally, the pivot should be the median value in the array, so that the subarrays $S_1$ and $S_2$ each have the same—or nearly the same—number of entries.
- In the following example, we will take as our pivot the median of three entries in the array: the first entry, the middle entry, and the last entry.



# Quick Sort Implementation (1 of 6)

```
static const int MIN_SIZE = 10;// Smallest size of an array that quicksort will sort

/** Arranges two specified array entries into sorted order by exchanging them, if necessary.
 @param theArray  The given array.
 @param i  The index of the first entry to consider in theArray.
 @param j  The index of the second entry to consider in theArray. */
template<class ItemType>
void order(ItemType theArray[], int i, int j)
{
   if (theArray[i] > theArray[j])
      std::swap(theArray[i], theArray[j]); // Exchange entries
}  // end order
```

# Quick Sort Implementation (2 of 6)

```
/** Arranges the first, middle, and last entry in an array in sorted order.
 @pre  theArray[first..last] is an array; first <= last.
 @post  theArray[first..last] is arranged so that its first, middle, and last entries are sorted.
 @param theArray  The given array.
 @param first  The first entry to consider in theArray.
 @param last  The last entry to consider in theArray.
 @return  The index of the middle entry. */
template<class ItemType>
int sortFirstMiddleLast(ItemType theArray[], int first, int last)
{
   int mid = first + (last - first) / 2;
   order(theArray, first, mid); // Make theArray[first] <= theArray[mid]
   order(theArray, mid, last);  // Make theArray[mid] <= theArray[last]
   order(theArray, first, mid); // Make theArray[first] <= theArray[mid]

   return mid;
}  // end sortFirstMiddleLast
```

# Quick Sort Implementation (3 of 6)

```
/** Partitions the entries in an array about a pivot entry for quicksort.
 @pre  theArray[first..last] is an array; first <= last.
 @post  theArray[first..last] is partitioned such that: S1 = theArray[first..pivotIndex-1] <= pivot
    theArray[pivotIndex]  == pivot and  S2 = theArray[pivotIndex+1..last]  >= pivot
 @param theArray  The given array.
 @param first  and last entries to consider in theArray.
@return  The index of the pivot. */
template<class ItemType>
int partition(ItemType theArray[], int first, int last)
{   // Choose pivot using median-of-three selection
    int pivotIndex = sortFirstMiddleLast(theArray, first, last);
    // Reposition pivot so it is last in the array
    std::swap(theArray[pivotIndex], theArray[last - 1]);
    pivotIndex = last - 1;
    ItemType pivot = theArray[pivotIndex];
    // Determine the regions S1 and S2
    int indexFromLeft = first + 1;
    int indexFromRight = last - 2;
    bool done = false;
```

# Quick Sort Implementation (4 of 6)

```
while (!done)  {
 // Locate first entry on left that is >= pivot
  while (theArray[indexFromLeft] < pivot)
      indexFromLeft = indexFromLeft + 1;
 // Locate first entry on right that is <= pivot
  while (theArray[indexFromRight] > pivot)
      indexFromRight = indexFromRight - 1;

  if (indexFromLeft < indexFromRight)   {
   std::swap(theArray[indexFromLeft], theArray[indexFromRight]);
   indexFromLeft = indexFromLeft + 1;
   indexFromRight = indexFromRight - 1;  }
  else   done = true;
 }  // end while
// Place pivot in proper position between S1 and S2, and mark its new location
  std::swap(theArray[pivotIndex], theArray[indexFromLeft]);
  pivotIndex = indexFromLeft;
  return pivotIndex;
}  // end partition
```

# Quick Sort Implementation (5 of 6)

```
/** Sorts an array into ascending order. Uses the quick sort with median-of-three pivot selection for
 arrays of at least MIN_SIZE entries, and uses the insertion sort for other arrays.
 @pre  theArray[first..last] is an array.
 @post  theArray[first..last] is sorted.
 @param theArray  The given array.
 @param first  The first element to consider in theArray.
 @param last  The last element to consider in theArray. */
template<class ItemType>
void quickSort(ItemType theArray[], int first, int last) {
   int sz = last - first + 1;
   if (sz < MIN_SIZE)
      insertionSort(theArray+first, sz);
   else {
      // Create the partition: S1 | Pivot | S2
      int pivotIndex = partition(theArray, first, last);
      // Sort subarrays S1 and S2
      quickSort(theArray, first, pivotIndex - 1);
      quickSort(theArray, pivotIndex + 1, last);
   }  // end if
}  // end quickSort
```

# Quick Sort Implementation (6 of 6)

```
int main()
{
   std::string a[6] = {"Z", "X", "R", "K", "F", "B"};
   quickSort(a, 0, 5);
   for (int i = 0; i < 6; i++)
      std::cout << a[i] << " ";
   std::cout << std::endl;

   std::string b[26] = {"Z", "Y", "X", "W", "V", "U", "T", "S",
"R", "Q", "P", "O", "N", "M", "L", "K", "J", "I", "H", "G", "F",
"E", "D", "C", "B", "A"};
   quickSort(b, 0, 25);
   for (int i = 0; i < 26; i++)
      std::cout << b[i] << " ";
   std::cout << std::endl;
}  // end main
```


# The Quick Sort Analysis

- Analysis
  - Partitioning is an O($n$) task
  - On average, there are $\log_2 n$ recursive calls to quickSort.
  - In the worst case, where the pivot always ends up at one end of the list, there are $n$ recursive calls to quickSort

- We conclude
  - Worst case O($n^2$)
  - Average case O($n \log n$)


# The Radix Sort

- Different from other sorts
  - Does not compare entries in an array
- Begins by organizing data (say strings) according to least significant letters
  - Then combine the groups
- Next form groups using next least significant letter

# The Radix Sort Example

| | |
|---|---|
| 0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150 | Original integers |
| (1560, 2150) (1061) (0222) (0123, 0283) (2154, 0004) | Grouped by fourth digit |
| 1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004 | Combined |
| (0004) (0222, 0123) (2150, 2154) (1560, 1061) (0283) | Grouped by third digit |
| 0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283 | Combined |
| (0004, 1061) (0123, 2150, 2154) (0222, 0283) (1560) | Grouped by second digit |
| 0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560 | Combined |
| (0004, 0123, 0222, 0283) (1061, 1560) (2150, 2154) | Grouped by first digit |
| 0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154 | Combined (sorted) |

# The Radix Sort Pseudocode

```
 // Sorts n d-digit integers in the array theArray.
radixSort(theArray: ItemArray, n: integer, d: integer): void
 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.
 }
```


# The Radix Sort Analysis

- Analysis
  - Requires *n* moves each time it forms groups
  - *n* moves to combine again into one group
  - Performs these $2 \times n$ moves *d* times
  - Thus requires $2 \times n \times d$ moves

- For a fixed *d*, radix sort is of order O(*n*)


# Sorting Algorithms Comparison

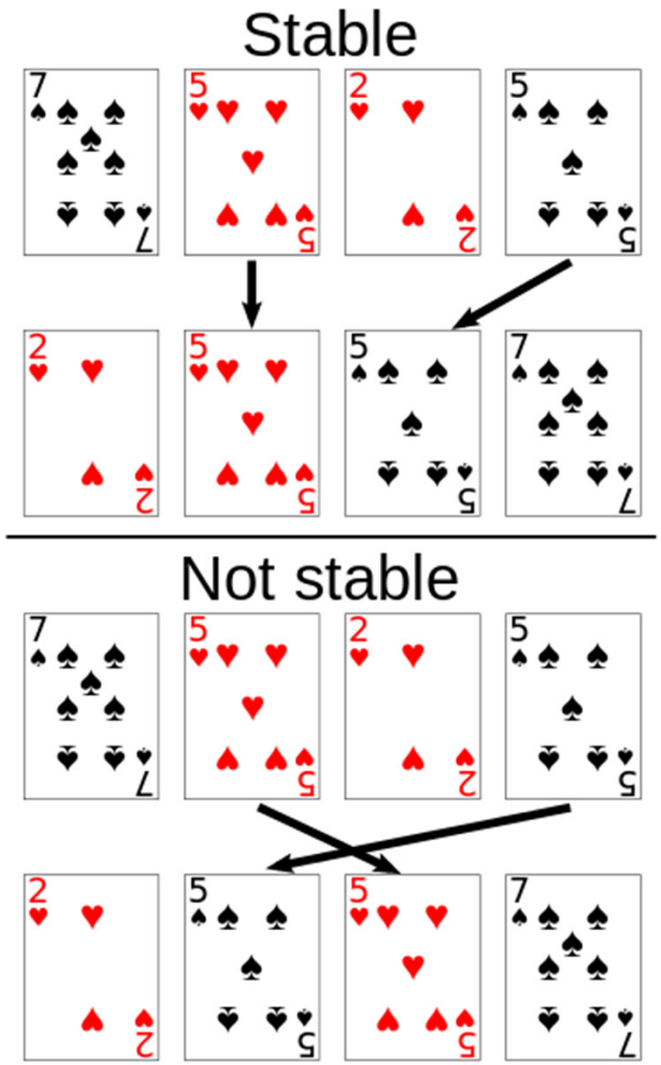
| | Worst case | Average case |
|---|---|---|
| Selection sort | $n^2$ | $n^2$ |
| Bubble sort | $n^2$ | $n^2$ |
| Insertion sort | $n^2$ | $n^2$ |
| Merge sort | $n \times \log n$ | $n \times \log n$ |
| Quick sort | $n^2$ | $n \times \log n$ |
| Radix sort | $n$ | $n$ |
| Tree sort | $n^2$ | $n \times \log n$ |
| Heap sort | $n \times \log n$ | $n \times \log n$ |


# In-Place Sorting Algorithm

- A sorting algorithm sorts ***in place*** if only a constant number of elements of the input array are ever stored outside the array.
- Merge sort requires a temporary array to perform the merge operation.
  - The implementation of merge sort with arrays does not sort ***in place***.

# Stability of Sorting Algorithms

- A sorting algorithm is stable if it preserves the original order of elements with equal key values as shown in the example.
- *Example:* given an array of students sorted by name and re-sort it by year of graduation. Using a stable sorting algorithm to sort the array by year will ensure that within each year the students will remain sorted by name.




©2019 Brilliant.org, "Sorting Algorithms"

# Counting Sort

- ***Counting sort*** assumes that each of the $n$ input elements is an integer in the range 0 to $k$, for some integer $k$. When $k = O(n)$, the sort runs in $\Theta(n)$ time.
- Counting sort determines, for each input element $x$, the number of elements less than $x$. It uses this information to place element $x$ directly into its position in the output array.
- In the algorithm procedure, we assume that the input is an array $A[1 .. n]$, and thus $A.length = n$.
- We require two other arrays: the array $B[1..n]$ holds the sorted output, and the array $C[0..k]$ provides temporary working storage to hold the occurrences of each element in $A$.

# Counting Sort Procedure

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] = number of
    //        elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] = number of
    // elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| A | 2 | 5 | 3 | 0 | 2 | 3 | 0 | 3 |

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| C | 2 | 0 | 2 | 3 | 0 | 1 |

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| C | 2 | 2 | 4 | 7 | 7 | 8 |

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| B | | | | | | | 3 | |

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| C | 2 | 2 | 4 | 6 | 7 | 8 |


# Counting Sort Stability

- The loop in step 10 of the Counting Sort algorithm starts from the last element in array *A* to maintain the stability property of the algorithm.
  - Stability in this case guarantees that numbers with the same value appear in the output array in the same order as they do in the input array.
  - That is, it breaks ties between two numbers by the rule that whichever number appears first in the input array appears first in the output array.