CS302 - Data Structures
using C++

Topic: Basic Sorting Algorithms

Kostas Alexis

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

Basic Sorting Algorithms

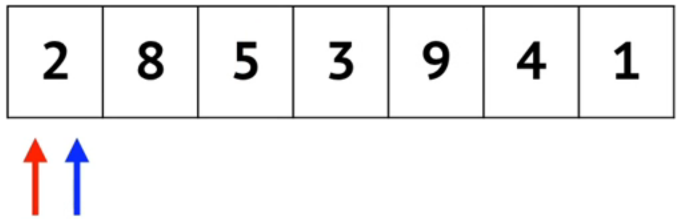
- The Selection Sort
- The Bubble Sort
- The Insertion Sort

The Selection Sort

- Visualization

current minimum

current item



The Selection Sort

- Grey 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

- An implementation of the selection sort

```
// Finds the largest item in an array
template<class ItemType>
int findIndexOfLargest(const ItemType theArray[], int size);

// Sorts the items in an array into ascending order.
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
            // theArray[0..last].
            // Select the largest entry in theArray[0..last]
            int largest = findIndexOfLargest(theArray, last+1);

            // Swap the largest entry, theArray[largest], with
            // theArray[last]
            std::swap(theArray[largest], theArray[last]);
      } // end for
} // end selectionSort
```

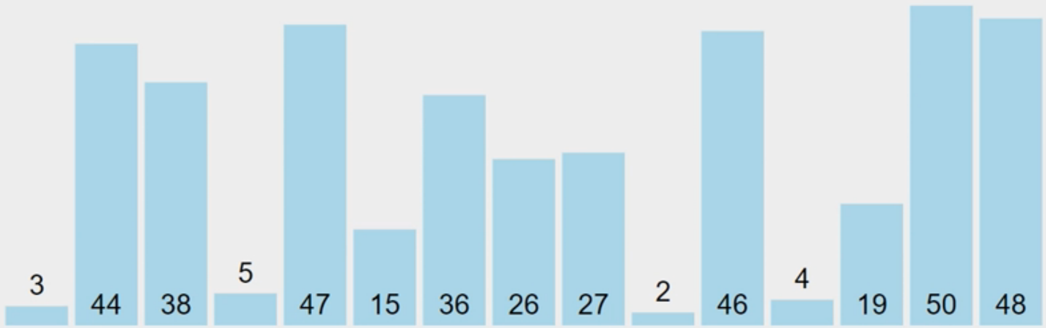
```
template<class ItemType>
int findIndexOfLargest(const ItemType theArray[], int size);
{
      int indexSoFar = 0; // Index of largest entry found so far
      for (int currentIndex = 1; currentIndex < size; currentIndex++)
      {
            // At this point, theArray[indexSoFar] >= all entries 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

- `For` loop executes $n-1$ times.
  - `selectionSort` calls each of the functions `findIndexOfLargest` and swap $n-1$ times.
- Each call to `findIndexOfLargest` causes its loop to execute `last` times (that is `size-1` times when size is `last+1`).
- Thus the $n-1$ calls to `findIndexOfLargest`, for values that range from $n-1$ down to 1, cause the loop in `findIndexofLargest` to execute a total of $(n-1) + (n-2) + \ldots + 1 = n \times (n-1)/2$ times.
- Because each execution of `findIndexOfLargest’s` loop performs one comparison, the calls to `findIndexLargest` require $n \times (n-1)/2$ comparisons.
- The $n-1$ calls to `swap` result in $n-1$ exchanges. Each exchange requires three assignments or data moves. Thus the calls to `swap` require $3 \times (n-1)$
- Together, a selection sort of n items requires $n \times (n-1)/2 + 3\,x\,(n-1)$ major operations.

The Selection Sort

- Analysis
    - Selection sort is $O(n^2)$
    - Appropriate only for small n
- Could be a good choice when
    - Data moves are costly
    - But comparisons are not



Selection Sort

```
repeat (numOfElements - 1) times
  set the first unsorted element as the minimum
  for each of the unsorted elements
    if element < currentMinimum
      set element as new minimum
  swap minimum with first unsorted position
```

The Bubble Sort

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

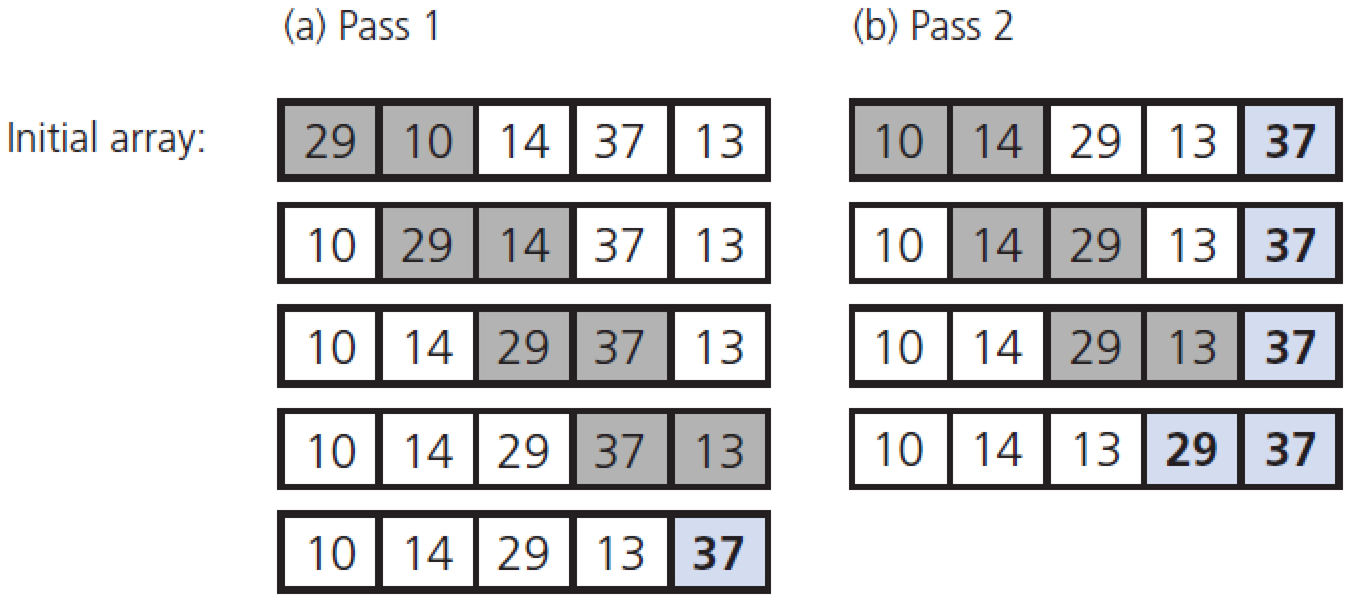
The Bubble Sort

- Visualization

2	8	5	3	9	4	1

The Bubble Sort

- First two passes of a bubble sort of an array of five integers



The Bubble Sort

- An implementation of the insertion sort

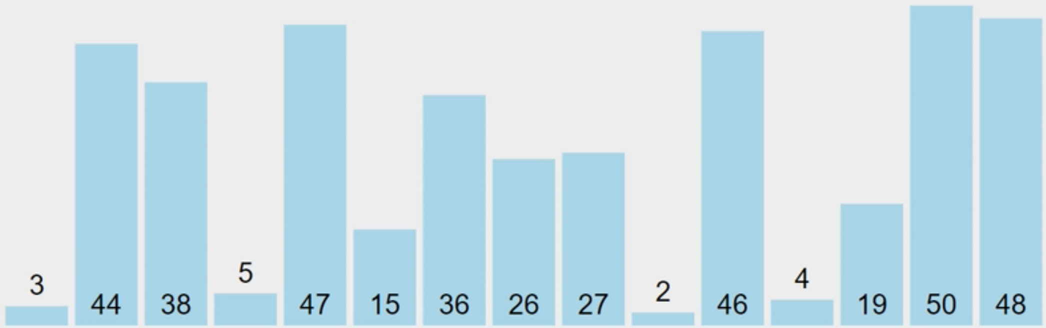
```
// Sorts the items in an array into ascending order
template<class ItemType>
void bubbleSort ItemType theArray[], int n);
{
      bool sorted = false; // False when swaps occur
      int pass = 1;
      while (!sorted && (pass < n))
      {
            // At this point, theArray[n+1-pass..n-1] is 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]
                  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
```

The Bubble Sort

- Requires at most $n - 1$ passes through the array
- Pass 1 requires $n - 1$ comparisons and at most $n - 1$ exchanges
- Pass 2 requires $n - 2$ comparisons and at most $n - 2$ exchanges
- [$\ldots$] Pass i requires $n - i$ comparisons and at most $n - i$ exchanges.
- Therefore, in the worst case, a bubble sort will require a total of $(n-1)\ +\ (n-2)\ +\ \ldots\ 1\ =\ n\ \times\ (n-1)/2$ comparisons and the same number of exchanges.
- Recall that each exchange requires three data moves. Thus altogether we have $2\ x\ n\ x\ (n-1)\ =\ 2\ \times\ n^2\ -2\ \times\ n$ major operations in the worst case.

The Bubble Sort

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

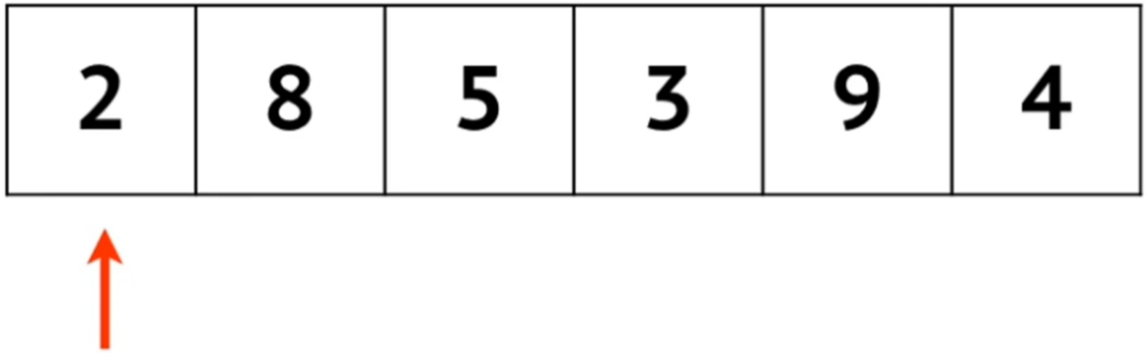


Bubble Sort

```
do
  swapped = false
  for i = 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap(leftElement, rightElement)
      swapped = true
while swapped
```

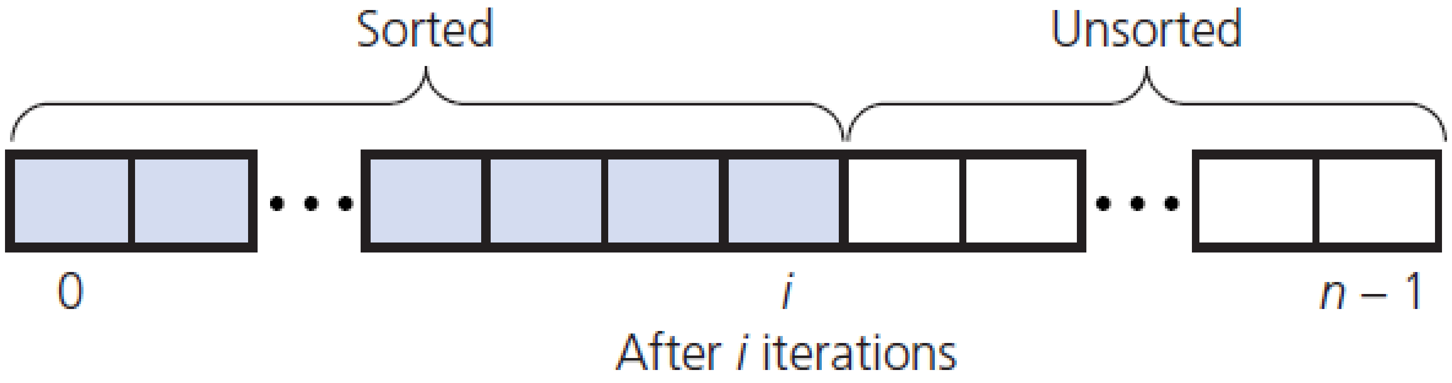
The Insertion Sort

- Visualization



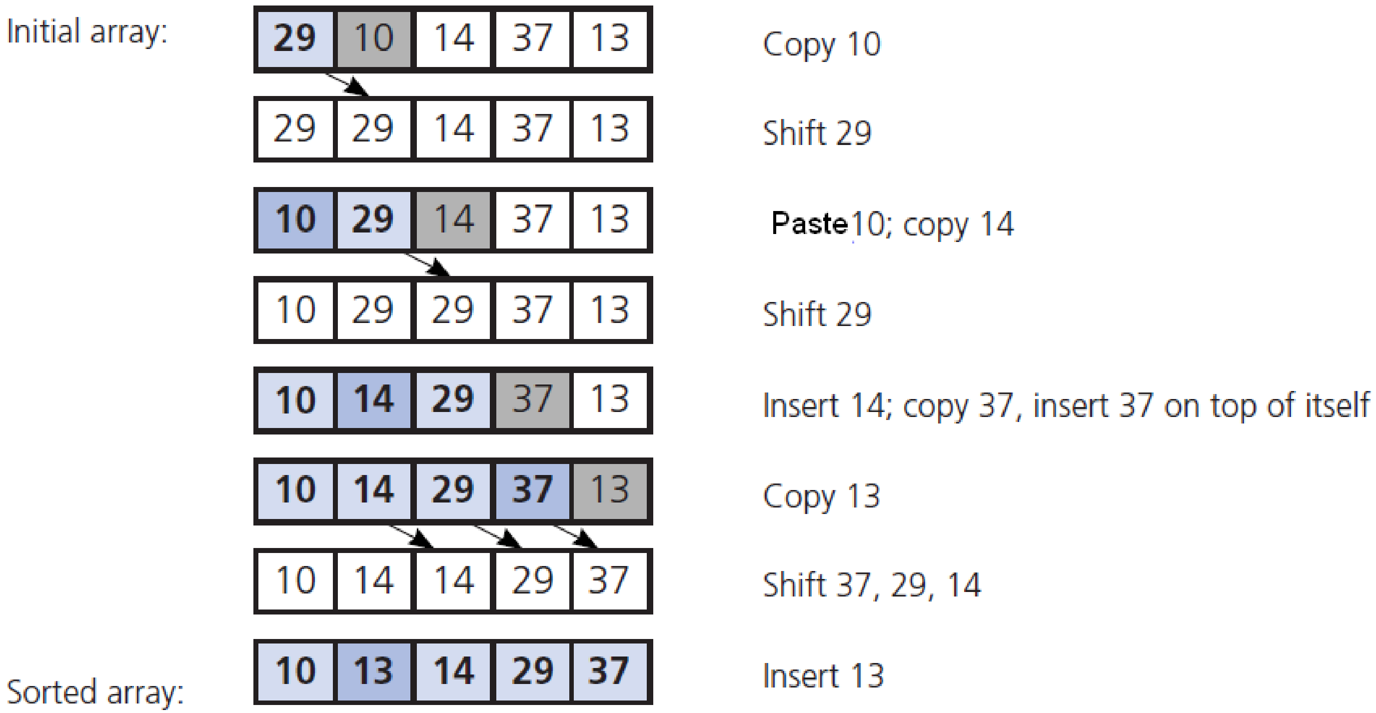
The Insertion Sort

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



The Insertion Sort

- An insertion sort of an array of five integers



The Insertion Sort

- An implementation of the bubble sort

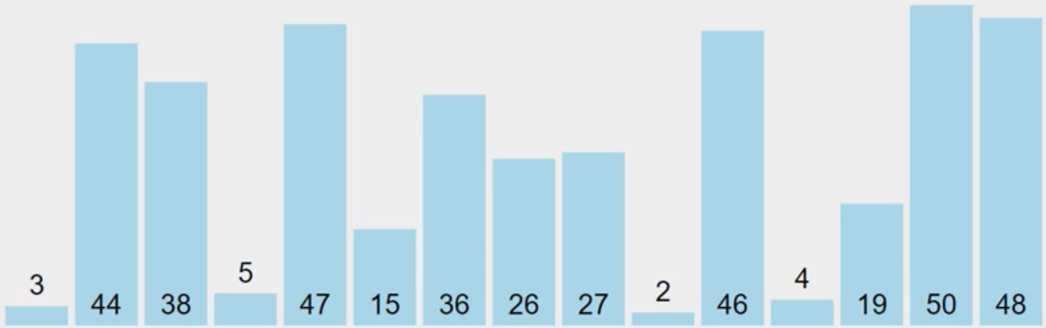
```
// Sorts the items in an array into ascending order
template<class ItemType>
void insertionSort ItemType theArray[], int n);
{
     // unsorted = first index of the unsorted region
     // loc = index of insertion in the sorted region
     // nextItem = next item in the unsorted region
     // Initially,   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]
     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
          theArray[loc] = nextItem; // Insert maxItem into sorted region
     } // end ofr
} // end insertionSort
```

The Insertion Sort

- The `outer for` loop in the function `insertionSort` executes $n - 1$ times.
- This loop contains an inner `while` loop that executes at most $unsorted$ times for values of $unsorted$ that range from 1 to $n - 1$.
- Thus, in the worst case, the algorithm's comparison occurs $1 + 2 + \ldots + (n - 1) = n \times (n - 1)/2$ times.
- In addition, the inner loop moves data items at most the same number of items.
- The `outer loop` moves data items twice per iteration, or $2 \times (n - 1)$ times.
- Together, there are $n \times (n - 1) + 2 \times (n - 1) = n^2 + n - 2$ major operations.

The Insertion Sort

- 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



Insertion Sort

```
mark first element as sorted
for each unsorted element X
  'extract' the element X
  for j = lastSortedIndex down to 0
    if current element j > X
      move sorted element to the right by 1
    break loop and insert X here
```

Faster Sorting Algorithms

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

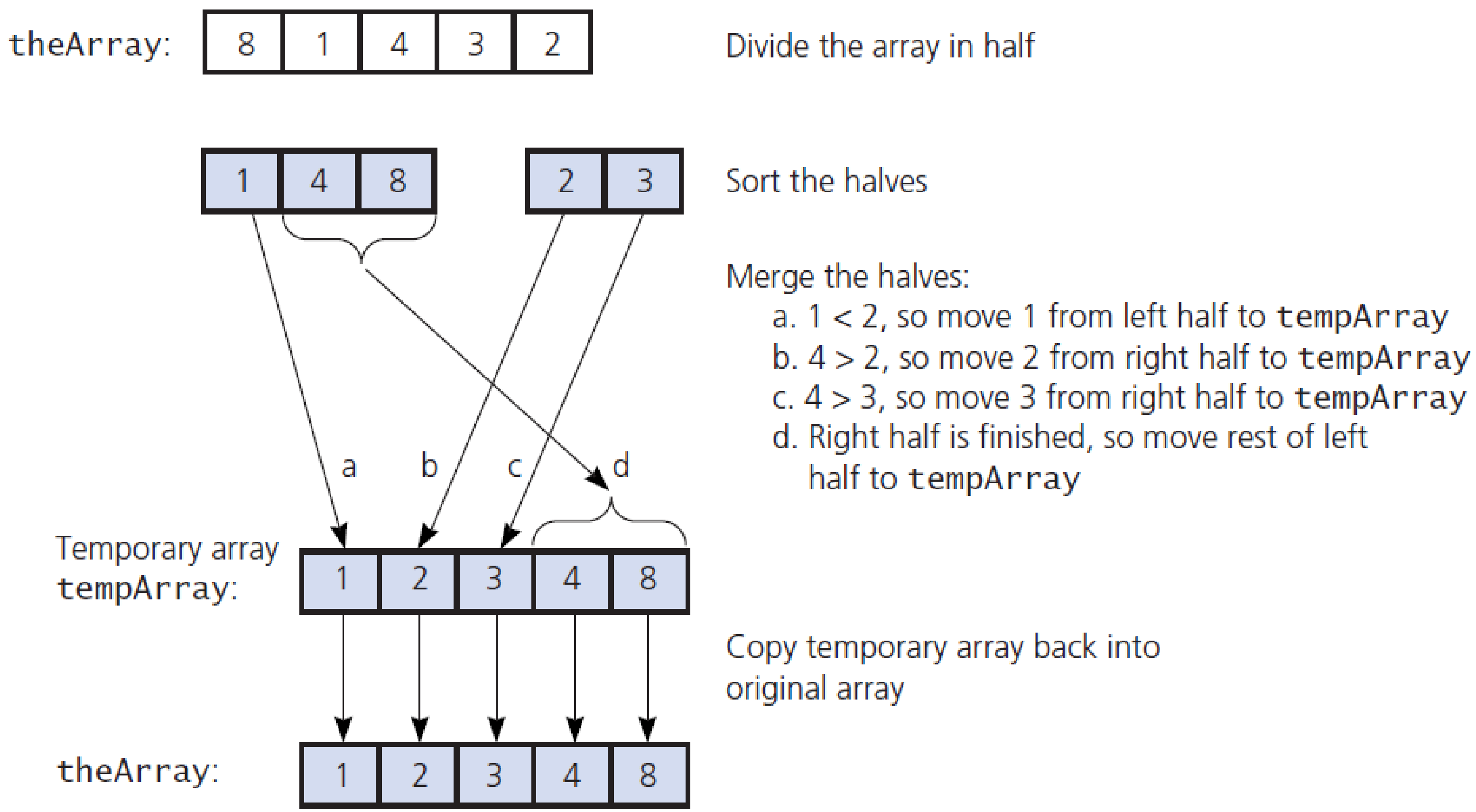
Merge Sort

- Visualization

2	8	5	3	9	4	1	7

Merge Sort

- A merge sort with an auxiliary temporary array



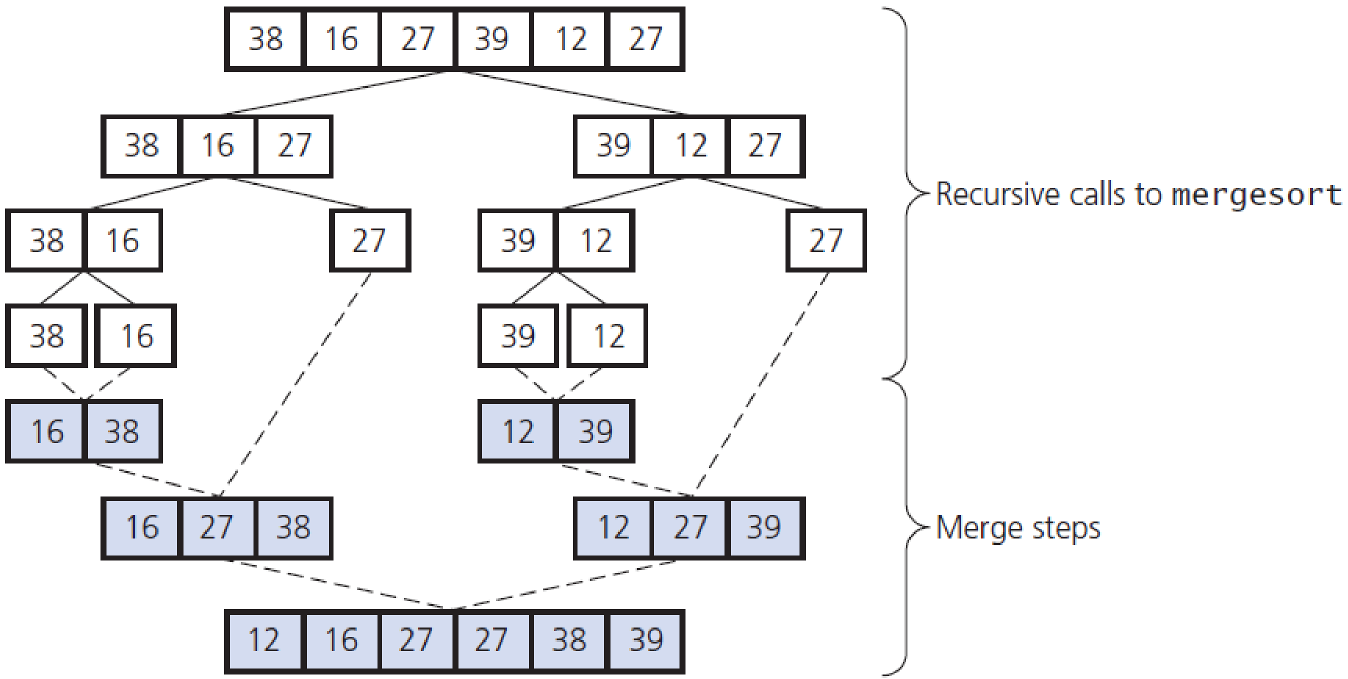
Merge Sort

Pseudocode for the merge sort

```
// 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

- A merge sort of an array of six integers



Merge Sort

- An implementation of the bubble sort

```
const int MAX_SIZE = 100; // maximum number of items in 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

      // 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];
                  first++
            }
            else
            {
                  tempArray[index] = theArray[first2];
                  first2++
            } // end if
            index++;
      } // end while
      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
```

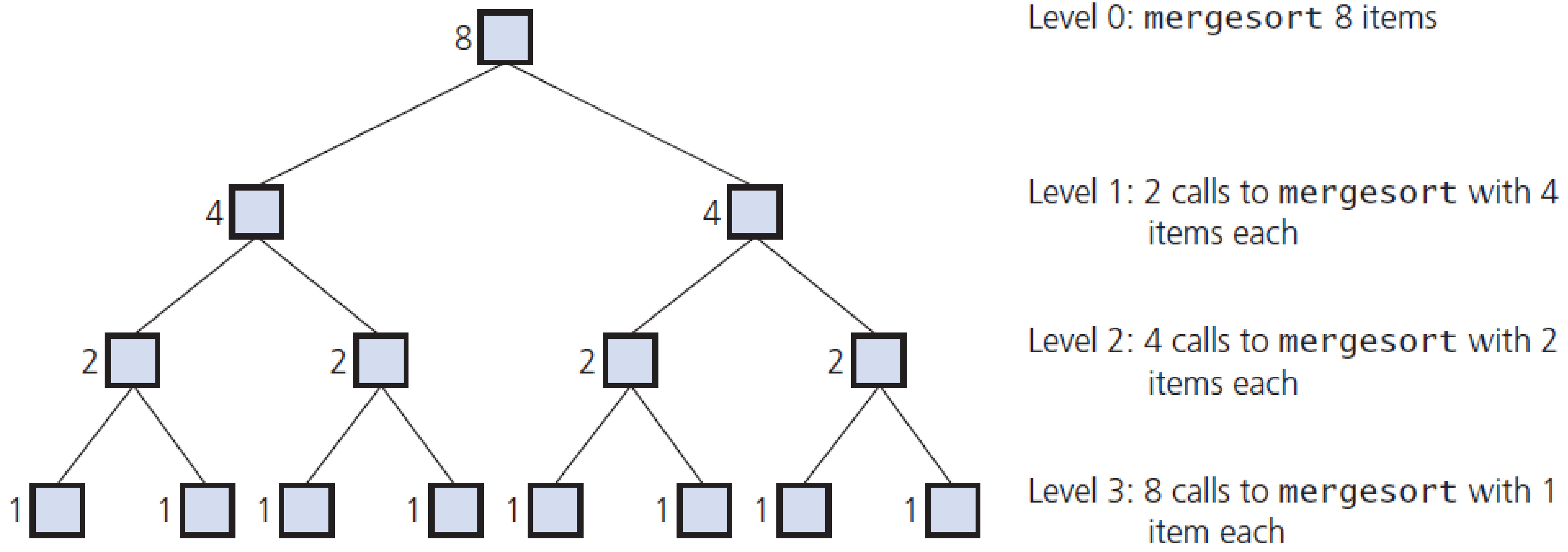
```
template <class ItemType>
void mergeSort(ItemType theArray[], int first, int last)
{
      if (first < last)
      {
            int mid = first + (last-first)/2;
            mergeSort(theArray,first,mid)l
            mergeSort(theArray,mid+1,last);
            merge(theArray, first, mid, last);
      } // end if
}
```

Merge Sort

- Each `merge` step merges `theArray[first..mid]` and `theArray[mid+1..last]`.
- 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.
- In addition, there are n moves from the original array to the temporary array, and n moves from the temporary array back to the original array. Thus, each merge step requires $3 \times n - 1$ major operations.
- Each call to `mergeSort` recursively calls itself twice. Each call to `mergeSort` halves the array. If n is a power of 2, the recursion goes $k = \log_2 n$ levels deep. If n is not a power of 2, there are $1 + \log_2 n$ levels of recursive calls to `mergeSort`.
- The original call to `mergeSort` calls `merge` once. Then `merge` merges all n items and requires $3 \times n - 1$ operations. At level m of the recursion, 2^m calls `merge` to occur, each of these calls merges $n/2^m$ items and so requires $3 \times (n/2^m) - 1$ operations. Together, the 2^m calls to `merge` require $3 \times n - 2^m$ operations.
- Thus each level of recursion requires $O(n)$ giving a total of $O(nlogn)$

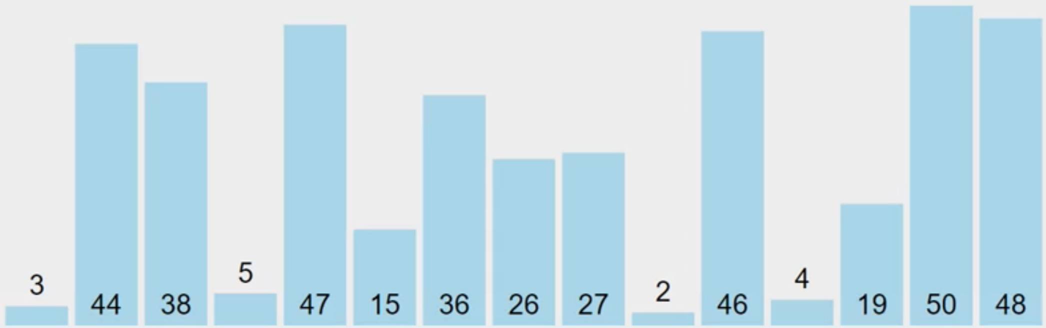
Merge Sort

Recursive calls to mergeSort, given an array of Levelseight items



Merge Sort

- Analysis
  - Worst case O(nlogn)
  - Average case O(nlogn)



Merge Sort

```
split each element into partitions of size 1
recursively merge adjancent partitions
  for i = leftPartIdx to rightPartIdx
    if leftPartHeadValue <= rightPartHeadValue
      copy leftPartHeadValue
    else: copy rightPartHeadValue
copy elements back to original array
```

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 or equal to the pivot
- Partitioning places pivot in its correct position within the array
  - Place chosen pivot in theArray[last] before partitioning

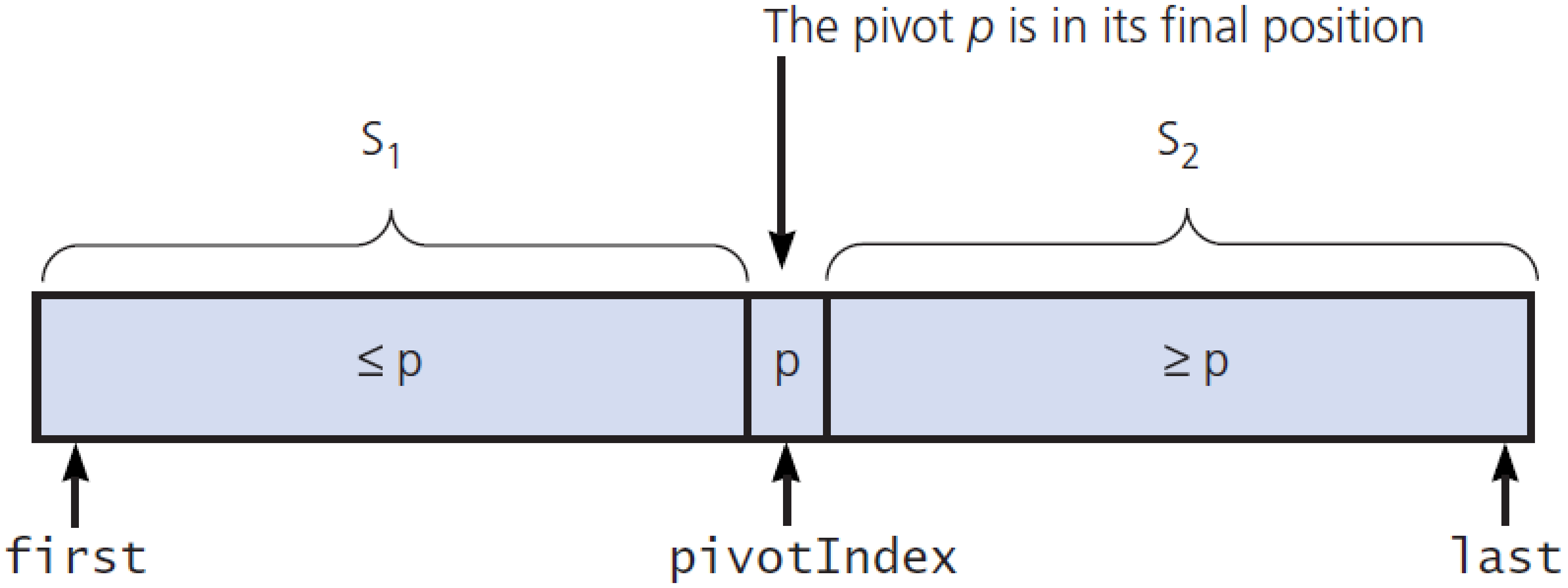
The Quick Sort

- Visualization

2	6	5	3	8	7	1	0

The Quick Sort

- A partition about a pivot



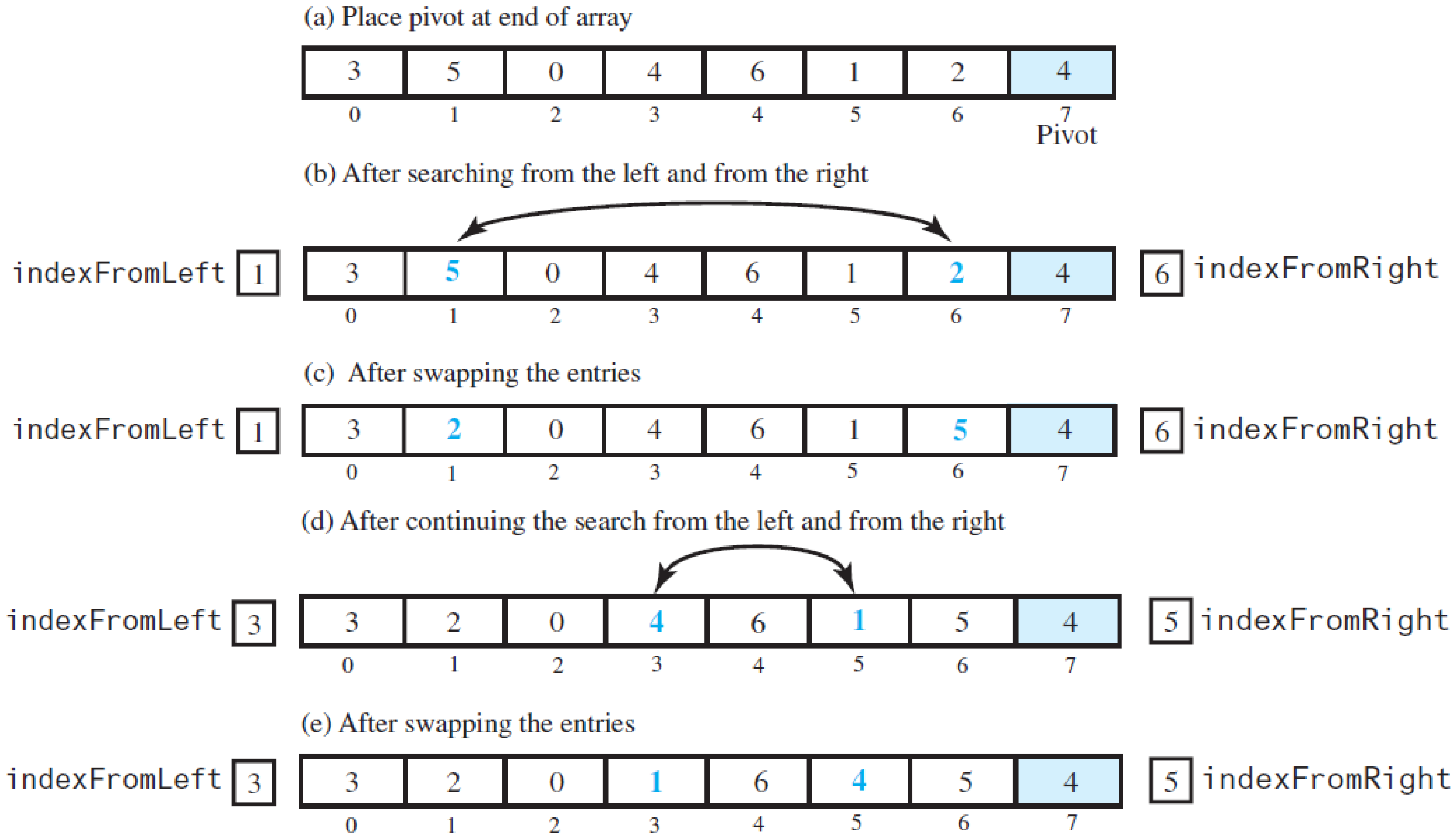
The Quick Sort

- First draft of pseudocode for the quick sort algorithm

```
// 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
}
```

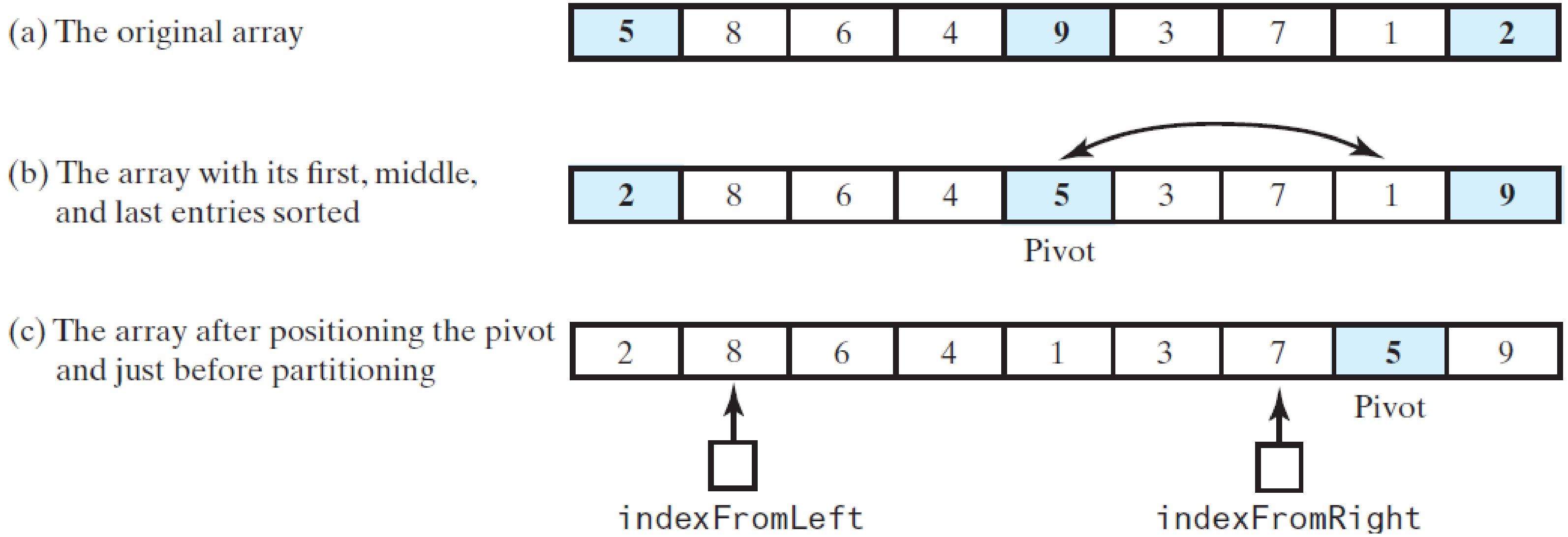
The Quick Sort

- A partitioning of an array during a quicj sort



The Quick Sort

- Median-of-three pivot selection



The Quick Sort

- Adjusting the partition algorithm

```
// Arranges the first, middle, and last entries in an array into ascending order.
sortFirstMiddleLast(theArray: ItemArray, first: integer, mid: integer,
                    last: integer): void
{
   if (theArray[first] > theArray[mid])
      Interchange theArray[first] and theArray[mid]

   if (theArray[mid] > theArray[last])
      Interchange theArray[mid] and theArray[last]

   if (theArray[first] > theArray[mid])
      Interchange theArray[first] and theArray[mid]
}
```

The Quick Sort

- Pseudo code describes the partitioning algorithm for an array of at least 4 entries

```
// Partitions theArray[first..last].
partition(theArray: ItemArray, first: integer, last: integer): integer
{
   // Choose pivot and reposition it
   mid = first + (last - first) / 2
   sortFirstMiddleLast(theArray, first, mid, last)
   Interchange theArray[mid] and theArray[last - 1]
   pivotIndex = last - 1
   pivot = theArray[pivotIndex]

   // Determine the regions S1 and S2
   indexFromLeft = first + 1
   indexFromRight = last - 2

   done = false
   while (not done)
   {
      // Locate first entry on left that is ≥ pivot
```

The Quick Sort

- Pseudo code describes the partitioning algorithm for an array of at least 4 entries

```
done = false
while (not 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)
   {
      Interchange theArray[indexFromLeft] and theArray[indexFromRight]
      indexFromLeft = indexFromLeft + 1
      indexFromRight = indexFromRight - 1
   }
   else
      done = true
}
```

The Quick Sort

- Pseudo code describes the partitioning algorithm for an array of at least 4 entries

```
            indexFromRight = indexFromRight - 1
        }
        else
            done = true
    }
    // Place pivot in proper position between S1 and S2, and mark its new location
    Interchange theArray[pivotIndex] and theArray[indexFromLeft]
    pivotIndex = indexFromLeft

    return pivotIndex
}
```

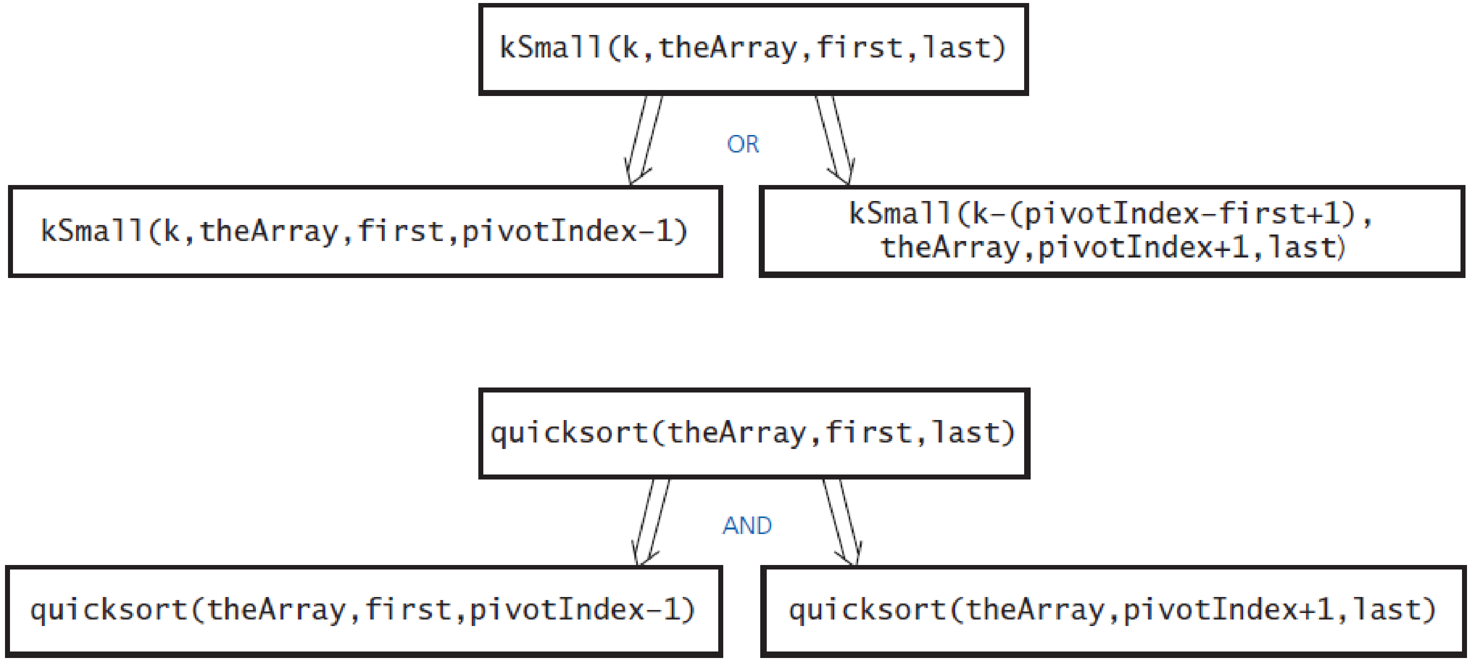
The Quick Sort

- A function performing quick sort

```
** 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 index of 1st element to consider in theArray
@param     last – the index of last element to cnsdr in theArray
template <class ItemType>
void quicksort(ItemType theArray[], int first, int last)
{
     if ((last-first+1)<MIN_SIZE)
     {
          insertionSort(theArray, first, last);
     }
     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
```

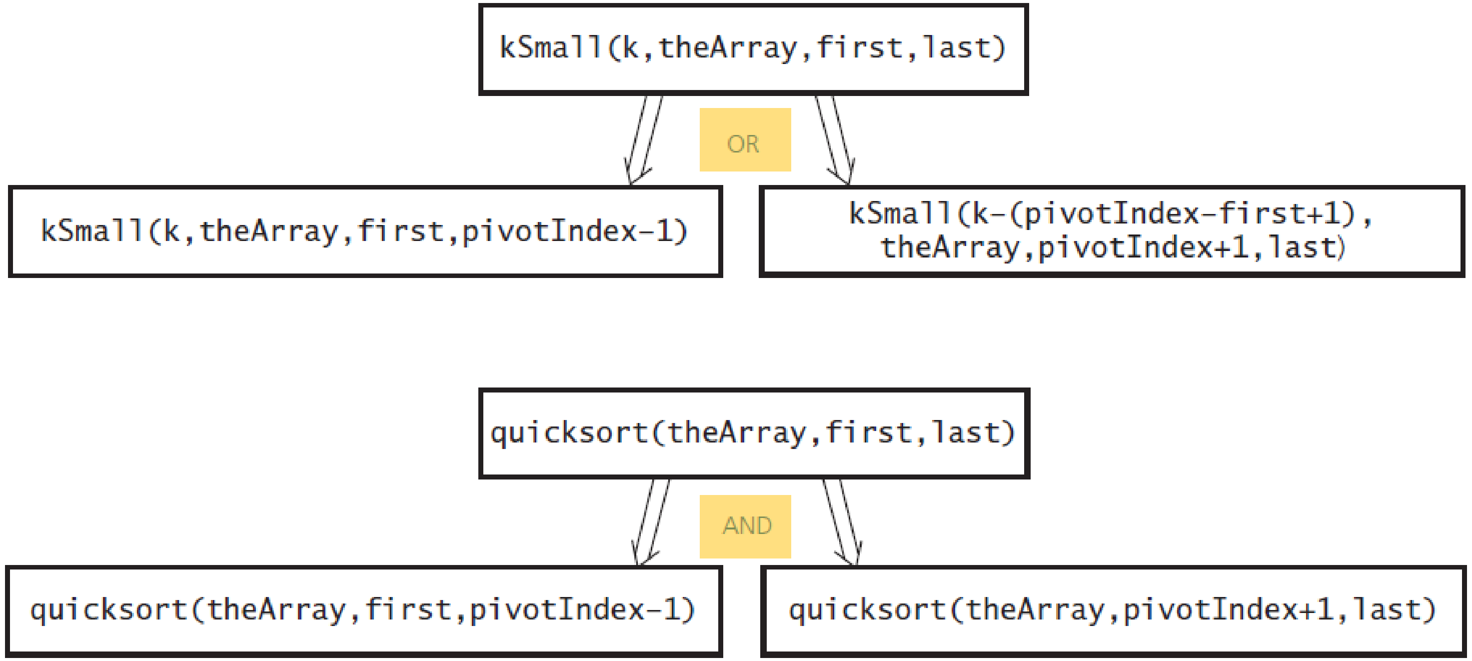
The Quick Sort

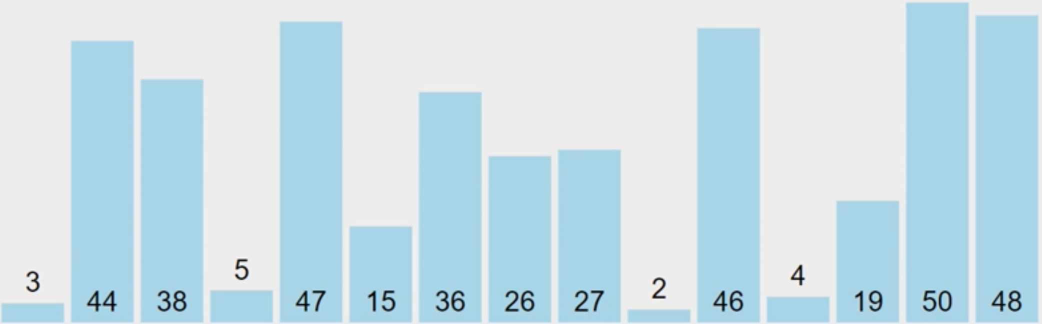
kSmall versus quickSort



The Quick Sort

kSmall versus quickSort





Quick Sort

```
for each (unsorted) partition
set first element as pivot
  storeIndex = pivotIndex + 1
  for i = pivotIndex + 1 to rightmostIndex
    if element[i] < element[pivot]
      swap(i, storeIndex); storeIndex++
  swap(pivot, storeIndex - 1)
```

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

3	44	38	5	47	15	36	26	27	2	46	4	19	50	48

0	1	2	3	4	5	6	7	8	9

Radix Sort

```
create 10 buckets (queues) for each digit (0 to 9)
for each digit placing
  for each element in list
    move element into respective bucket
  for each bucket, starting from smallest digit
    while bucket is non-empty
      restore element to list
```

Create

Sort

The Radix Sort

- A radix sort of eight intevers

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)

The Radix Sort

- Pseudocode for algorithm for a radix sort of n decimal integers of d digits each

```
// 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

```
// C++ implementation of Radix Sort
#include<iostream>
using namespace std;

// A utility function to get maximum value in arr[]
int getMax(int arr[], int n)
{
	int mx = arr[0];
	for (int i = 1; i < n; i++)
		if (arr[i] > mx)
			mx = arr[i];
	return mx;
}

// A function to do counting sort of arr[] according to
// the digit represented by exp.
void countSort(int arr[], int n, int exp)
{
	int output[n]; // output array
	int i, count[10] = {0};

	// Store count of occurrences in count[]
	for (i = 0; i < n; i++)
		count[ (arr[i]/exp)%10 ]++;

	// Change count[i] so that count[i] now contains actual
	// position of this digit in output[]
	for (i = 1; i < 10; i++)
		count[i] += count[i - 1];

	// Build the output array
	for (i = n - 1; i >= 0; i--)
	{
		output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];
		count[ (arr[i]/exp)%10 ]--;
	}

	// Copy the output array to arr[], so that arr[] now
	// contains sorted numbers according to current digit
	for (i = 0; i < n; i++)
		arr[i] = output[i];
}
```

```
// The main function to that sorts arr[] of size n using
// Radix Sort
void radixsort(int arr[], int n)
{
	// Find the maximum number to know number of digits
	int m = getMax(arr, n);

	// Do counting sort for every digit. Note that instead
	// of passing digit number, exp is passed. exp is 10^i
	// where i is current digit number
	for (int exp = 1; m/exp > 0; exp *= 10)
		countSort(arr, n, exp);
}

// A utility function to print an array
void print(int arr[], int n)
{
	for (int i = 0; i < n; i++)
		cout << arr[i] << " ";
}

// Driver program to test above functions
int main()
{
	int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
	int n = sizeof(arr)/sizeof(arr[0]);
	radixsort(arr, n);
	print(arr, n);
	return 0;
}
```

The Radix Sort

- The algorithm requires n moves each time it forms groups and n moves to combine them again into one group.
- The algorithm performs these $2\ x\ n$ moves d times.
- Therefore, radix sort requires $2\ x\ n\ \times\ d$ moves to sort n strings of d characters each.
- No comparisons are necessary.
- Radix Sort is $O(n)$

The Radix Sort

- Analysis
  - Requires n moves each time it forms groups
  - n moves to combine again into one group
  - Performs these 2 x n moves d times
  - Thus requires 2 x n x d moves
- Radix sort is of order O(n)
- More appropriate for chain of linked lists than for an array
- Challenge: it requires to accommodate many groups. For example in English language if you are to sort strings of uppercase letters, you need to accommodate 27 groups – one group for blanks and one for each letter.
  - For large n this requirement requires a lot of memory. Therefore better to use a chain of linked nodes for each of the 27 groups.

A comparison of Sorting Algorithms

- Approximate growth rates of time required for eight sorting algorithms

	Worst Case	Average Case
Selection sort	n^2	n^2
Bubble Sort	n^2	n^2
Insertion Sort	n^2	n^2
Merge Sort	nlogn	nlogn
Quick Sort	n^2	nlogn
Radix Sort	n^2	nlogn
Tree Sort	n^2	nlogn
Heap Sort	nlogn	nlogn

Thank you