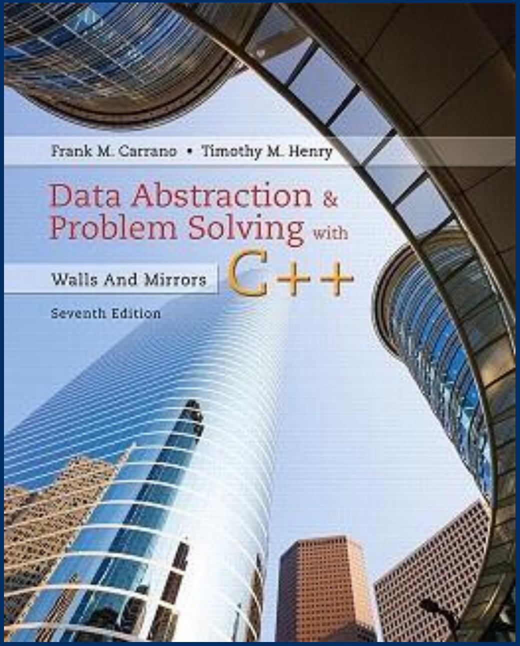
Chapter 11
Sorting Algorithms and their Efficiency

CS 302 - Data Structures

M. Abdullah Canbaz



Modified from authors' slides

Merge and Quick Sort

Reminders

- Midterm is on **Wednesday March 14th**
- Review session on **Monday March 12th**

- Quiz 6 is due on Wednesday
  – between 4pm to 11:59pm

Some Interview Questions

Alternate sorting of Linked list

...s of the list in such a way that the data in first node is first ... node is second maximum and so on.

Which sorting algorithms is most efficient to sort string consisting of ASCII characters?

- A Quick sort
- B Heap sort
- C Merge sort
- D

Sul...

We ha... order ... $A[i]$ > ... is by ... s

Probl... ...arify, one must deterr...

Input: The first ... integers where the ith integer i...

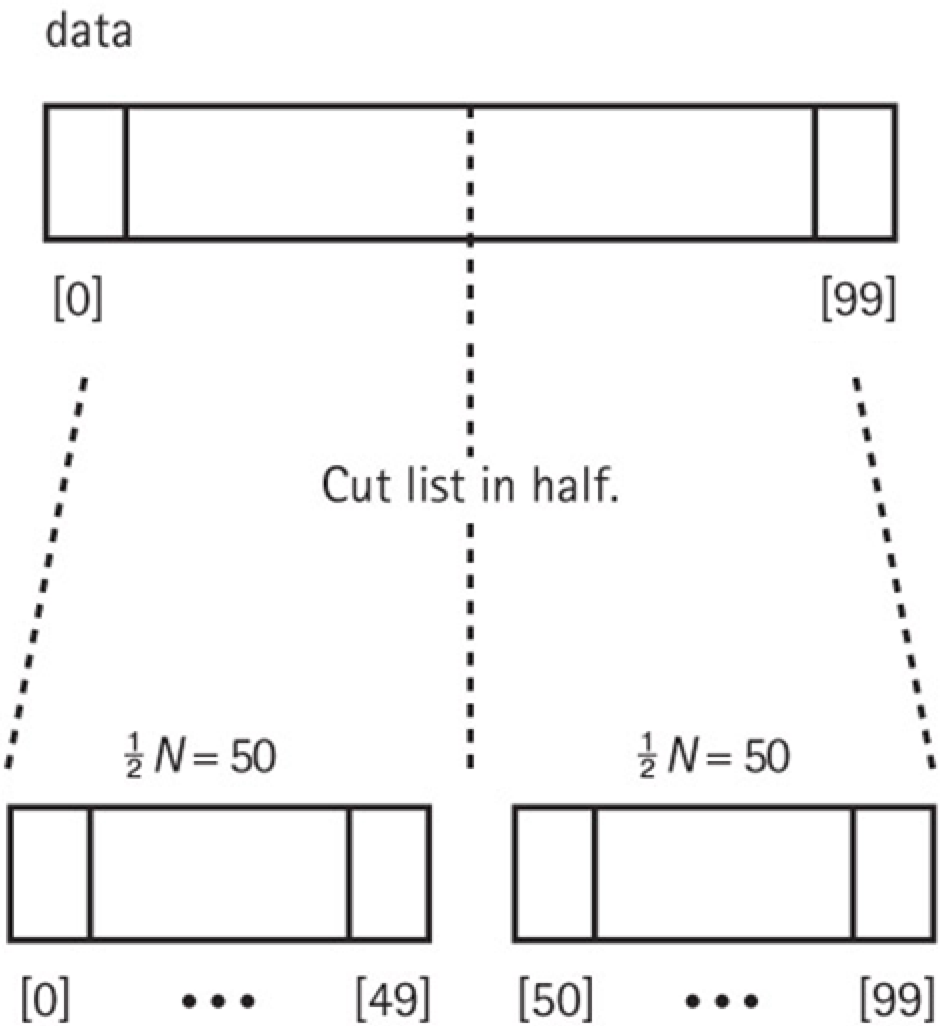
You have to sort 1 GB of data with only 100 MB of available main memory. Which sorting technique will be most appropriate?

- A Heap sort
- B Merge sort
- C Quick sort
- D Insertion sort

O(n logn) Sorts

Merge Sort

Divide and Conquer Sorts



$N = 100$

$N^2 = (100)^2 = 10{,}000$

$(\frac{1}{2}N)^2 + (\frac{1}{2}N)^2$ To sort

$+ \quad N$ To merge

$= (50)^2 + (50)^2 + 100$

$= 5100$

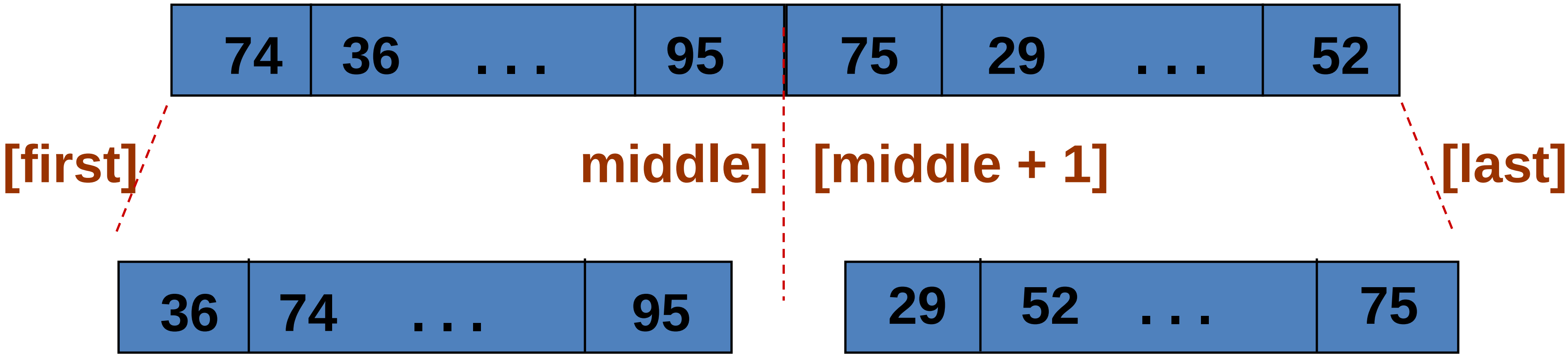
Merge Sort Algorithm

Cut the array in half.

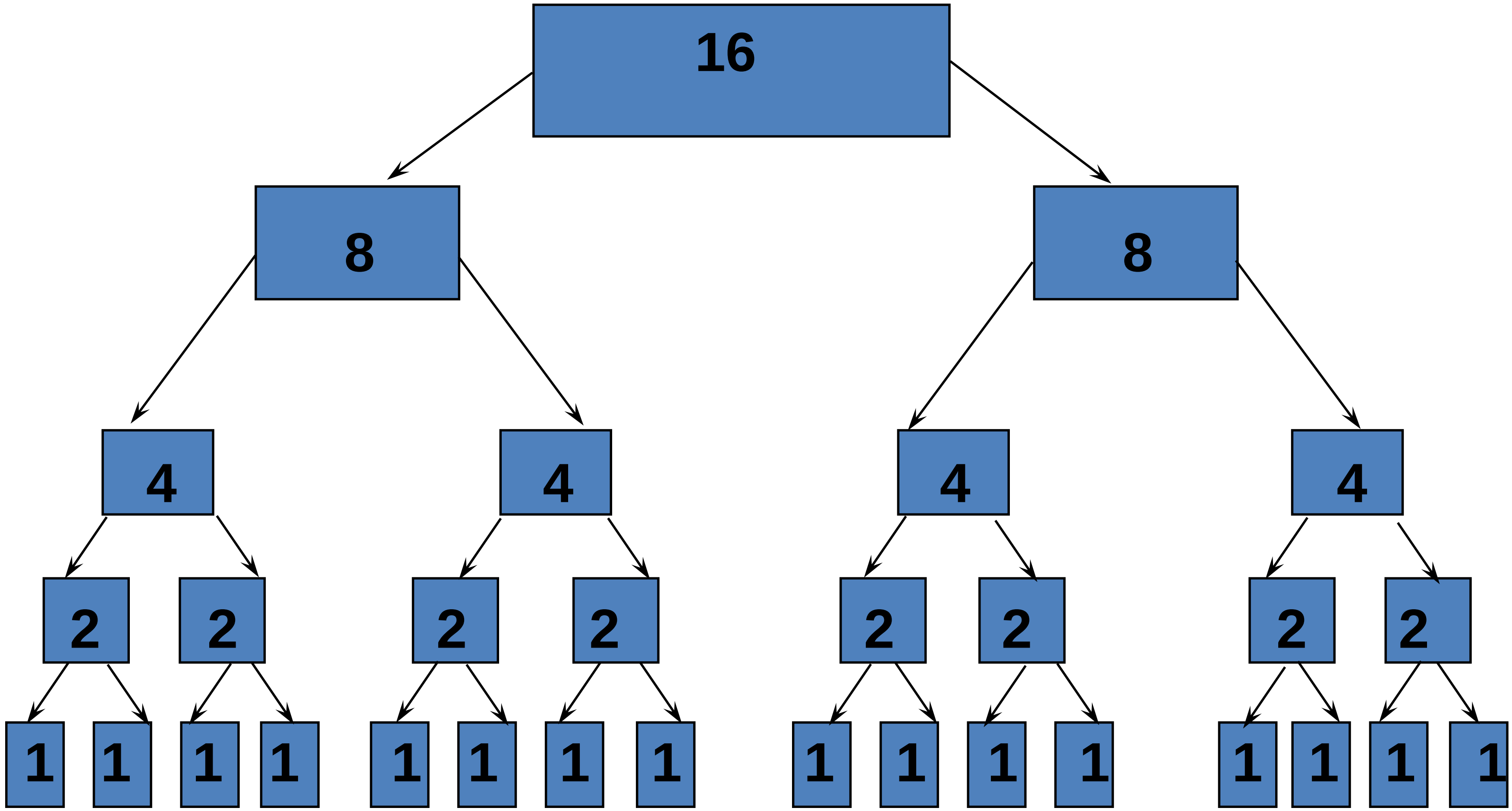
Sort the left half.

Sort the right half.

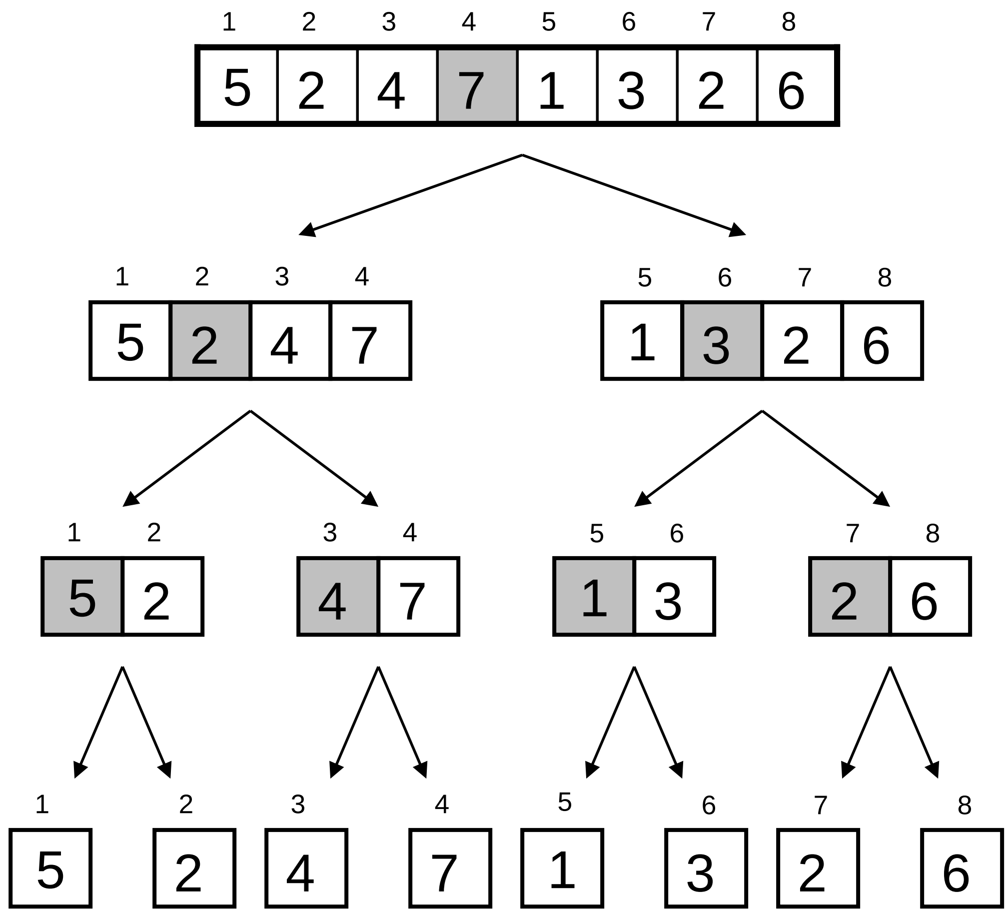
Merge the two sorted halves into one sorted array.



Using Merge Sort Algorithm

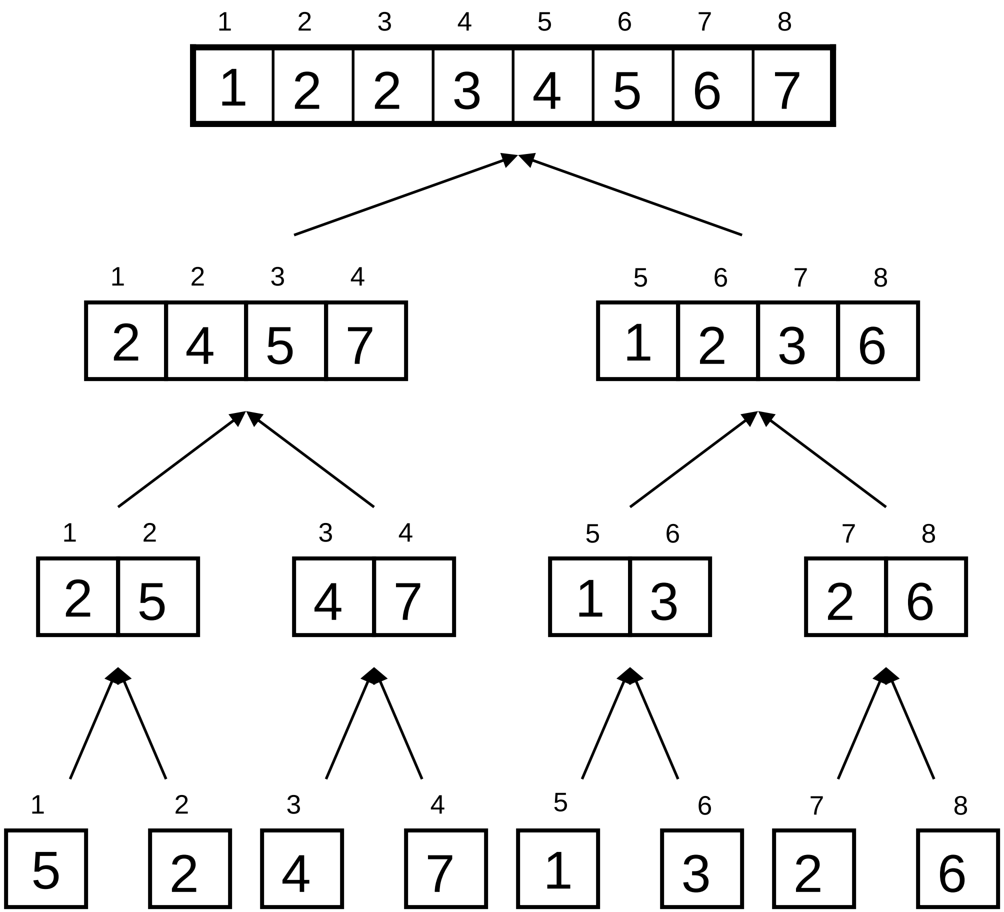


Example



middle = 4

Example (cont.)

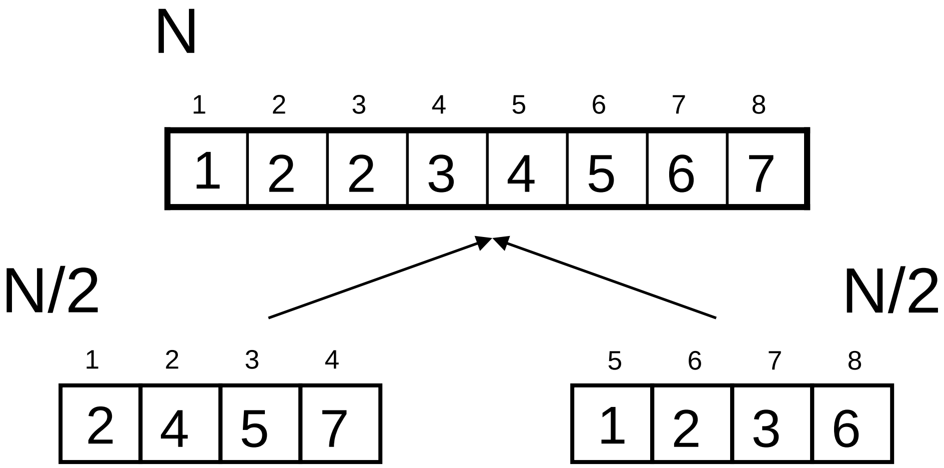


Merging

- Idea for merging:
  - Two piles of sorted cards
    - Choose the smaller of the two top cards
    - Remove it and place it in the output pile
  - Repeat the process until one pile is empty
  - Take the remaining input pile and place it face-down onto the output pile

Merge Step

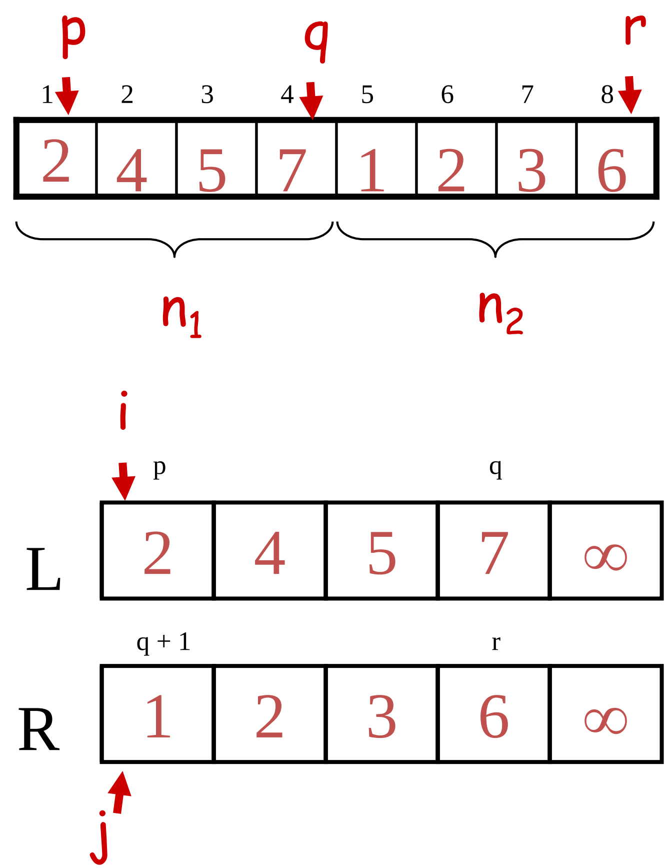
- **Merge two “sorted” lists into a new “sorted” list**
- **Can be done in O(N) time**



Merge - Pseudocode

Alg.: MERGE(A, p, q, r)

1. Compute n_1 and n_2
2. Copy the first n_1 elements into $L[1 \, . \, . \, n_1 + 1]$ and the next $\mathbf{n_2}$ elements into $\mathbf{R[1 \, . \, . \, n_2 + 1]}$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r
6. **do if** $L[\,i\,] \le R[\,j\,]$
7. **then** $A[k] \leftarrow L[\,i\,]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[\,j\,]$
10. $j \leftarrow j + 1$



```
// Recursive merge sort algorithm

template <class  ItemType >
void MergeSort ( ItemType  values[ ], int first, int last )

//  Pre:   first <= last
//  Post: Array values[first..last] sorted into
//    ascending order.
{
   if  ( first < last )                    //  general case
   {
       int  middle = ( first  +  last ) / 2;
        MergeSort ( values, first,      middle );
        MergeSort ( values, middle + 1, last   );

        // now  merge two subarrays
        // values [ first . . . middle ] with
        // values [ middle + 1,  . . . last ].

        Merge(values,  first, middle, middle + 1, last);
   }
}
```

Merge Sort of N elements: How many comparisons?

The entire array can be subdivided into halves only $\log_2 N$ times

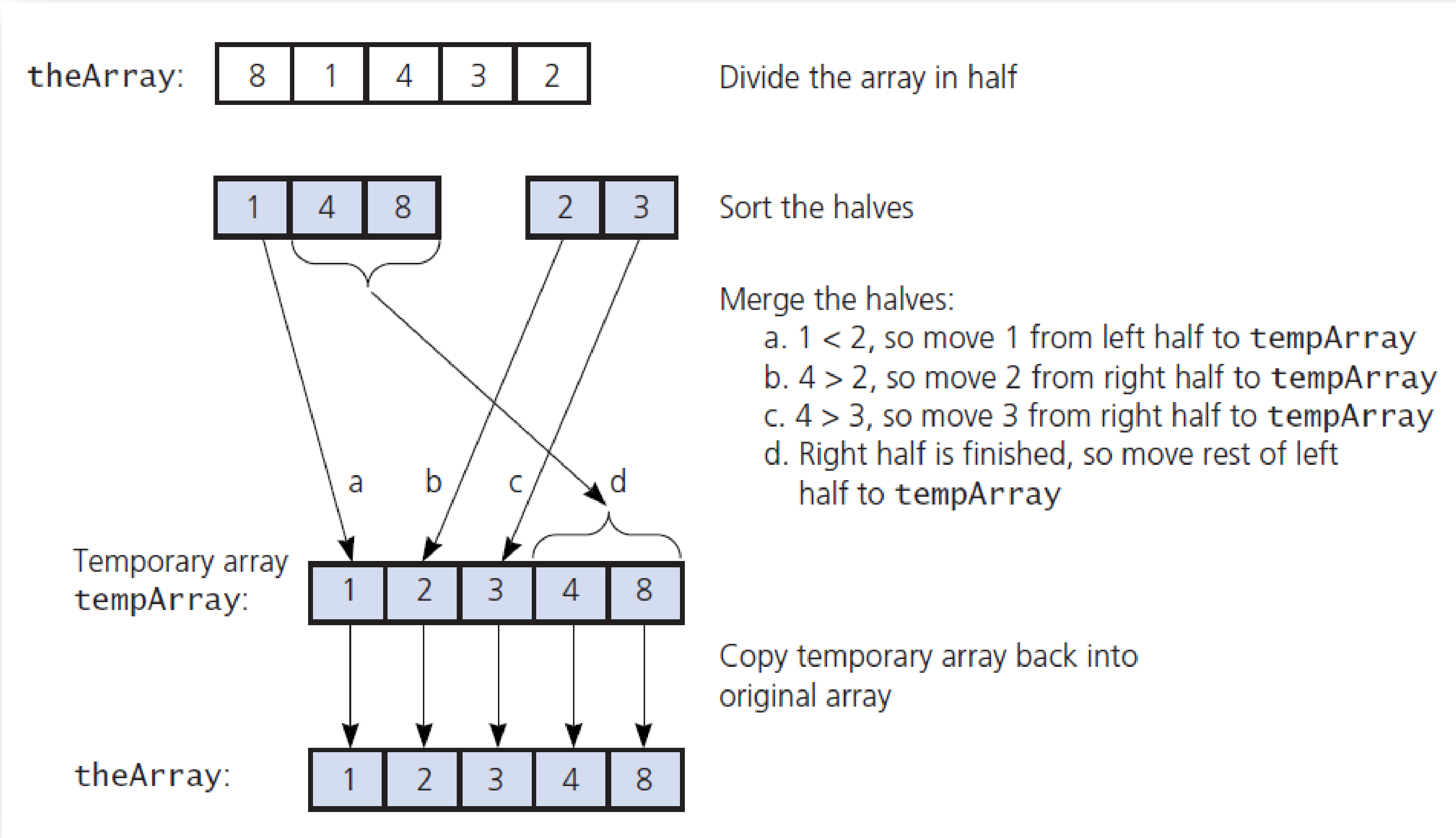
Each time it is subdivided, function Merge is called to re-combine the halves

- Function Merge uses a temporary array to store the merged elements
- Merging is O(N) because it compares each element in the subarrays

- Copying elements back from the temporary array to the values array is also O(N)

MERGE SORT IS $O(N*\log_2 N)$.

Merge Sort



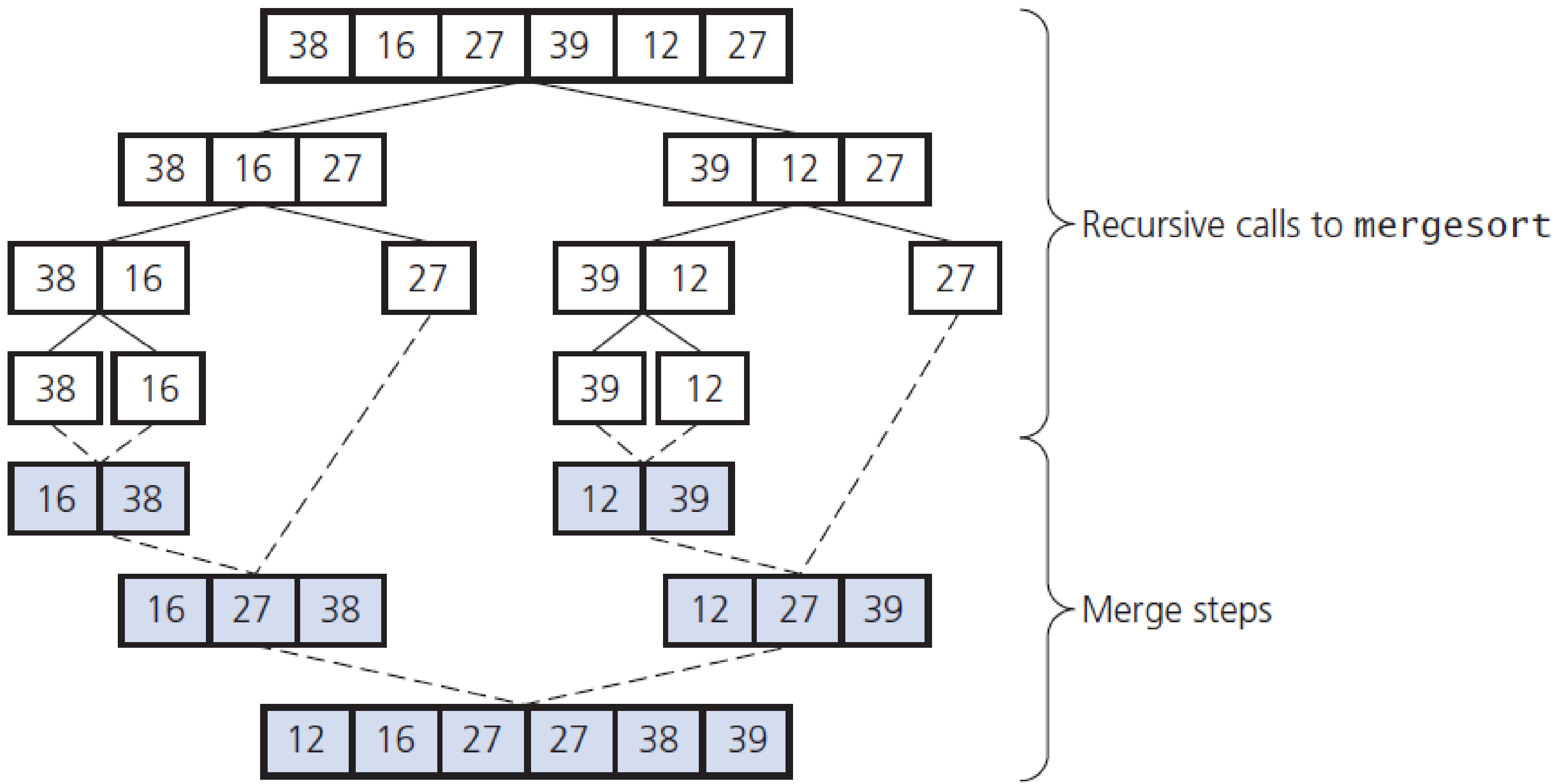
- A merge sort with an auxiliary temporary array

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
}
```

- Pseudocode for the merge sort

Merge Sort



- A merge sort of an array of six integers

Merge Sort

```
const int MAX_SIZE = maximum-number-of-items-in-array;

/** Merges two sorted array segments 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 each sorted in increasing order.
  @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 first segment in theArray;
       mid + 1 marks the beginning of the second segment.
  @param last  The index of the last element in the second segment in
       theArray.
```

- An implementation of merge and mergeSort

Merge Sort

```
 @note  This function merges the two subarrays into a temporary
    array and copies the result into the original array theArray. */
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))
   {
```

- An implementation of merge and mergeSort

Merge Sort

```
   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
   // Finish off the first subarray, if necessary
   while (first1 <= last1)
```

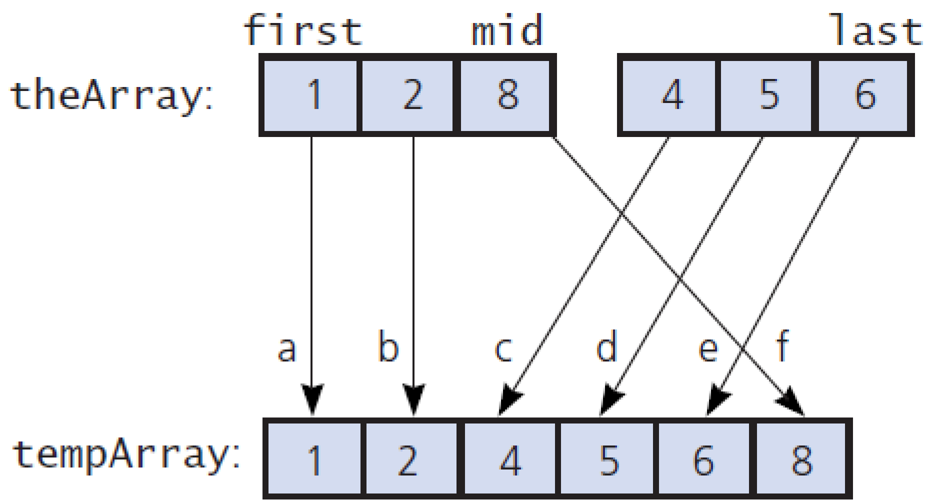
- An implementation of merge and mergeSort

Merge Sort

```
   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
```

- An implementation of merge and mergeSort

Merge Sort

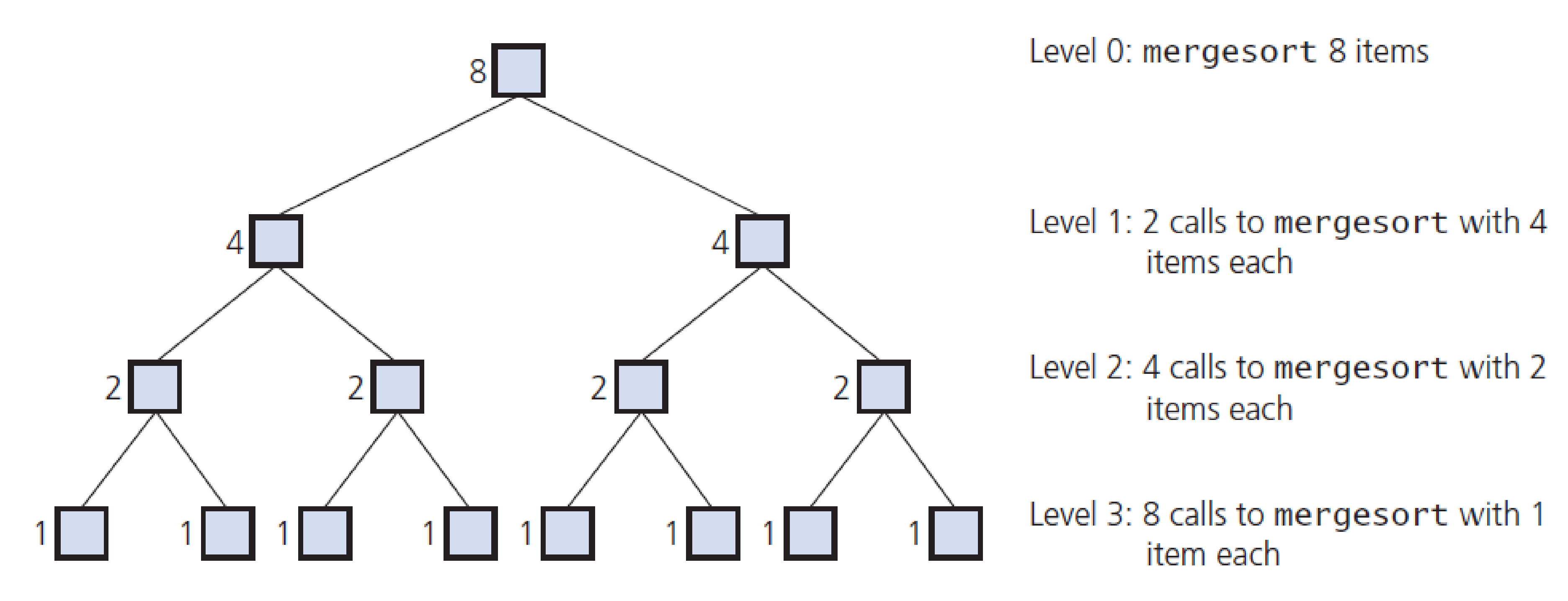


A worst-case instance of the merge step in a merge sort

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`

Merge Sort



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

Merge Sort - Discussion

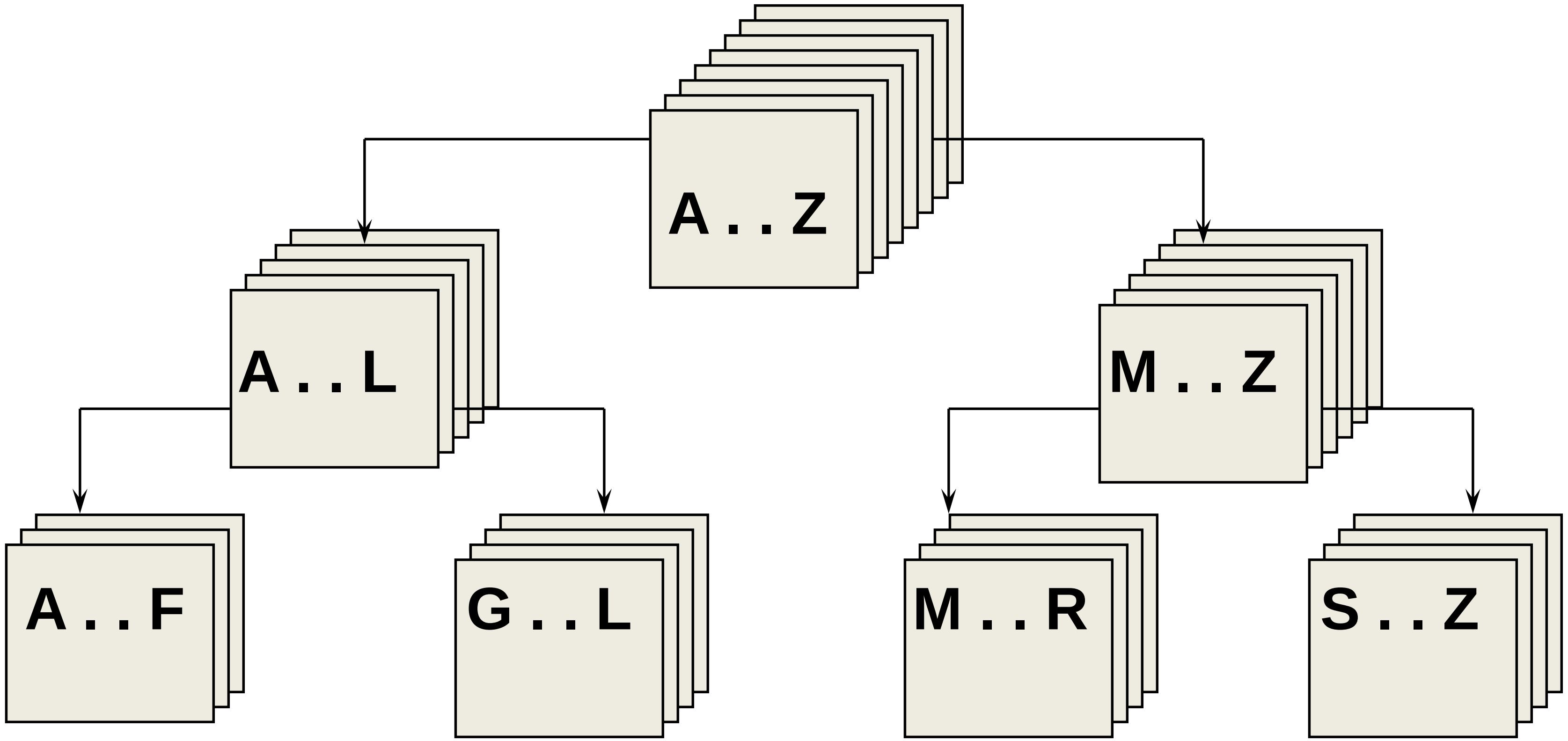
- Running time insensitive of the input
- Advantages:
  - Guaranteed to run in **$\Theta(n\lg n)$**
- Disadvantage
  - Requires extra space ≈N
- Applications
  - Maintain a large ordered data file
  - How would you use Merge sort to do this?

Quick Sort

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
  - Place chosen pivot in theArray[last] before partitioning

Using quick sort algorithm



Split

20	*14*	*11*	*18*	*3*	*6*	*60*	*9*

splitVal = 9

smaller values in left part

larger values in right part

6	*3*	*9*	*18*	*14*	*20*	*60*	*11*

Before call to function Split

splitVal = 9

GOAL: place splitVal in its proper position with all values less than or equal to splitVal on its left and all larger values on its right

9	*20*	*6*	*18*	*14*	*3*	*60*	*11*

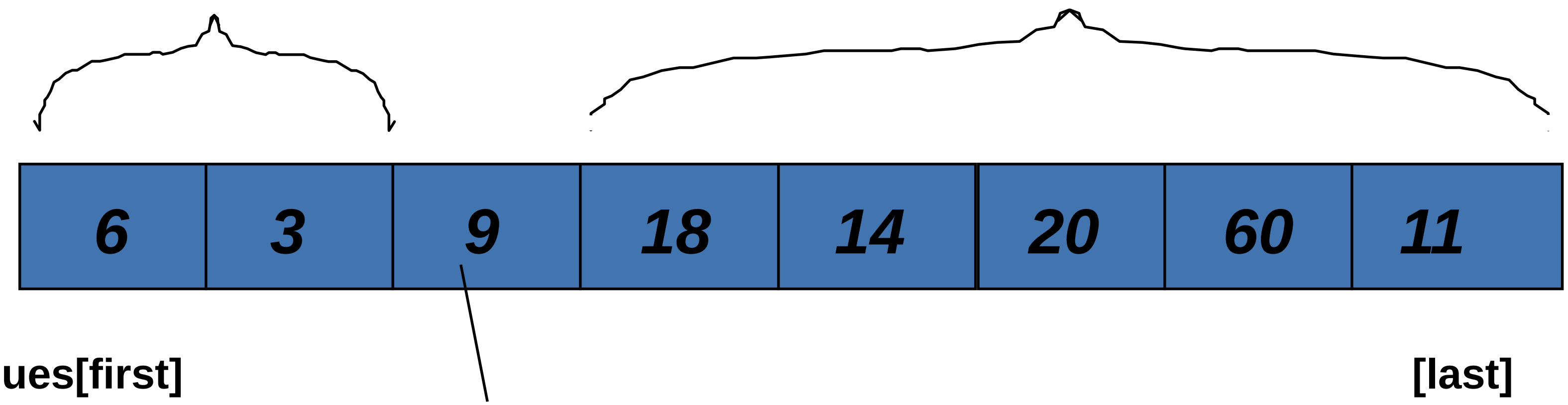
values[first] **[last]**

After call to function Split

splitVal = 9

smaller values
in left part

larger values
in right part

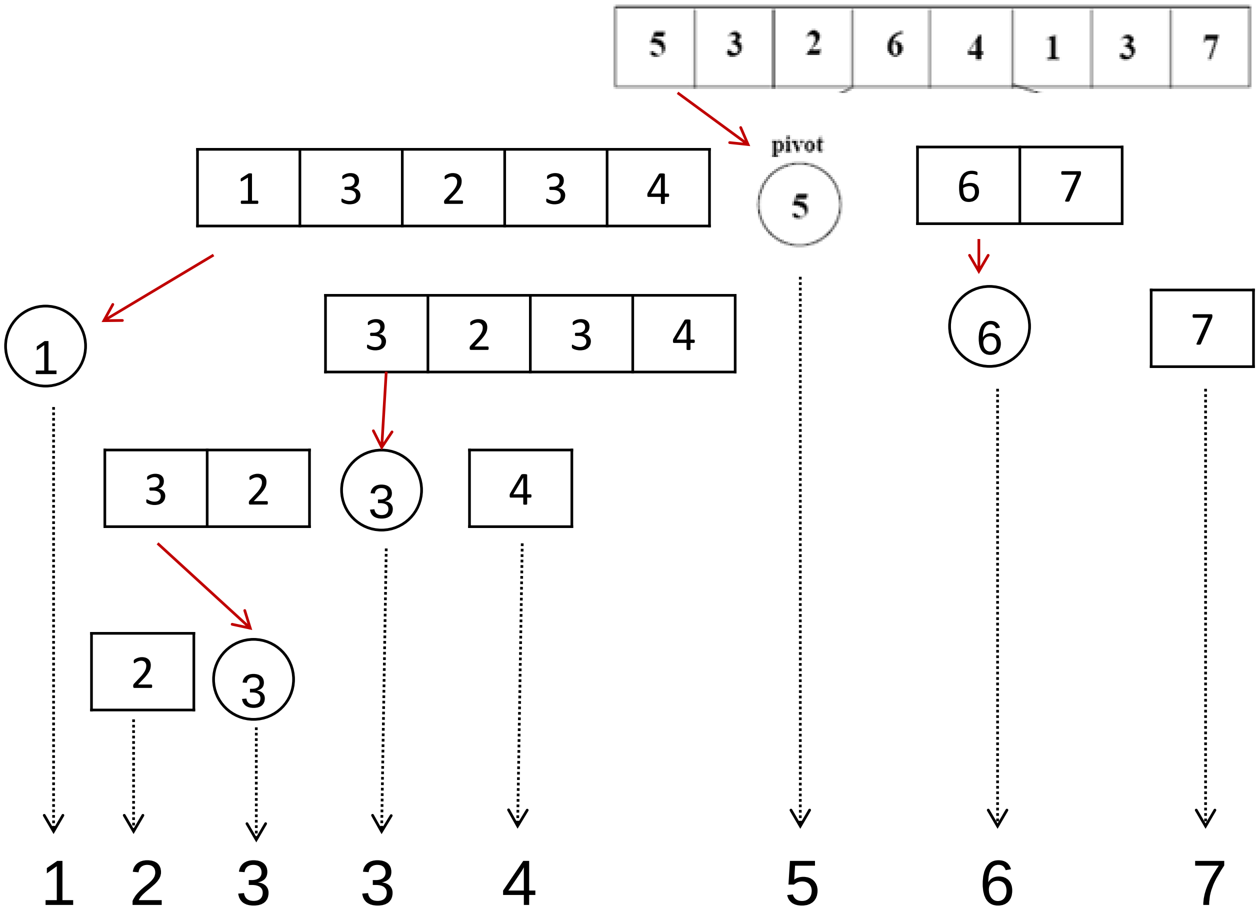


values[first]

[last]

splitVal in correct position

Example



```
// Recursive quick sort algorithm

template <class  ItemType >
void QuickSort ( ItemType  values[ ], int first, int last )

//  Pre:   first <= last
//  Post: Sorts array values[ first .   . last ] into
   ascending order
{
   if  ( first < last )                   //  general case
   {
      int  splitPoint = first;
      Split ( values, first, last, splitPoint ) ;
      // values [first]..values[splitPoint - 1] <= splitVal
      // values  [splitPoint] = splitVal
      // values [splitPoint + 1]..values[last] > splitVal
      QuickSort(values,  first,  splitPoint - 1);
      QuickSort(values,  splitPoint + 1,  last);
   }
} ;
```

Quick Sort of N elements: How many comparisons?

N	For first call, when each of N elements is compared to the split value
2 * N/2	For the next pair of calls, when N/2 elements in each "half" of the original array are compared to their own split values.
4 * N/4	For the four calls when N/4 elements in each "quarter" of original array are compared to their own split values.

.

.

.

HOW MANY SPLITS CAN OCCUR?

Quick Sort of N elements: How many splits can occur?

It depends on the order of the original array elements!

If each split divides the subarray approximately in half, there will be only $\log_2 N$ splits, and QuickSort is $O(N*\log_2 N)$.

But, if the original array was sorted to begin with, the recursive calls will split up the array into parts of unequal length, with one part empty, and the other part containing all the rest of the array except for split value itself.

In this case, there can be as many as N-1 splits, and QuickSort is $O(N^2)$.

Before call to function Split

splitVal = 9

GOAL: place splitVal in its proper position with all values less than or equal to splitVal on its left and all larger values on its right

9	***20***	***26***	***18***	***14***	***53***	***60***	***11***

values[first] **[last]**

After call to function Split

splitVal = 9

no smaller values
empty left part

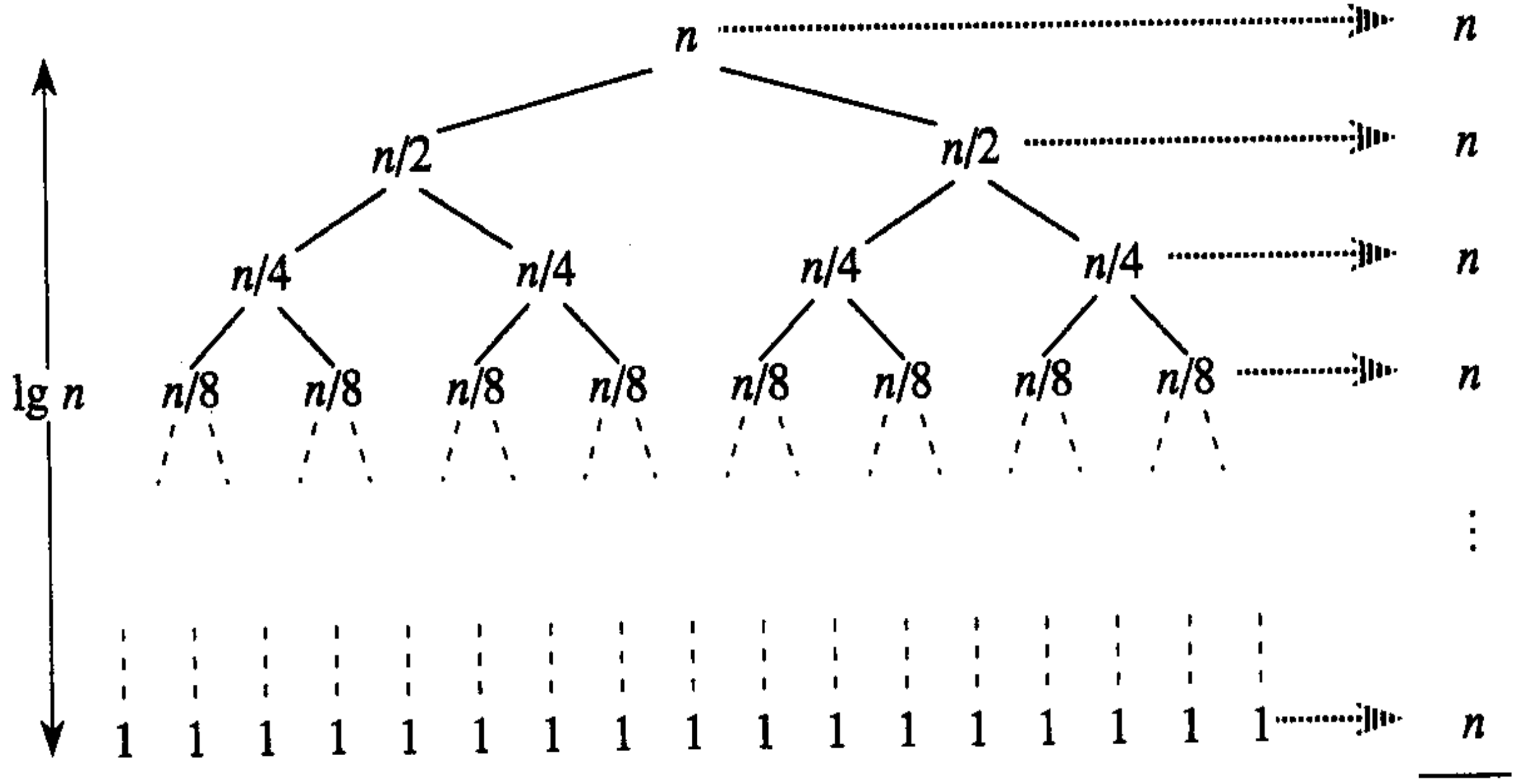
larger values
in right part with N-1 elements

9	*20*	*26*	*18*	*14*	*53*	*60*	*11*

values[first] **[last]**

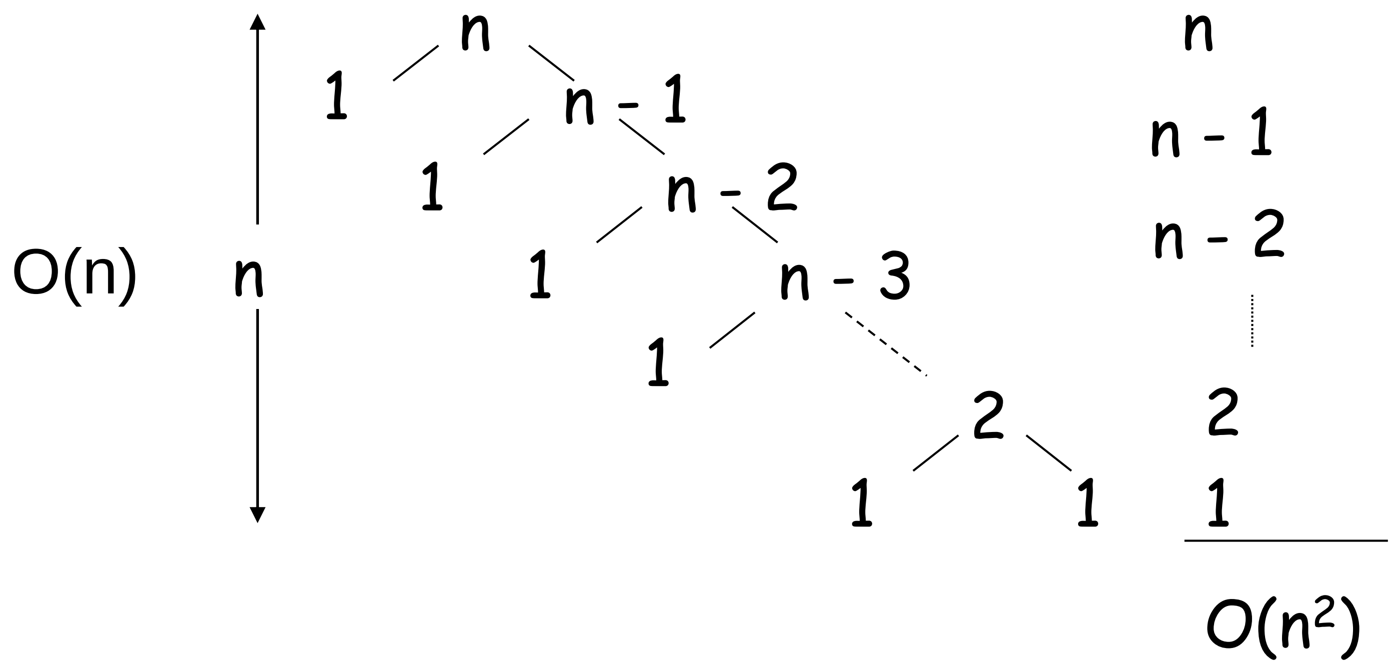
splitVal in correct position

Best case: balanced splits

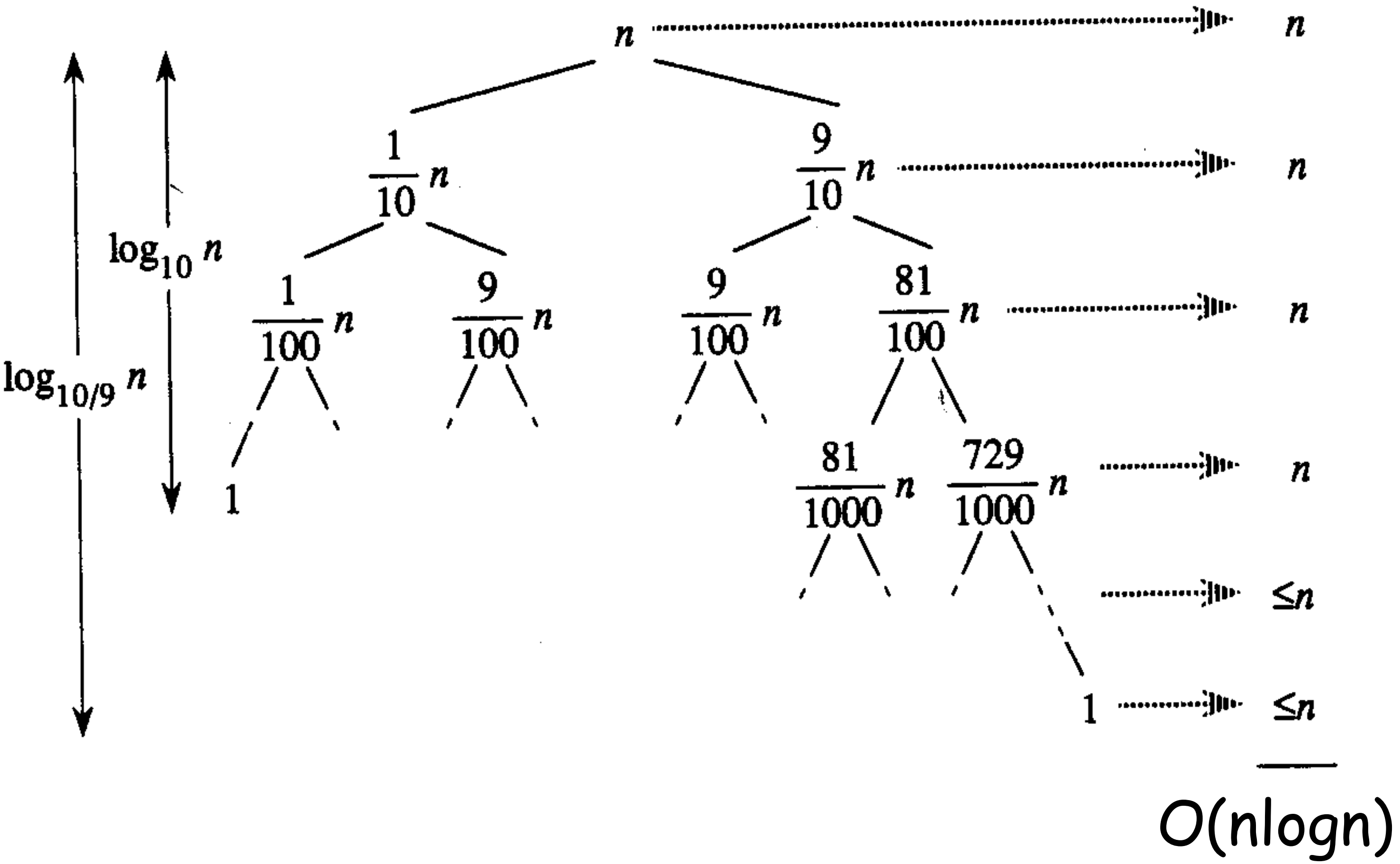


O(nlogn)

Worst case: unbalanced splits

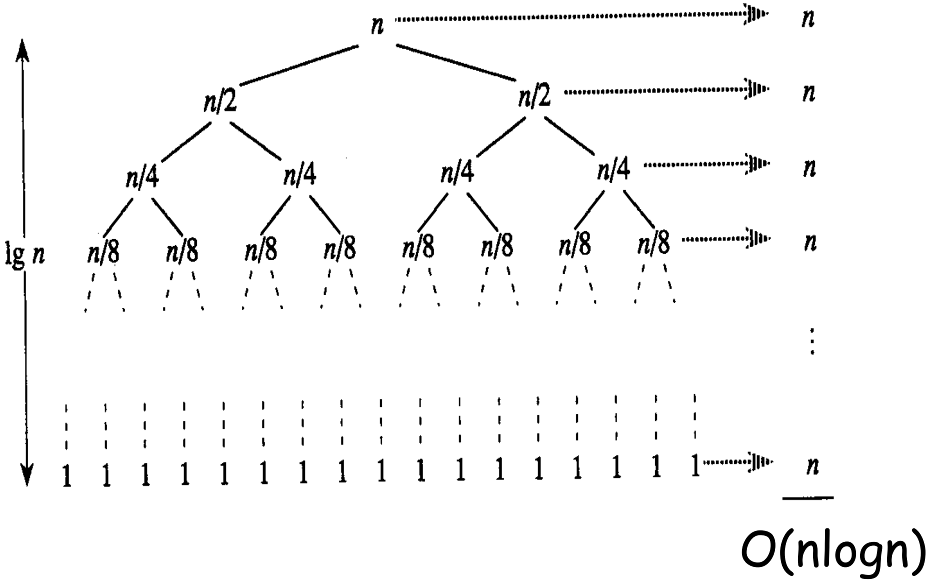


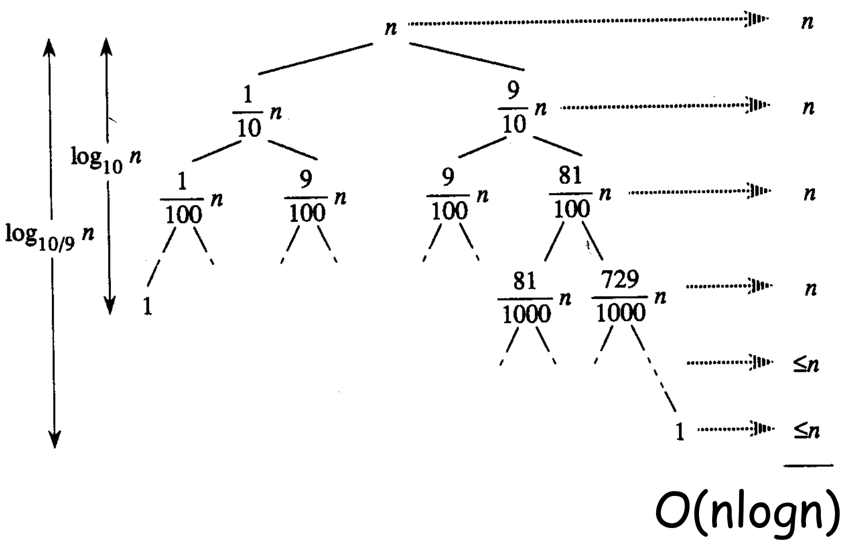
But … is every unbalanced split a bad split?



Need to look at split ratio!

Split ratio

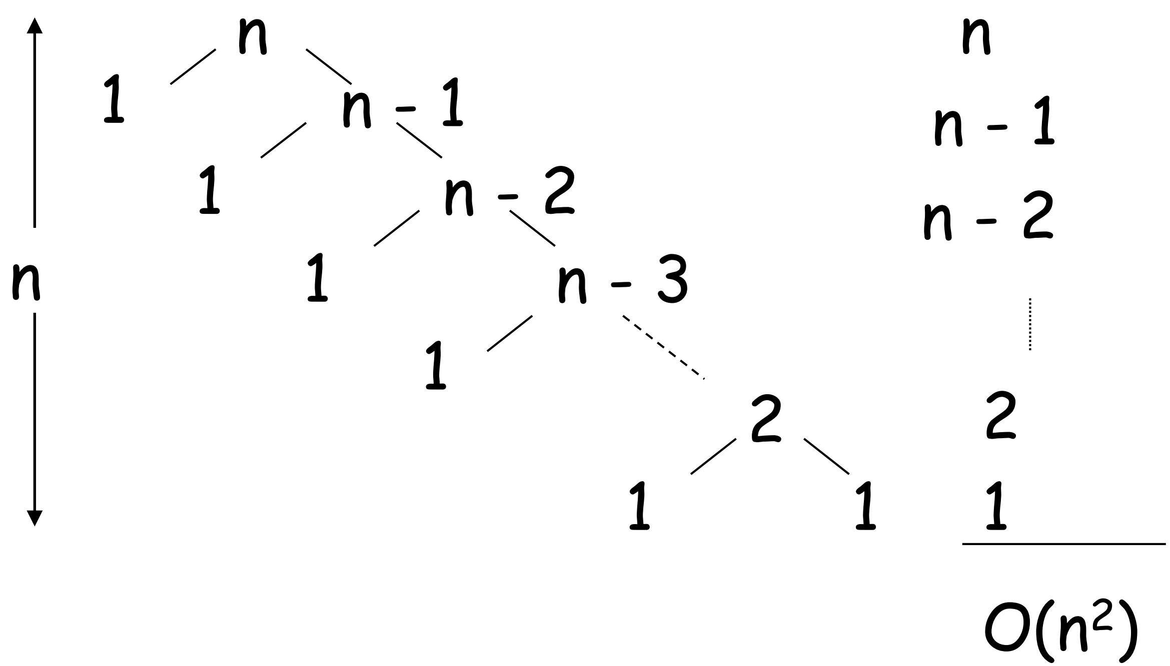




split ratio: (n/2) / (n/2) =const

split ratio: (n/10) / (9n/10) =const

Split ratio (cont'd)



split ratio: n / 1 = n not const

Randomized Quicksort

- Randomly permute the elements of the input array before sorting.

- Or, choose splitPoint randomly.

Randomized Quicksort

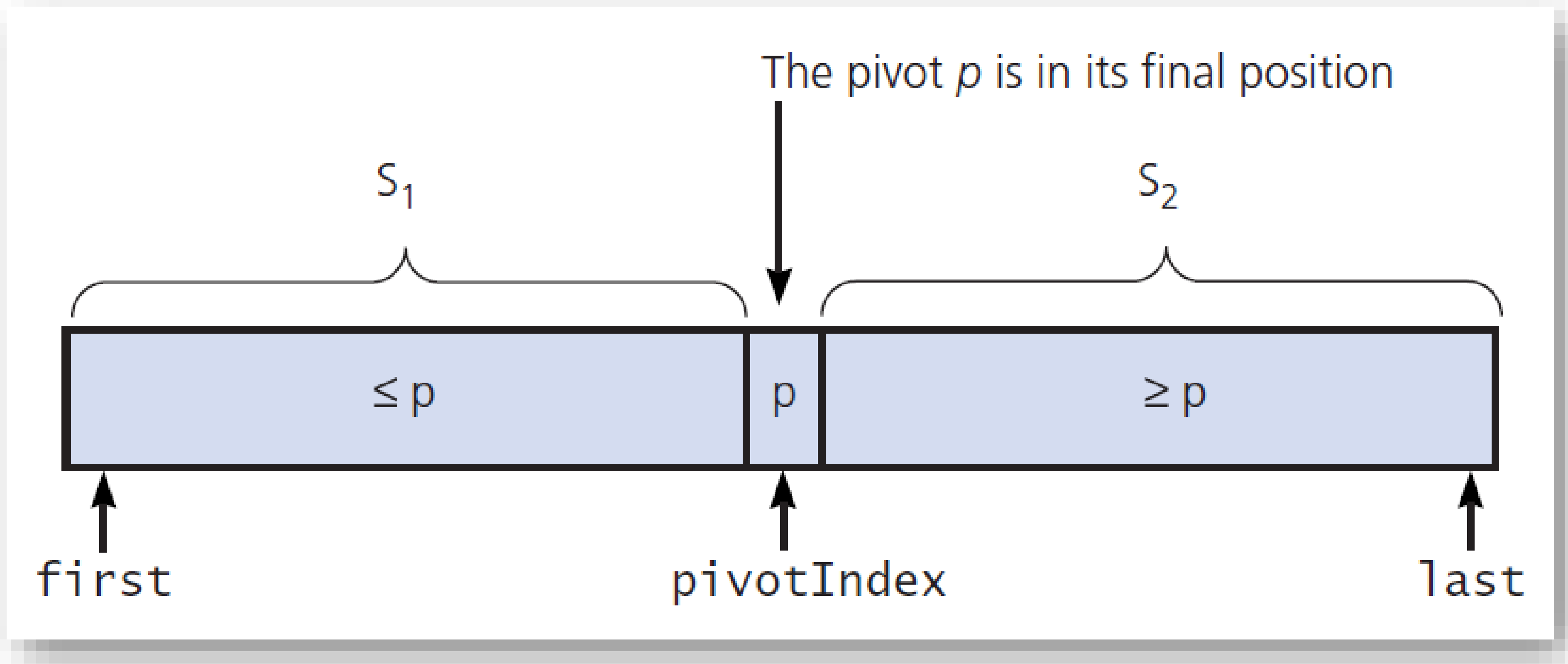
- At each step of the algorithm we exchange element $A[p]$ with an element chosen at **random** from $A[p \ldots r]$

- The pivot element $x = A[p]$ is equally likely to be any one of the $r - p + 1$ elements of the subarray

Randomized Quicksort

- Worst case becomes less likely
  - Worst case occurs only if we get “unlucky” numbers from the random number generator.
  - Randomization can NOT eliminate the worst-case but it can make it less likely!

The Quick Sort



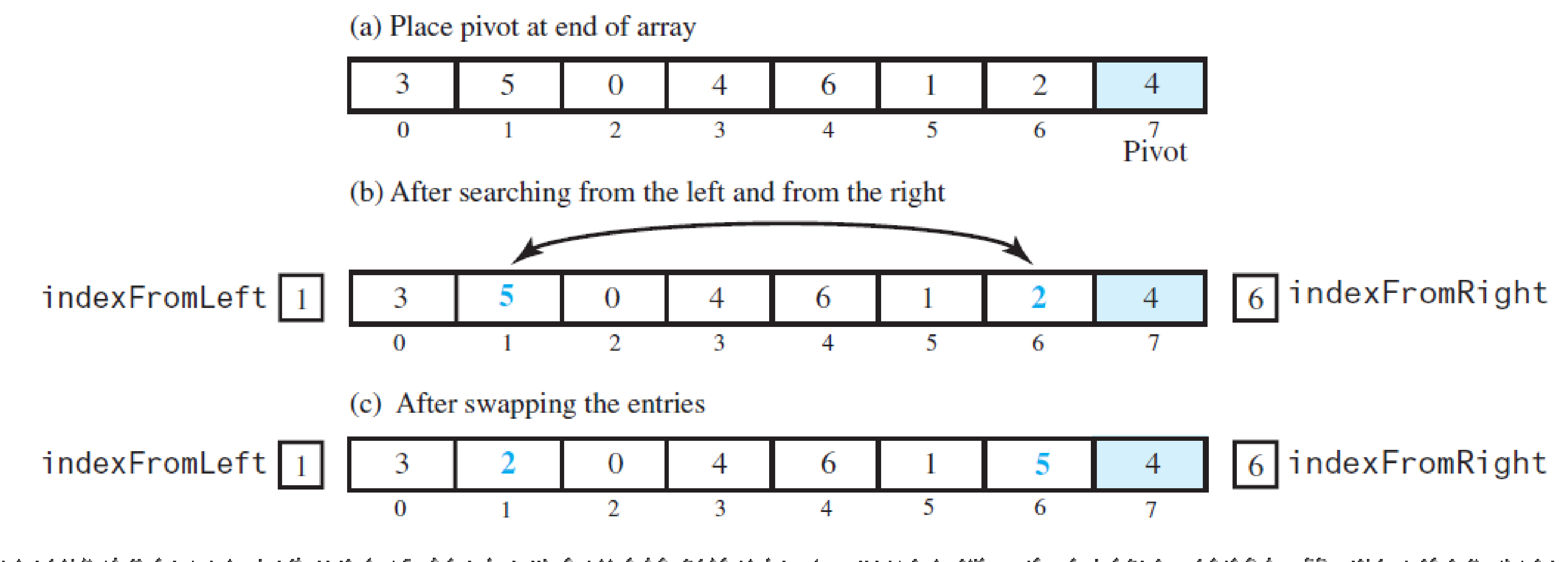
- A partition about a pivot

The Quick Sort

```
// 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
}
```

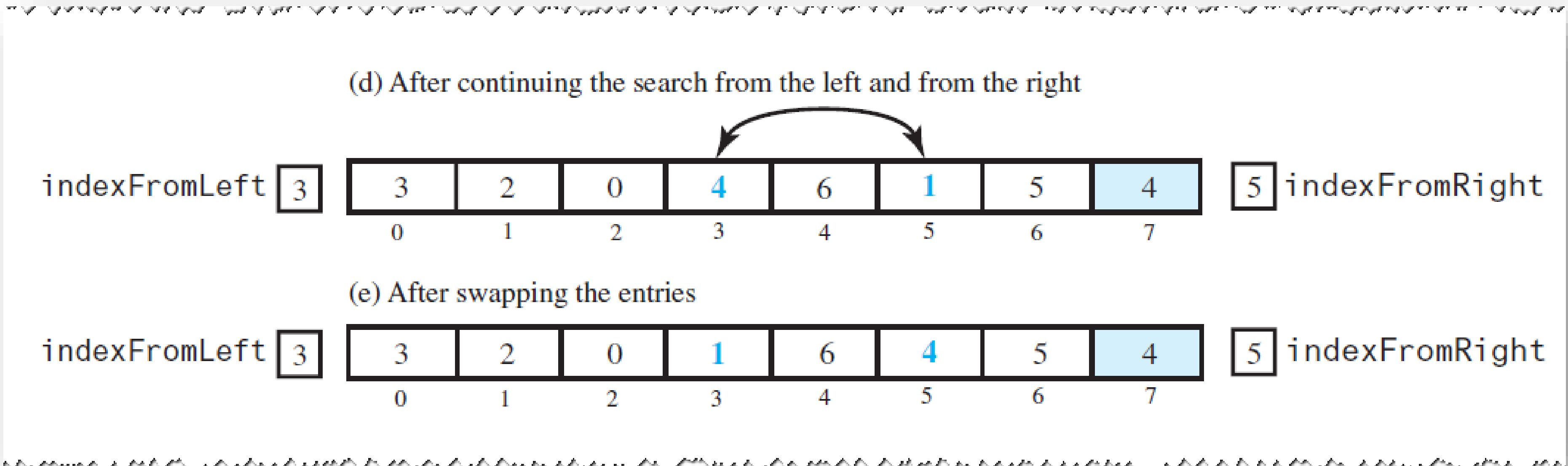
First draft of pseudocode for the quick sort algorithm

The Quick Sort



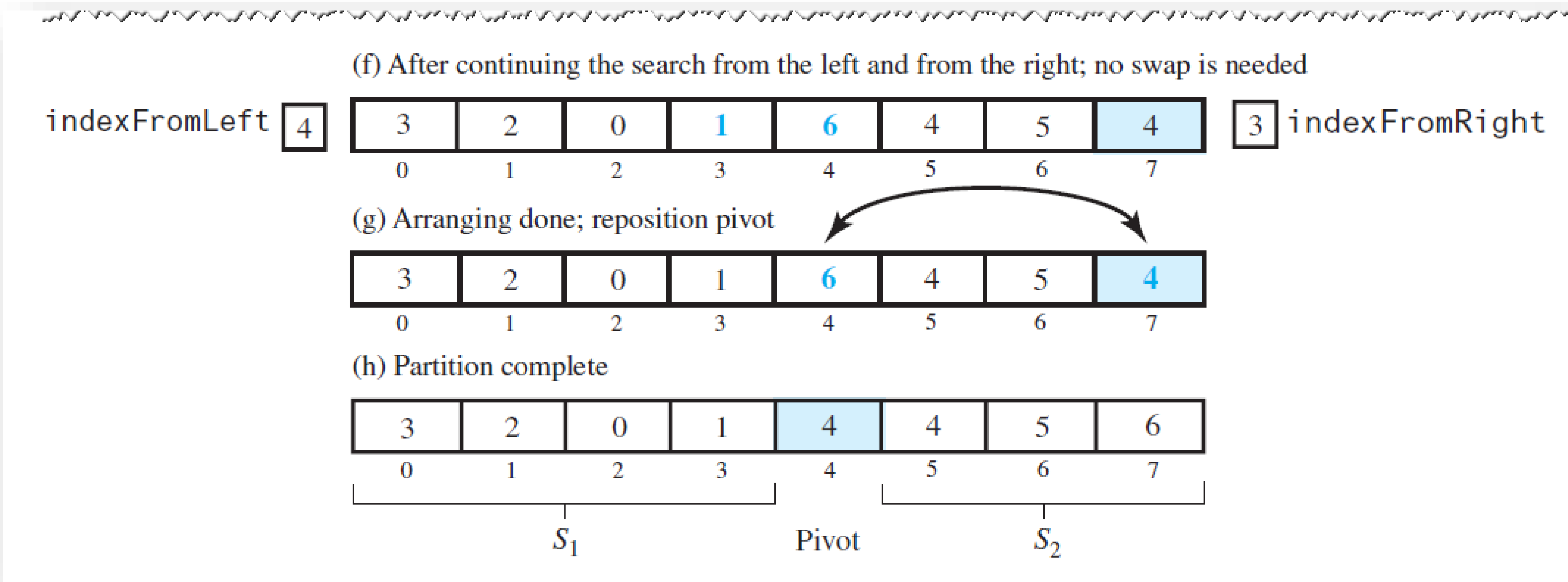
- A partitioning of an array during a quick sort

The Quick Sort



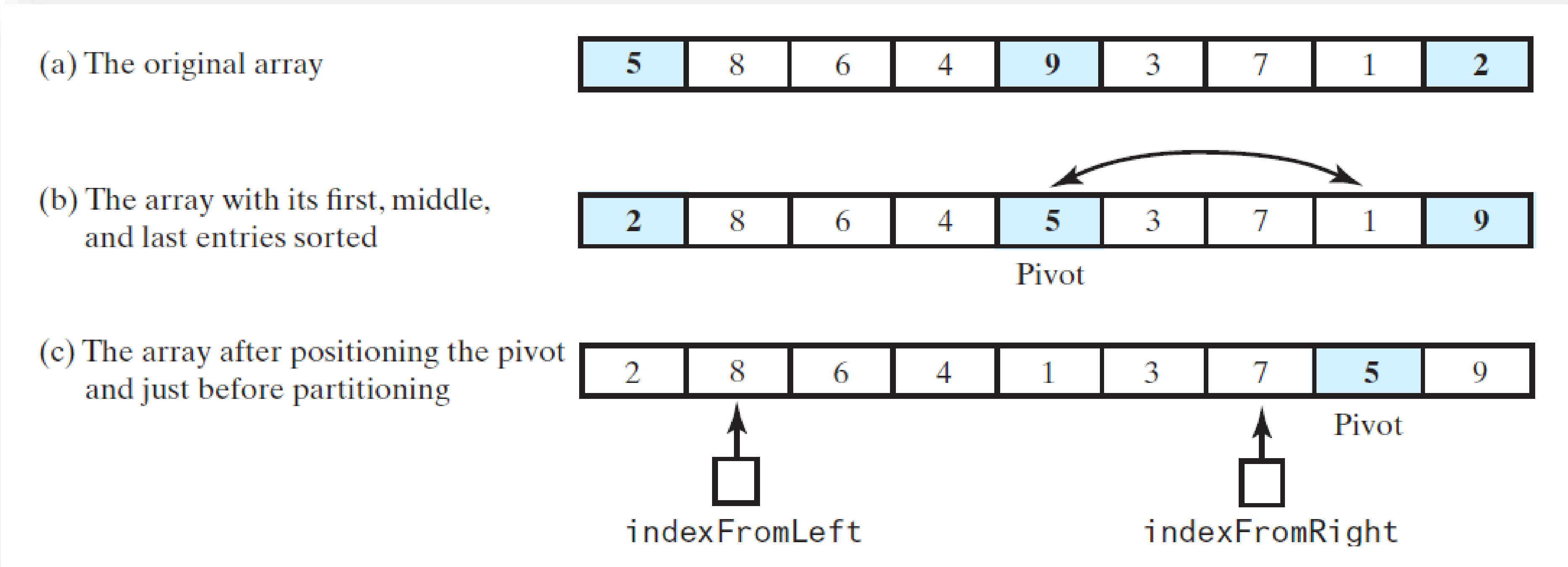
- A partitioning of an array during a quick sort

The Quick Sort



- A partitioning of an array during a quick sort

The Quick Sort



- Median-of-three pivot selection

The Quick Sort

```
// 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]
}
```

- Adjusting the partition algorithm.

The Quick Sort

```
// 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
```

- Pseudocode describes the partitioning algorithm for an array of at least four entries

The Quick Sort

```
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
}
```

- Pseudocode describes the partitioning algorithm for an array of at least four entries

The Quick Sort

```
        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
}
```

- Pseudocode describes the partitioning algorithm for an array of at least four entries

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

- A function that performs a quick sort

The Quick Sort

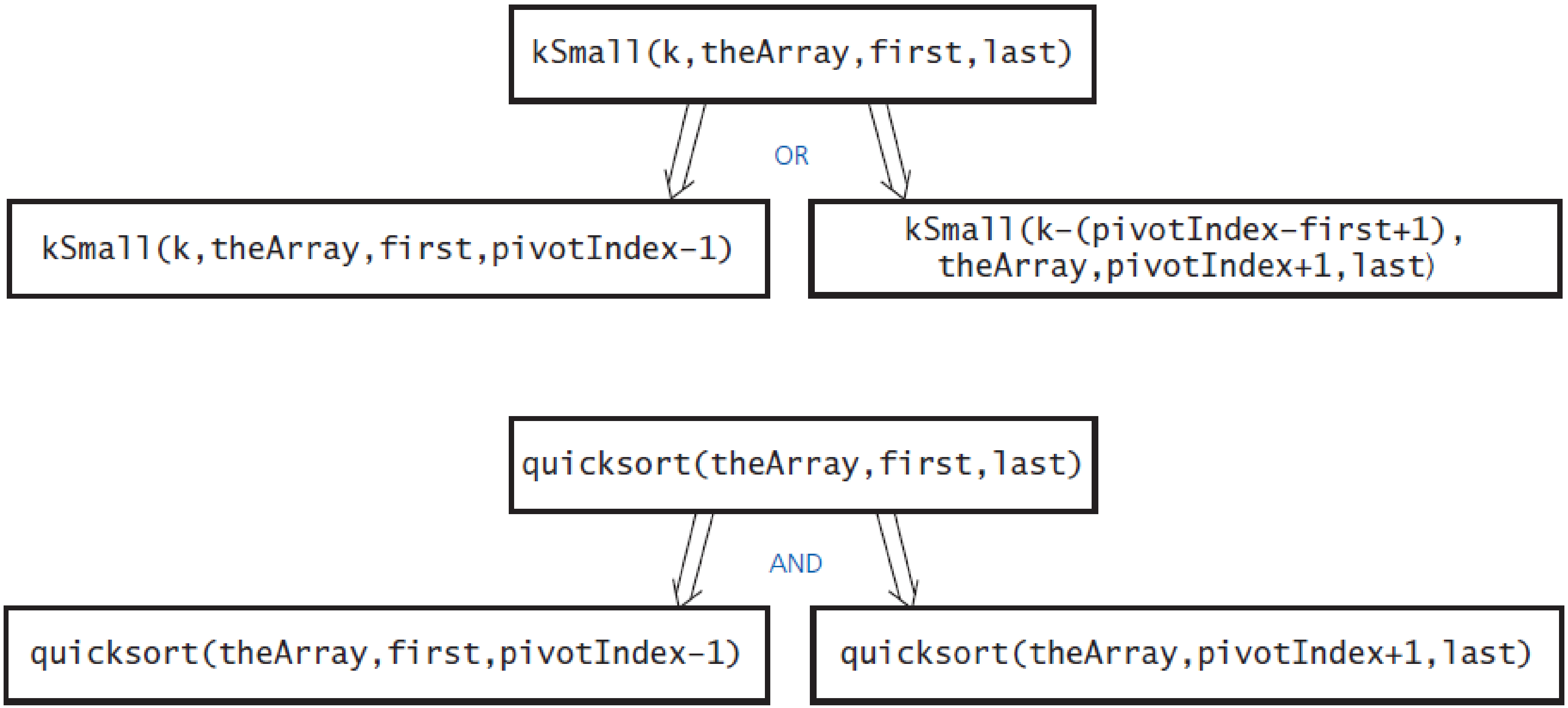
```
    }
    else
    {
       // Create the partition: S1 | Pivot | S2
       int pivotIndex = partition(theArray, first, last);

       // Sort subarrays S1 and S2
       quickSort(theArray, first, pivotIndex - 1);
       quickSort(theArray, pivotIndex + 1, last);
    }  // end if
 }  // end quickSort
```

- A function that performs a quick sort

The Quick Sort

- Analysis
  - Partitioning is an O(n) task
  - There are either $\log_2 n$ or $1 + \log_2 n$ levels of recursive calls to quickSort
- We conclude
  - Worst case O(n^2)
  - Average case O($n \log n$)

The Quick Sort



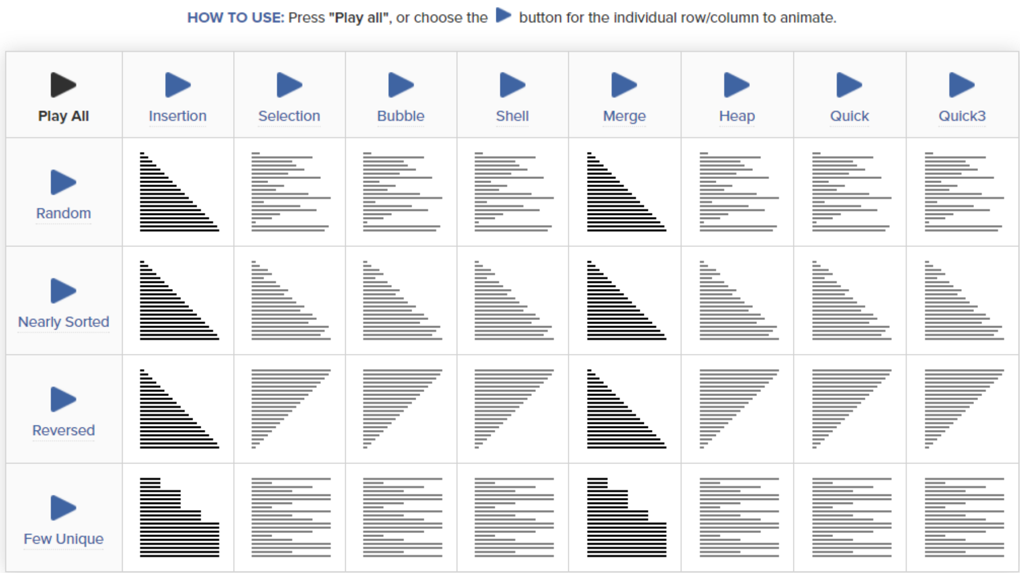
kSmall versus quickSort

$\Theta(n\lg n)$ Quick Sort vs Merge Sort $\Theta(n\lg n)$

- A common question *(in Google, Apple, and Amazon interviews)*
  - Despite of better worst case performance of merge sort, ***quicksort*** *is considered better than merge sort*.
  - *Auxiliary Space*
    - Merge sort uses extra space, quicksort requires little space and exhibits good cache locality.
  - *Worst Cases*
    - The worst case of quicksort $O(n^2)$ can be avoided by using randomized quicksort.

- Merge sort is better for large data structures
  - easily adaptable to data structures
  - Great on slow-to-access media (i.e. disk storage or network attached storage)

Sorting Algorithms Animations



https://www.toptal.com/developers/sorting-algorithms

https://www.youtube.com/watch?v=ZZuD6iUe3Pc&t=69s

Coming Up Next

- Liner Time Sorting algorithms