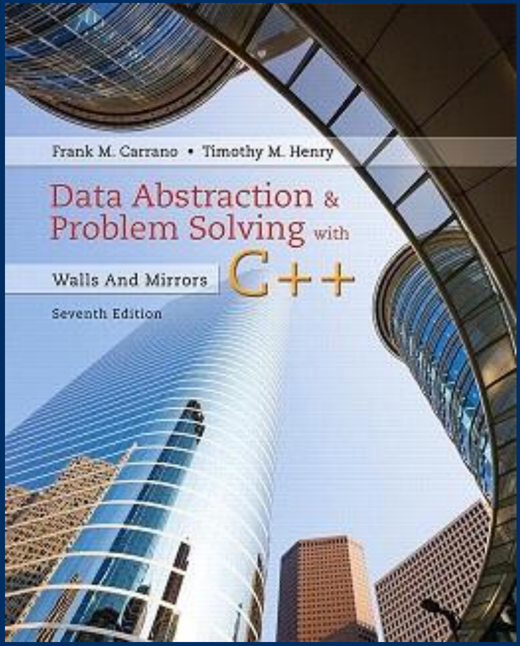
Chapter 11

Sorting Algorithms and their Efficiency

CS 302 - Data Structures

M. Abdullah Canbaz



Modified from authors' slides

Reminders

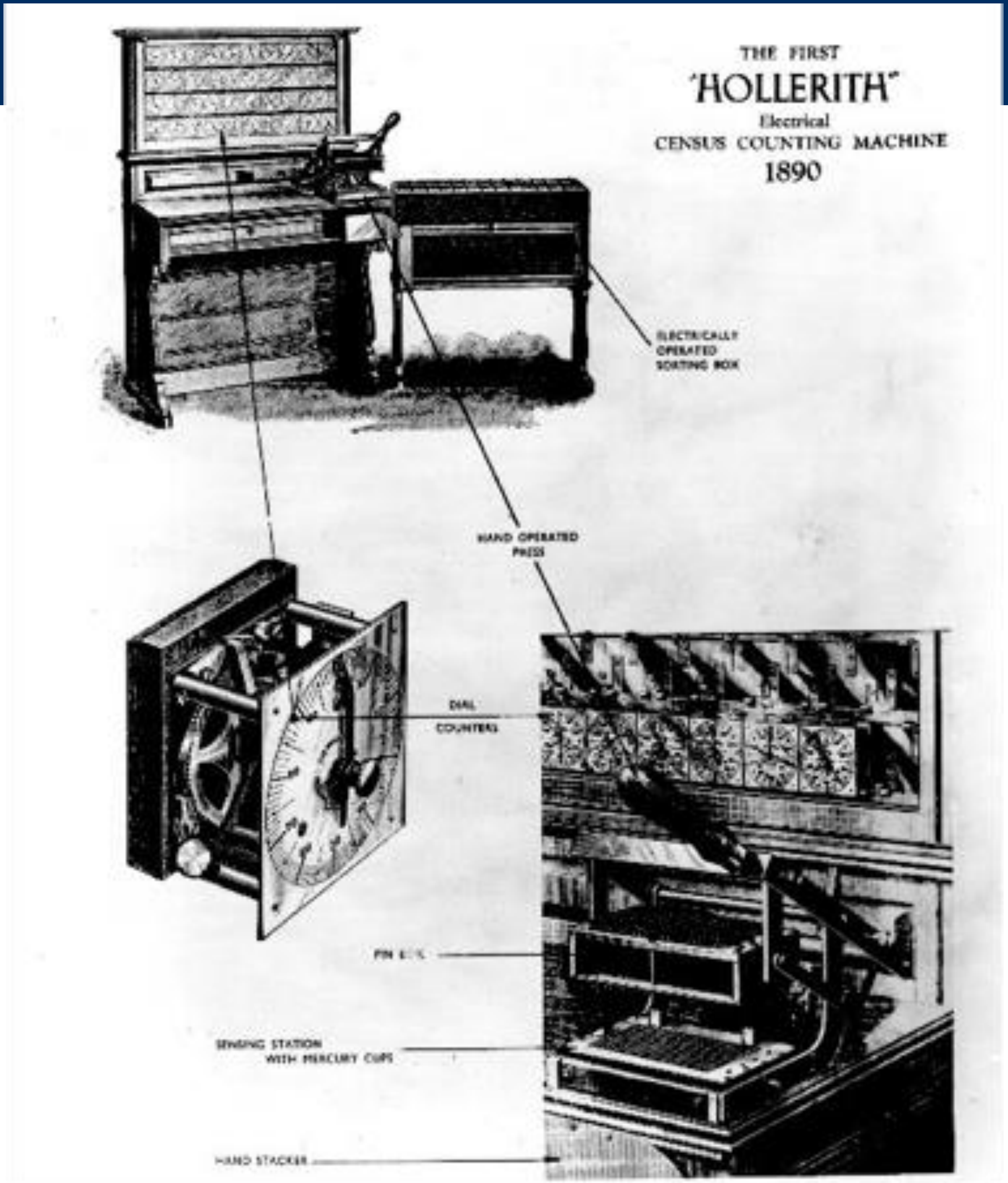
- Assignment 3 is available
  - Due Mar 5th at 2pm
- TA
  - Athanasia Katsila,
    Email: *akatsila [at] nevada {dot} unr {dot} edu*,
    Office Hours: Thursdays, 10:30 am - 12:30 pm at SEM 211
- Quiz 5 due today
  - between 4pm to 11:59pm

Herman Hollerith (1860-1929)

- The 1880 U.S. Census took almost 10 years to process.
- While a lecturer at MIT, Hollerith prototyped punched-card technology.
- His machines, including a "card sorter," allowed the 1890 census total to be reported in 6 weeks.
- He founded the Tabulating Machine Company in 1911, which merged with other companies in 1924 to form International **B**usiness **M**achines.

Hollerith's Tabulating System

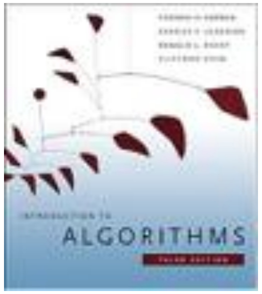
- Pantograph card punch
- Hand-press reader
- Dial counters
- Sorting box



A great reference

Introduction to Algorithms

book by Charles E. Leiserson, Clifford Stein, Ronald Rivest, and Thomas H. Cormen



Sorting Algorithms

- Selection Sort
- Bubble Sort
- Recursive Bubble Sort
- Insertion Sort
- Recursive Insertion Sort
- Merge Sort
- Iterative Merge Sort
- Quick Sort
- Iterative Quick Sort
- Heap Sort
- Counting Sort
- Radix Sort
- Bucket Sort
- ShellSort
- TimSort
- Comb Sort
- Pigeonhole Sort
- Cycle Sort
- Cocktail Sort
- Bitonic Sort
- Pancake sorting
- Binary Insertion Sort
- Permutation Sort
- Gnome Sort
- Sleep Sort
- Structure Sorting
- Stooge Sort
- Tag Sort
- Tree Sort
- Cartesian Tree Sorting
- Odd-Even Sort / Brick Sort
- 3-Way QuickSort
- 3-way Merge Sort

Sorting Algorithms Visualized

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

Sorting means . . .

- Sorting rearranges the elements into either ascending or descending order within the array.
  - We'll use ascending order.

- The values stored in an array have keys of a type for which the relational operators are defined.
  - We also assume unique keys.

36	6
24	10
10	12
6	24
12	36

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

Selection Sort

Straight Selection Sort

values	[0]	36
	[1]	24
	[2]	10
	[3]	6
	[4]	12

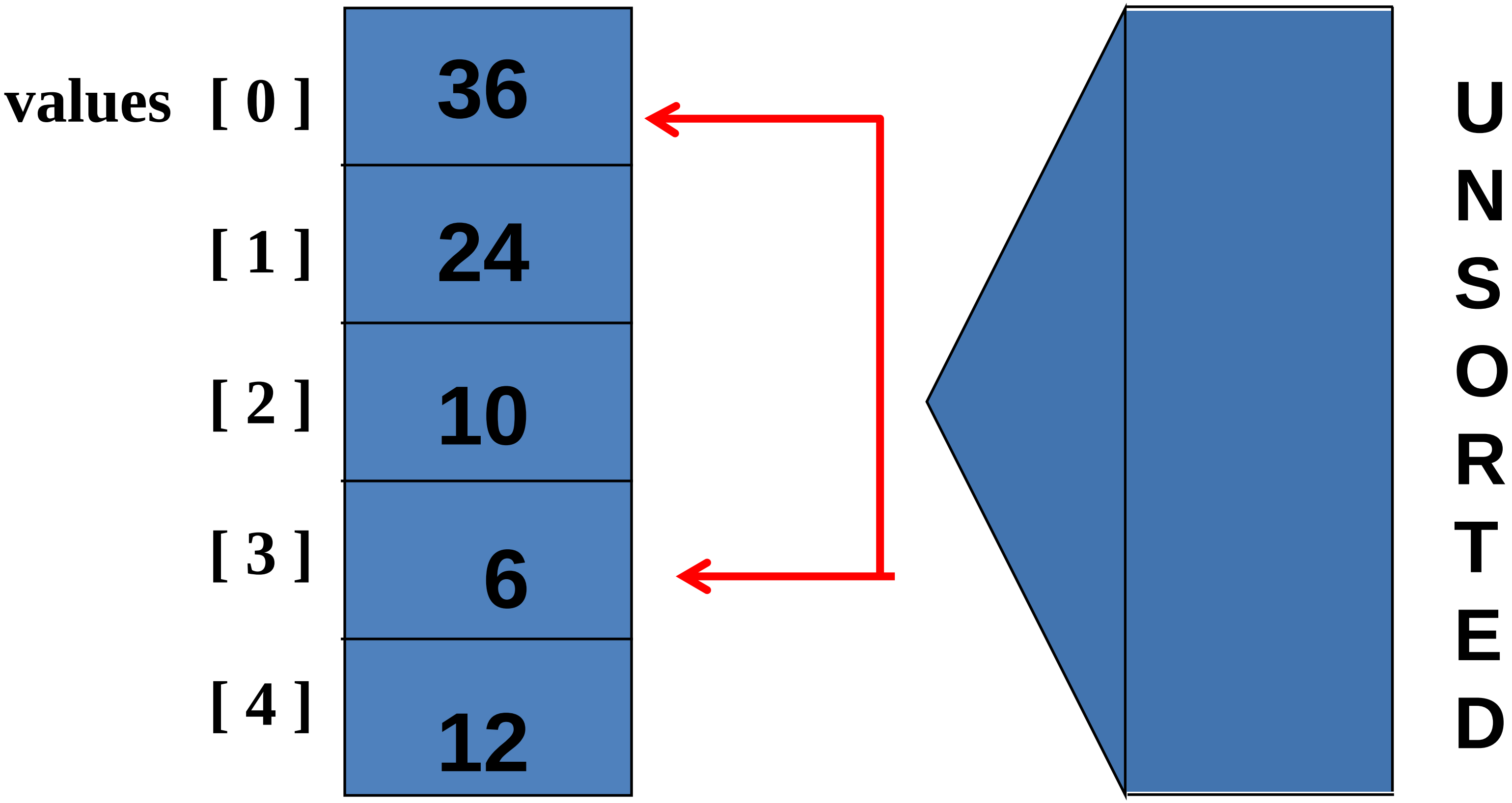
Divides the array into two parts:

already sorted, and
not yet sorted.

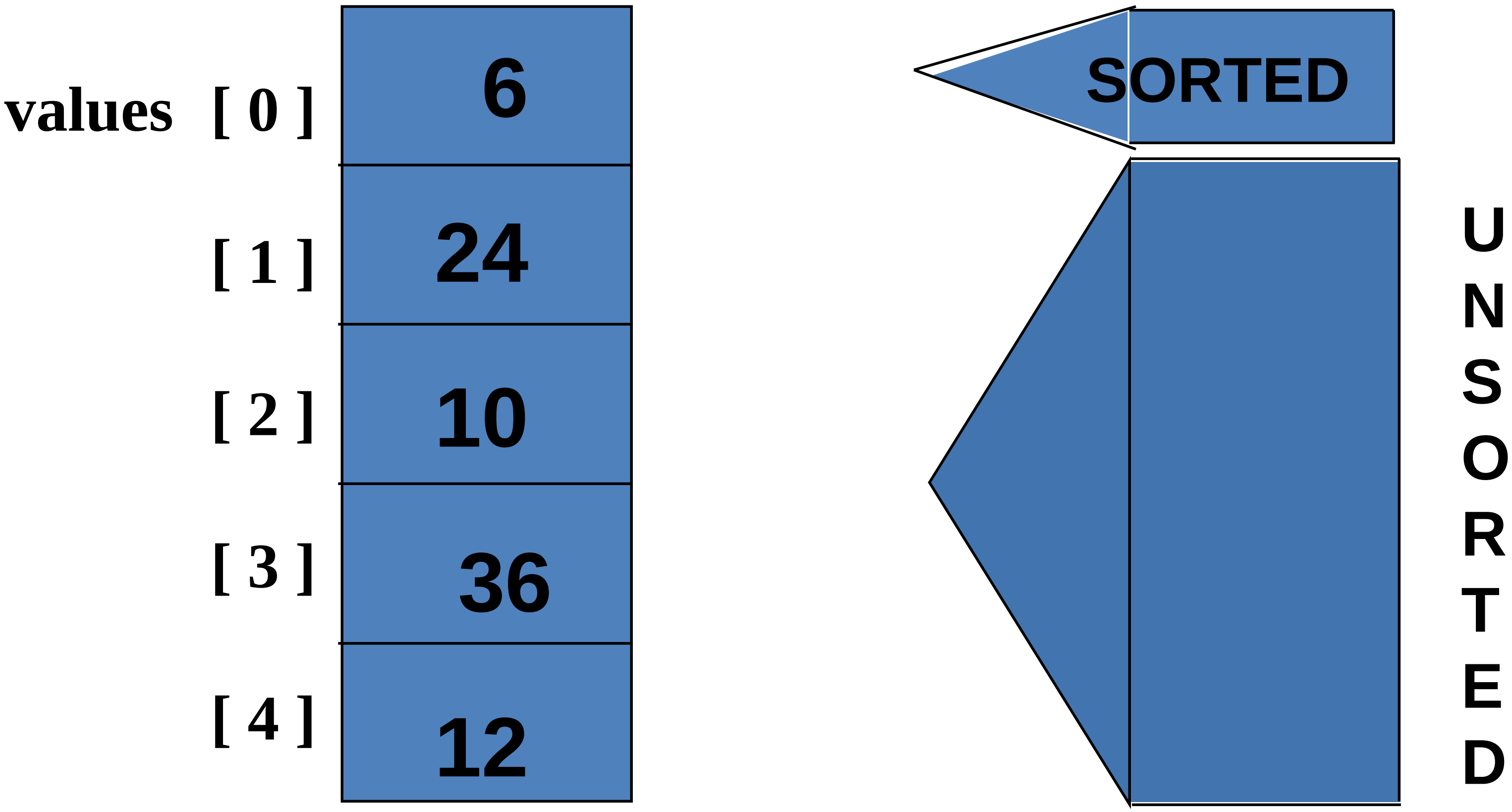
On each pass,

- **finds the smallest of the unsorted elements, and**
- **swaps it into its correct place,**
- **thereby increasing the number of sorted elements by one.**

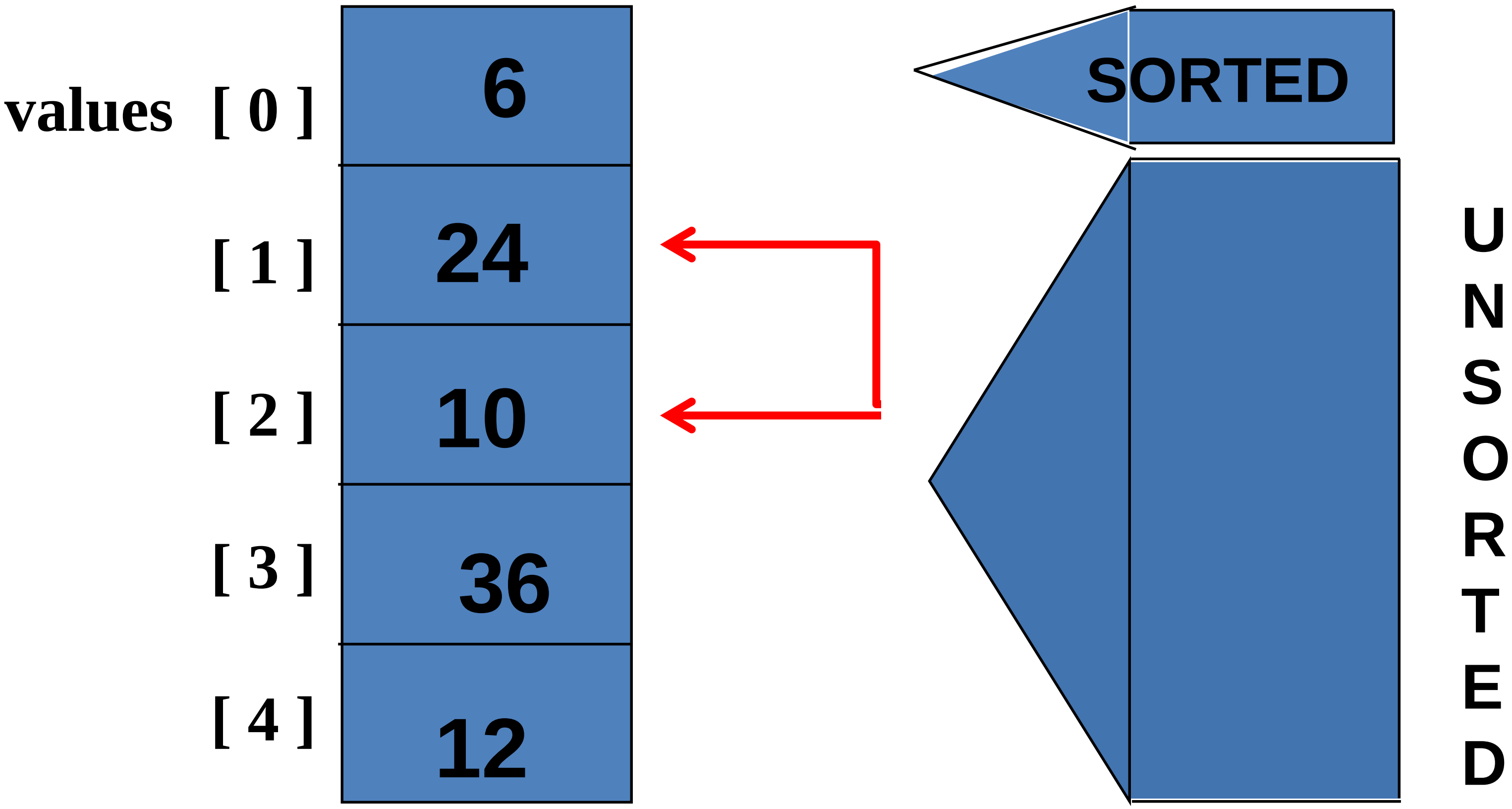
Selection Sort: Pass One



Selection Sort: End Pass One

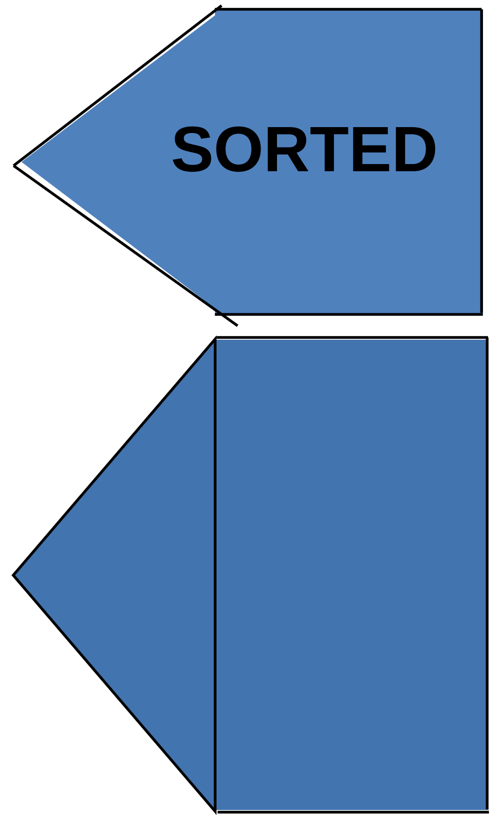


Selection Sort: Pass Two



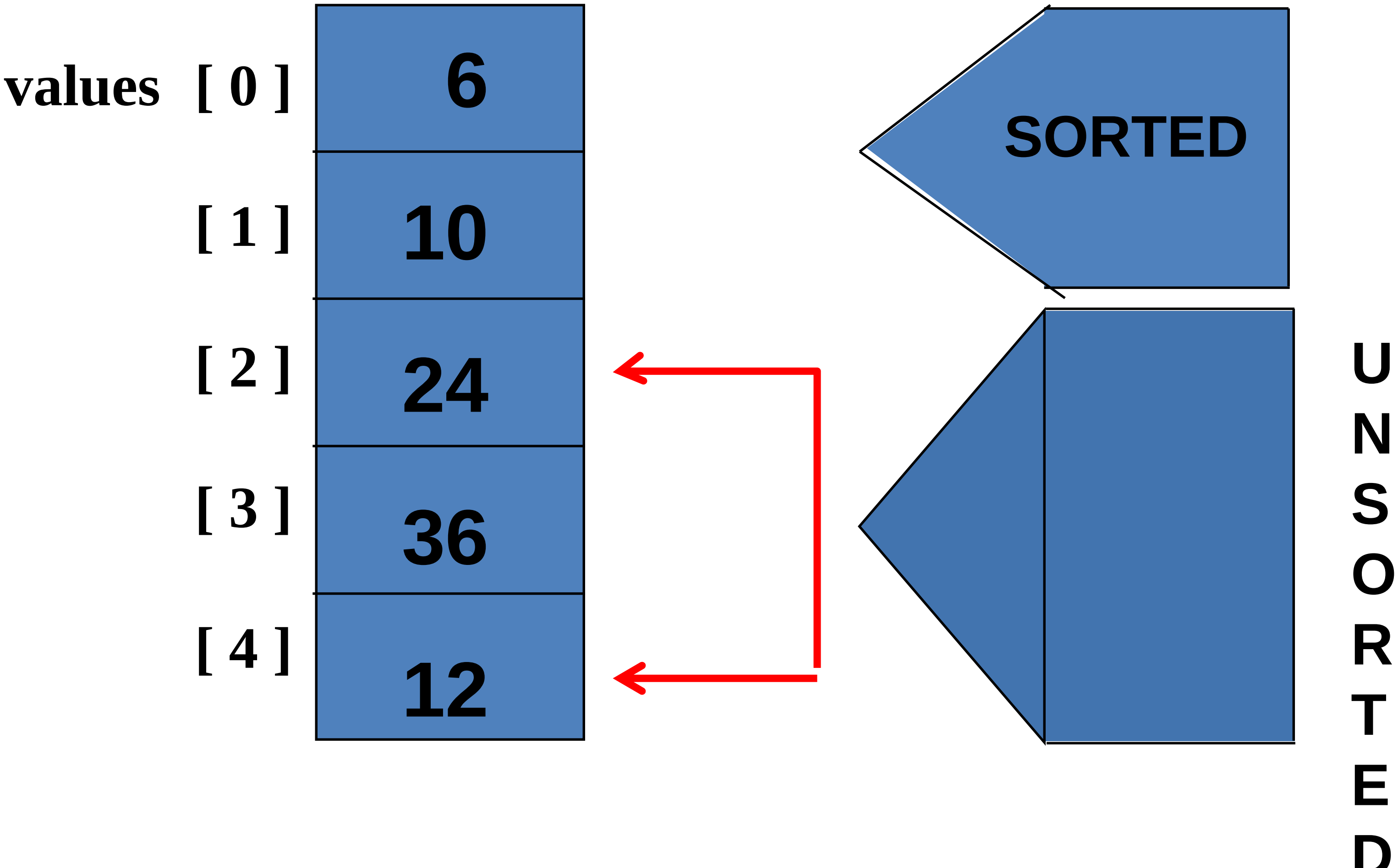
Selection Sort: End Pass Two

values	
[0]	6
[1]	10
[2]	24
[3]	36
[4]	12



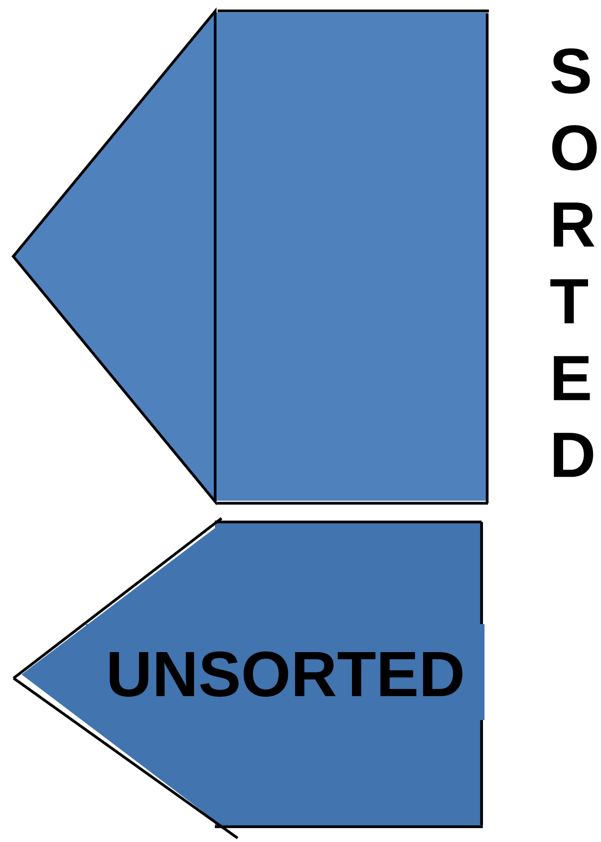
UNSORTED

Selection Sort: Pass Three

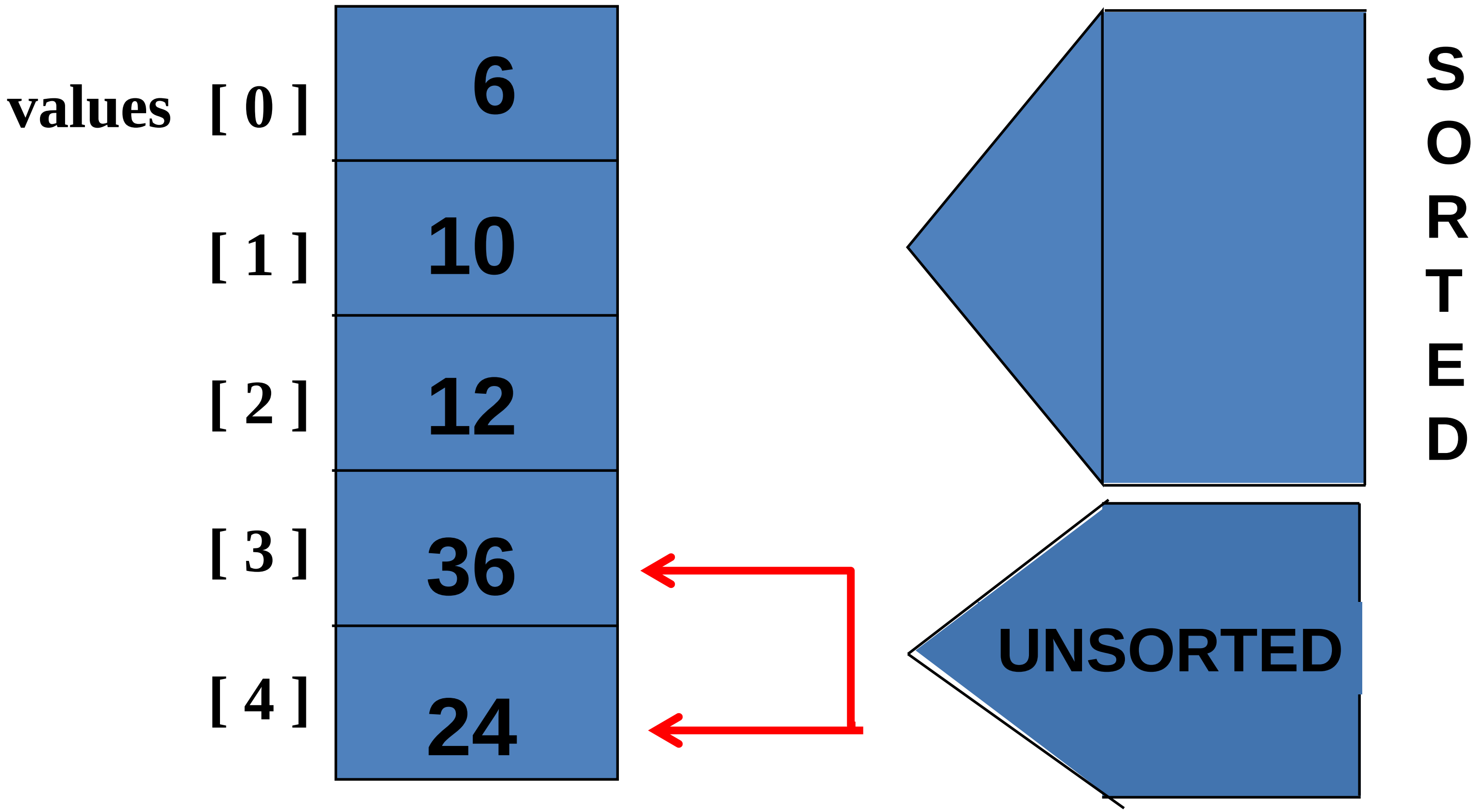


Selection Sort: End Pass Three

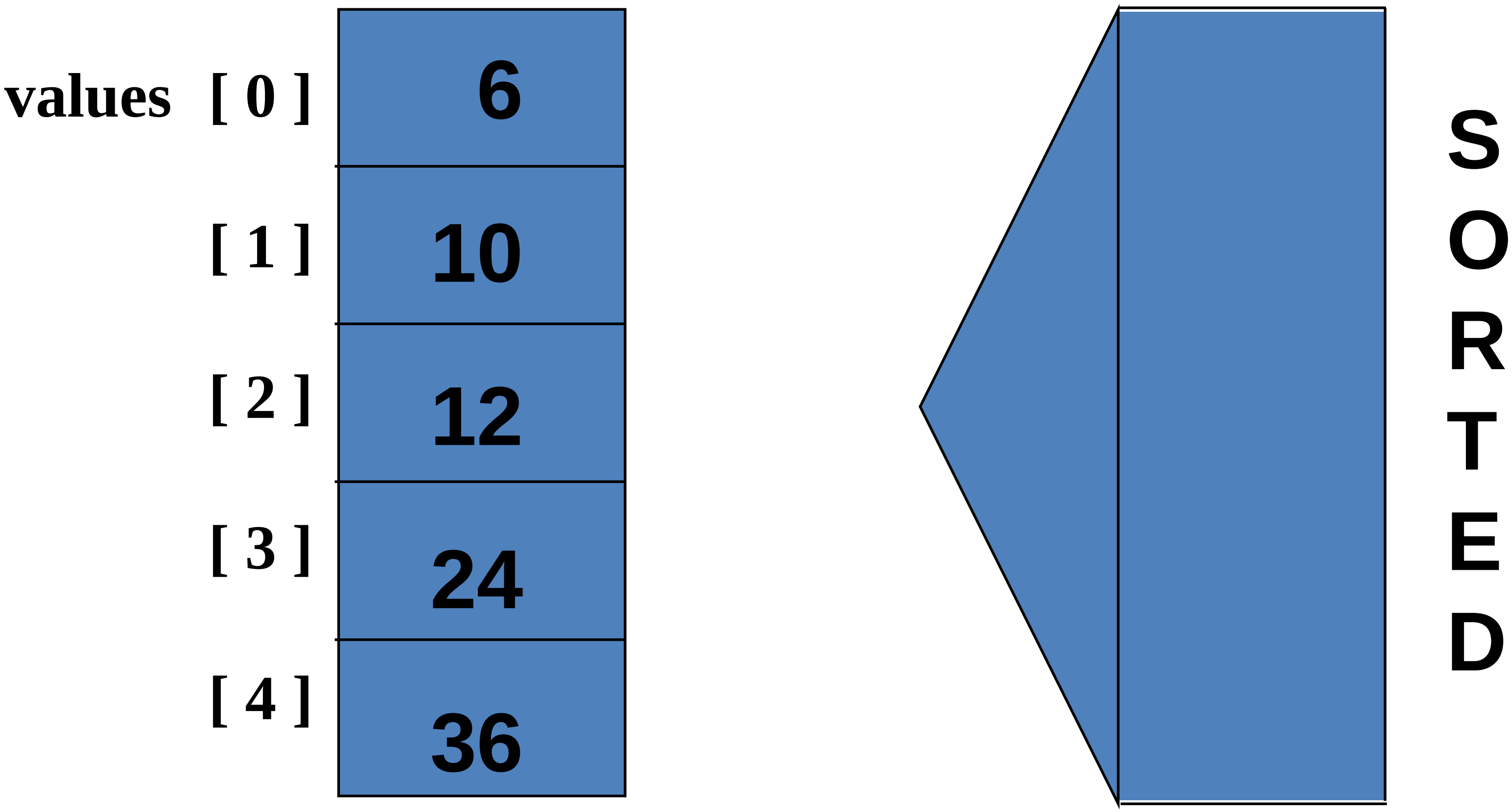
values	[0]	6
	[1]	10
	[2]	12
	[3]	36
	[4]	24



Selection Sort: Pass Four



Selection Sort: End Pass Four



```
template <class  ItemType >
int  MinIndex(ItemType values [ ], int  start, int end)
//  Post: Function value = index of the smallest value
//  in values [start]  . . values [end].
{
  int  indexOfMin = start ;

  for(int index = start + 1 ; index <= end ; index++)
    if  (values[ index] < values [indexOfMin])
       indexOfMin = index ;

  return   indexOfMin;

}
```

```
template <class  ItemType >
void  SelectionSort (ItemType values[ ],
  int  numValues )

// Post: Sorts array values[0 . . numValues-1 ]
// into ascending order by key
{
  int  endIndex = numValues - 1;

  for (int current=0; current<endIndex; current++)


    Swap (values[current],
      values[MinIndex(values, current, endIndex)]);

}
```

Selection Sort: How many comparisons?

values	[0]	6
	[1]	10
	[2]	12
	[3]	24
	[4]	36

4 compares for values[0]

3 compares for values[1]

2 compares for values[2]

---

1 compare for values[3]

= 4 + 3 + 2 + 1

For selection sort in general

- **The number of comparisons when the array contains N elements is**

Sum = (N-1) + (N-2) + . . . + 2 + 1

Arithmetic series:

$$Sum = \sum_{i=1}^{N-1} i = \frac{(N-1)N}{2} \qquad \mathbf{O(N^2)}$$

The Selection Sort

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**

- A selection sort of an array of five integers

The Selection Sort

```
/** 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);

/** 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
```

The Selection Sort

```
    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].
        // 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>
```

- An implementation of the selection sort

The Selection Sort

```
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
```

- An implementation of the selection sort

The Selection Sort

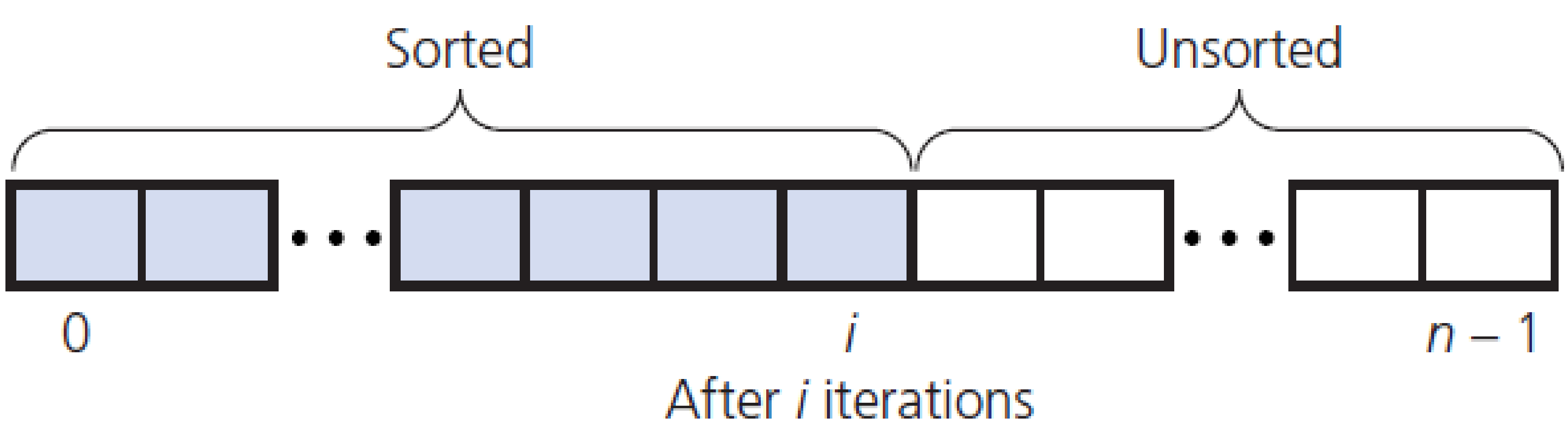
- Analysis
  - Selection sort is $O(n^2)$
  - Appropriate only for small *n,*
  - $O(n^2)$ grows rapidly

- Could be a good choice when
  - Data moves are costly,
  - But comparisons are not

Insertion Sort

The Insertion Sort

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



An insertion sort partitions the array into two regions

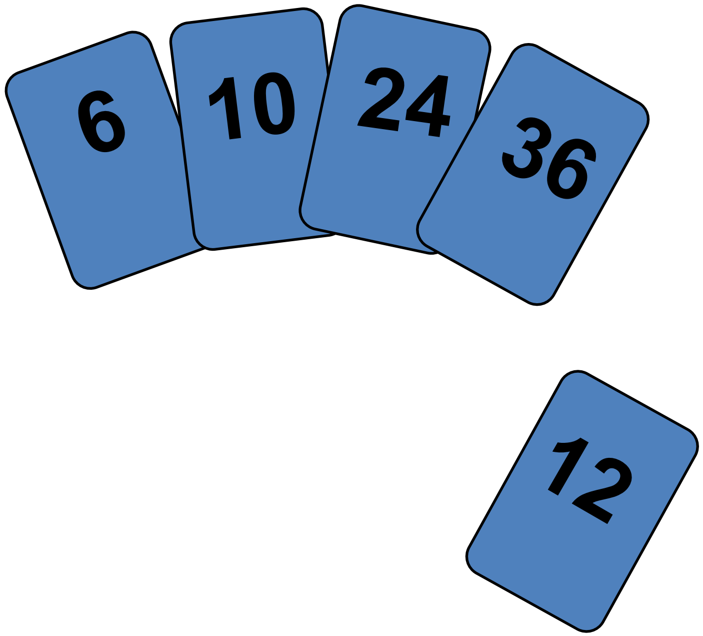
Insertion Sort

values	[0]	36
	[1]	24
	[2]	10
	[3]	6
	[4]	12

One by one, each as yet unsorted array element is inserted into its proper place with respect to the already sorted elements.

On each pass, this causes the number of already sorted elements to increase by one.

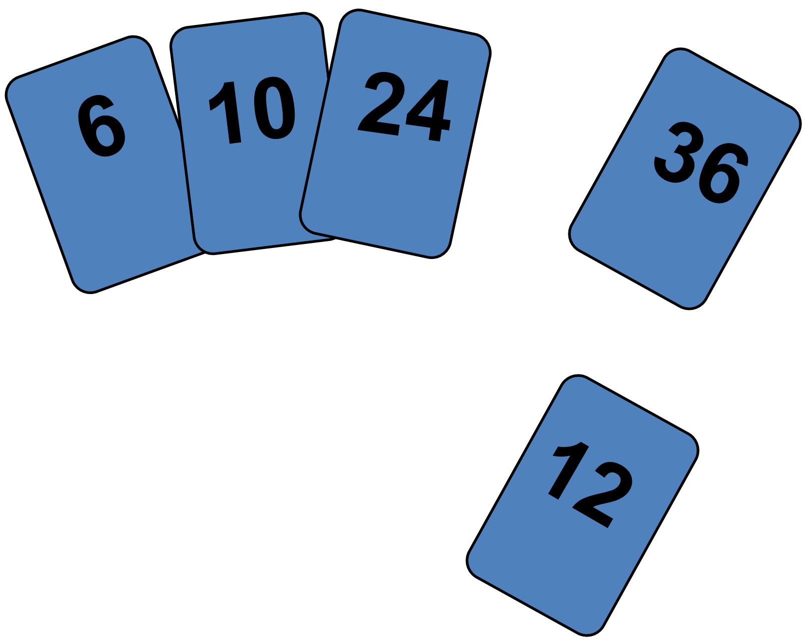
Insertion Sort



Works like someone who "inserts" one more card at a time into a hand of cards that are already sorted.

To insert 12,
we need to make room for it by moving first 36 and
then 24.

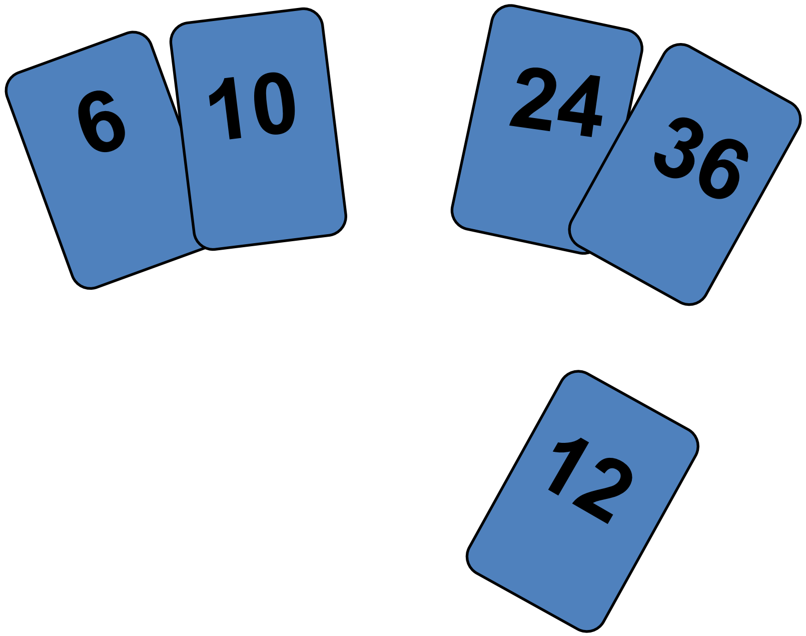
Insertion Sort



Works like someone who "inserts" one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.

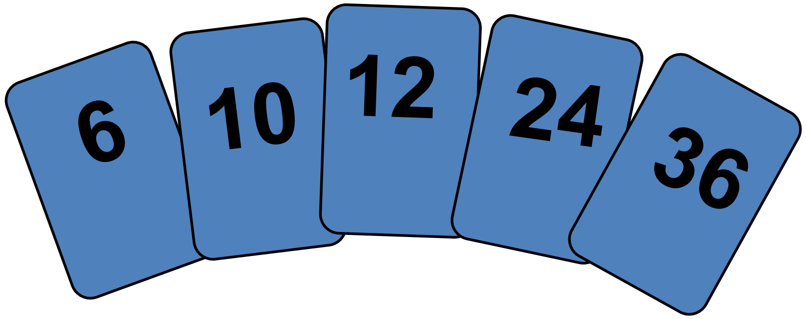
Insertion Sort



Works like someone who “inserts” one more card at a time into a hand of cards that are already sorted.

**To insert 12,
we need to make room for it by moving first 36 and
then 24.**

Insertion Sort



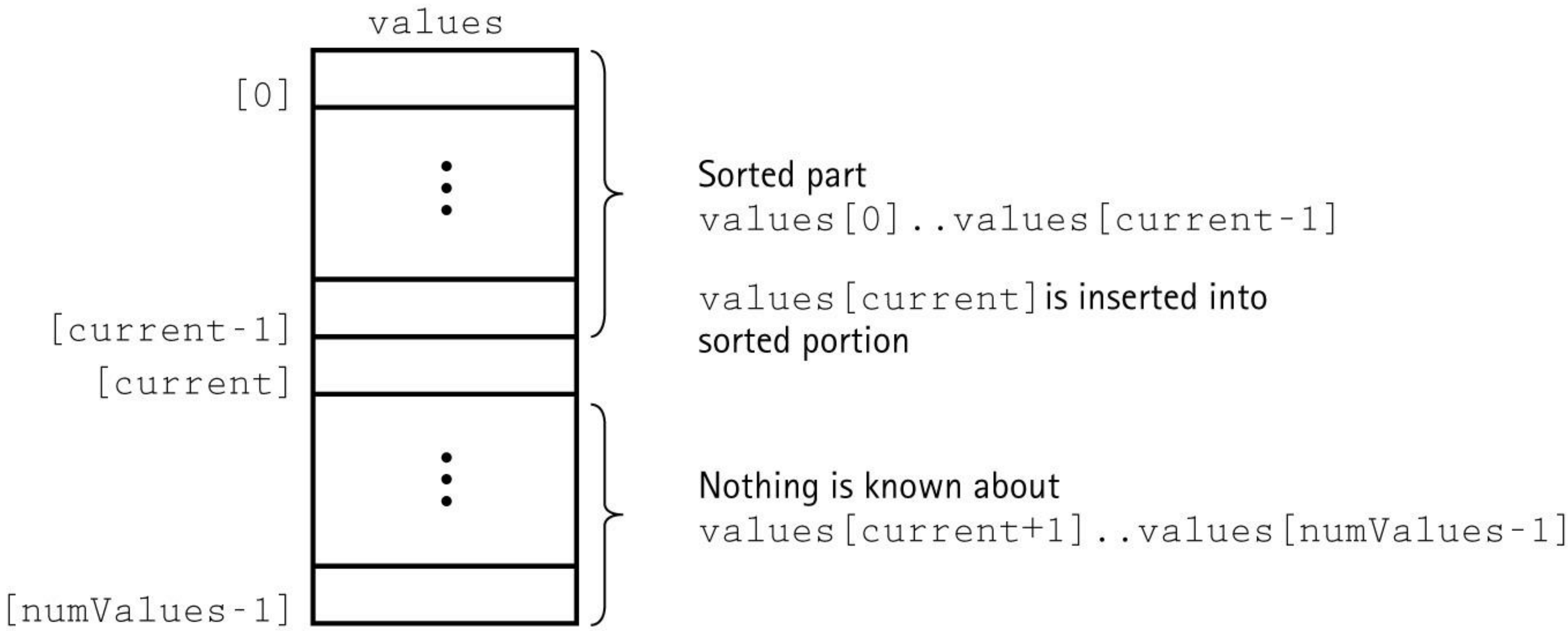
Works like someone who "inserts" one more card at a time into a hand of cards that are already sorted.

To insert 12,
we need to make room for it by moving first 36 and
then 24.

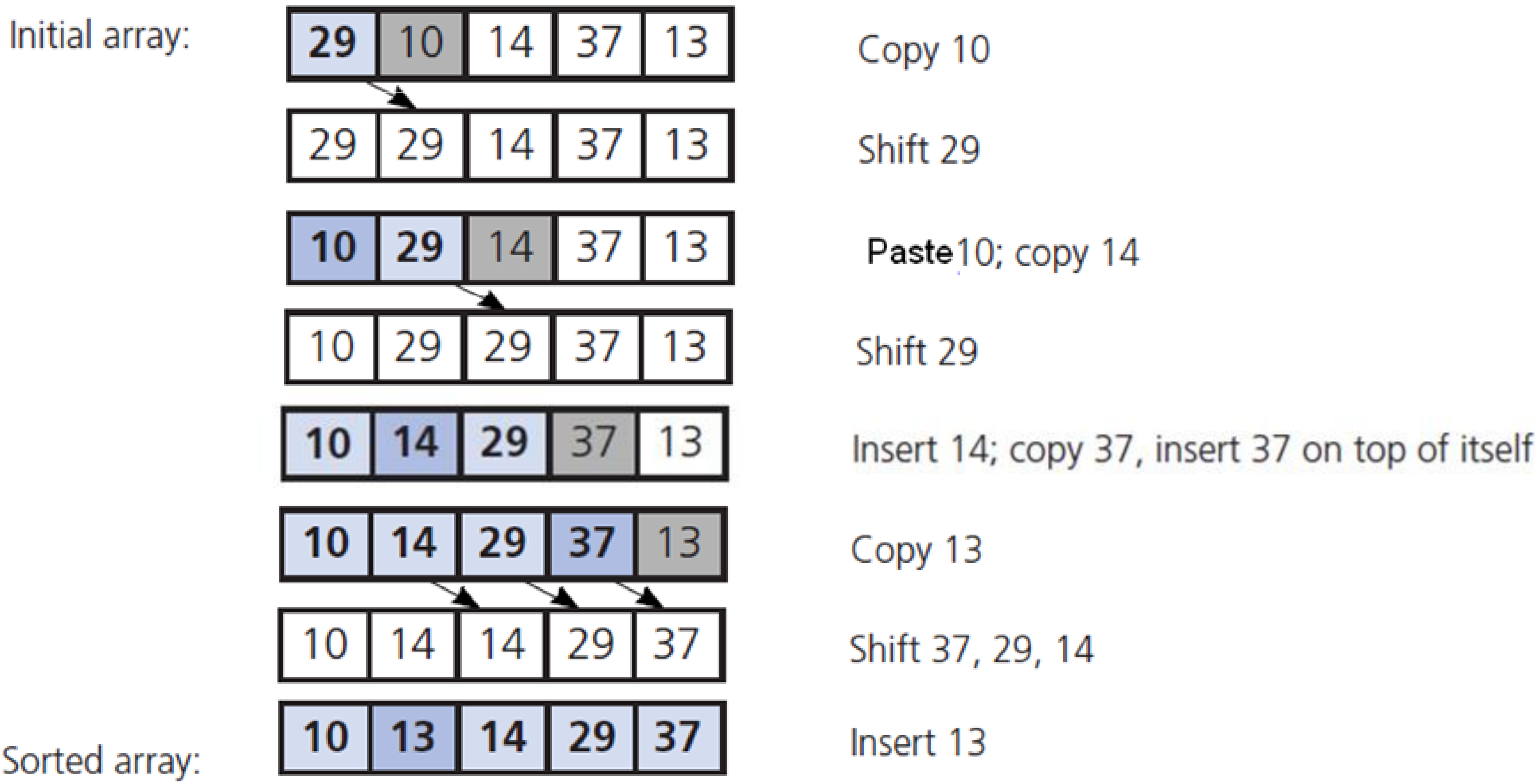
```
template <class  ItemType >
void  InsertionSort ( ItemType  values[], int numValues)


//  Post: Sorts array values[0 . . numValues-1 ] into
//   ascending order by key
{
   for (int count = 0 ; count < numValues; count++)

     InsertItem ( values , 0 , count );
}
```

```
template <class  ItemType >
void InsertItem (ItemType  values[], int  start, int end)

//  Post: Elements between values[start] and  values
//   [end] have been sorted into ascending order by key.
{
   bool  finished = false ;
   int   current  =  end ;
   bool  moreToSearch = (current != start);

   while (moreToSearch  &&  !finished )
   {
     if  (values[current] < values[current - 1])
     {
         Swap(values[current], values[current - 1);
        current--;
        moreToSearch = ( current != start );
     }
     else
        finished = true;
   }
}
```

A Snapshot of the Insertion Sort Algorithm



The Insertion Sort



- An insertion sort of an array of five integers

The Insertion Sort

```
/** 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]
```

- An implementation of the insertion sort

The Insertion Sort

```
   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 nextItem into sorted region
   }  // end for
}  // end insertionSort
```

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
  - Unless already sorted

Buble Sort

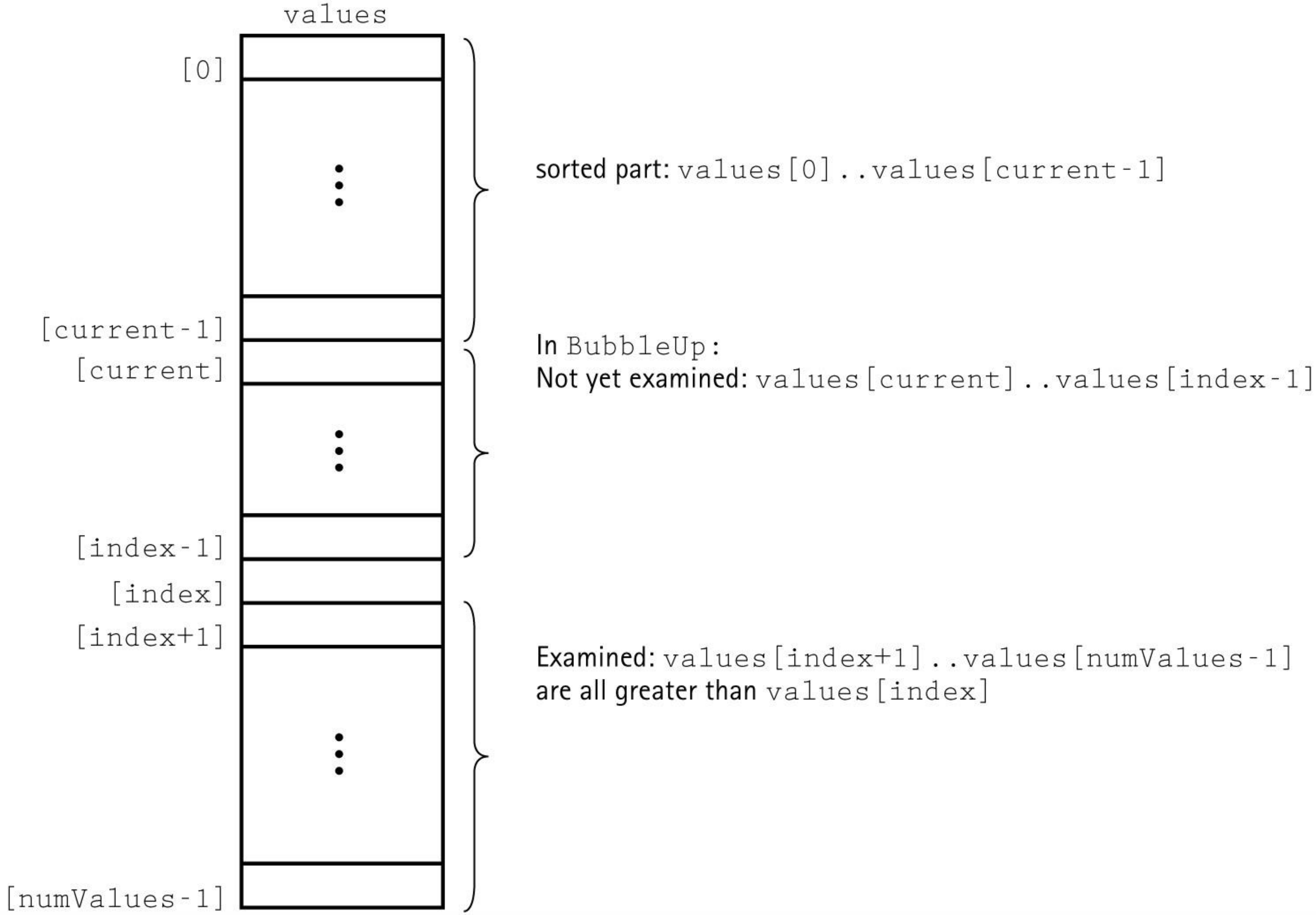
Bubble Sort

36
24
10
6
12

Compares neighboring pairs of array elements,
- starting with the last array element, and
- swaps neighbors whenever they are not in correct order.

On each pass, this causes the smallest element to "bubble up" to its correct place in the array.

Snapshot of BubbleSort



Code for BubbleSort

```
template<class ItemType>
void BubbleSort(ItemType values[], int numValues)
{
  int current = 0;
  while (current < numValues-1)
  {
    BubbleUp(values, current, numValues-1);
    current++;
  }
}
```

Code for BubbleUp

```
template<class ItemType>
void BubbleUp(ItemType values[],
              int startIndex,
              int endIndex)
// Post: Adjacent pairs that are out of
//   order have been switched between
//   values[startIndex]..values[endIndex]
//   beginning at values[endIndex].

{
  for (int index=endIndex; index>startIndex; index--)
    if (values[index] < values[index-1])
      Swap(values[index], values[index-1]);
}
```

Observations on BubbleSort

This algorithm is *always* $O(N^2)$.

There can be a large number of intermediate swaps.

Can this algorithm be improved?

Add a “flag” to exit iterations
if nothing changes in a single iteration

Code for BubbleSort2

```
template<class ItemType>
void BubbleSort2(ItemType values[], int numValues)
{
  int current = 0;
  bool sorted = false;
  while (current < numValues-1 && ! sorted)
  {
    BubbleUp2(values, current, numValues-1, sorted);
    current++;
  }
}
```

Code for BubbleUp2

```
template<class ItemType>
void BubbleUp(ItemType values[],
              int startIndex,
              int endIndex,
              bool& sorted )
// Post: Adjacent pairs that are out of
//   order have been switched between
//   values[startIndex]..values[endIndex]
//   beginning at values[endIndex].
{
  sorted = true;
  for (int index=endIndex; index>startIndex; index--)
    if (values[index] < values[index-1]){
      Swap(values[index], values[index-1]);
      sorted = false;
    }
}
```

BubbleSort2 Complexity

$(N - 1) + (N - 2) + (N - 3) + \ldots + (N - K)$

1st call   2nd call   3rd call   Kth call

$= KN - (\text{sum of 1 through } K)$

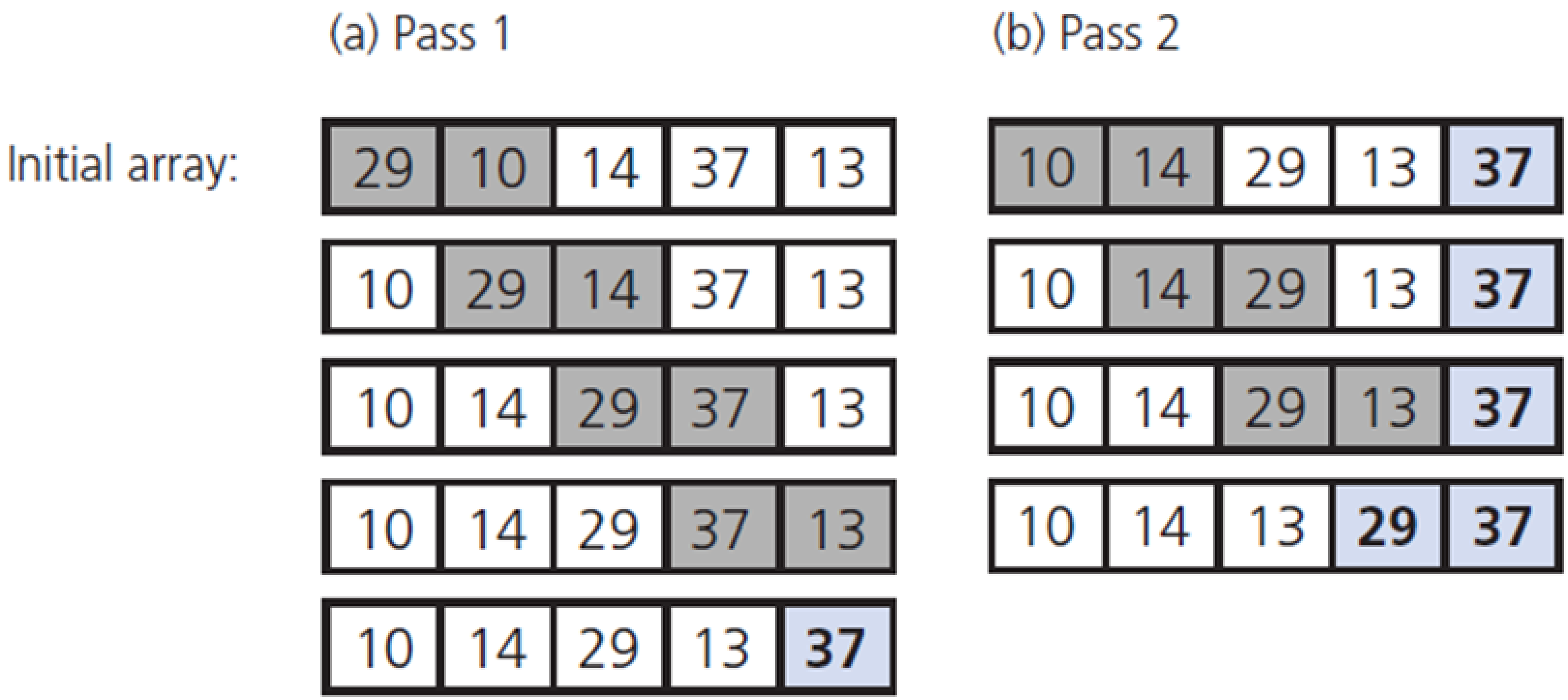
$= KN - K(K+1)/2$

$= (2KN - K^2 - K) / 2$

$O(N^2)$

Good for almost in order items!

The Bubble Sort



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

The Bubble Sort

```
/** 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++)
      {
```

- An implementation of the bubble sort

The Bubble Sort

```cpp
      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
```

- An implementation of the bubble sort

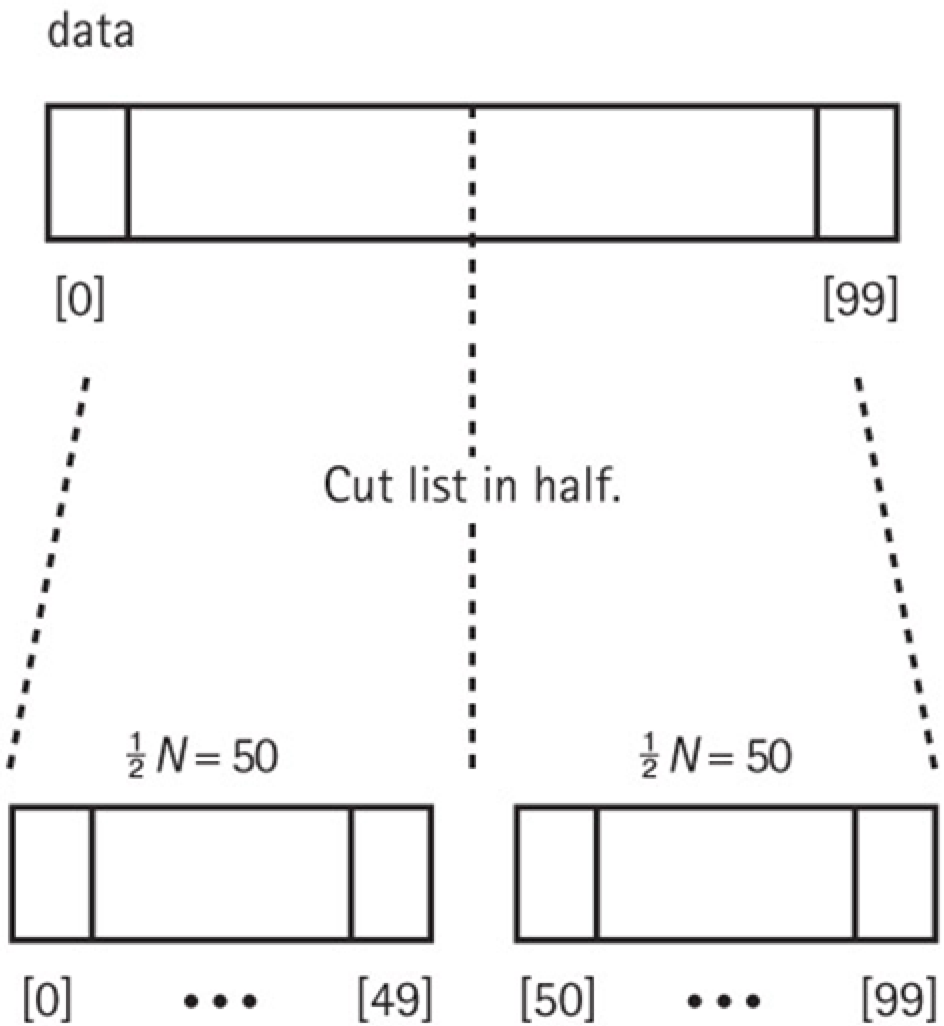
The Bubble Sort

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

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

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

$= 5100$

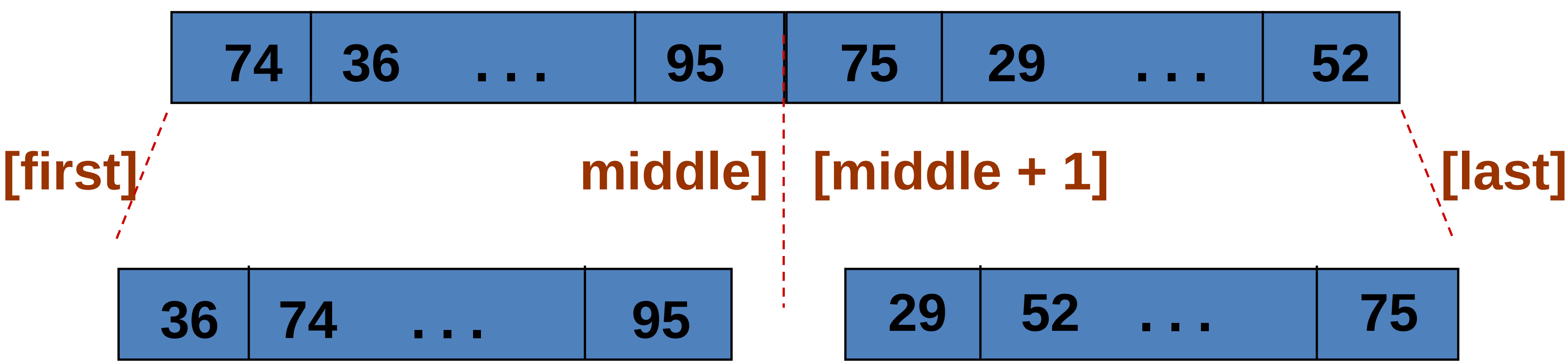
Merge Sort Algorithm

Cut the array in half.

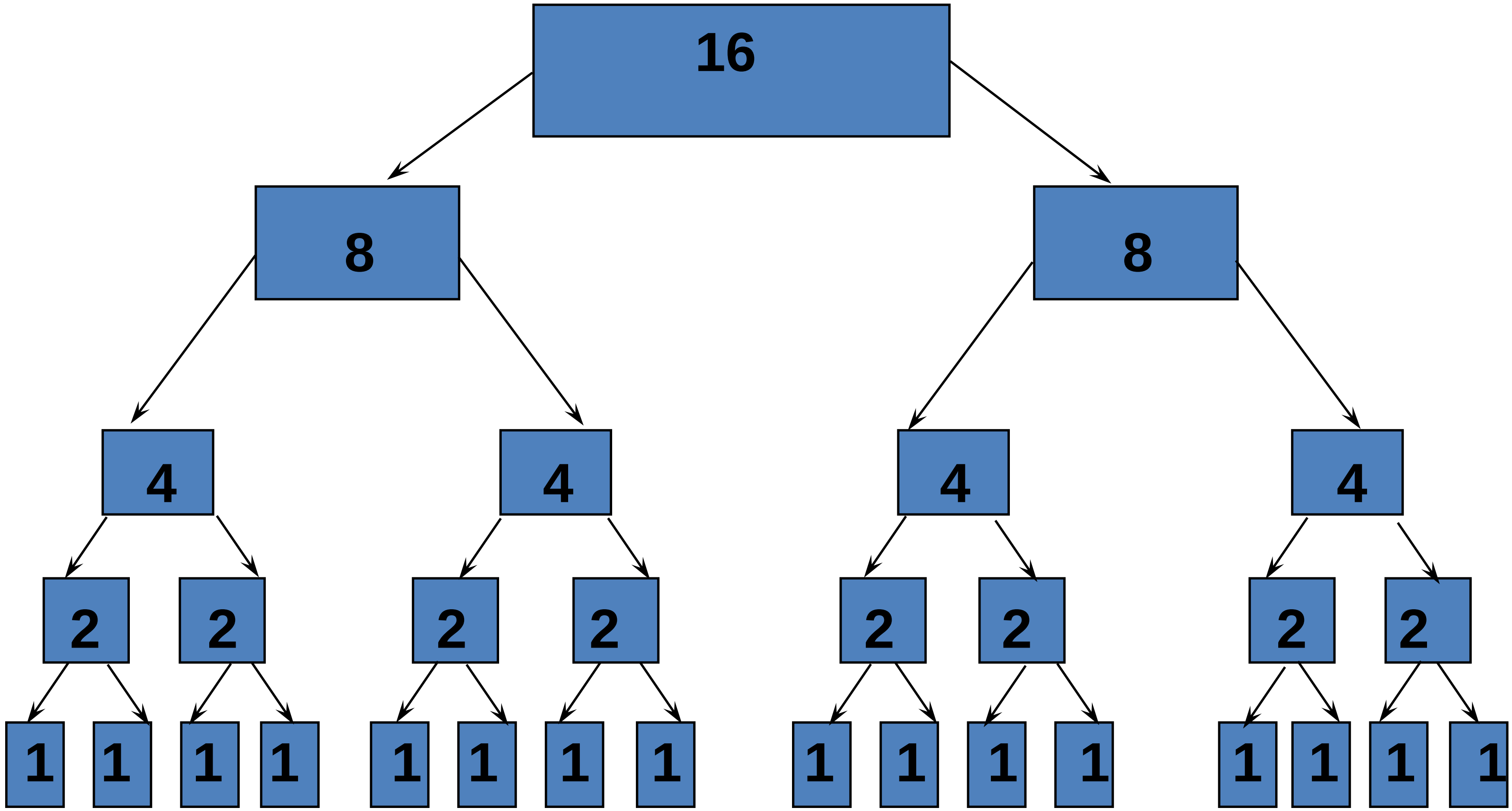
Sort the left half.

Sort the right half.

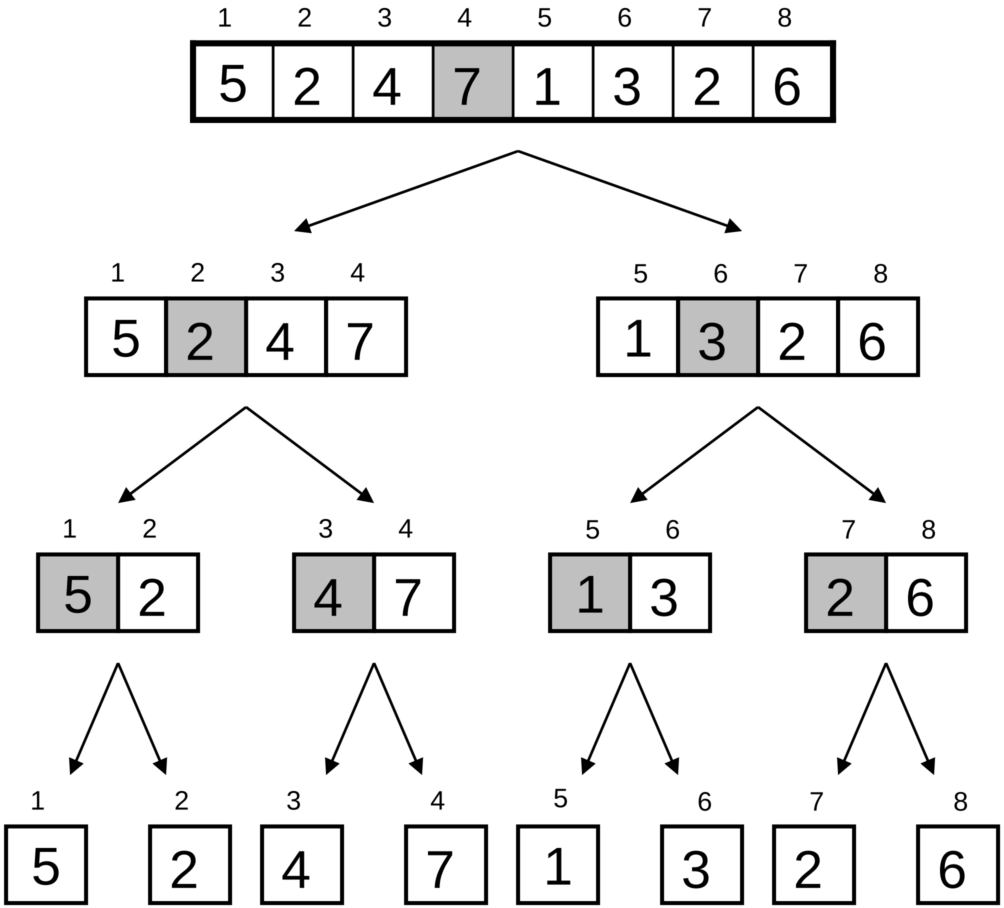
Merge the two sorted halves into one sorted array.



Using Merge Sort Algorithm

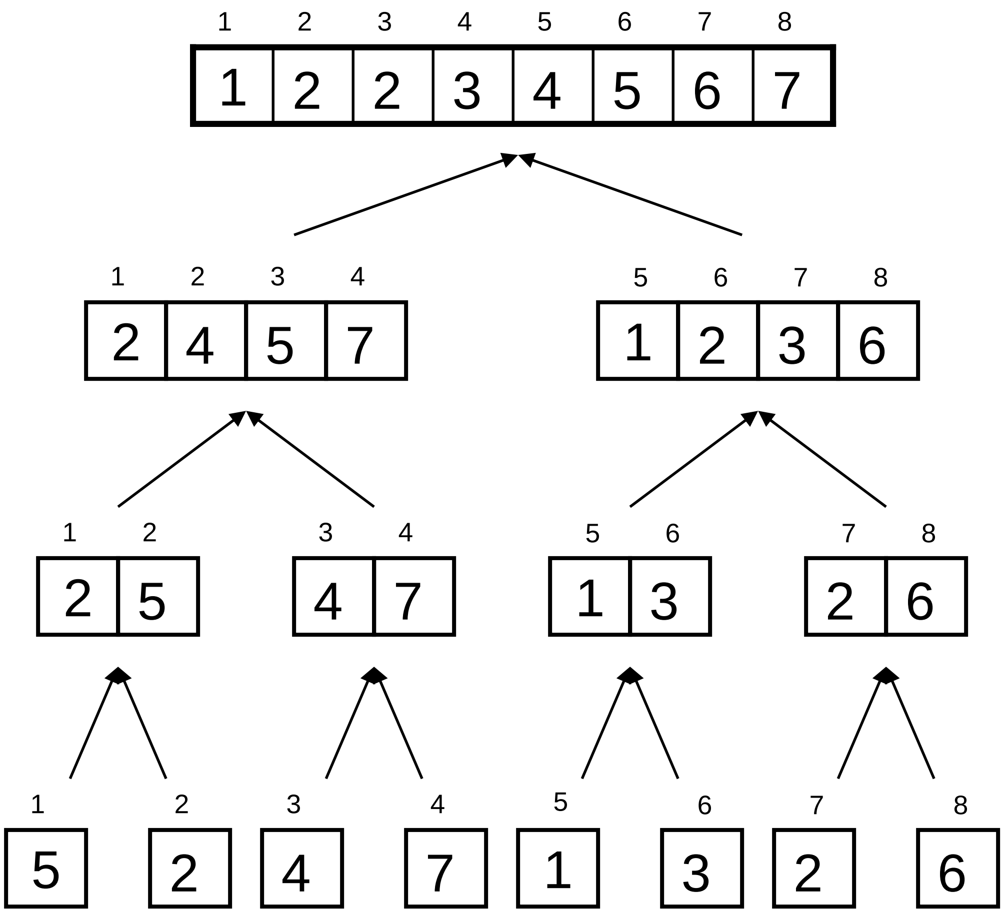


Example



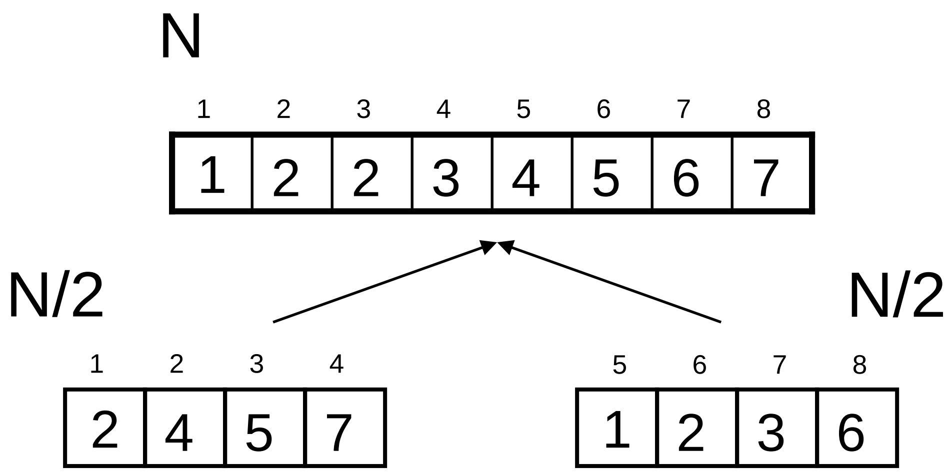
middle = 4

Example (cont.)



Merge Step

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



```
// 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

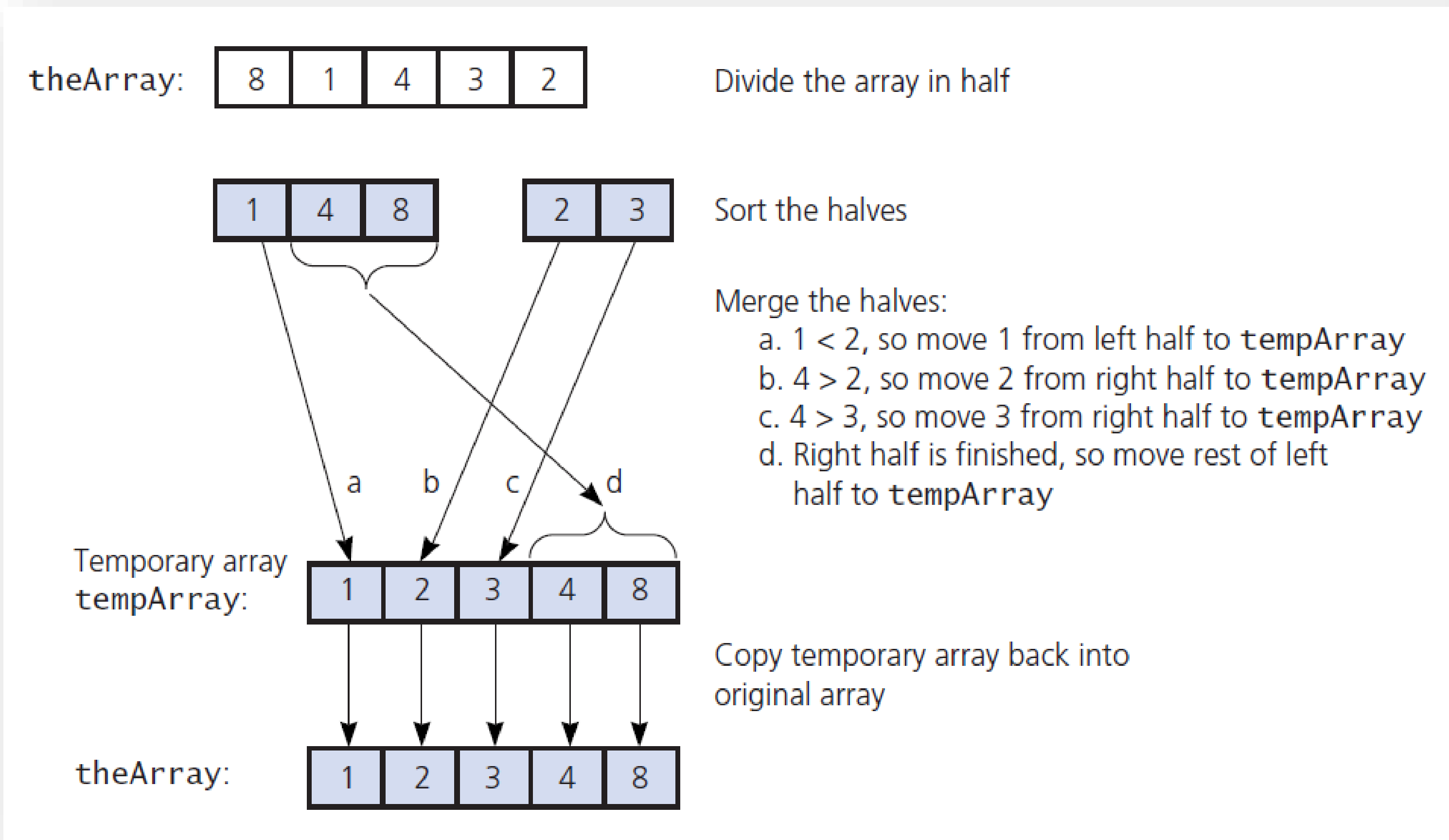
Each time it is subdivided, function Merge is called to re-combine the halves

- 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



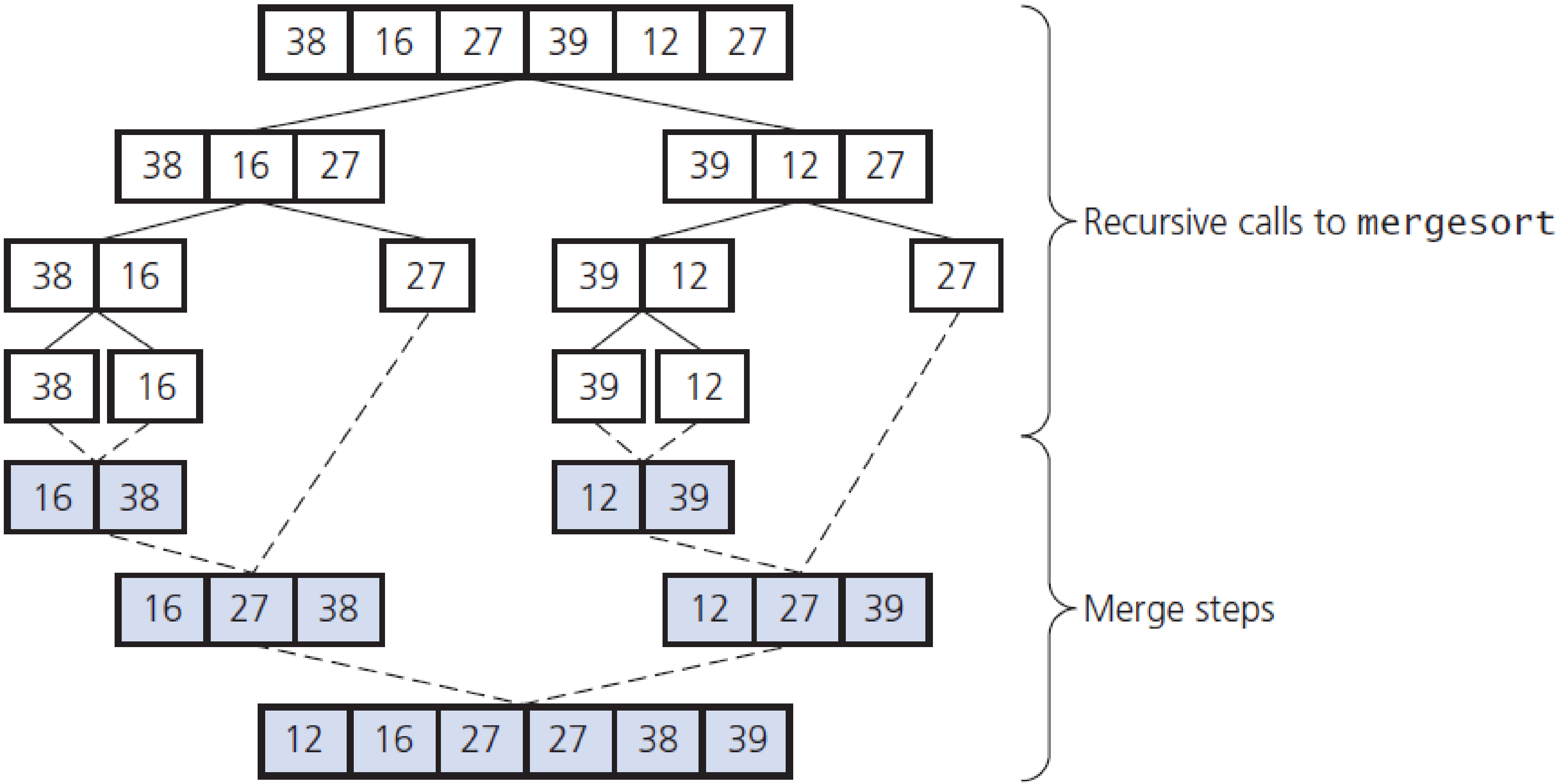
- A merge sort with an auxiliary temporary array

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)
```

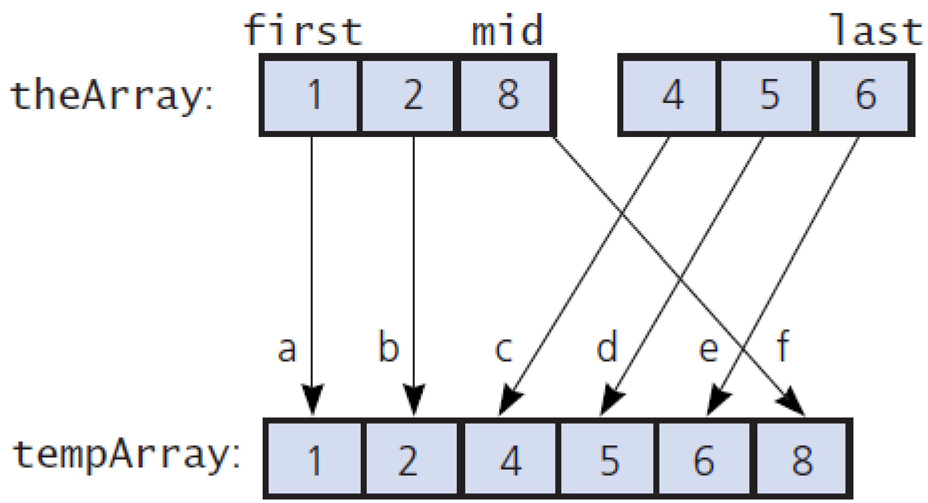
- An implementation of merge and mergeSort

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

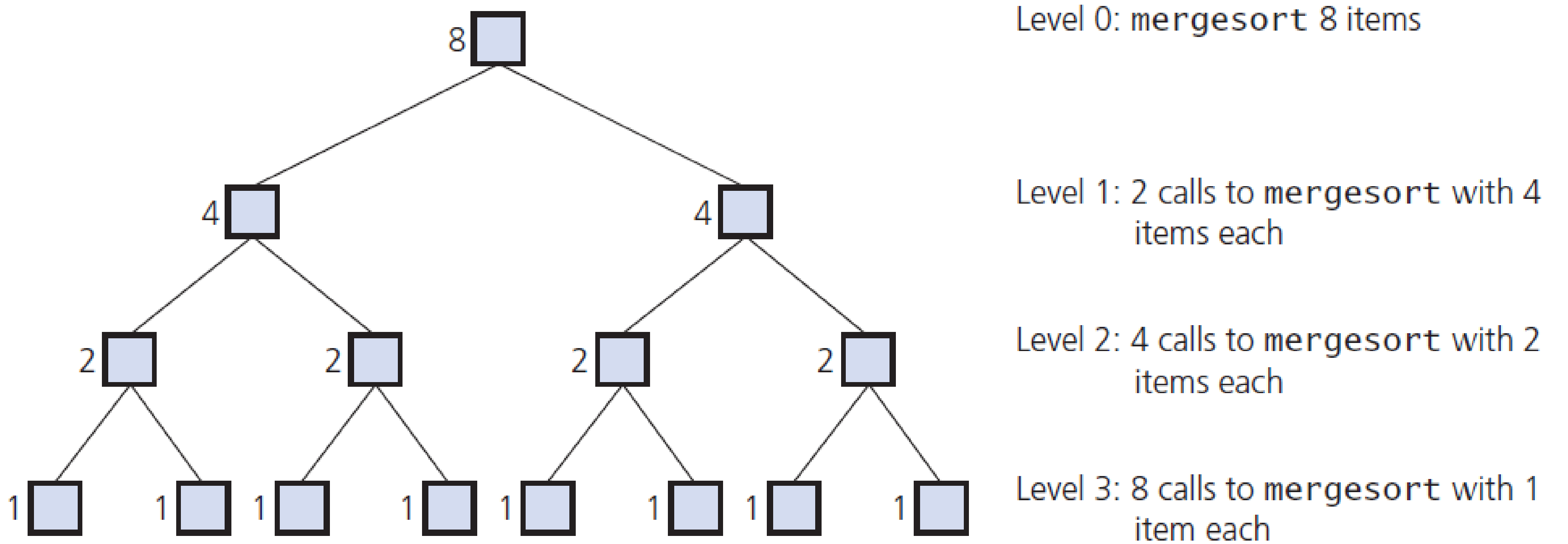


A worst-case instance of the merge step in a merge sort

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



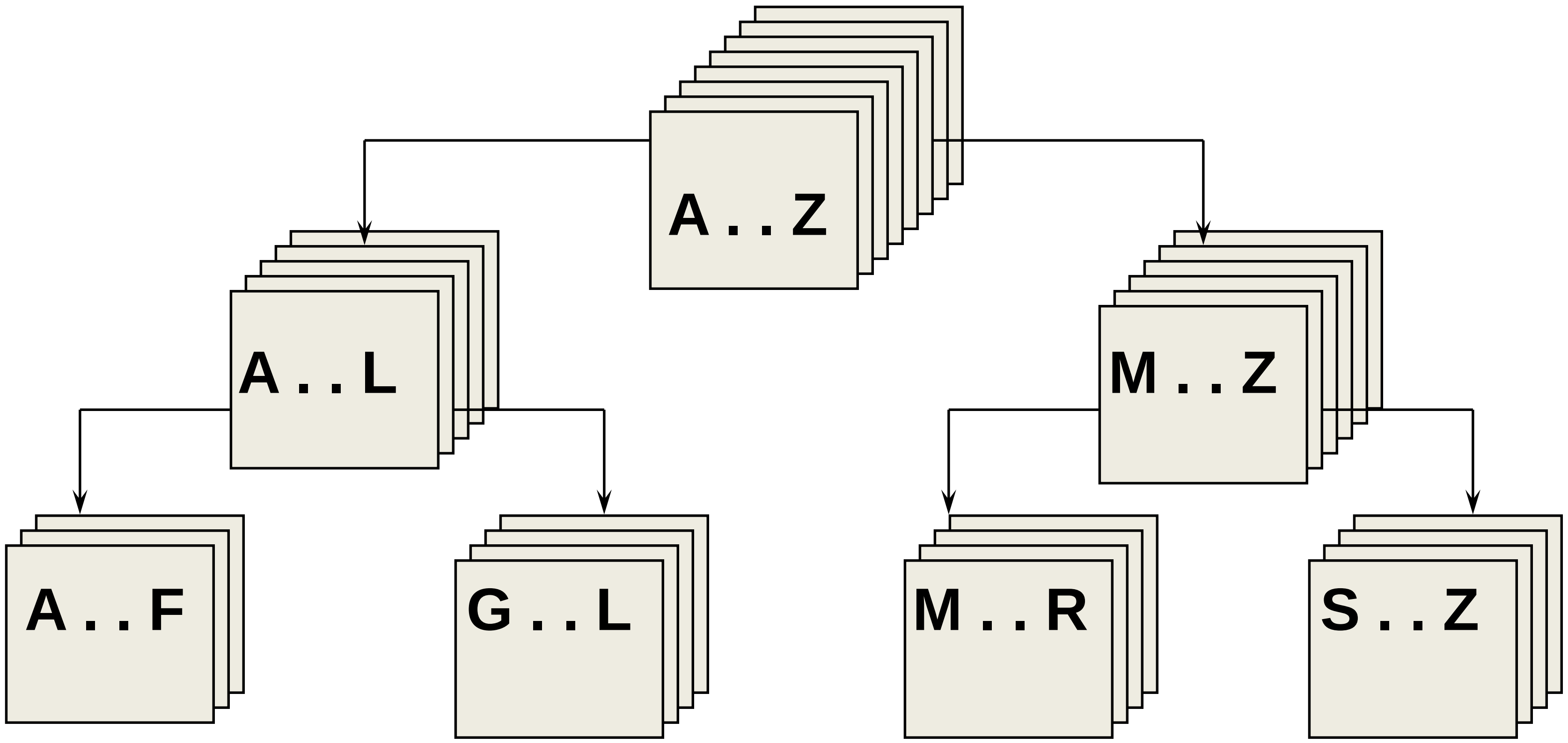
Levels of recursive calls to mergeSort, given an array of eight items

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

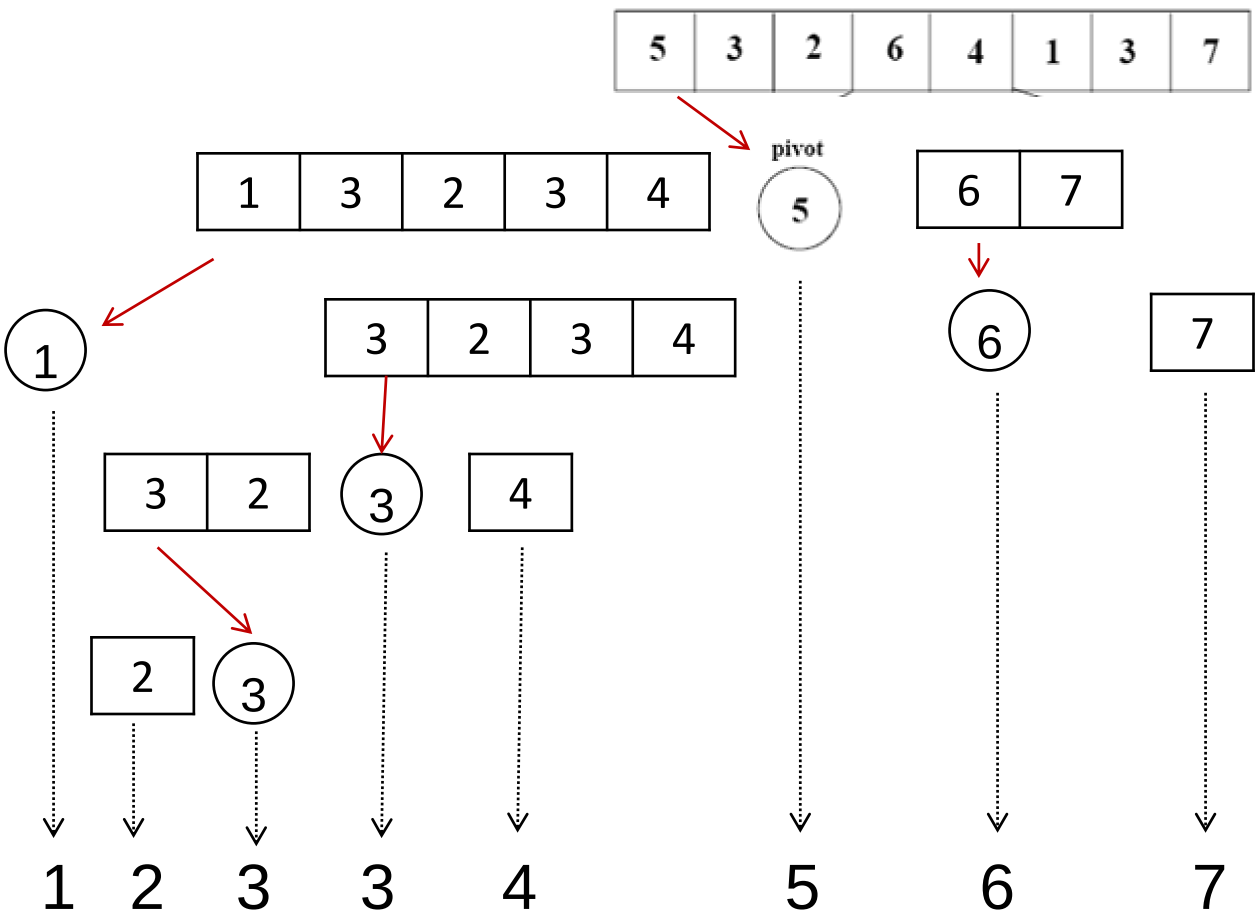
larger values in right part

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

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

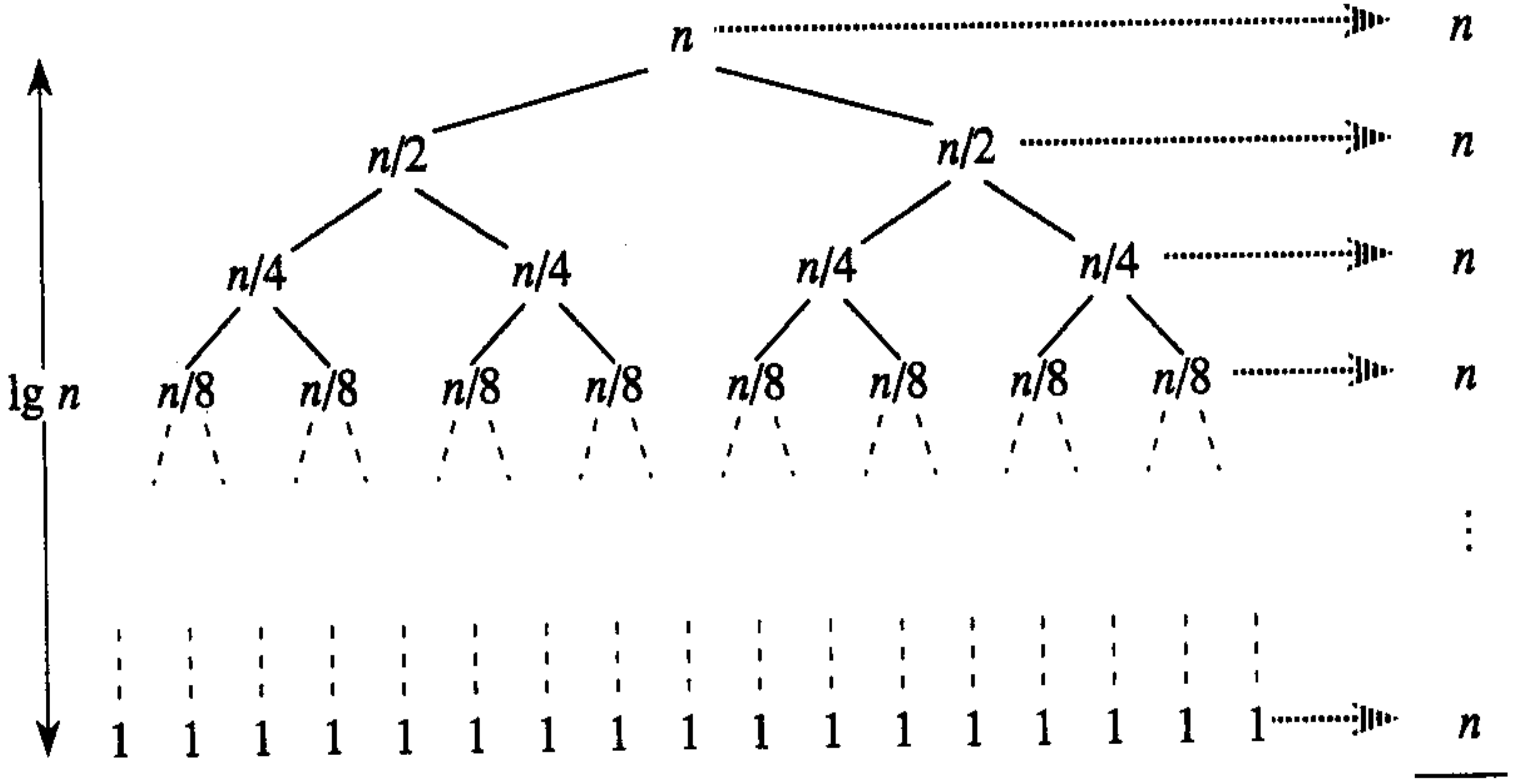
larger values
in right part with N-1 elements

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

values[first] **[last]**

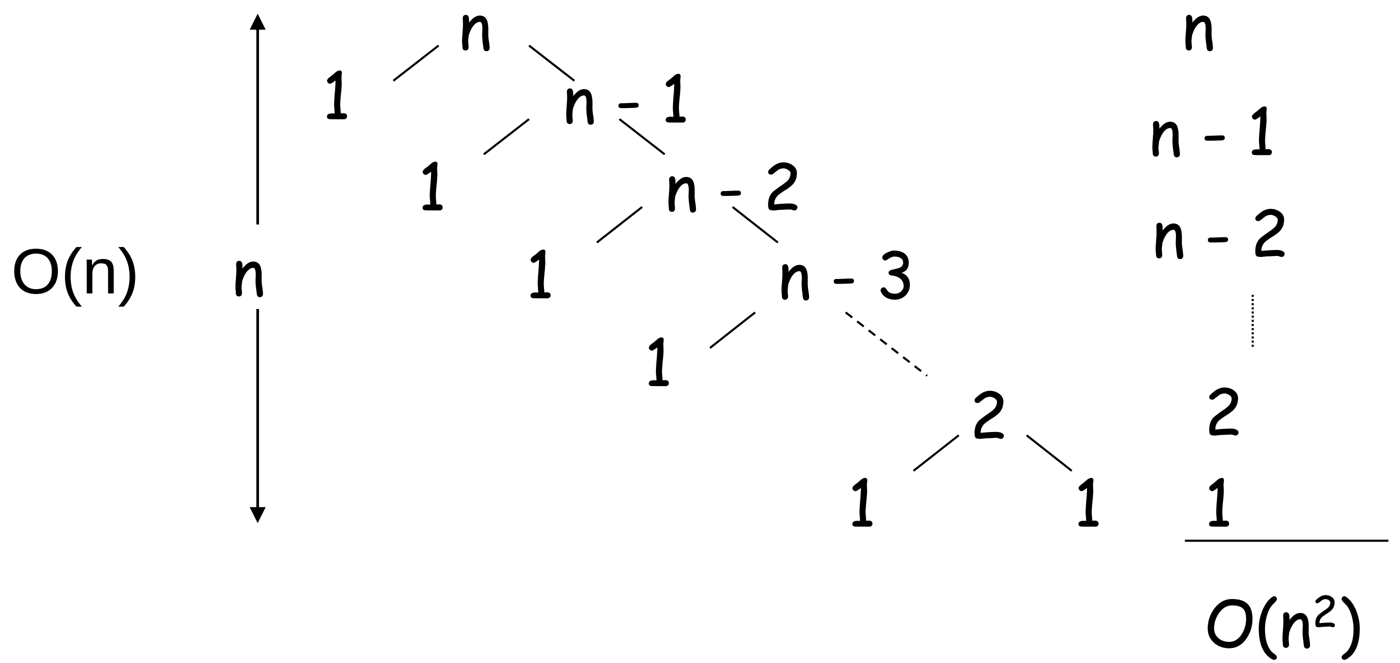
splitVal in correct position

Best case: balanced splits

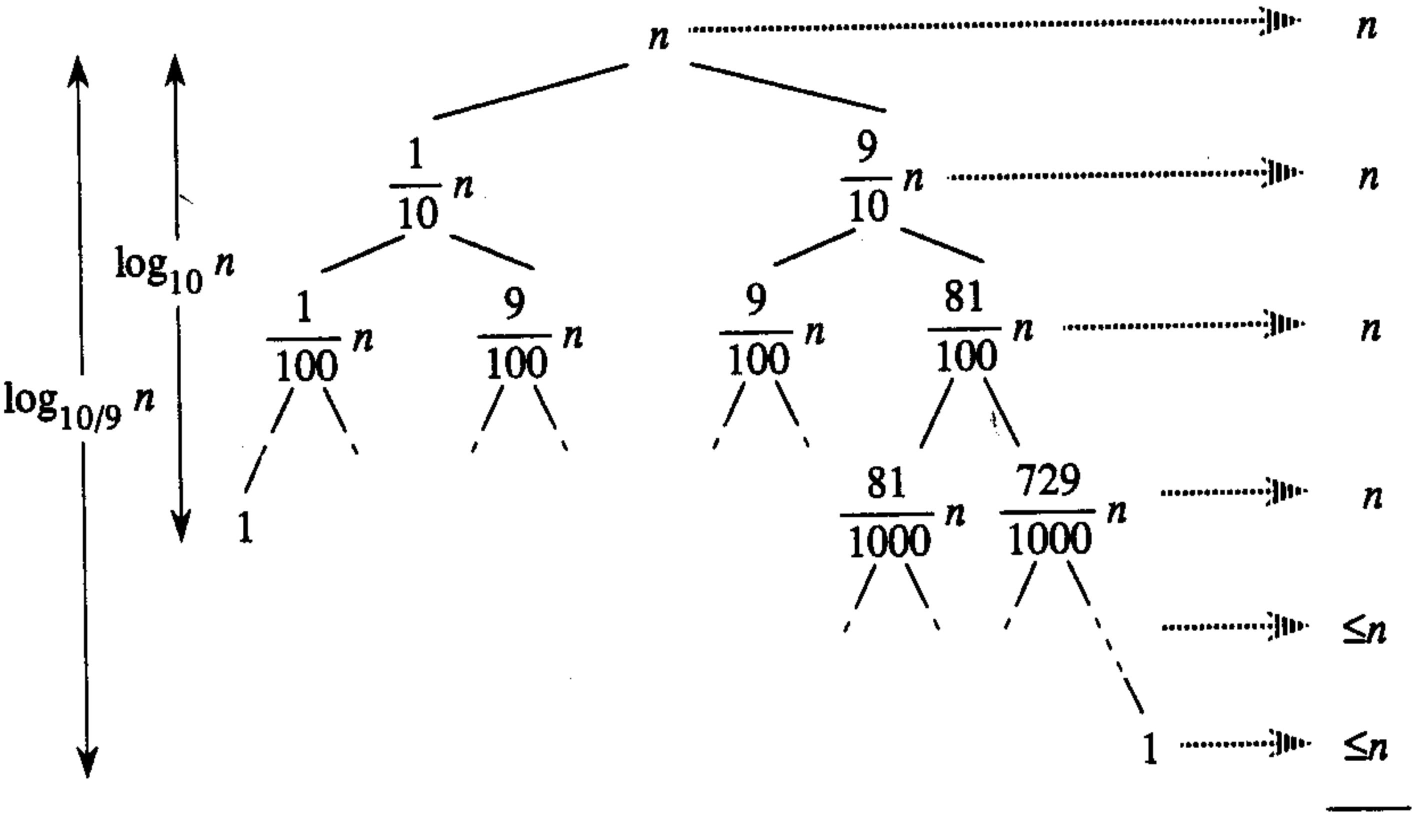


O(nlogn)

Worst case: unbalanced splits



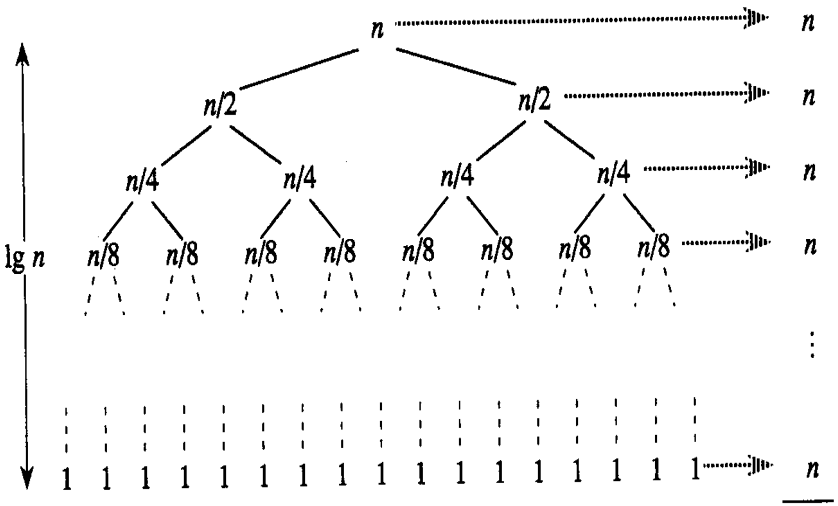
But … is every unbalanced split a bad split?



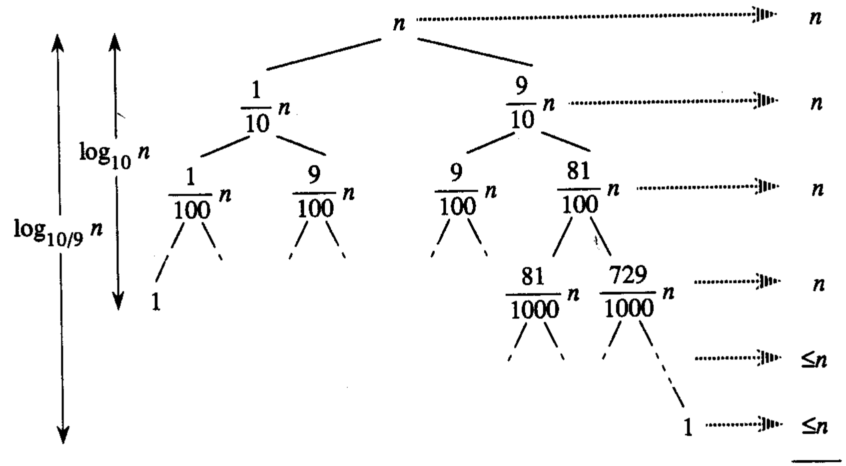
O(nlogn)

Need to look at split ratio!

Split ratio



$O(n\log n)$

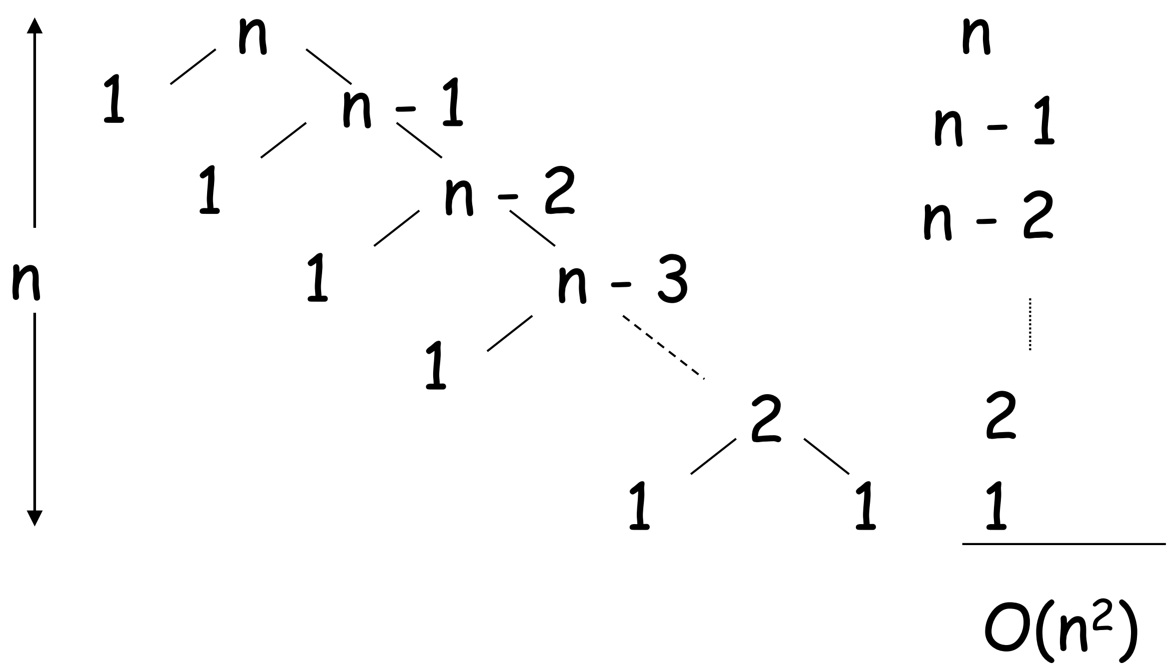


$O(n\log n)$

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

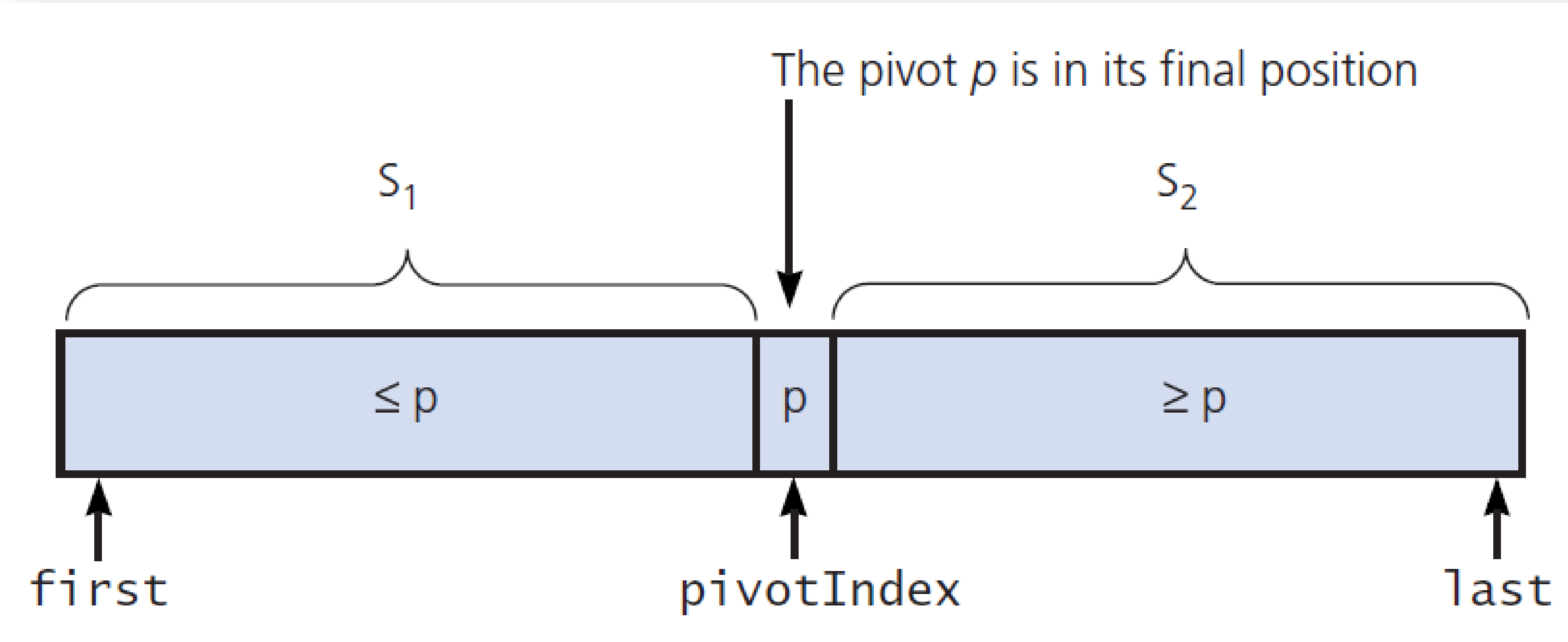
- At each step of the algorithm we exchange element $A[p]$ with an element chosen at **random** from $A[p...r]$

- 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



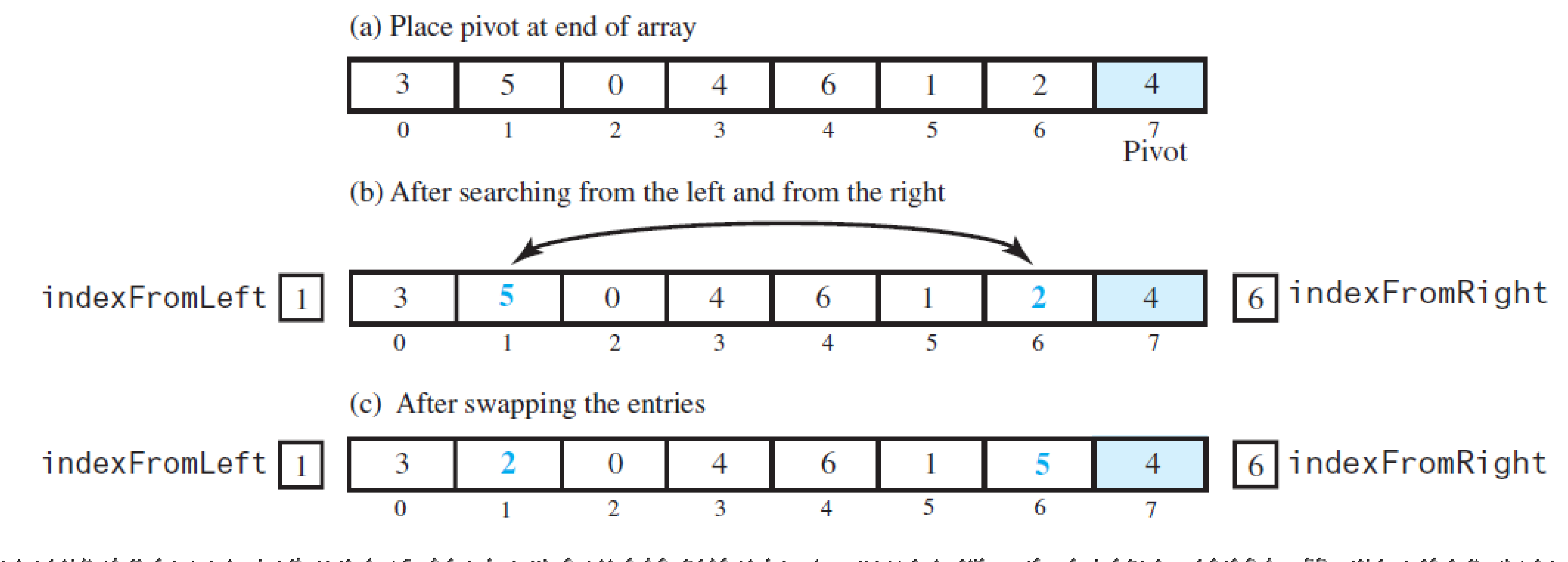
- A partition about a pivot

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
}
```

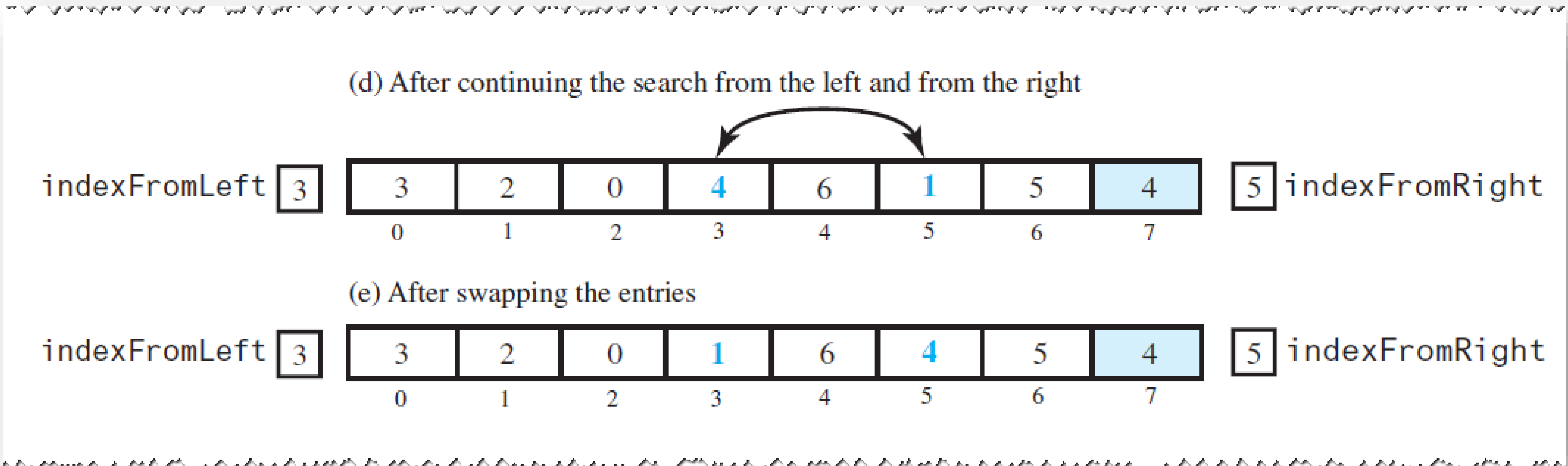
First draft of pseudocode for the quick sort algorithm

The Quick Sort



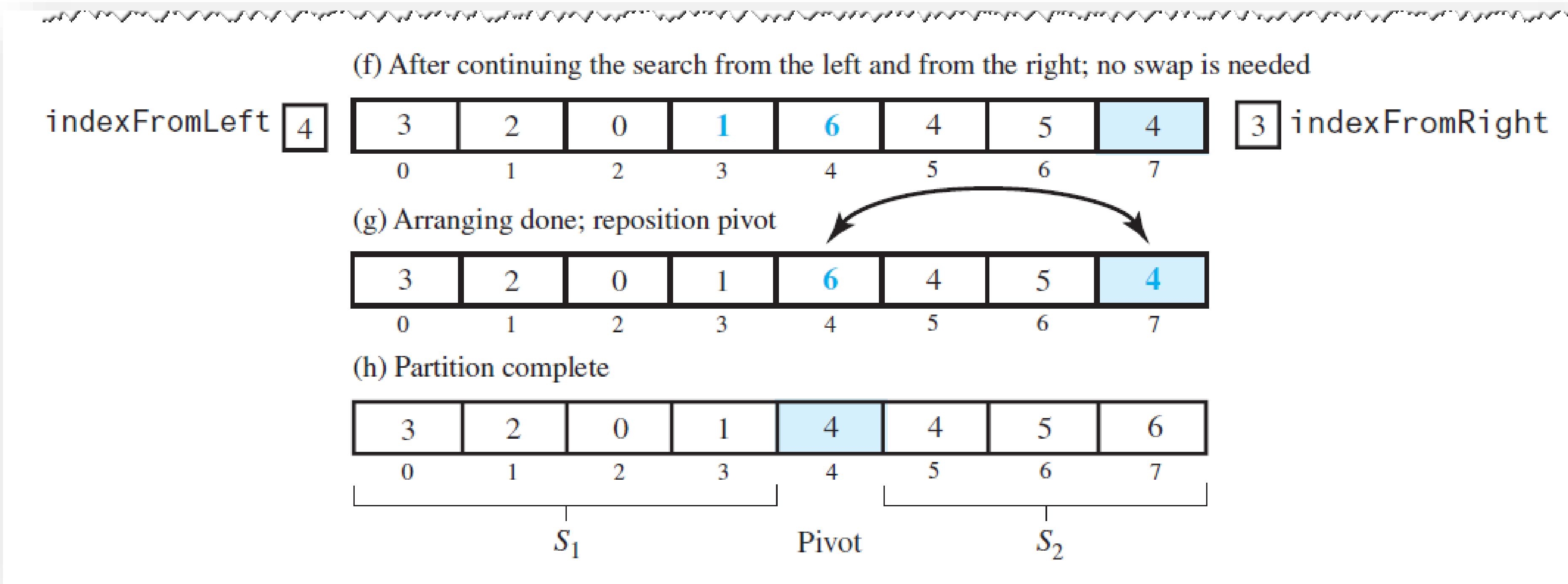
- A partitioning of an array during a quick sort

The Quick Sort



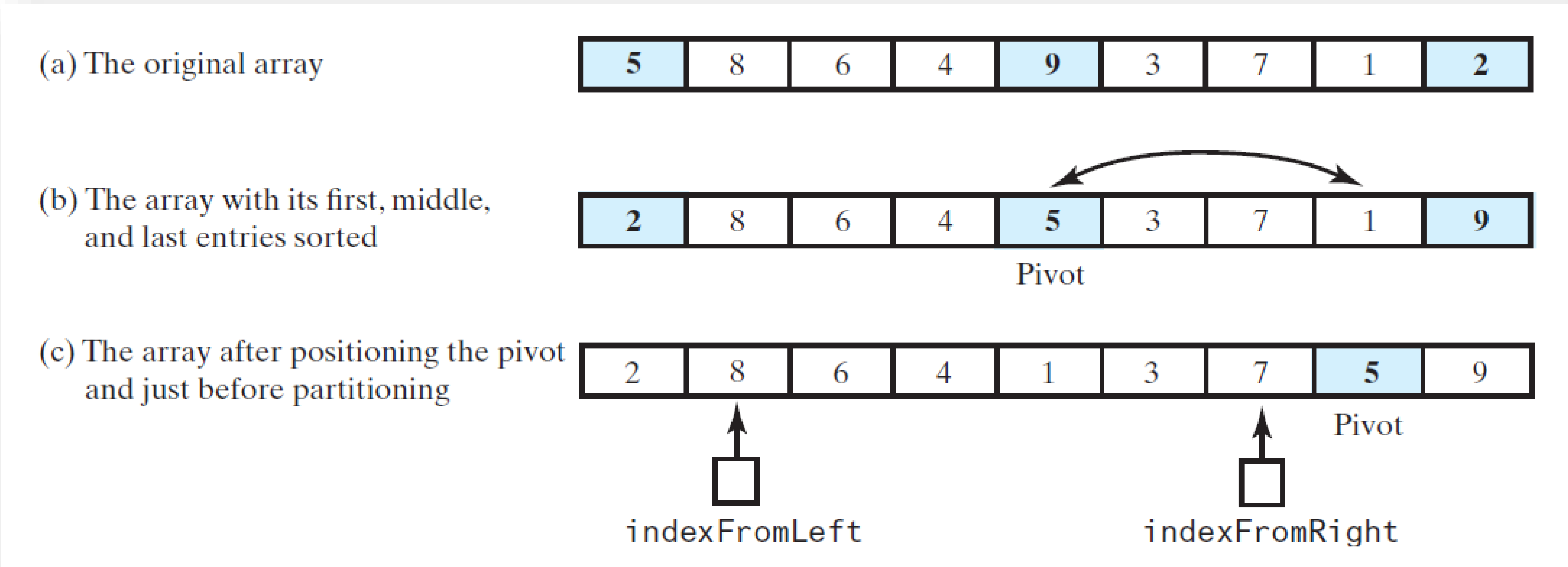
- A partitioning of an array during a quick sort

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

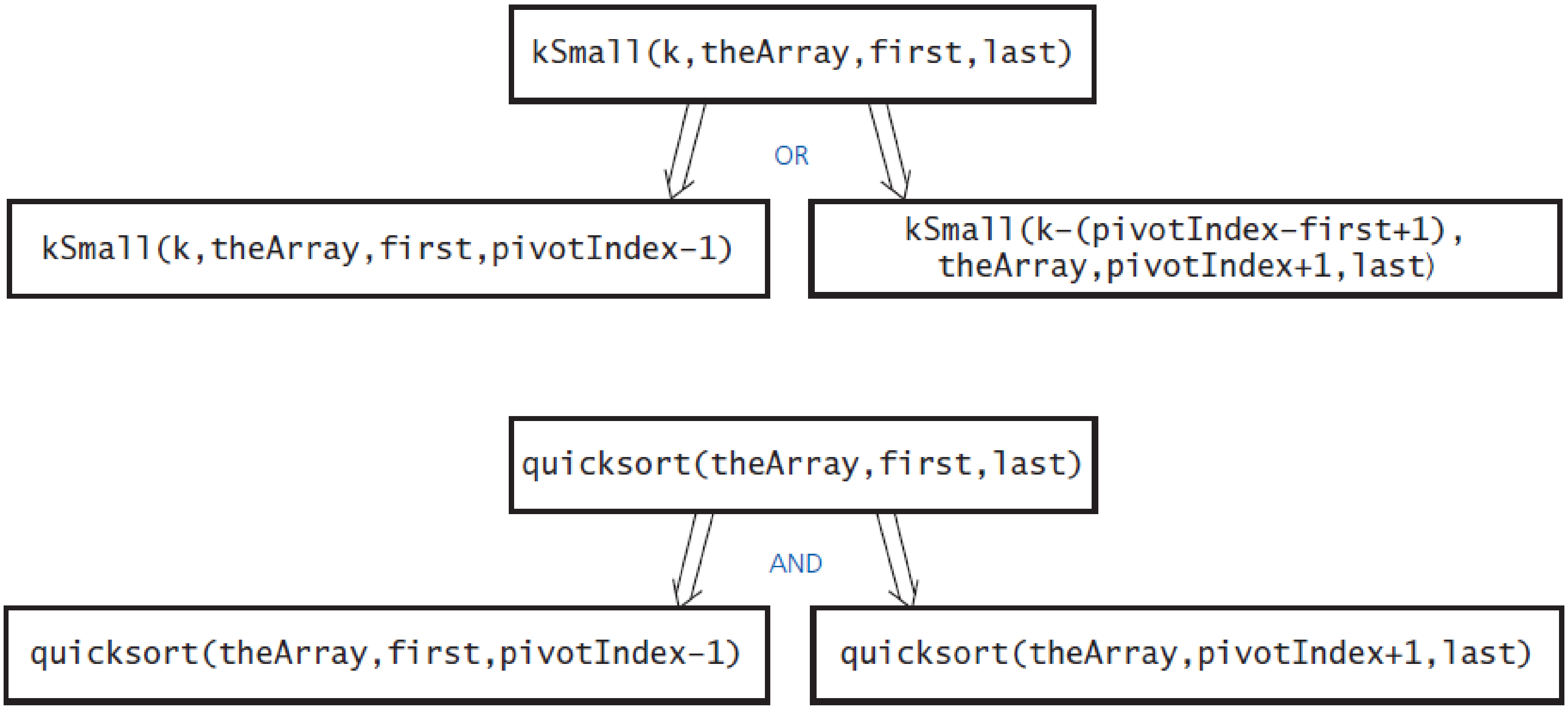
```
   }
   else
   {
      // Create the partition: S1 | Pivot | S2
      int pivotIndex = partition(theArray, first, last);

      // 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

Linear Sorts

How Fast Can We Sort?

- Selection Sort, Bubble Sort, Insertion Sort: $O(n^2)$
- Heap Sort, Merge sort: $O(nlgn)$
- Quicksort: $O(nlgn)$ - *average*
- What is common to all these algorithms?
  - Make **comparisons** between input elements

$a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, or $a_i > a_j$

Lower-Bound for Sorting

- Theorem: To sort n elements, comparison sorts **must** make $\Omega(n\lg n)$ comparisons in the worst case.

(see CS477 for a proof)

Can we do better?

- Linear sorting algorithms
  - Radix Sort
  - Counting Sort
  - Bucket sort

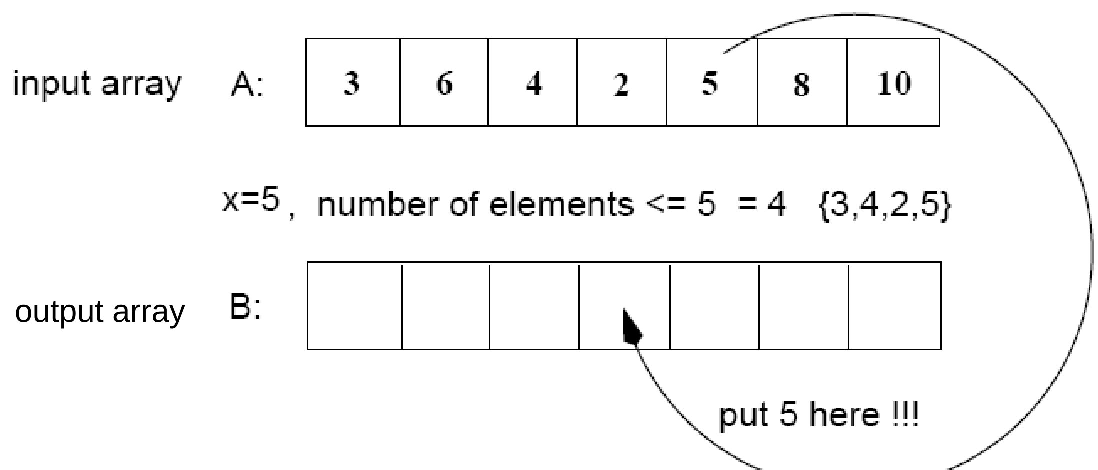
- Make certain assumptions about the data

- Linear sorts are NOT “comparison sorts”

Counting Sort

Counting Sort

- Assumptions:
  - n integers which are in the range [0 ... r]
  - r is in the order of n, that is, $r=O(n)$
- Idea:
  - For each element x, find the number of elements $\leq x$
  - Place x into its correct position in the output array



Step 1

Find the number of times $A[i]$ appears in A

(i.e., frequencies)

input array A:

3	6	4	1	3	4	1	4

allocate C

1	2	3	4	5	6
0	0	0	0	0	0

Allocate $C[1..r]$ (r=6)

i=1, A[1]=3

1	2	3	4	5	6
0	0	1	0	0	0

C[A[1]]=C[3]=1

For $1 \leq i \leq n, ++C[A[i]];$

C[i] = number of times element i appears in A

i=2, A[2]=6

1	2	3	4	5	6
0	0	1	0	0	1

C[A[2]]=C[6]=1

i=3, A[3]=4

1	2	3	4	5	6
0	0	1	1	0	1

C[A[3]]=C[4]=1

⋮

i=8, A[8]=4

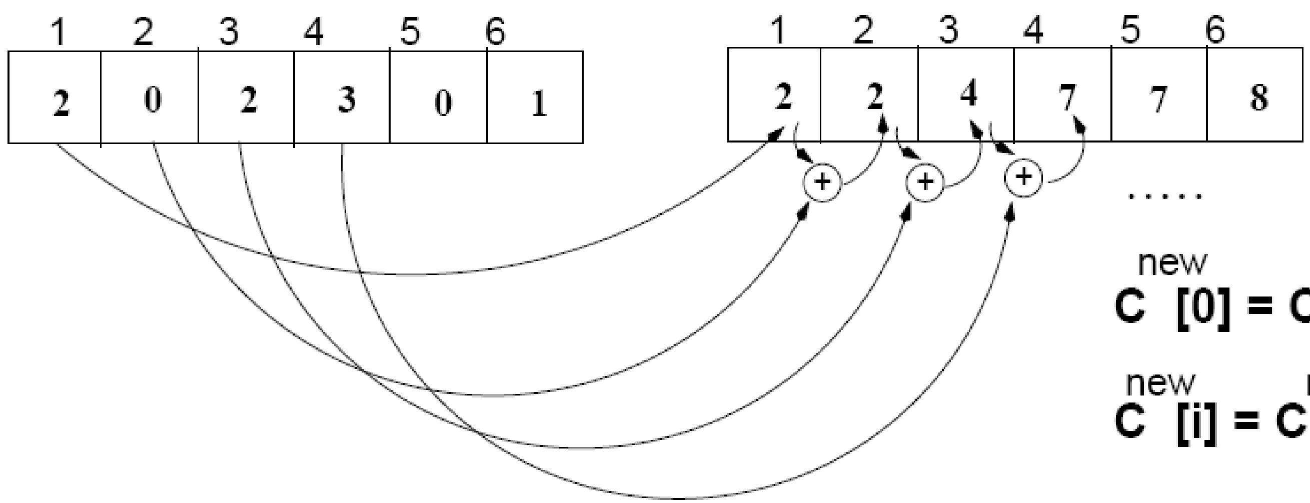
1	2	3	4	5	6
2	0	2	3	0	1

C[A[8]]=C[4]=3

Step 2

Find the number of elements $\leq A[i]$,

C (frequencies) $\quad C^{new}$ (cumulative sums)



$C^{new}[0] = C^{old}[0]$

$C^{new}[i] = C^{new}[i-1] + C^{old}[i]$

C[i] = # elements <= i

Algorithm

- Start from the last element of A
- Place A[i] at its correct place in the output array
- Decrease C[A[i]] by one

A

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C^{new}

0	1	2	3	4	5
2	2	4	7	7	8

Example

A

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C^{new}

0	1	2	3	4	5
2	2	4	7	7	8

B

1	2	3	4	5	6	7	8
						3	

C^{new}

0	1	2	3	4	5
2	2	4	6	7	8

B

1	2	3	4	5	6	7	8
	0					3	

C^{new}

0	1	2	3	4	5
1	2	4	6	7	8

B

1	2	3	4	5	6	7	8
	0				3	3	

C^{new}

0	1	2	3	4	5
1	2	4	5	7	8

B

1	2	3	4	5	6	7	8
	0		2		3	3	

C^{new}

0	1	2	3	4	5
1	2	3	5	7	8

Example (cont.)

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

	1	2	3	4	5	6	7	8
B	0	0		2		3	3	

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

	1	2	3	4	5	6	7	8
B	0	0		2	3	3	3	

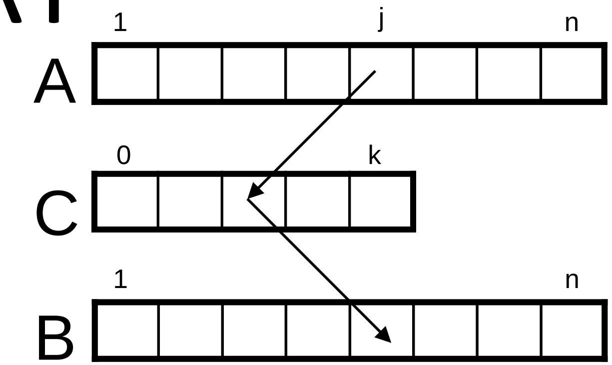
	0	1	2	3	4	5
C	0	2	3	4	7	8

	1	2	3	4	5	6	7	8
B	0	0		2	3	3	3	5

	0	1	2	3	4	5
C	0	2	3	4	7	7

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

COUNTING-SORT



Alg.: COUNTING-SORT(A, B, n, k)

1. **for** i ← 0 **to** r

2. **do** $C[\ i\ ] \leftarrow 0$

3. **for** j ← 1 **to** n

4. **do** $C[A[\ j\ ]] \leftarrow C[A[\ j\ ]] + 1$

5. ▷ $C[i]$ contains the number of elements equal to i

6. **for** i ← 1 **to** r

7. **do** $C[\ i\ ] \leftarrow C[\ i\ ] + C[i -1]$

8. ▷ $C[i]$ contains the number of elements ≤ i

9. **for** j ← n **downto** 1

10. **do** $B[C[A[\ j\ ]]] \leftarrow A[\ j\ ]$

11. $C[A[\ j\ ]] \leftarrow C[A[\ j\ ]] - 1$

Analysis of Counting Sort

Alg.: COUNTING-SORT(A, B, n, k)

```
for i ← 0 to r                              } O(r)
    do C[ i ] ← 0
for j ← 1 to n                              } O(n)
    do C[A[ j ]] ← C[A[ j ]] + 1
▷ C[i] contains the number of elements equal to i
for i ← 1 to r                              } O(r)
    do C[ i ] ← C[ i ] + C[i -1]
▷ C[i] contains the number of elements ≤ i
for j ← n downto 1                          } O(n)
    do B[C[A[ j ]]] ← A[ j ]
       C[A[ j ]] ← C[A[ j ]] - 1
```

Overall time: $O(n + r)$

Analysis of Counting Sort

- Overall time: $O(n + r)$

- In practice, we use COUNTING sort when $r = O(n)$

$\Rightarrow$ running time is $O(n)$

Bucket Sort

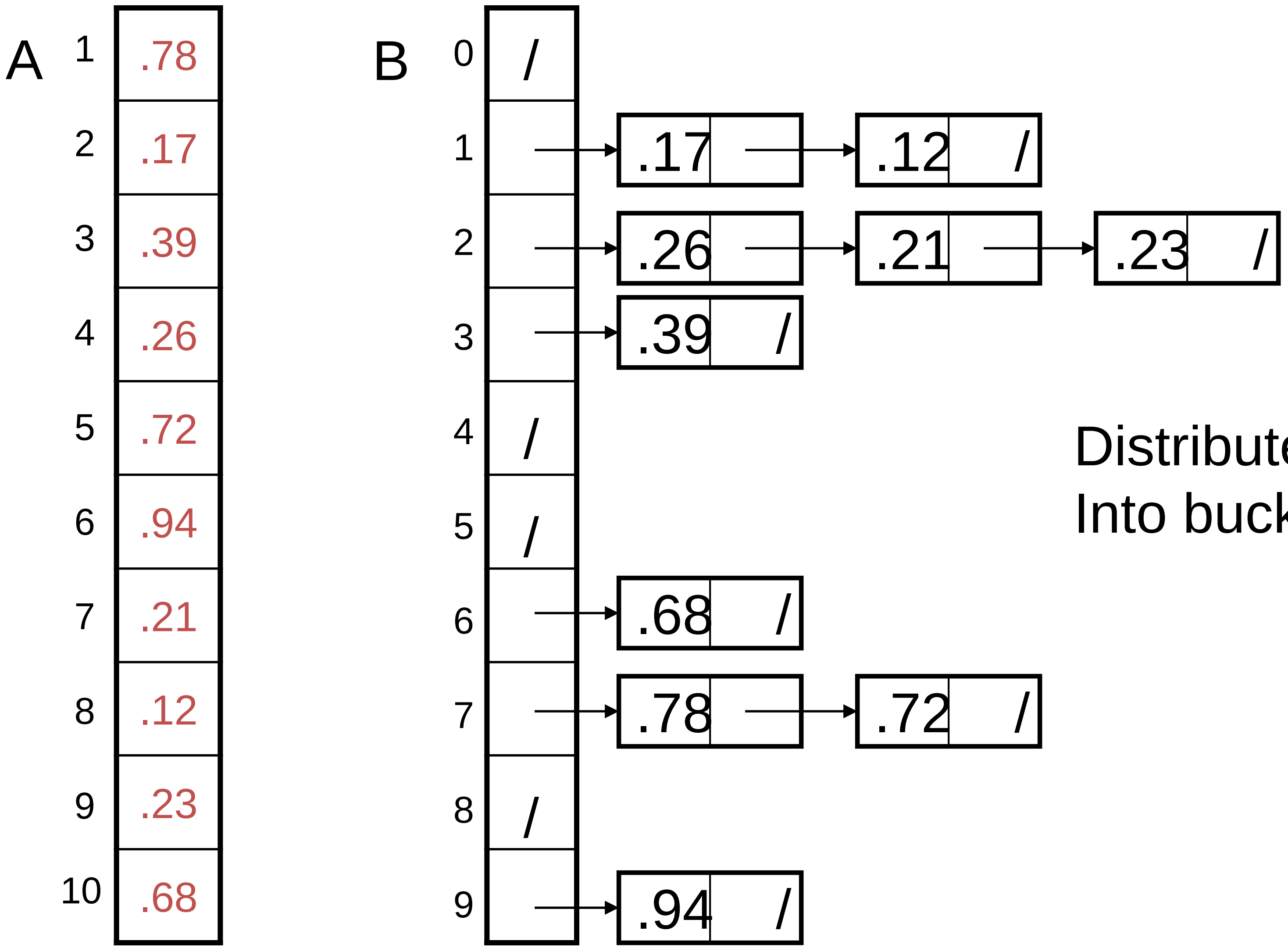
Bucket Sort

- Assumption:
  - the input is generated by a random process that distributes elements uniformly over [0, 1)
- Idea:
  - Divide [0, 1) into k equal-sized buckets (**k=Θ(n)**)
  - Distribute the n input values into the buckets
  - Sort each bucket (e.g., using mergesort)
  - Go through the buckets in order, listing elements in each one

- **Input:** $A[1 \,..\, n]$, where $0 \leq A[i] < 1$ for all i

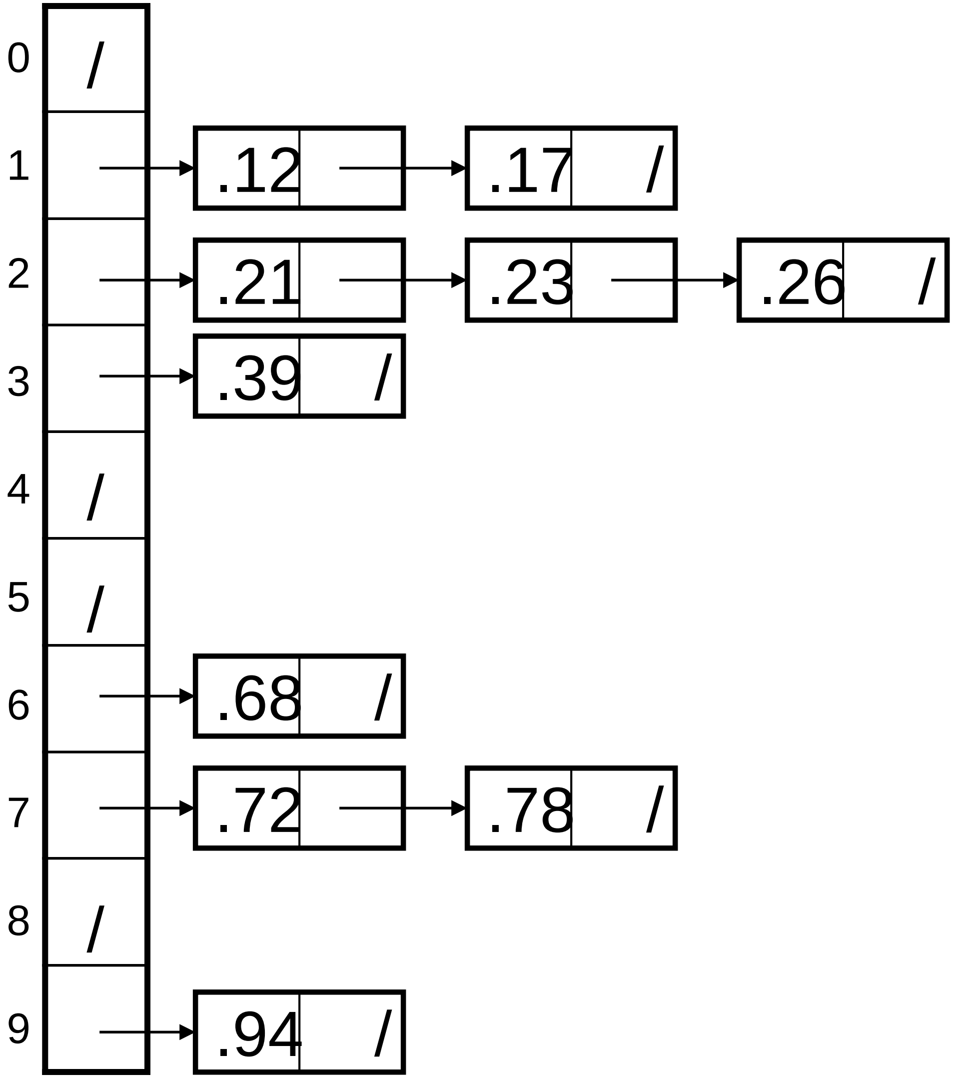
- **Output:** elements $A[i]$ sorted

Example - Bucket Sort



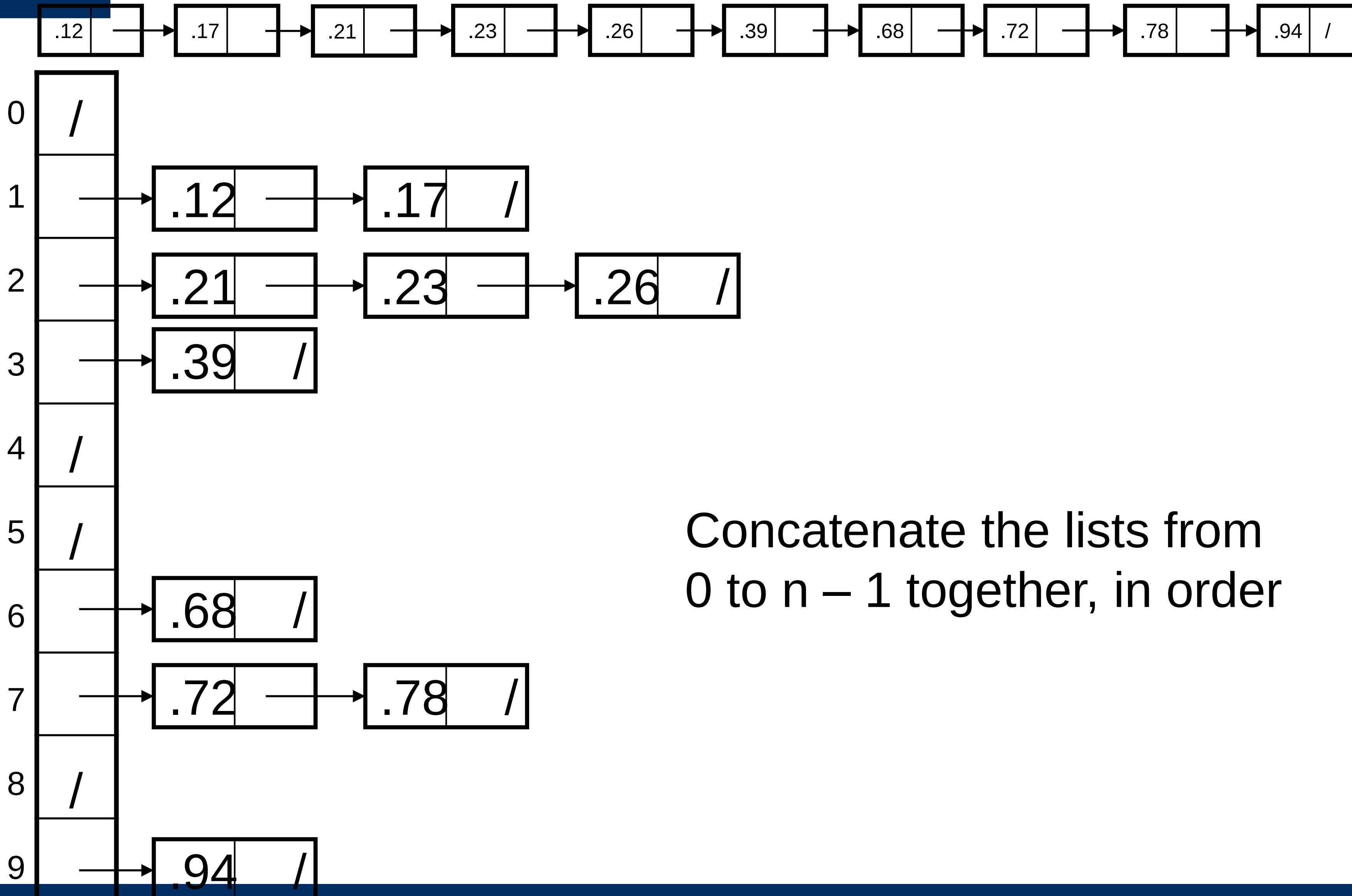
Distribute
Into buckets

Example - Bucket Sort



Sort within each bucket

Example - Bucket Sort



Concatenate the lists from 0 to n – 1 together, in order

Analysis of Bucket Sort

Alg.: BUCKET-SORT(A, n)

for i ← 1 **to** n

  do insert A[i] into list $B[\lfloor nA[i] \rfloor]$

O(n)

for i ← 0 **to** k - 1

  do sort list $B[i]$ with mergesort sort

k O(n/k log(n/k))
= O(nlog(n/k))

concatenate lists $B[0]$, $B[1]$, . . . , $B[n -1]$

together in order

O(k)

return the concatenated lists

O(n) (if k=Θ(n))

Radix Sort

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

Radix Sort

- Represents keys as d-digit numbers in some base-k

$$key = x_1 x_2 \ldots x_d \quad \text{where } 0 \le x_i \le k-1$$

- Example: key=15

$$key_{10} = 15, \quad d=2,\ k=10 \quad \text{where } 0 \le x_i \le 9$$

$$key_{2} = 1111,\ d=4,\ k=2 \quad \text{where } 0 \le x_i \le 1$$

Radix Sort

- Assumptions

  $d = O(1)$ and $k = O(n)$

- Sorting looks at one column at a time
  - For a d digit number, sort the least significant digit first
  - Continue sorting on the next least significant digit,
    - until all digits have been sorted
  - Requires only d passes through the list

326
453
608
835
751
435
704
690

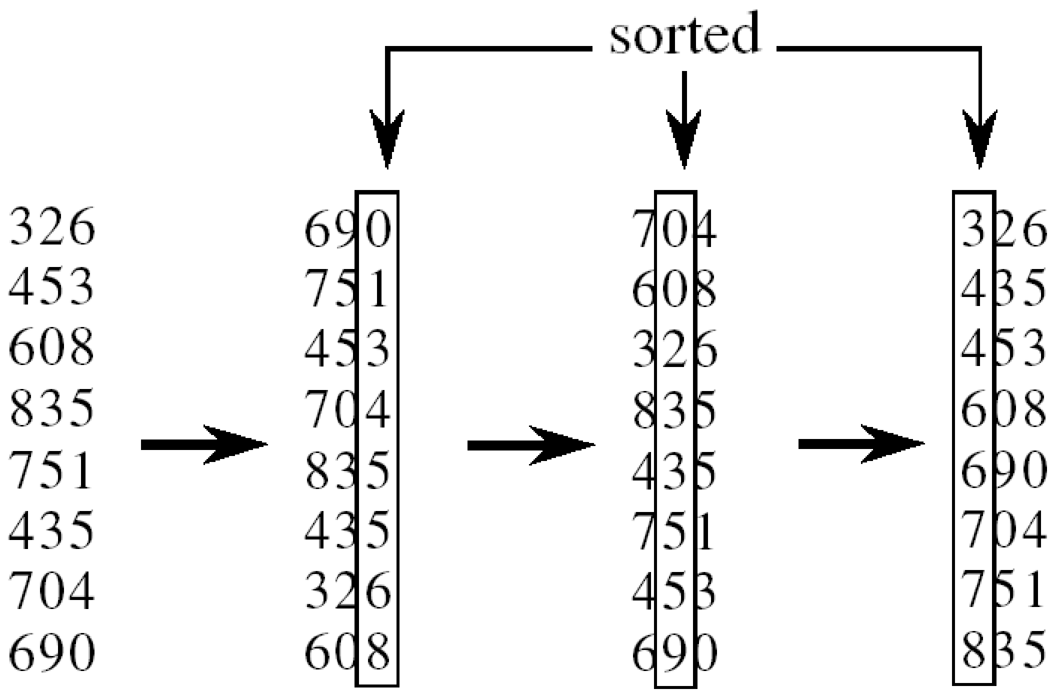
RADIX-SORT

Alg.: RADIX-SORT(A, d)

for i ← 1 **to** d

do use a stable sort to sort array A on digit i

(stable sort: preserves order of identical elements)



The Radix Sort

- Uses the idea of forming groups, then combining them to sort a collection of data

- Consider collection of three letter groups

  ABC, XYZ, BWZ, AAC, RLT, JBX, RDT, KLT, AEO, TLJ

- Group strings by rightmost letter

  (AB**C**, AA**C**) (TL**J**) (AE**O**) (RL**T**, RD**T**, KL**T**) (JB**X**) (XY**Z**, BW**Z**)

- Combine groups

  ABC, AAC, TLJ, AEO, RLT, RDT, KLT, JBX, XYZ, BWZ

The Radix Sort

- Group strings by middle letter

(A**A**C) (A **B** C, J **B** X) (R **D** T) (A **E** O) (T **L** J, R **L** T, K **L** T) (B **W** Z) (X **Y** Z)

- Combine groups

AAC, ABC, JBX, RDT, AEO, TLJ, RLT, KLT, BWZ, XYZ

- Group by first letter, combine again

(**A** AC, **A** BC, **A** EO) (**B** WZ) (**J** BX) (**K** LT) (**R** DT, **R** LT) (**T** LJ) (**X** YZ)

- Sorted strings

AAC, ABC, AEO, BWZ, JBX, KLT, RDT, RLT, TLJ, XYZ

The Radix Sort

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)

- A radix sort of eight integers

The Radix Sort

```
// 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.
   }
}
```

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

The Radix Sort

- 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
- Radix sort is of order O(*n*)

Analysis of Radix Sort

- Given n numbers of d digits each, where each digit may take up to k possible values, RADIX-SORT correctly sorts the numbers in $O(d(n+k))$
  - One pass of sorting per digit takes $O(n+k)$ assuming that we use **counting sort**
  - There are d passes (for each digit)
  - Assuming d=O(1) and k=O(n), running time is O(n)

Stable Sorting Algorithms

- $O(N^2)$
  - Selection Sort, Insertion Sort, Bubble Sort
- $O(N \log N)$
  - Merge Sort
- $O(N)$
  - Counting Sort, Bucket Sort, Radix Sort

A Comparison of Sorting Algorithms

	Best case	Average case	Worst case
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Buble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Counting sort	$O(n)$	$O(n)$	$O(n + r)$
Bucket sort	$O(n)$	$O(n)$	$O(n \log n)$
Radix sort	$O(n)$	$O(n)$	$O(n + k)$

- Approximate growth rates of time required for sorting algorithms

Heap Sort