The Merge Sort

Pearson

Sorting With Recursion

- $O(n^2)$ **sorting algorithms are fine for small arrays**
  - Sorting time grows rapidly as size increases

n	n^2
1	1
10	100
100	10,000
1,000	1,000,000
10,000	100,000,000
100,000	1,000,000,000

- **Recursion can help us sort more efficiently**
  - "Divide and Conquer"
  - Average time: $O(n \log_2 n)$

n	$n \log_2 n$
1	< 1
10	33
100	664
1,000	9,966
10,000	123,877
100,000	1,660,964

Pearson

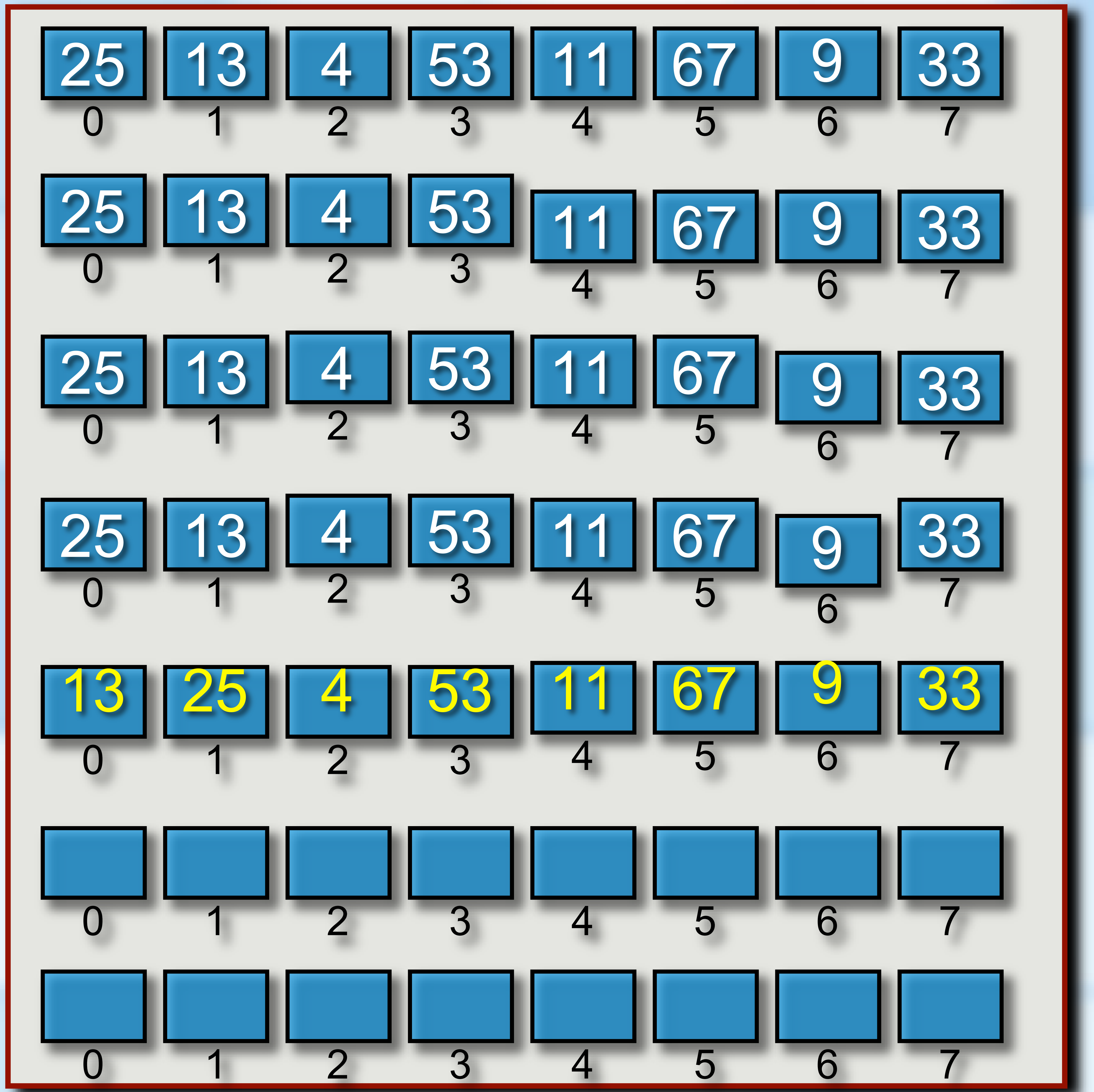
The Merge Sort

- Divide the array in half
- Sort each half recursively
- Merge the sorted halves back together

```
void mergeSort(ItemType theArray[], int start, int end)
{
   if (start < end)
   {
      // Find midpoint
      int mid = start + (end - start) / 2;

      // Sort each half
      mergeSort(theArray, start, mid);

      mergeSort(theArray, mid + 1, end);

      // Merge the two halves
      merge(theArray, start, mid, end);

   }  // end if
}  // end mergeSort
```

Pearson

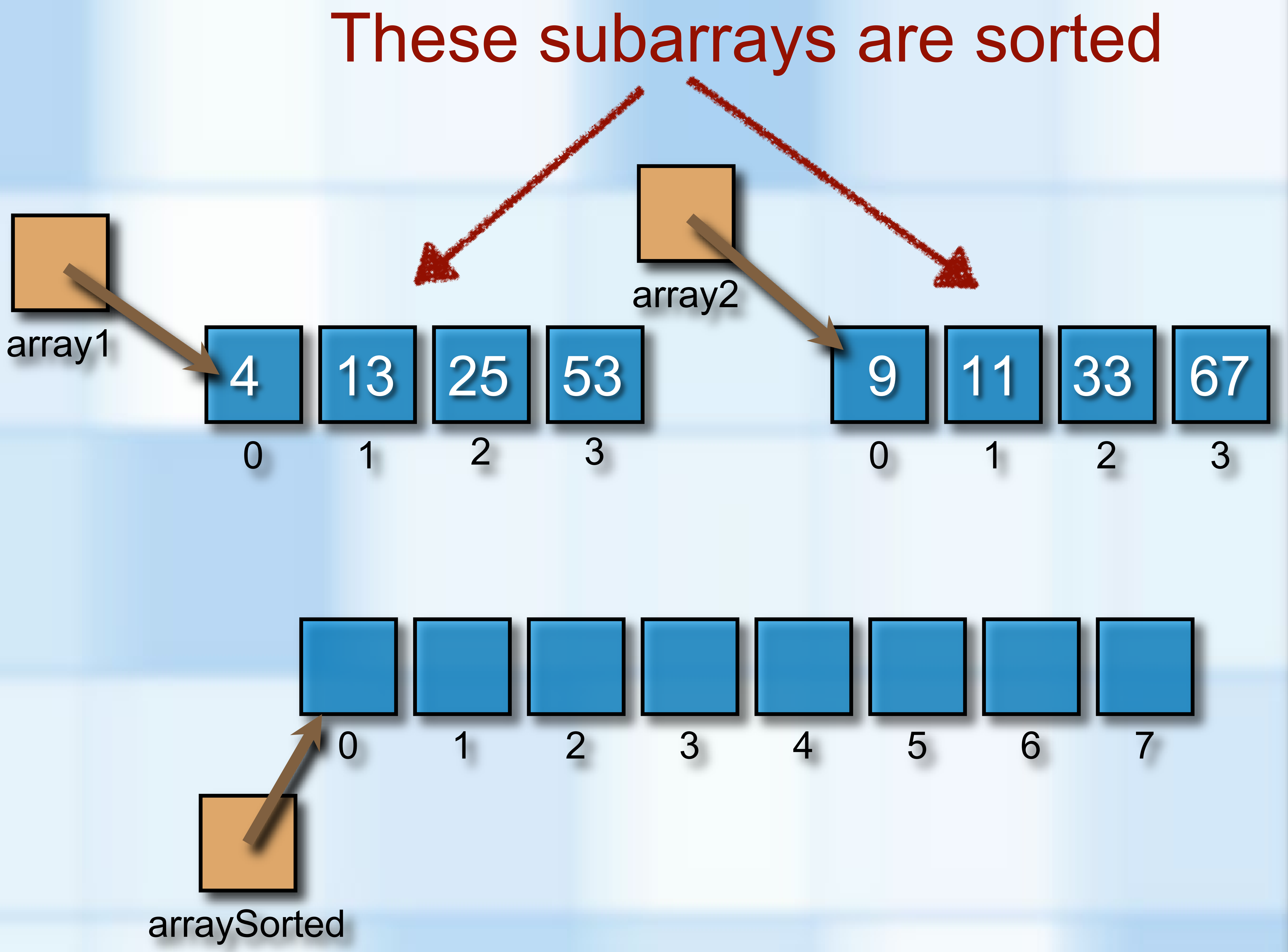
The Merge Sort




```
void mergeSort(ItemType theArray[], int start, int end)
{
  if (start < end)
  {
    // Find midpoint
    int mid = start + (end - start) / 2;

    // Sort each half
    mergeSort(theArray, start, mid);

    mergeSort(theArray, mid + 1, end);

    // Merge the two halves
    merge(theArray, start, mid, end);

  } // end if
} // end mergeSort
```

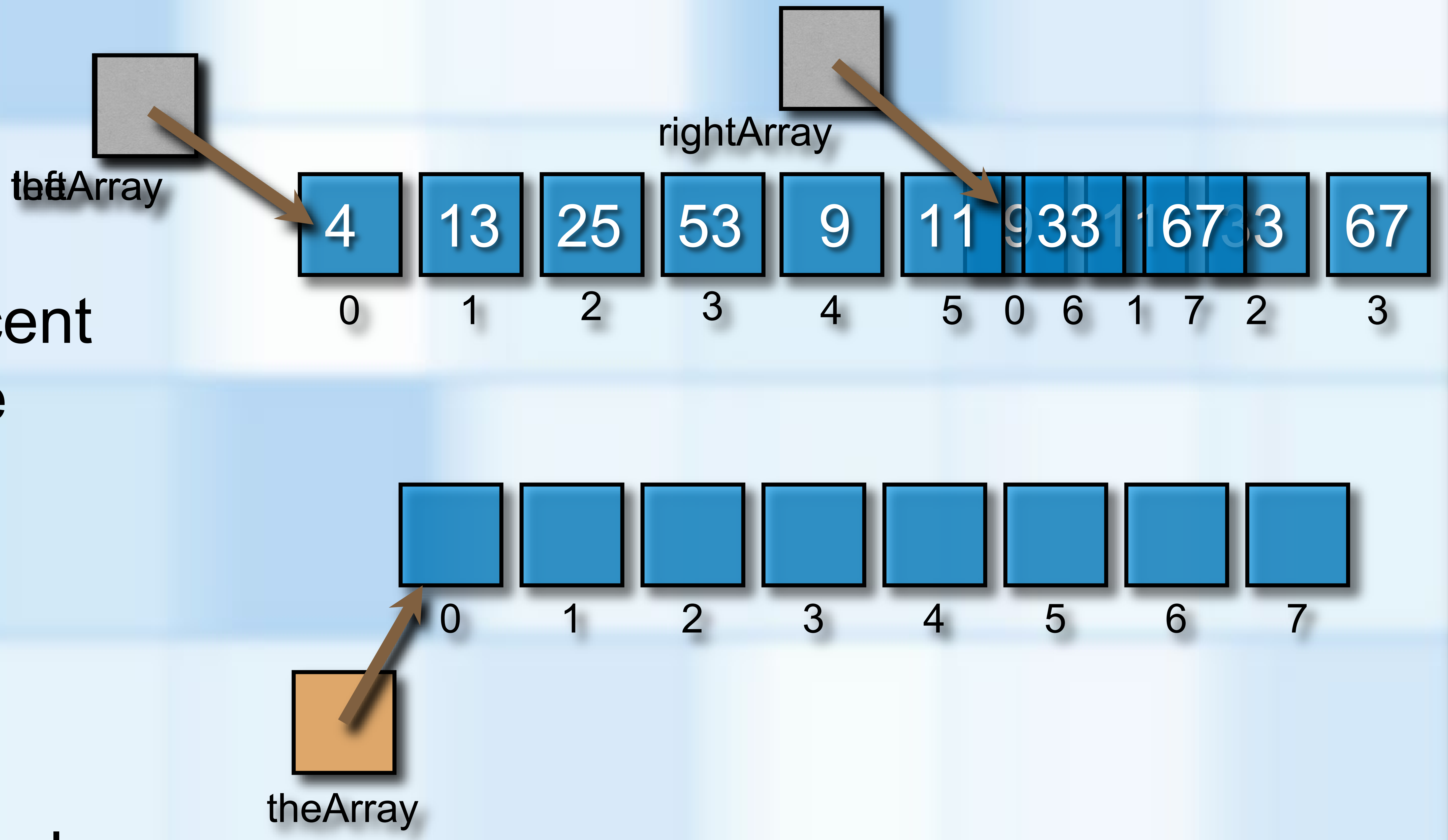
Merging Two Sorted Arrays

- **Compare first item in each array**
- **Copy the smaller to a new array**
- **Continue until one array is empty**
- **Copy remaining items from other array**



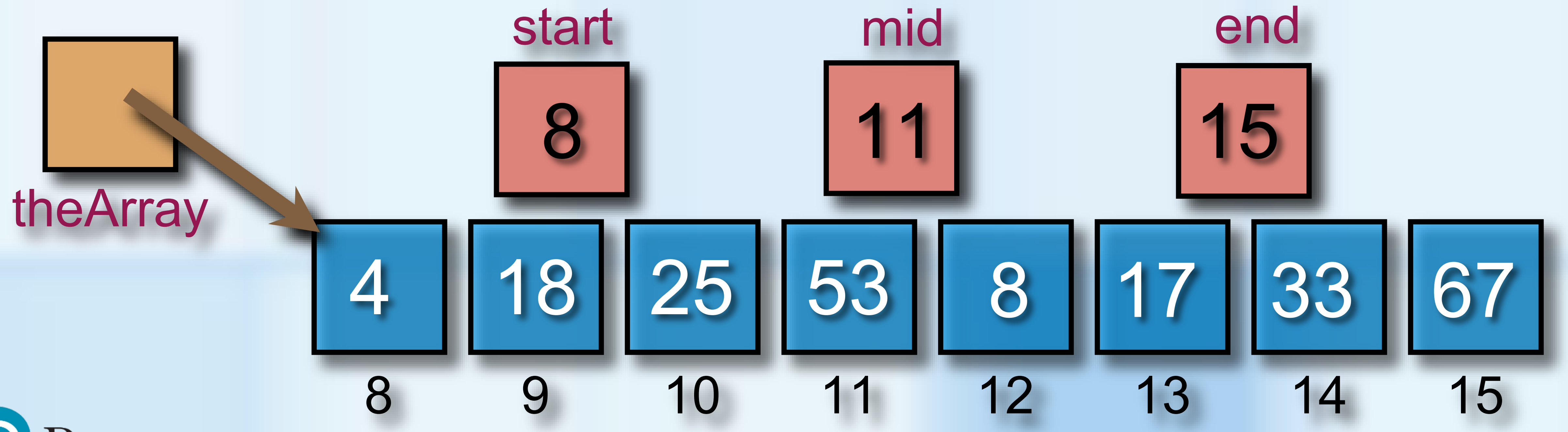
Merging Two Sorted Arrays

- **For our implementation**
  - The two sorted subarrays are adjacent sequences of elements in the same array
  - Copy elements into two temporary arrays
  - Merge elements back into the original array



The Merge Algorithm

- **Parameters**
  - `theArray` - the array to merge
  - `start` - first item of left sorted subarray
  - `mid` - last item of the left sorted subarray
  - `end` - last item in the right sorted subarray
- **Other important variables**
  - `leftIndex` - next left subarray item
  - `rightIndex` - next right subarray item
  - `mergeLocation` - where to place next sorted value

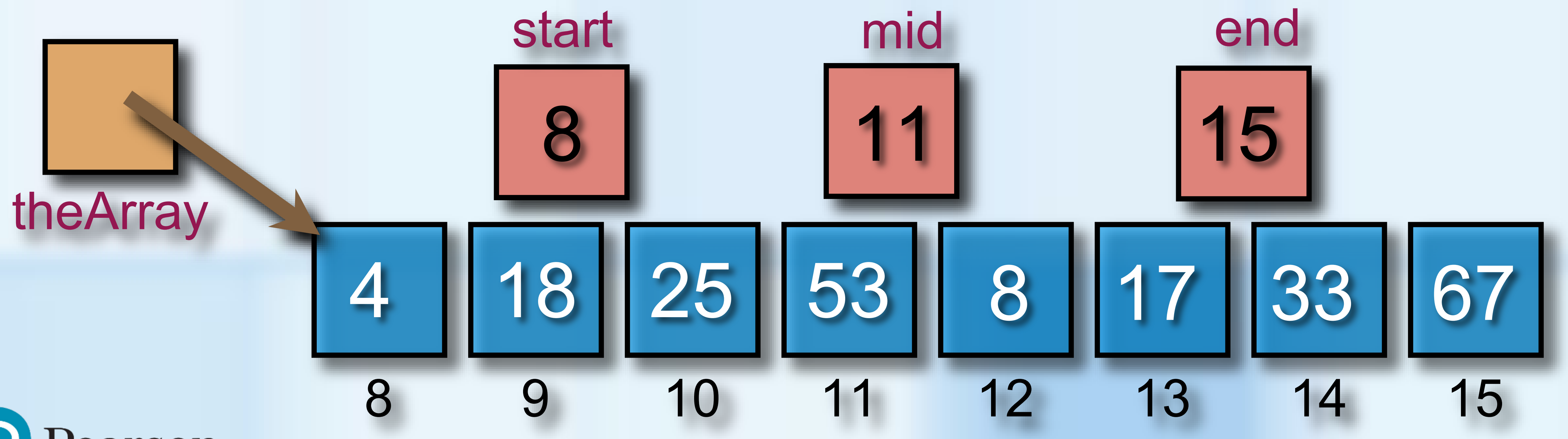



```
template <class ItemType>
void merge(ItemType theArray[], int start, int mid, int end)
{
  int sizeLeft = mid – start + 1;     // Size of left subarray
  int sizeRight = end - mid;  // Size of right subarray
  ItemType leftArray[sizeLeft];   // Temporary array
  ItemType rightArray[sizeRight]; // Temporary array

    // Move items to merge into temporary subarrays
  for (int index = 0; index < sizeLeft; index++)
    leftArray[index] = theArray[start + index];

  for (int index = 0; index < sizeRight; index++)
    rightArray[index] = theArray[mid + 1 + index];

    // While both subarrays are not empty, copy the
    // smaller item into the temporary array
  int leftIndex = 0;          // Beginning of first subarray
  int rightIndex = 0;          // Beginning of second subarray
  int mergeLocation = start;  // where to place next value

  while ((leftIndex < sizeLeft) && (rightIndex < sizeRight))
  {
      // At this point, leftArray and rightArray are in order
    if (leftArray[leftIndex ] <= rightArray[rightIndex])
    {
      theArray[mergeLocation] = leftArray[leftIndex];
      leftIndex++;
```

Pearson

The Merge Algorithm

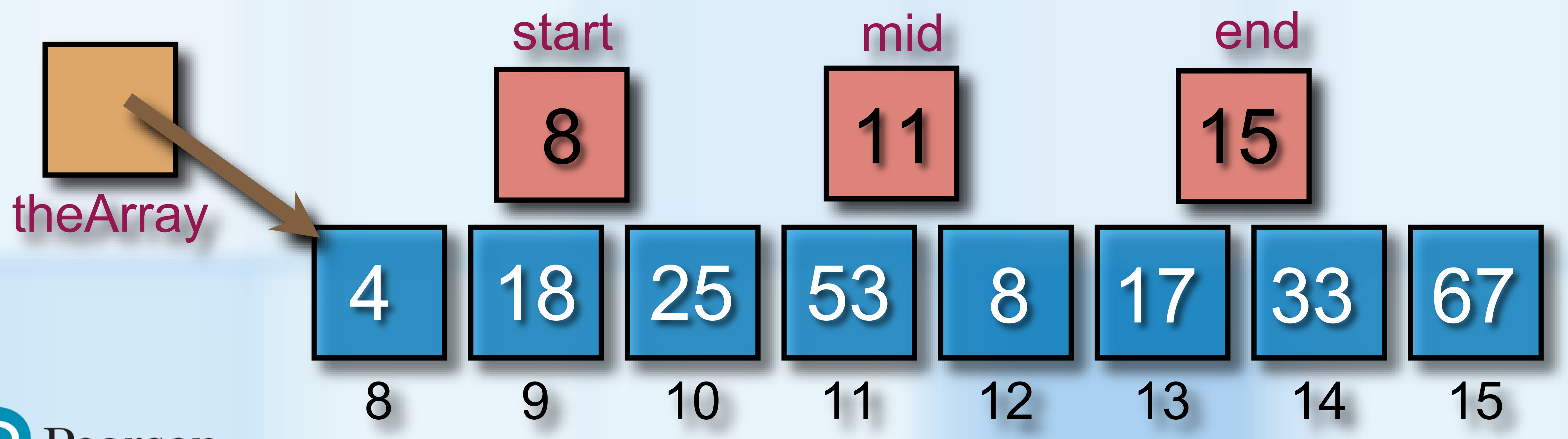
- **Parameters**
  - **`theArray`** - the array to merge
  - **`start`** - first item of left sorted subarray
  - **`mid`** - last item of the left sorted subarray
  - **`end`** - last item in the right sorted subarray
- **Other important variables**
  - **`leftIndex`** - next left subarray item
  - **`rightIndex`** - next right subarray item
  - **`mergeLocation`** - where to place next sorted value




```
while ((leftIndex < sizeLeft) && (rightIndex < sizeRight))
  {
      // At this point, leftArray and rightArray are in order
    if (leftArray[leftIndex ] <= rightArray[rightIndex])
    {
      theArray[mergeLocation] = leftArray[leftIndex];
      leftIndex++;
    }
    else
    {
      theArray[mergeLocation] = rightArray[rightIndex];
      rightIndex++;
    }  // end if
    mergeLocation++;
  }  // end while

    // Finish off the first subarray, if necessary
  while (leftIndex < sizeLeft)
  {
      // At this point, leftArray is in order
    theArray[mergeLocation] = leftArray[leftIndex];
    leftIndex++;
    mergeLocation++;
  }  // end while

    // Finish off the second subarray, if necessary
  while (rightIndex < sizeRight)
  {
```

The Merge Algorithm

- **Parameters**
  - `theArray` - the array to merge
  - `start` - first item of left sorted subarray
  - `mid` - last item of the left sorted subarray
  - `end` - last item in the right sorted subarray
- **Other important variables**
  - `leftIndex` - next left subarray item
  - `rightIndex` - next right subarray item
  - `mergeLocation` - where to place next sorted value

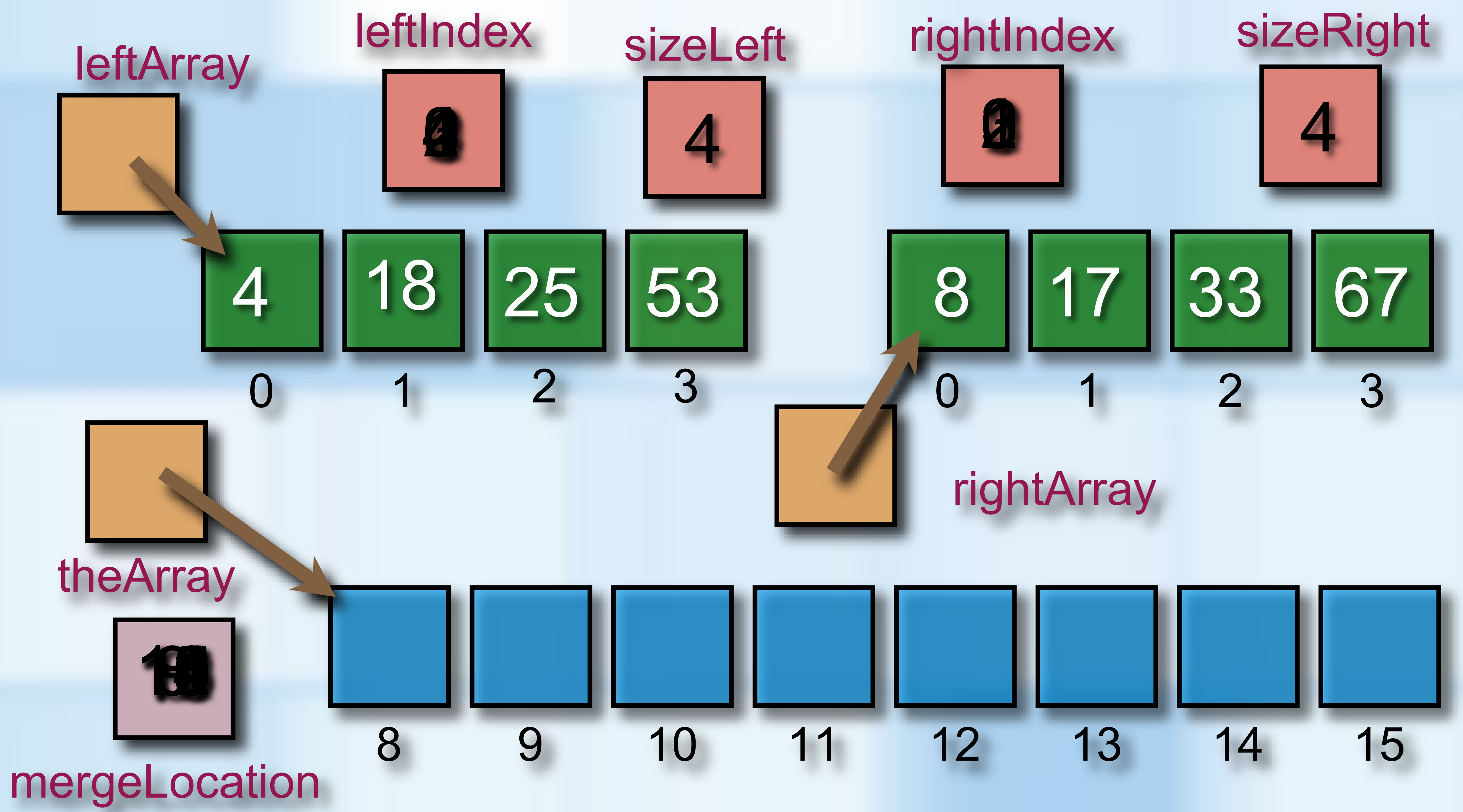



```
// Finish off the first subarray, if necessary
  while (leftIndex < sizeLeft)
  {
      // At this point, leftArray is in order
    theArray[mergeLocation] = leftArray[leftIndex];
    leftIndex++;
    mergeLocation++;
  } // end while

    // Finish off the second subarray, if necessary
  while (rightIndex < sizeRight)
  {
      // At this point, leftArray is in order
    theArray[mergeLocation] = rightArray[rightIndex];
    rightIndex++;
    mergeLocation++;
  } // end while
} // end merge
```

Pearson

The Merge Sort

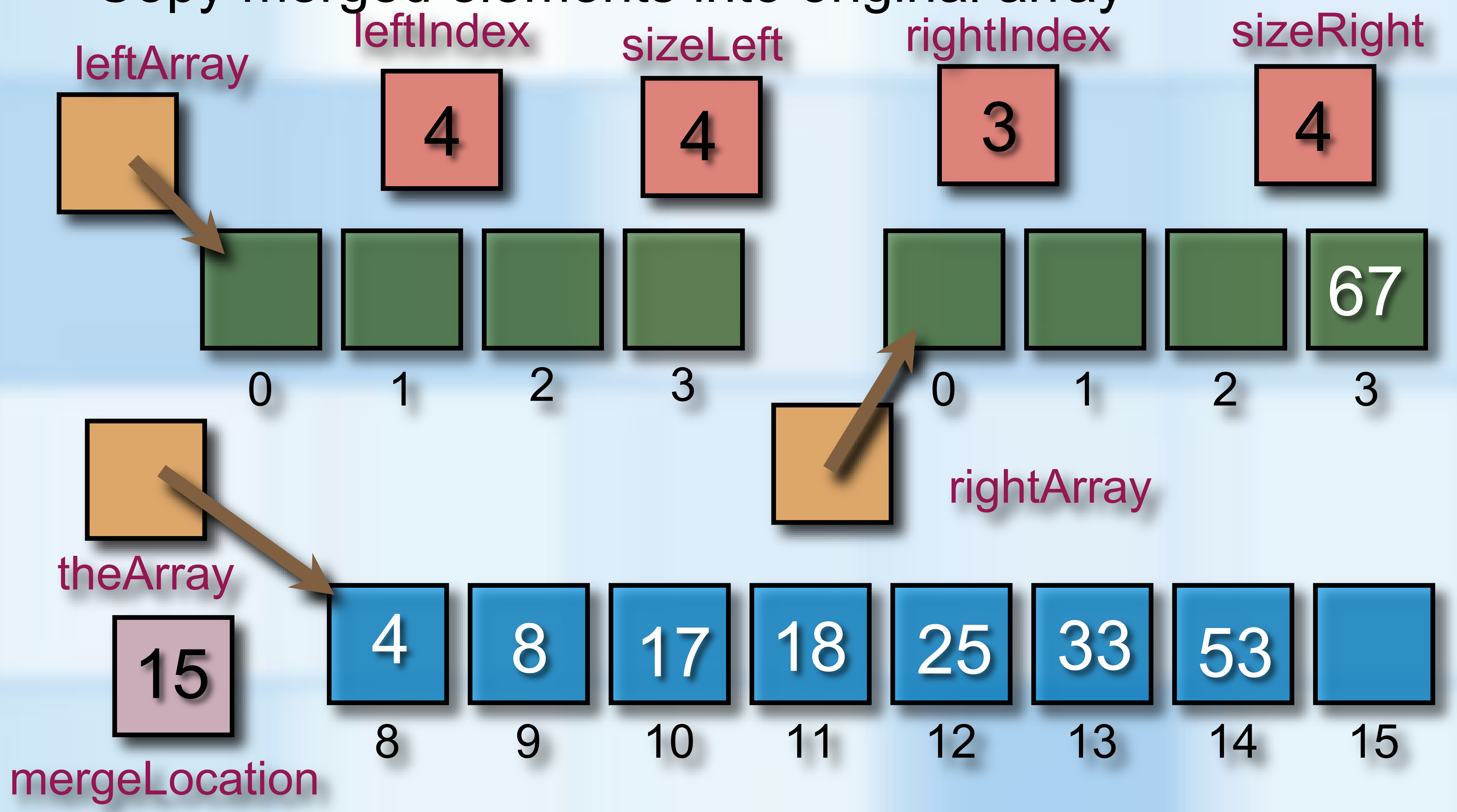
- **Merging Two Sorted Array Segments**
  - Compare first item in each array segment
  - Copy the smaller to a new array segment
  - Continue until one array segment is empty
  - Copy remaining items from other segment
  - Copy merged elements into original array




```
template <class ItemType>
void merge(ItemType theArray[], int start, int mid, int end)
{
  int sizeLeft = mid – start + 1;     // Size of left subarray
  int sizeRight = end - mid;  // Size of right subarray
  ItemType leftArray[sizeLeft];  // Temporary array
  ItemType rightArray[sizeRight]; // Temporary array

   // Move items to merge into temporary subarrays
  for (int index = 0; index < sizeLeft; index++)
    leftArray[index] = theArray[start + index];

  for (int index = 0; index < sizeRight; index++)
    rightArray[index] = theArray[mid + 1 + index];

   // While both subarrays are not empty, copy the
   // smaller item into the temporary array
  int leftIndex = 0;          // Beginning of first subarray
  int rightIndex = 0;          // Beginning of second subarray
  int mergeLocation = start;  // where to place next value

  while ((leftIndex < sizeLeft) && (rightIndex < sizeRight))
  {
     // At this point, leftArray and rightArray are in order
    if (leftArray[leftIndex ] <= rightArray[rightIndex])
    {
      theArray[mergeLocation] = leftArray[leftIndex];
      leftIndex++;
    }
    else
    {
```

The Merge Sort

- **Merging Two Sorted Array Segments**
  - Compare first item in each array segment
  - Copy the smaller to a new array segment
  - Continue until one array segment is empty
  - Copy remaining items from other segment
  - Copy merged elements into original array



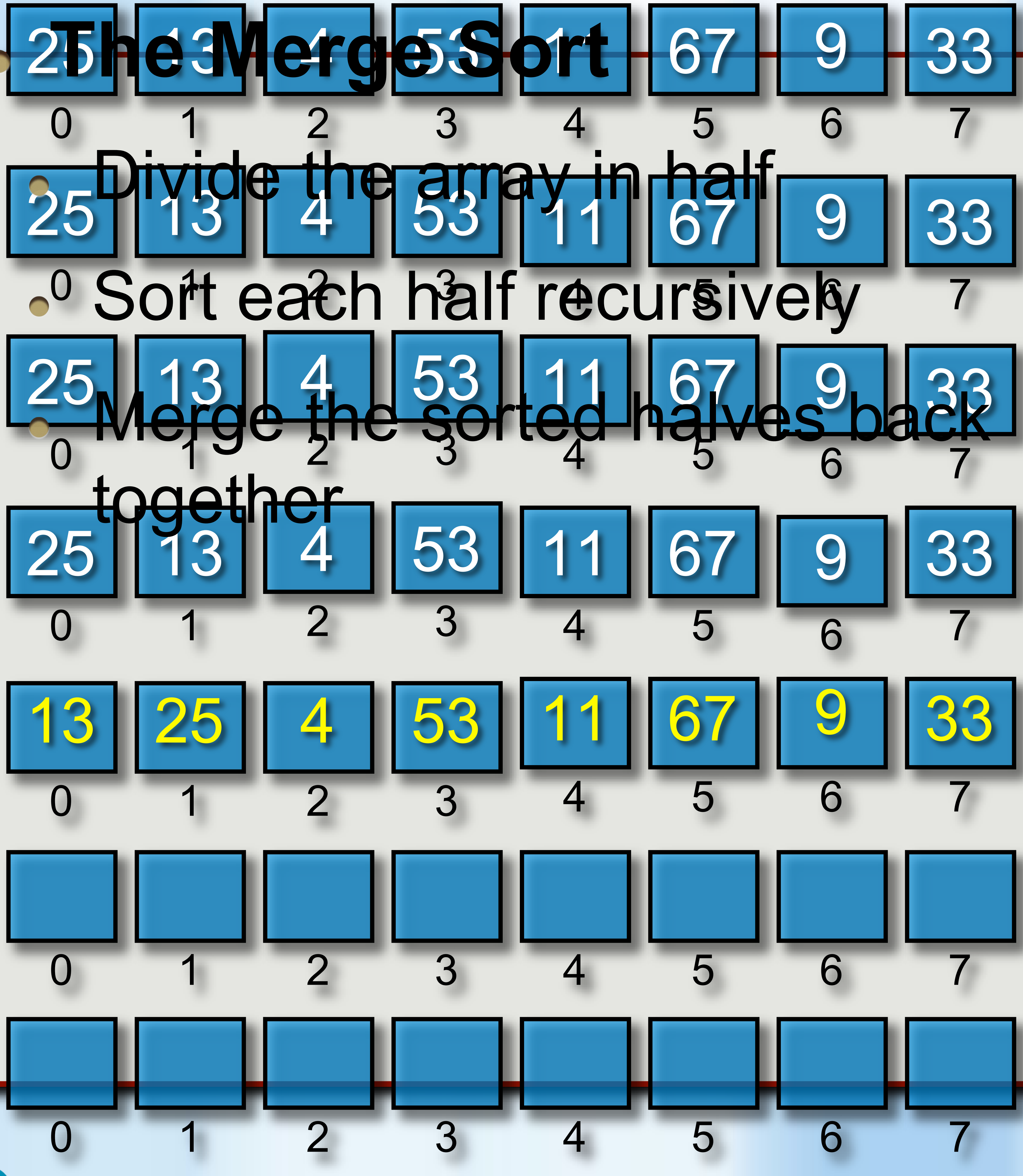

```
    rightArray[index] = theArray[mid + 1 + index];

  // While both subarrays are not empty, copy the
  // smaller item into the temporary array
int leftIndex = 0;         // Beginning of first subarray
int rightIndex = 0;         // Beginning of second subarray
int mergeLocation = start;  // where to place next value

while ((leftIndex < sizeLeft) && (rightIndex < sizeRight))
{
    // At this point, leftArray and rightArray are in order
  if (leftArray[leftIndex ] <= rightArray[rightIndex])
  {
    theArray[mergeLocation] = leftArray[leftIndex];
    leftIndex++;
  }
  else
  {
    theArray[mergeLocation] = rightArray[rightIndex];
    rightIndex++;
  }  // end if
  mergeLocation++;
}  // end while

  // Finish off the first subarray, if necessary
while (leftIndex < sizeLeft)
{
    // At this point, leftArray is in order
```

Pearson

The Merge Sort

The Merge Sort

- Divide the array in half
- Sort each half recursively
- Merge the sorted halves back together




```
void mergeSort(ItemType theArray[], int start, int end)
{
   if (start < end)
   {
      // Find midpoint
      int mid = start + (end - start) / 2;

      // Sort each half
      mergeSort(theArray, start, mid);

      mergeSort(theArray, mid + 1, end);

      // Merge the two halves
      merge(theArray, start, mid, end);

   }  // end if
}  // end mergeSort
```

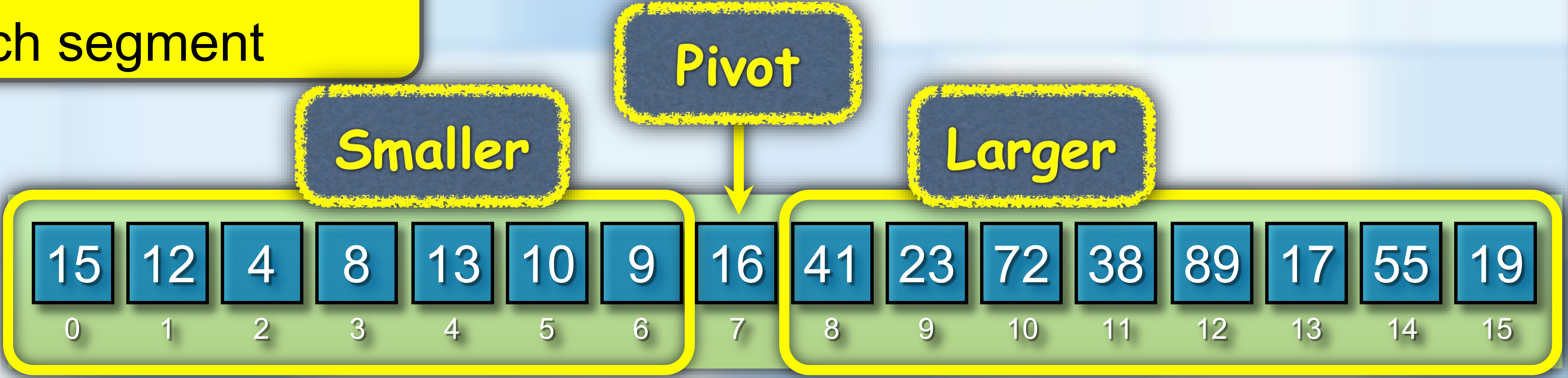
Faster Sorting Algorithms

The Quick Sort

- **Recursive divide and conquer**
- **Algorithm:**
  - Select a ***pivot*** entry
  - Rearrange array entries so that
    - Pivot is in its final sorted position
    - Entries ***smaller*** than the pivot are to its ***left***
    - Entries ***larger*** than the pivot are to its ***right***
  - Recursively sort each segment
- **Common Tasks**
  - Choosing a pivot
  - Partition array

```
Algorithm quickSort(theArray, start, end)
  // Sorts the array entries theArray[start]
  //                through a[end] recursively.
  if (start < end)

                    vot
                    he array about the pivot
                    = index of pivot
                    heArray, start, pivotIndex - 1)
                    heArray, pivotIndex + 1, end)
```

Partitioning the array



PARTITIONING

- **Choosing the pivot**
  - Pivot should be median value
  - Easier to find median of a subset of values
    - Select median of three entries
    - Sort first, middle and last elements
- **Partition the array**
  - Prepare for partitioning
  - Partition the entries
  - Move pivot into place

```
template<class ItemType>
int sortFirstMiddleLast(ItemType theArray[], int first, int last)
{
   int mid = first + (last - first) / 2;

   if (theArray[first] > theArray[mid])
      std::swap(theArray[first], theArray[mid]); // Exchange entries

   if (theArray[mid] > theArray[last])
      std::swap(theArray[mid], theArray[last]); // Exchange entries

   if (theArray[first] > theArray[mid])
      std::swap(theArray[first], theArray[mid]);// Exchange entries

   return mid;
}  // end sortFirstMiddleLast
```

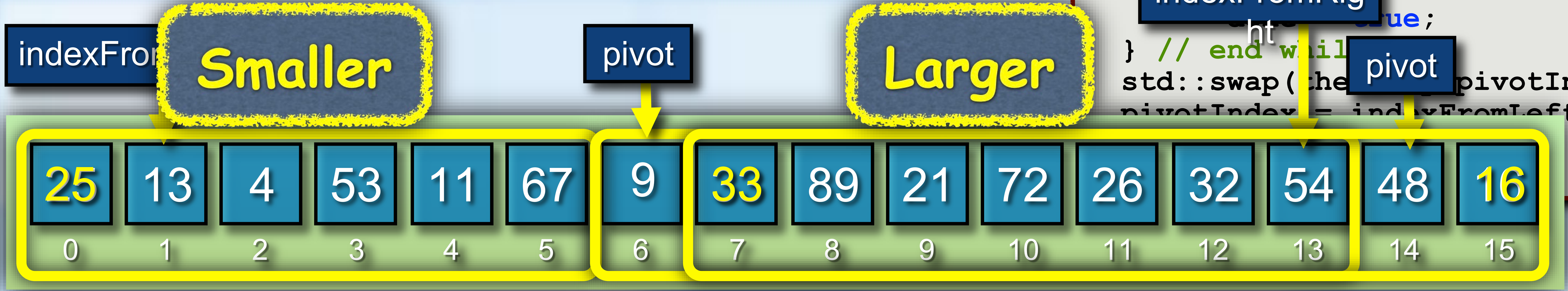
25	13	4	53	11	67	9	33	89	21	72	26	32	54	48	16
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pearson

PARTITIONING

- **Choosing the pivot**
  - Pivot should be median value
  - Easier to find median of a subset of values
    - Select median of three entries
    - Sort first, middle and last elements
- **Partition the array**
  - Prepare for partitioning
  - Partition the entries
  - Move pivot into place

```
int partition(ItemType[] theArray, int start, int end)
{
  int mid = (start + end) / 2;
  sortFirstMiddleLast(theArray, start, mid, end);
  std::swap(theArray, mid, end - 1); // move pivot
  int pivotIndex = end - 1;
  ItemType pivot = theArray[pivotIndex];

  int indexFromLeft = start + 1;
  int indexFromRight = end - 2;
  bool done = false;
  while (!done)
  {
     while (theArray[indexFromLeft] < pivot)
           indexFromLeft++;

     while (theArray[indexFromRight] > pivot)
           indexFromRight--;

     if (indexFromLeft < indexFromRight)
     {
        std::swap(theArray, indexFromLeft, indexFromRight);
        indexFromLeft++;
        indexFromRight--;
     }
        ue;
  } // end whil
  std::swap(     pivotIndex],theArray[indexFromLeft]);
```



Pearson

The Quick Sort

- The Algorithm

```
Algorithm quickSort(theArray, start, end)
  // Sorts the array entries theArray[start]
              through theArray[end] recursively.
  if (start < end)
  {
    Choose a pivot
    Partition the array about the pivot
    pivotIndex = index of pivot
    quickSort(theArray, start, pivotIndex - 1)
    quickSort(theArray, pivotIndex + 1, end)
}
```

```
void quicksort(ItemType theArray[], int start, int end)
{
  if (end - start + 1 < MIN_SIZE)
  {
    insertionSort(theArray, end, start);
  }
  else
  {
    int pivotIndex = partition(theArray, end, start);

    quicksort(theArray, end, pivotIndex - 1);
    quicksort(theArray, pivotIndex + 1, start);
  } // end if
} // end quickSort
```