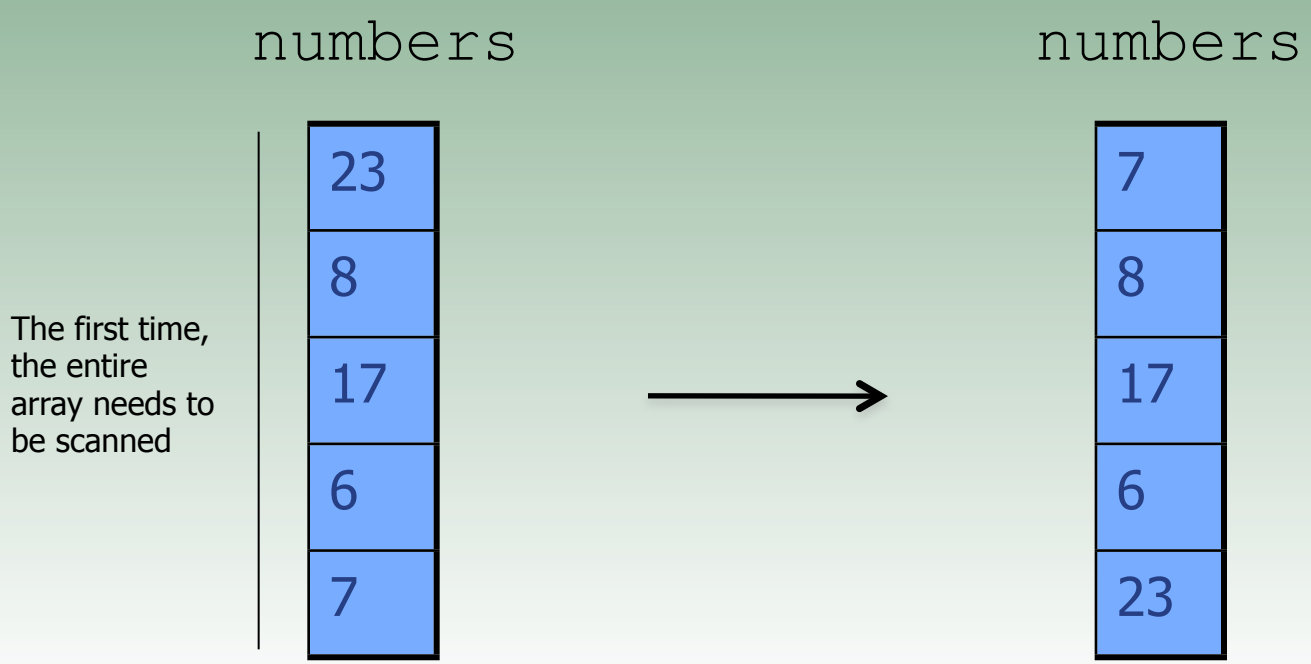
Topics in Computer Science
Sorting

Sorting an array

Selection Sort

Sorts a one dimensional array by scanning the array multiple times

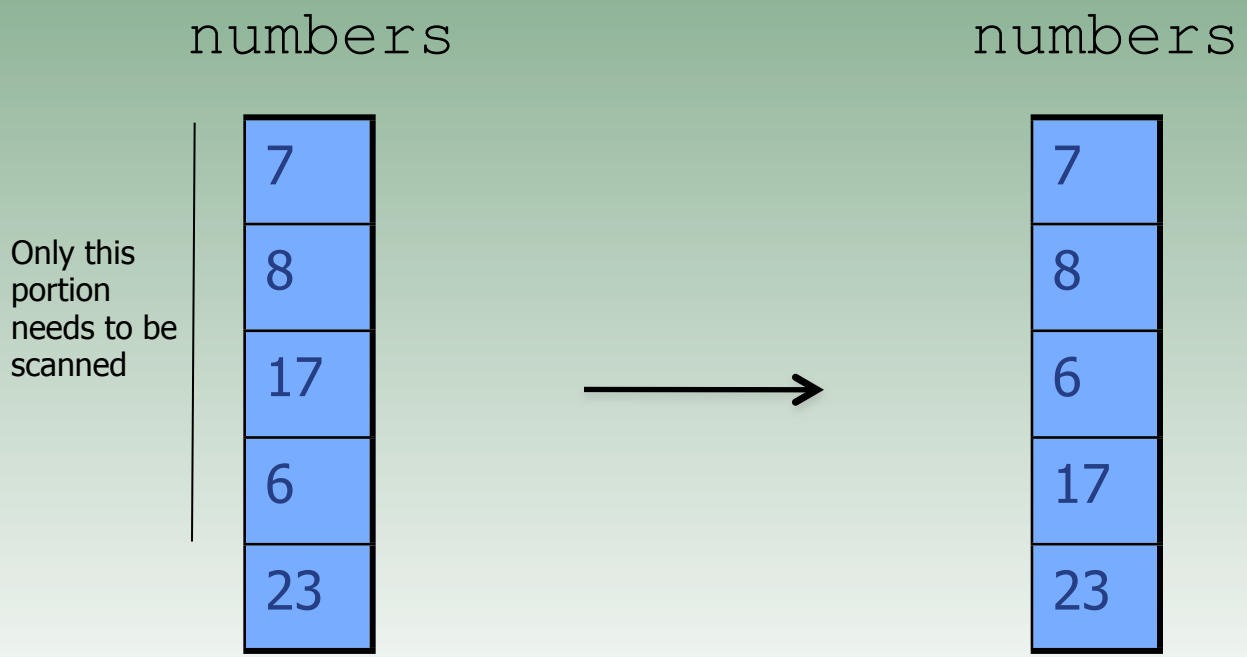
We start by locating the largest element in the array and swapping it with the last element of the array. After this, the largest element will be in the correct position



Sorting an array

Selection Sort

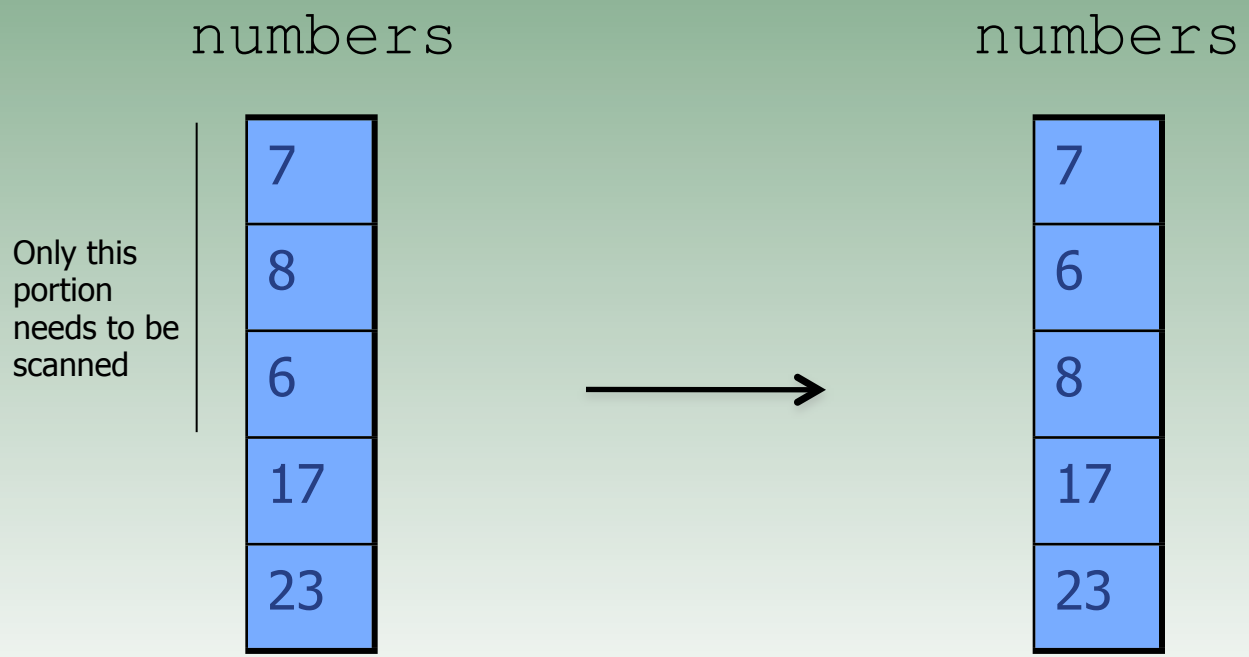
Next we locate the largest element contained in the range [0 to n - 2] in the array and we swap it with the second last element of the array. When done with this step, the second largest element is in the correct place



Sorting an array

Selection Sort

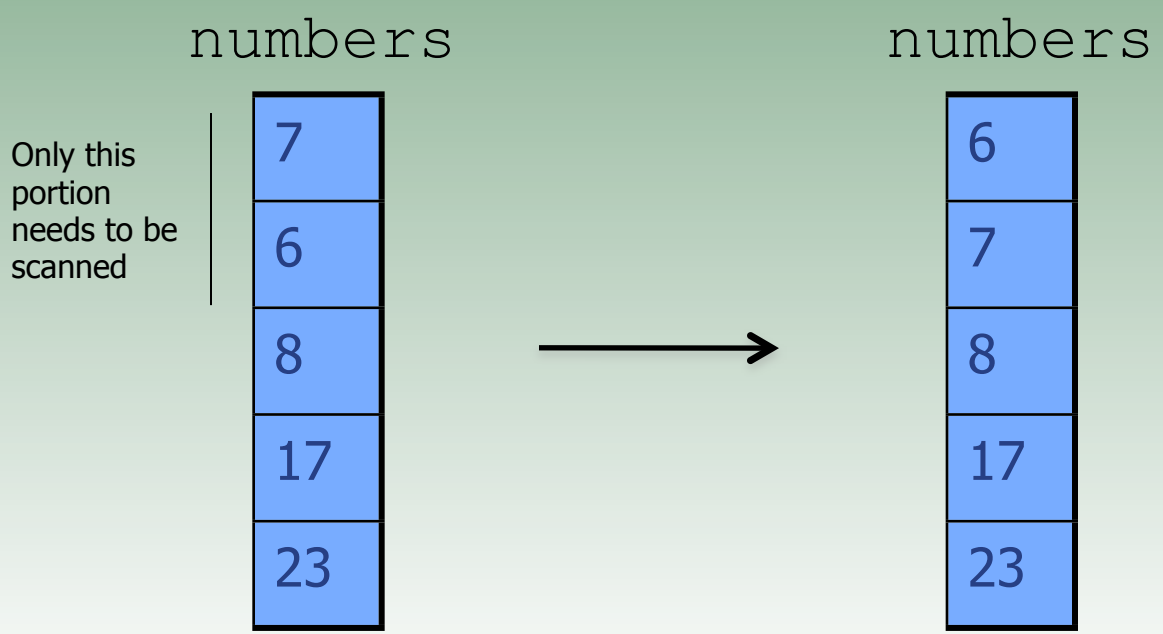
Then we locate the largest element contained in the range [0 to n - 3] in the array and we swap it with the third last element of the array. When done with this step, the third largest element is in the correct place



Sorting an array

Selection Sort

Finally, we locate the largest number in the range [0, n - 4] in the array and we swap it with the fourth last element of the array. When done with this step, the fourth largest element is in the correct place. The first element will automatically be the smallest element.



The whole process requires **n - 1** scans of the array, where n is the number of elements in the array.

Exercise 1

Implementing Selection Sort in Java

a) Create a class called `SearchAndSort`, and declare and instantiate an array of 20 numbers. Initialize the array to integers in the range [0, 50). Call the array `numbers`

Exercise 1

Implementing Selection Sort in Java

b) On the same class, write a method with the signature

```
public static void printNumbers(int[] numbers)
```

The method takes for parameter an array of integers and prints its contents.

Exercise 1

Implementing Selection Sort in Java

c) On the same class, write a method with the signature

```
public static void swapNumbers(int[] numbers, int i, int j)
```

The method takes for parameters an array of integers, and two index positions given by `i` and `j`. The method swaps the element in index `i` with the element in index `j`.

Test your method with the following code,

```
printNumbers(numbers);
swapNumbers(numbers, 4, 11);
printNumbers(numbers);
```

When the array is printed the second time, the numbers on index locations 4 and 11 should be swapped. If not, examine your code to make sure that the method is swapping correctly.

Exercise 1

Implementing Selection Sort in Java

d) On the same class, write a method with signature

```
public static int indexOfLargest(int[] list, int end)
```

The method performs a linear search on the array `list` from index position `0` to index position `end` (inclusive) and returns the index position of the largest value of that interval.

Exercise 1

Implementing Selection Sort in Java

e) On the same class, write a Java method with the signature

```
public static void selectionSort(int[] integers)
```

that that sorts the array `integers` using Selection Sort. Print your array to make sure that the sorting works properly. Use the following algorithm

algorithm

```
for i in (n - 1, n - 2, n - 3, down to 1) {
    k = location of largest element in the array portion [0 to i]
    swap element in position k with the element in position i
}
```

Note that the loop will force the searching of largest numbers to be from the 2nd element of the array to the last one. When the loop ends, the first element will automatically be the smallest.

Sorting an array

Selection Sort

This sorting algorithm requires finding the largest element of a portion of the array n - 1 times

Scan	List size	Comparisons
1	n	n - 1
2	n - 1	n - 2
3	n - 2	n - 3
4	n - 3	n - 4
5	n - 4	n - 5
...	...	...
n - 1	n - [(n - 1) - 1] = 2	n - (n - 1) = 1

Sorting an array

Selection Sort

Using the arithmetic series formula* $S_N = \frac{N}{2}(t_1 + t_N)$ the total comparisons are

$$(n-1)+(n-2)+\ldots+3+2+1=\frac{(n-1)}{2}[(n-1)+1)=\frac{n^2-n}{2}\approx n^2$$

The number of comparisons performed is directly proportional to the square of the number of elements. Since n^2 grows so much more quickly than n, the linear effect is not significant. <u>The relation is quadratic</u>

Complexity of Selection Sort algorithm is $O(n^2)$

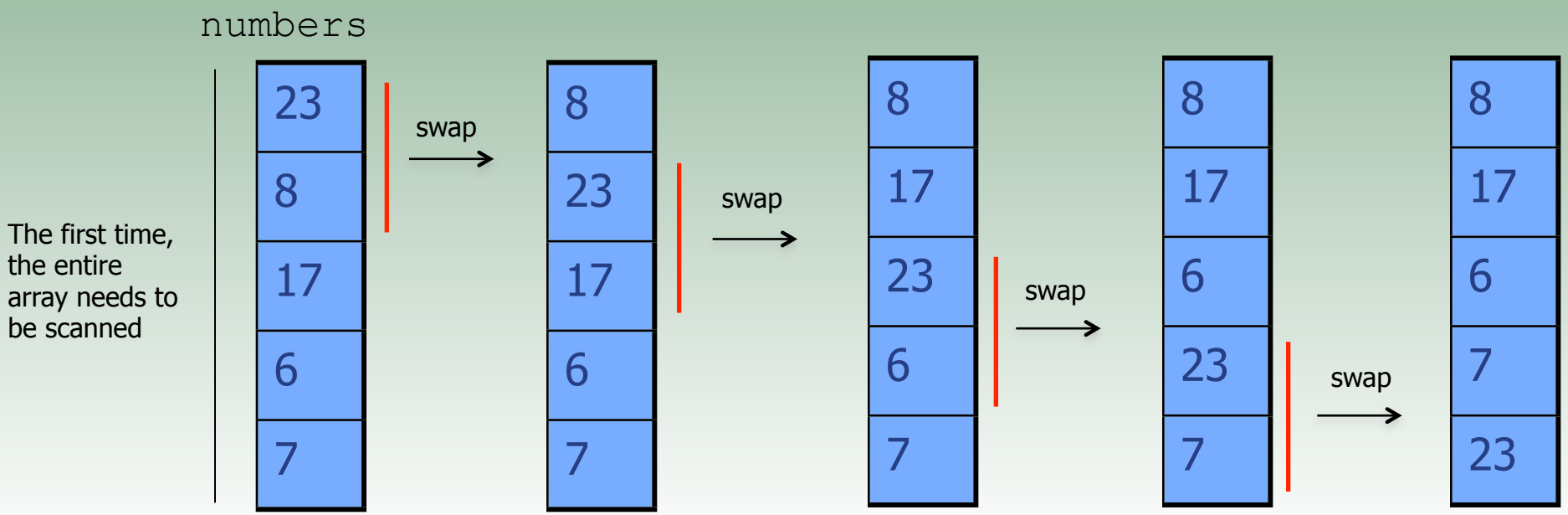
**Capital N stands for the number of elements in an arithmetic series. Lower case n stands for the number of elements in the array.*

Sorting an array

Bubble Sort

Like Selection Sort, Bubble Sort sorts a one dimensional array by scanning the array multiple times. These algorithms are known as **exchange sorts**

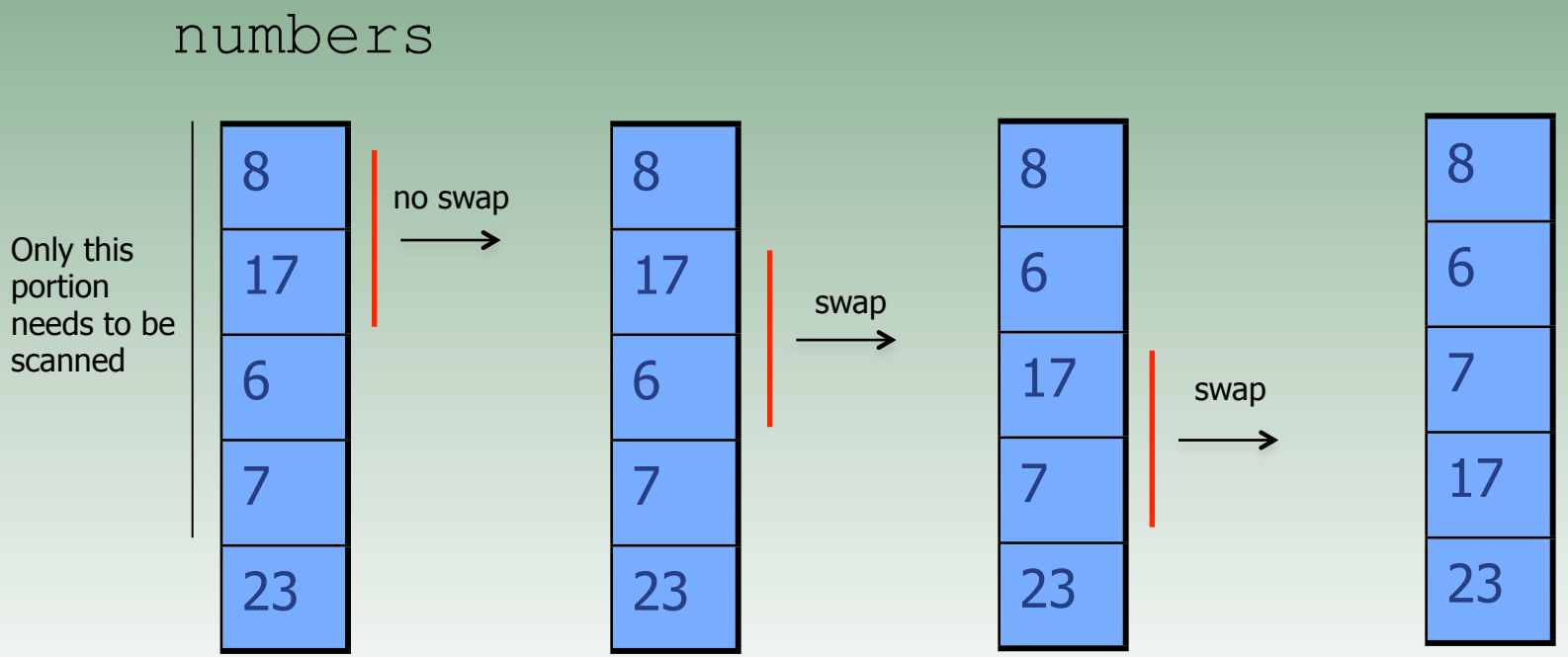
We scan the entire array, comparing adjacent elements and swapping them with each other if necessary. After the first scan, the largest element will be in the correct position.



Sorting an array

Bubble Sort

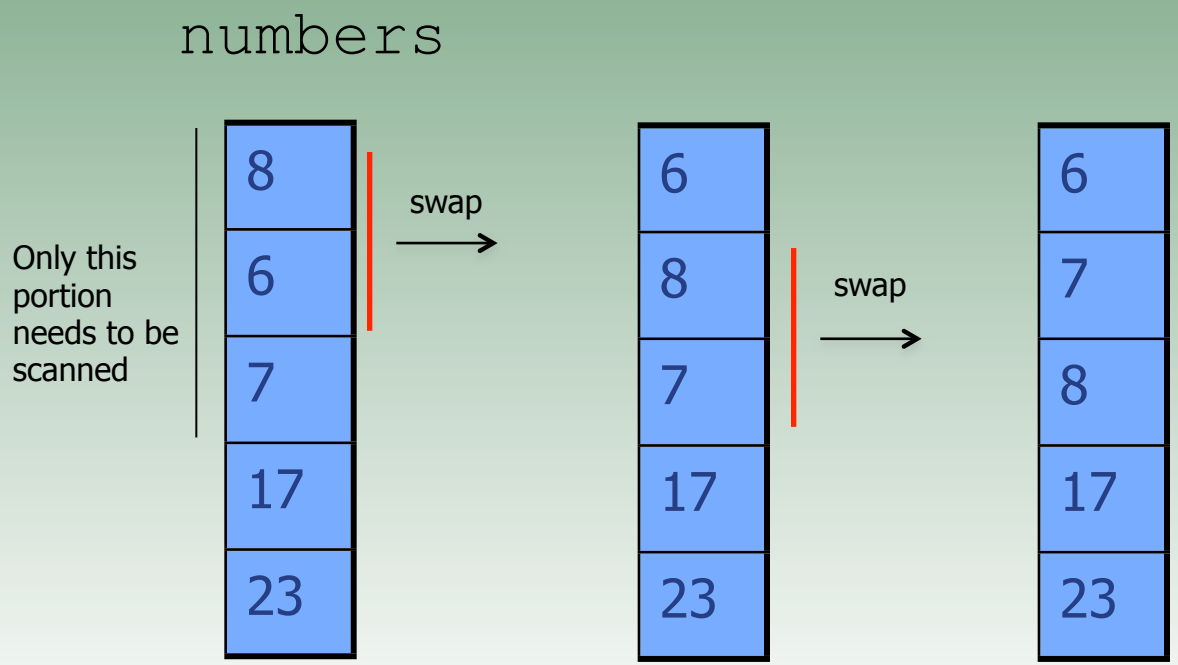
We perform the same operation, but this time we only scan the elements in the index range [0, n - 2]. After this operation, the second largest element is in the correct position.



Sorting an array

Bubble Sort

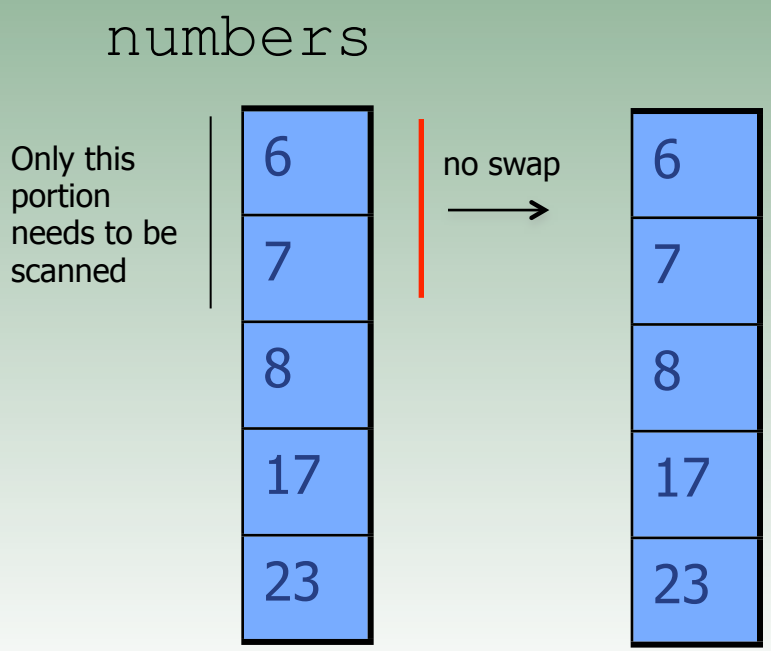
Then we scan the elements in the index range [0, n - 3]. After this operation, the third largest element is in the correct position.



Sorting an array

Bubble Sort

Finally, we scan the array number in the index [0, n - 4] When done with this step, the fourth largest element is in the correct place, and the first element will automatically be the smallest element.



The whole process requires **n - 1** scans of the array, where n is the number of elements in the array.

Exercise 2

a) The previous slides show how the array `numbers` becomes progressively sorted as elements are swapped during a Bubble Sort.

In the same way, for each of these arrays, show what each array looks like every time two adjacent elements have been compared, and whether or not there has been a swapping.

i)

87
34
42
50

ii)

72
18
21
56
39
48

iii)

6
2
2
2
2

Exercise 2

Implementing Bubble Sort in Java

b) Find your class

`SearchAndSort`

of Exercise 1.

Make sure that you already have these in working order:

- The array `numbers`
- The methods

```
printNumbers(int[] numbers)
swapNumbers(int[] numbers, int i, int j)
```

Exercise 2

Implementing Bubble Sort in Java

c) Write a Java method with the signature

```
public static void bubbleSort(int[] items)
```

that that sorts the array `items` using Bubble Sort. Print your array before and after sorting to make sure that the method works properly.

algorithm

```
for limit in (n - 1, n - 2, n - 3, down to 1) {
    for (i = 0 to limit - 1) {
        if items[i] is larger than items[i + 1] {
            swap them
        }
    }
}
```

Exercise 2

d) Write a more efficient sorting method as explained below

The `bubbleSort()` method we just wrote will continue to search the array and attempt to sort it, even if the array is already sorted. It may happen that halfway through the sorting process the array has become fully sorted.

If during a sorting pass no swapping happens, it means that the array is already sorted. We can then stop the process.

Modify the method of exercise 2 to include a mechanism to stop the `bubbleSort()` method if the array has become sorted.

In the interest of creating structured code, make these adjustments without the use of `break` statements.

Exercise 2

e) Using a similar analysis as we did for Selection Sort in previous slides, show that the number of comparisons in Bubble Sort grows as a quadratic function with respect to the number of elements in the array.

Complexity of Bubble Sort algorithm is $O(n^2)$

Sorting an array

Insertion Sort

Insertion Sort is an exchange algorithm that sorts in a way similar to the way we sort cards in a card game.

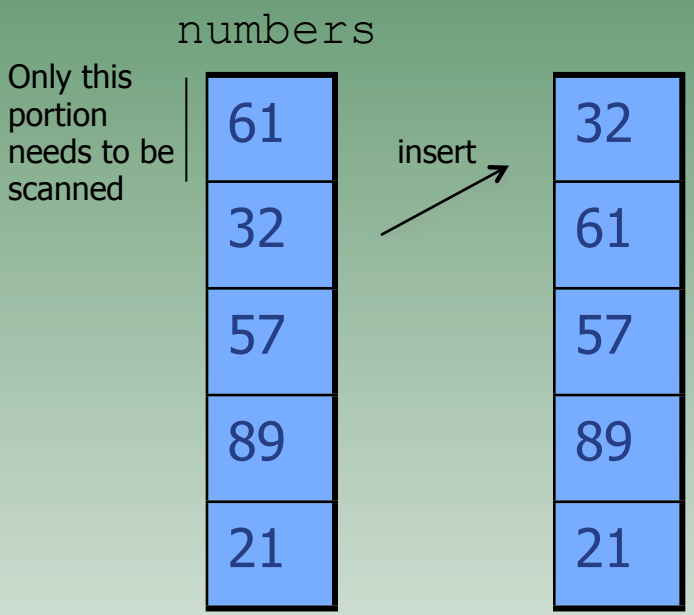
We start on item in position 1 and compare it to all items before it. (In this case just the item in position 0). If item at 1 is smaller than item at 0, we place item at 1 in a temporary variable, shift item at 0 to position 1, and then place the temporary value to item at 0. (Much like a swap)

Then, we go to item in position 2, compare it to all items before it until a larger item is found. We then place item at 2 in a temporary variable, shift all elements from the larger to position 2, then place the temporary value in the location where the larger was. (A more complicated swap)

We proceed this way until we have scanned to the end of the list. The next set of slides show this process.

Sorting an array

Insertion Sort



Starting with 32 (position 1), we scan all elements before 32 until we find a larger element. 61 is found at position 0. The sub-array

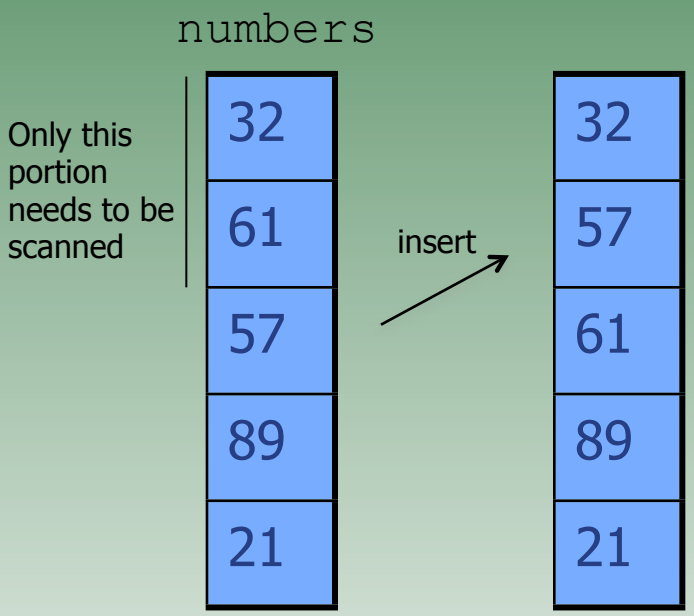
```
numbers[0] -> numbers[0]
```

is shifted down one position and 32 takes position 0

If no larger element is found, then 32 remains in its place and no shifting happens.

Sorting an array

Insertion Sort



Now we look at 57 (position 2). We scan all elements before 57 until we find a larger one. 61 is found in position 1. The sub-array

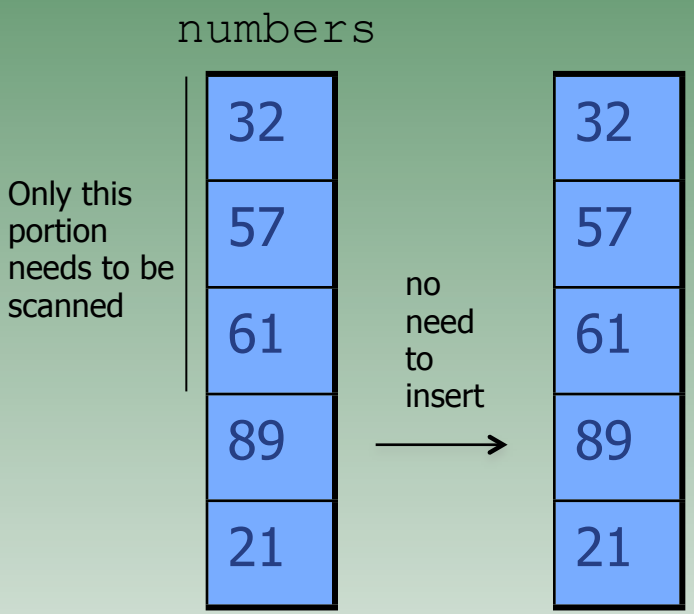
```
numbers[1] -> numbers[1]
```

is shifted down one position and 57 is inserted at position 1.

If no larger element is found, then 57 remains in place.

Sorting an array

Insertion Sort

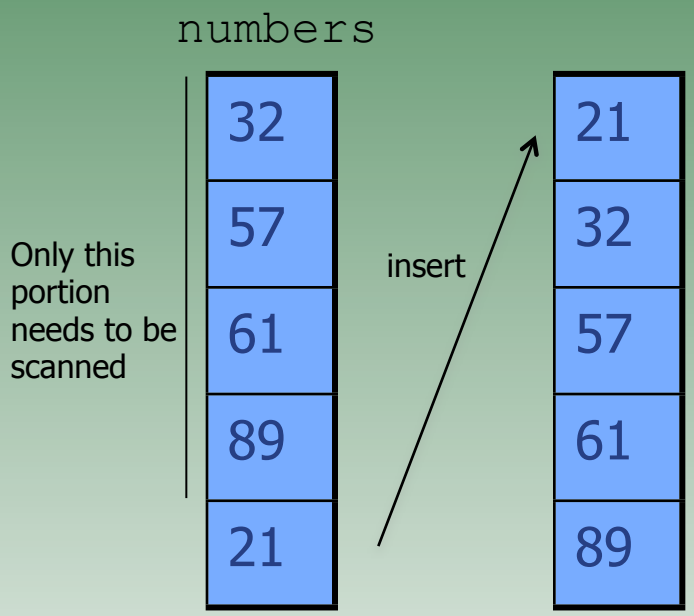


Now we look at 89 (position 3). We scan backwards all previous elements until we find a larger one.

None is found, then 89 remains in place.

Sorting an array

Insertion Sort



Now we look at 21 (position 4). We scan all elements before 21 until we find a larger one. 32 is found in position 0. The sub-array

```
numbers[0] -> numbers[3]
```

is shifted down one position and 21 is inserted at position 0.

Exercise 3a

Study the previous slides that illustrate an example of insertion sort. Then, write up instructions in English, just like the slides, that will describe insertion sort for a list of n elements.

Exercise 3b

Estimate the number of comparisons needed to execute an insertion sort on a list of n elements. Justify that the complexity of insertion sort is $O(n^2)$

Exercise 3c

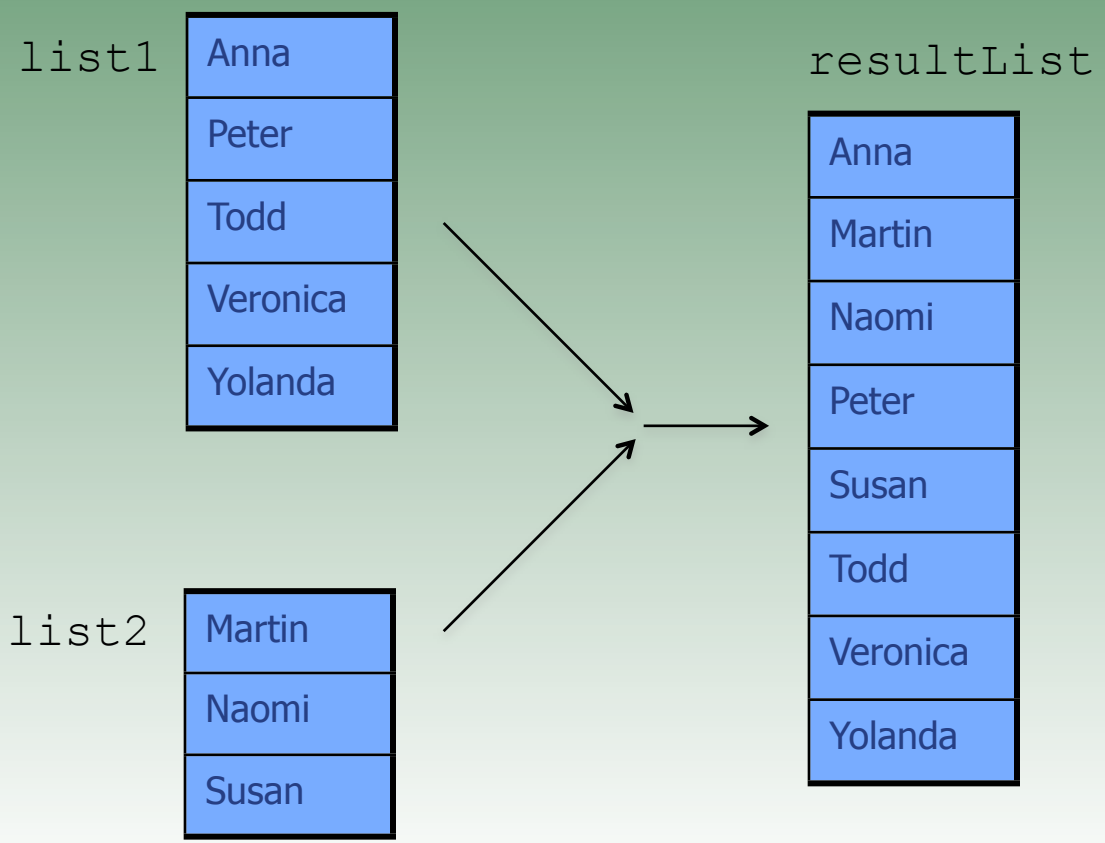
Write a Java method with the signature

```
public static void insertionSort(int[] list)
```

that performs insertion sort. Include this method in your SearchAndSort class.

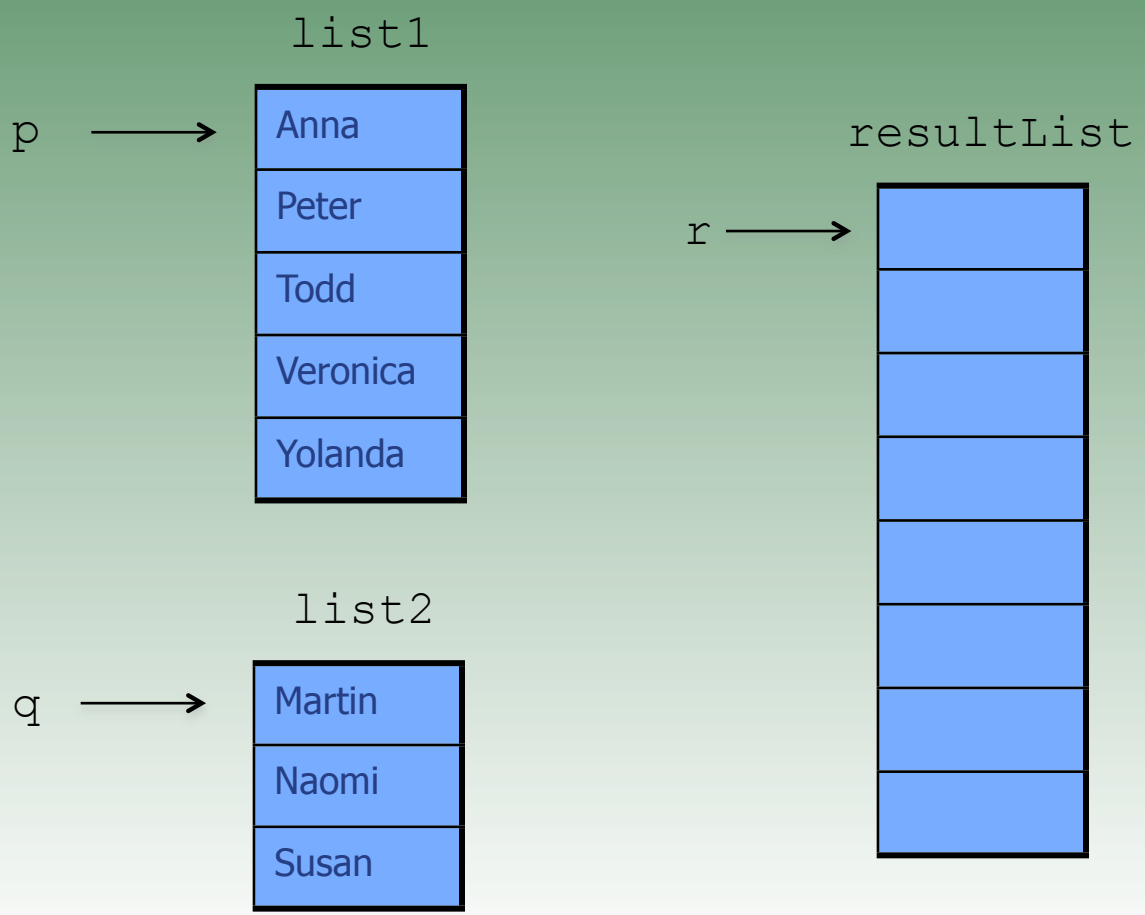
Merging Sorted Arrays

Merging sorted arrays is a process by which sorting time can be greatly reduced. The diagram belows shows how two sorted arrays are merged into a third array which will also be sorted



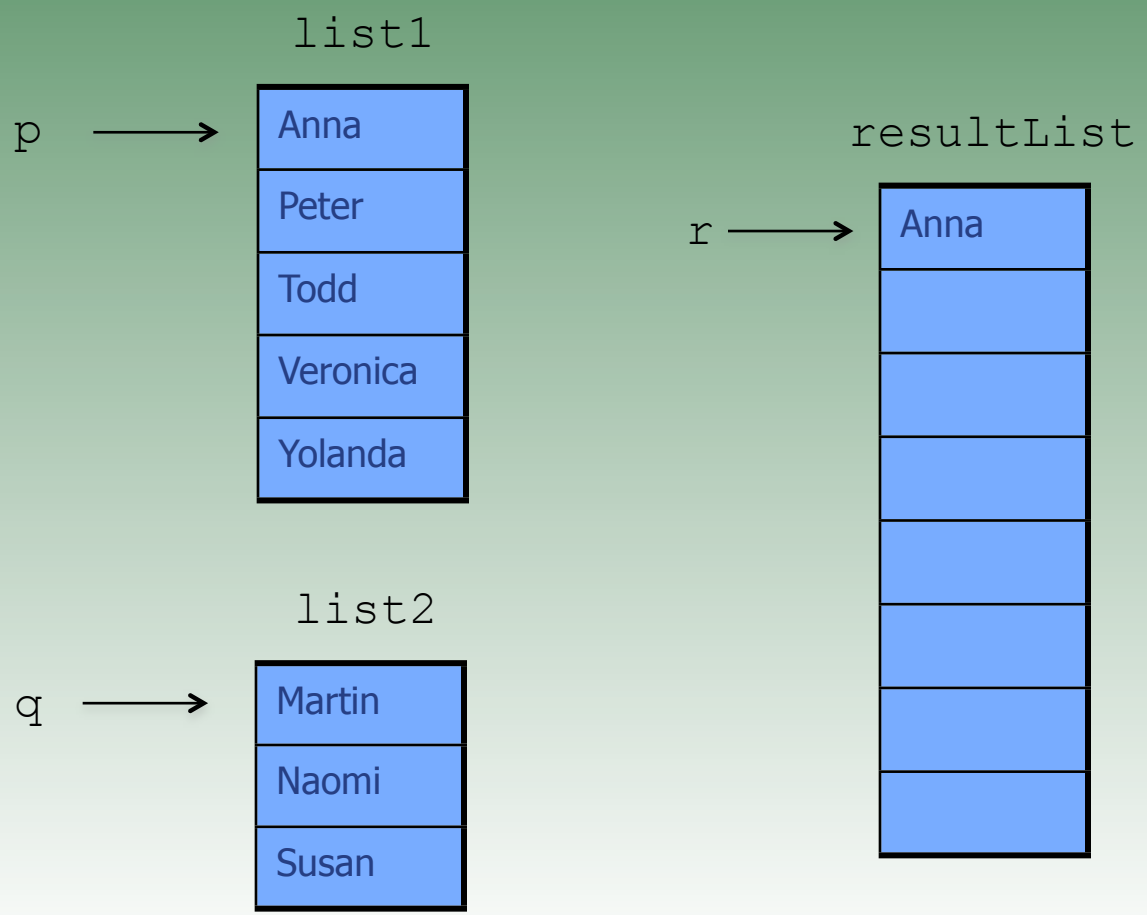
Merging Sorted Arrays

Step 1: Set three index variables `p`, `q`, `r` to the first position of arrays



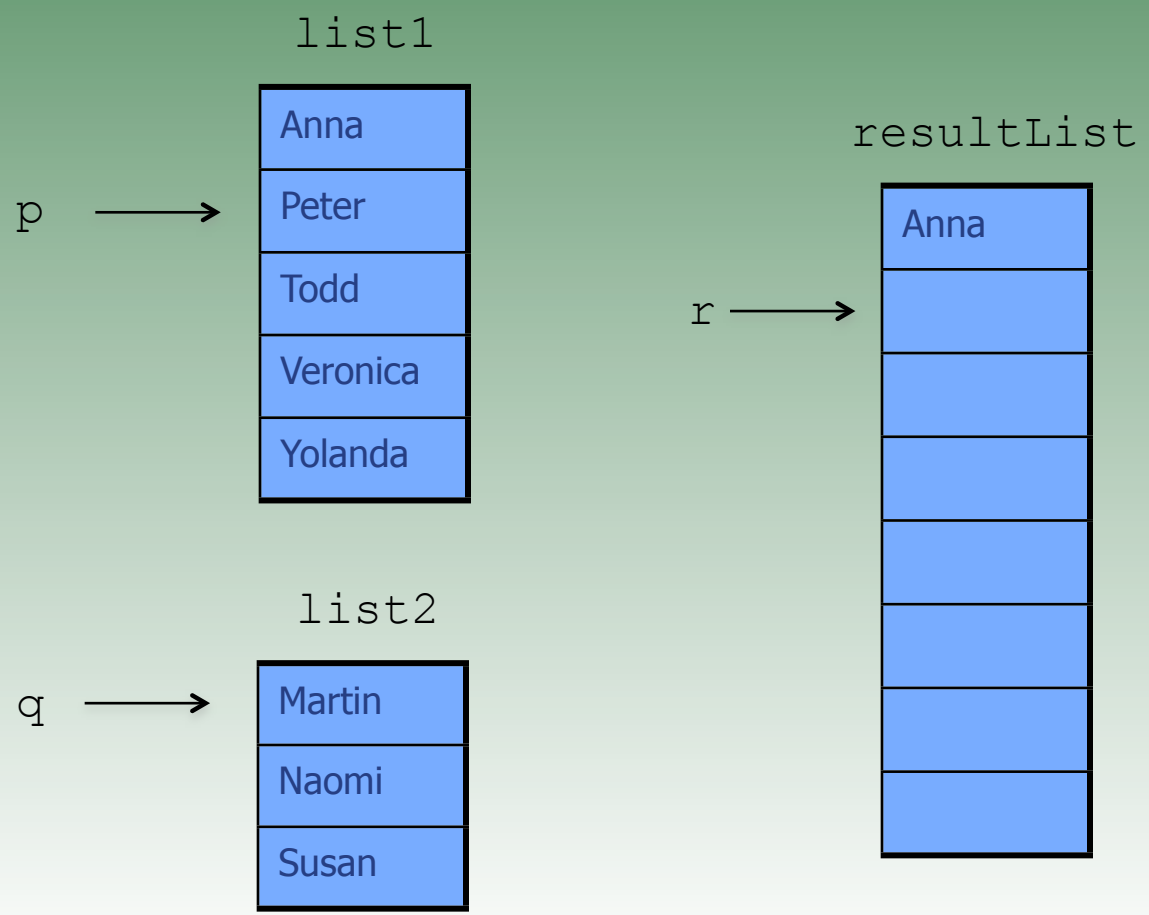
Merging Sorted Arrays

Step 2: Compare `list1[p]` and `list2[q]`. Since `list1[p]` has a lower or equal value than `list2[q]`, we copy `list1[p]` to the `resultList`, at position `r`



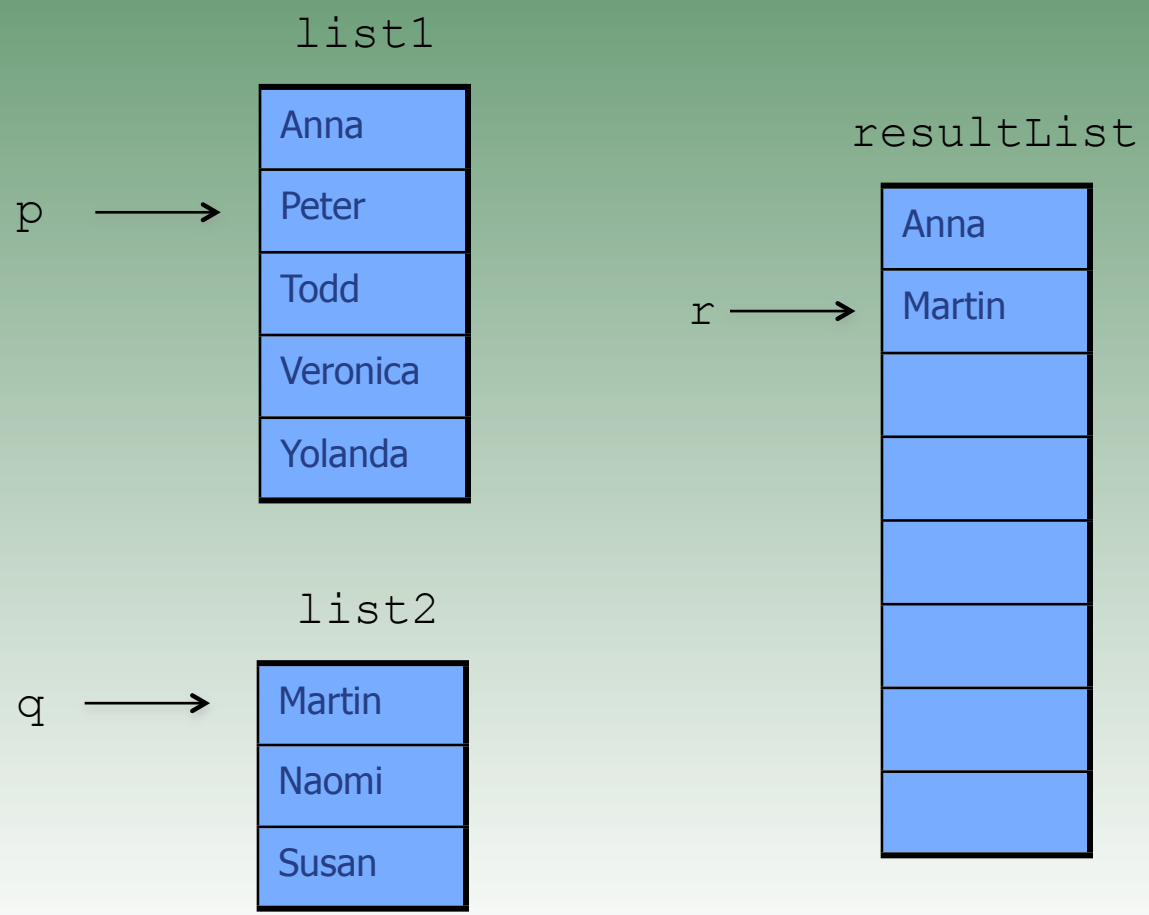
Merging Sorted Arrays

Step 3: We move `p` and `r` to the next position



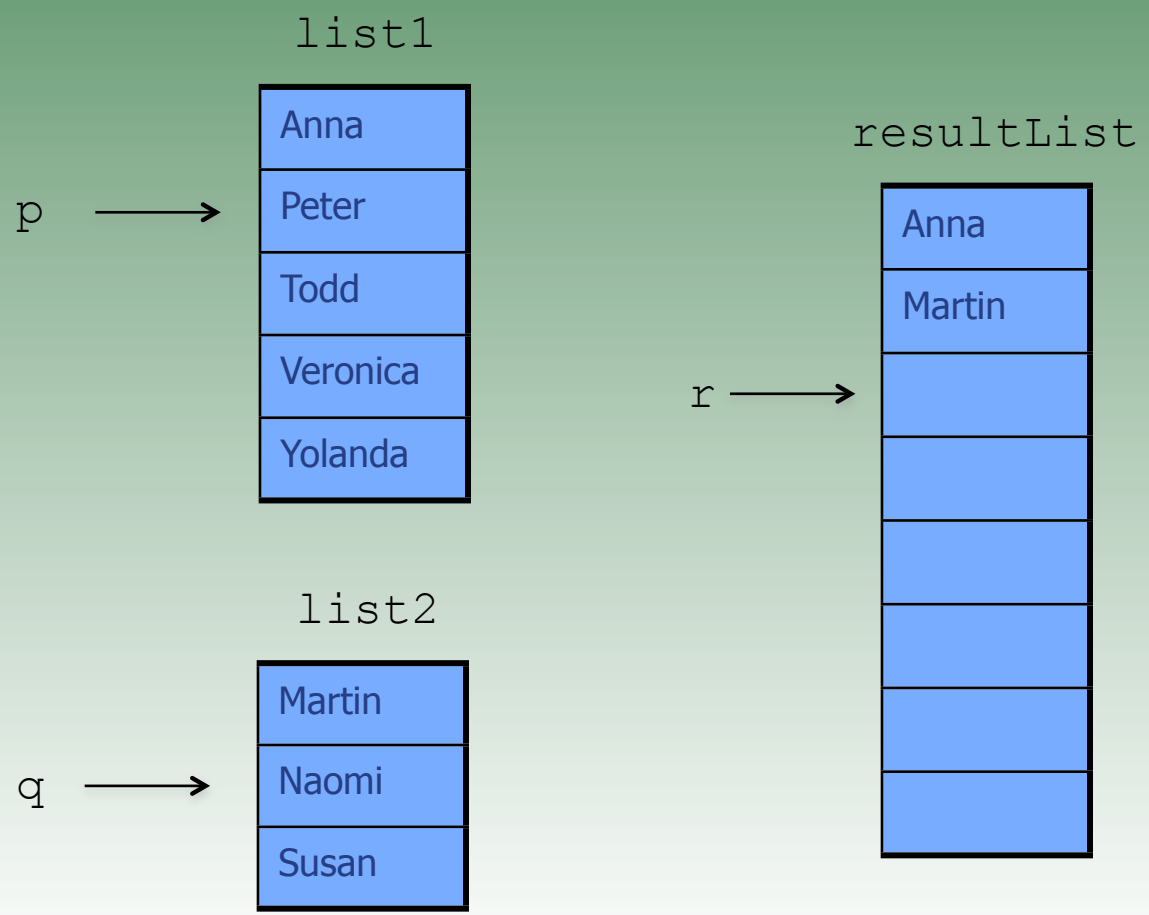
Merging Sorted Arrays

Step 4: Compare `list1[p]` and `list2[q]`. Since `list1[p]` has a larger value than `list2[q]`, we copy `list2[q]` to the `resultList`, at position `r`



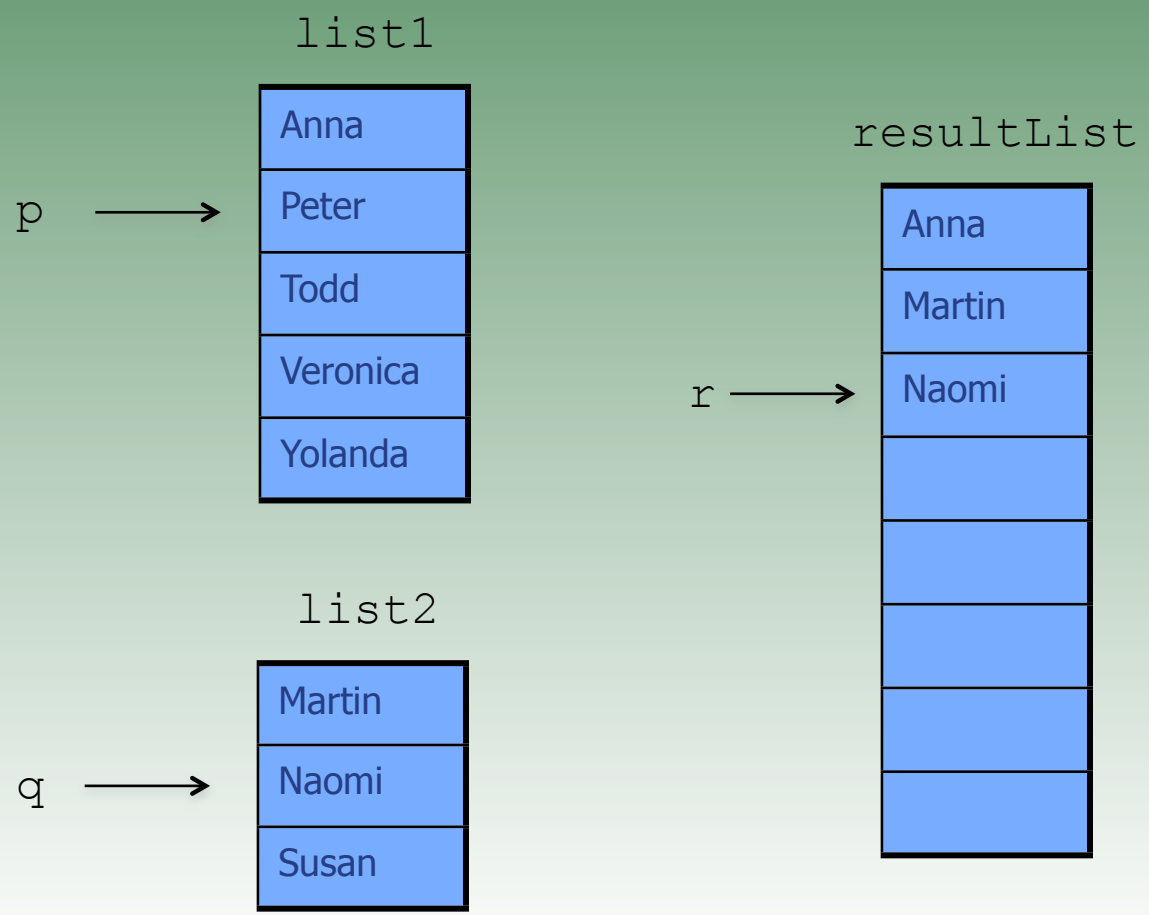
Merging Sorted Arrays

Step 5: We move `q` and `r` to the next position



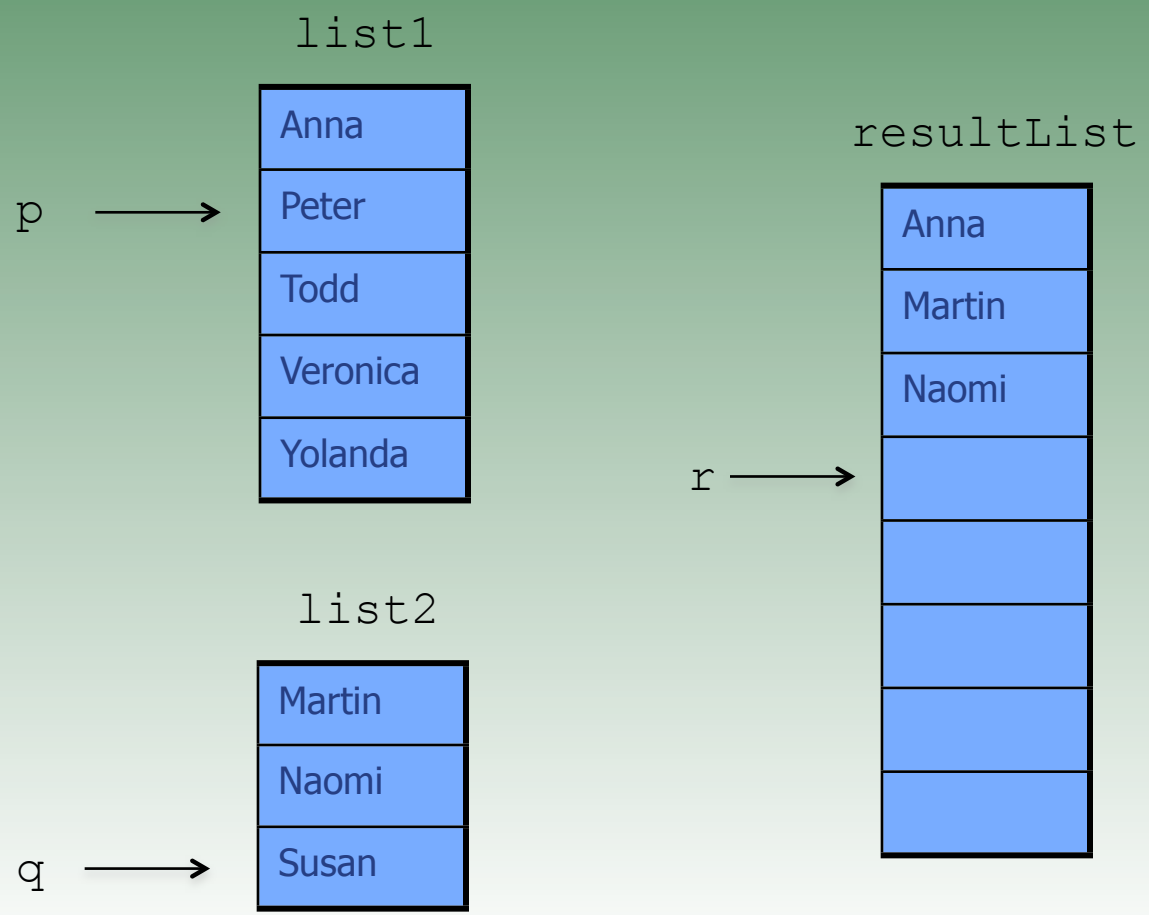
Merging Sorted Arrays

Step 6: Compare `list1[p]` and `list2[q]`. Since `list1[p]` has a larger value than `list2[q]`, we copy `list2[q]` to the `resultList`, at position `r`



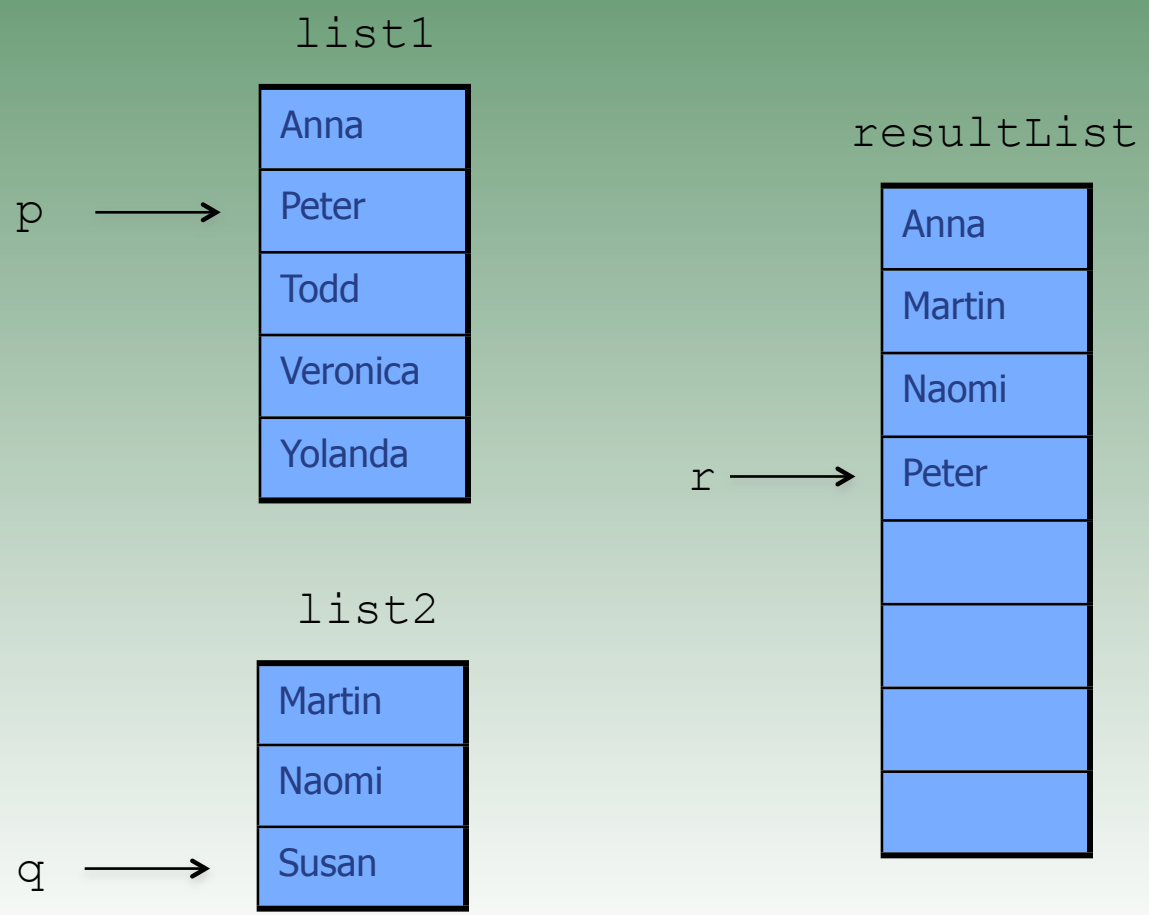
Merging Sorted Arrays

Step 7: We move `q` and `r` to the next position



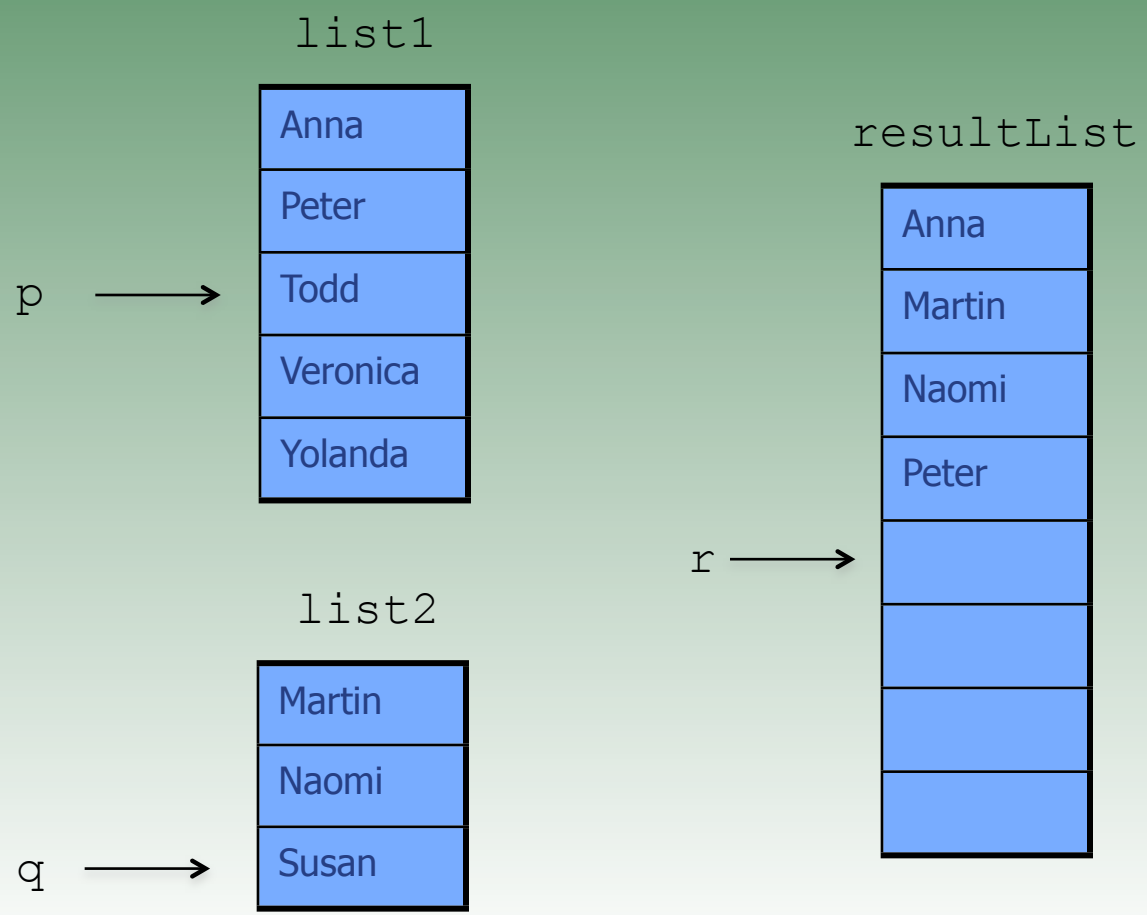
Merging Sorted Arrays

Step 8: Compare `list1[p]` and `list2[q]`. Since `list1[p]` has a lower or equal value than `list2[q]`, we copy `list1[p]` to the `resultList`, at position `r`



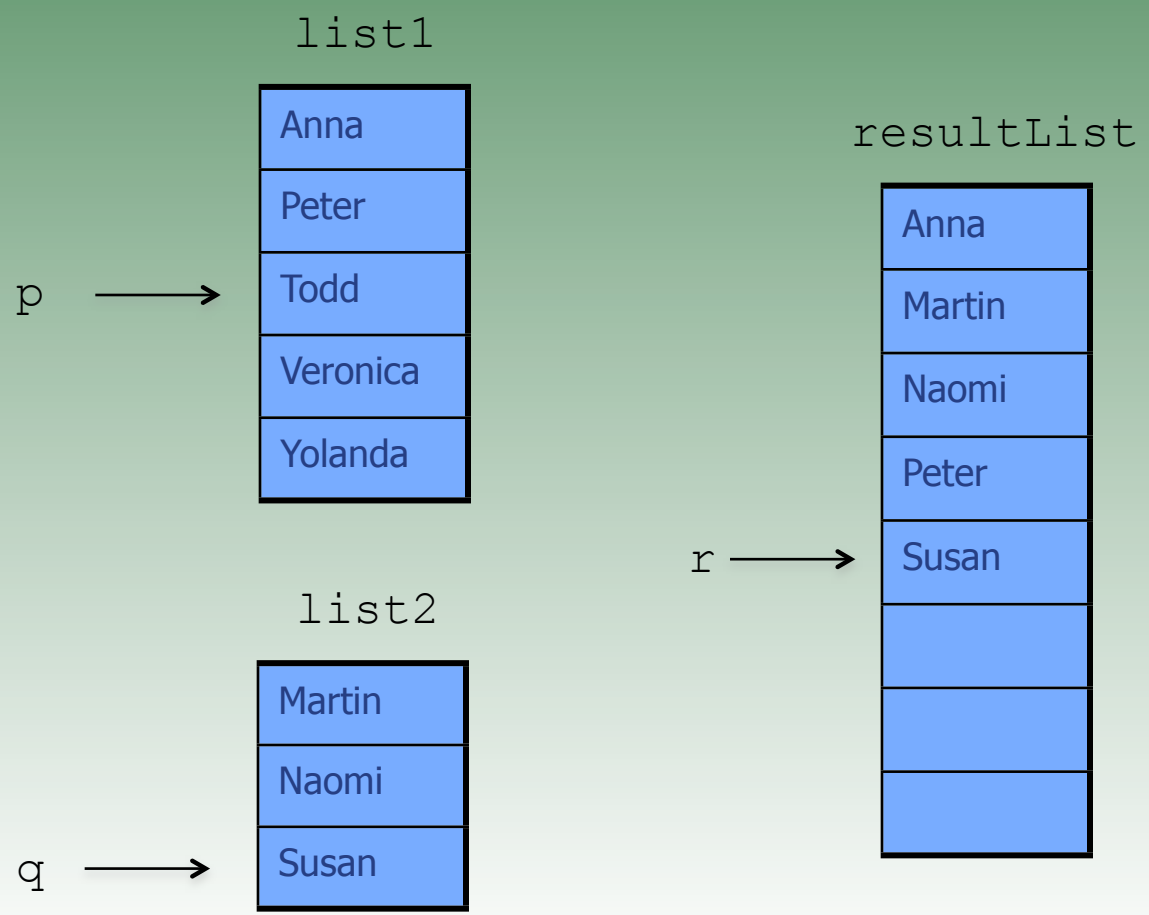
Merging Sorted Arrays

Step 9: We move `p` and `r` to the next position



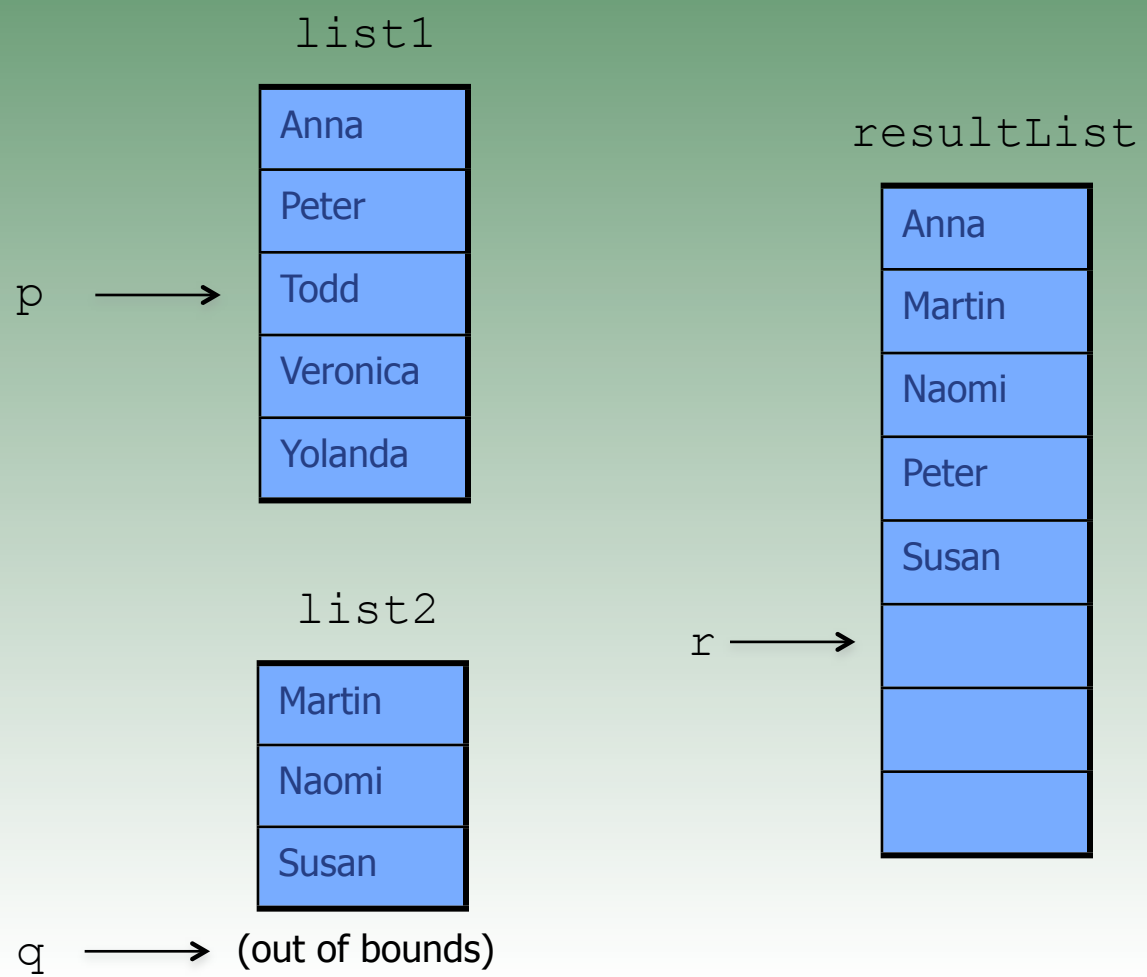
Merging Sorted Arrays

Step 10: Compare `list1[p]` and `list2[q]`. Since `list1[p]` has a larger value than `list2[q]`, we copy `list2[q]` to the `resultList`, at position `r`



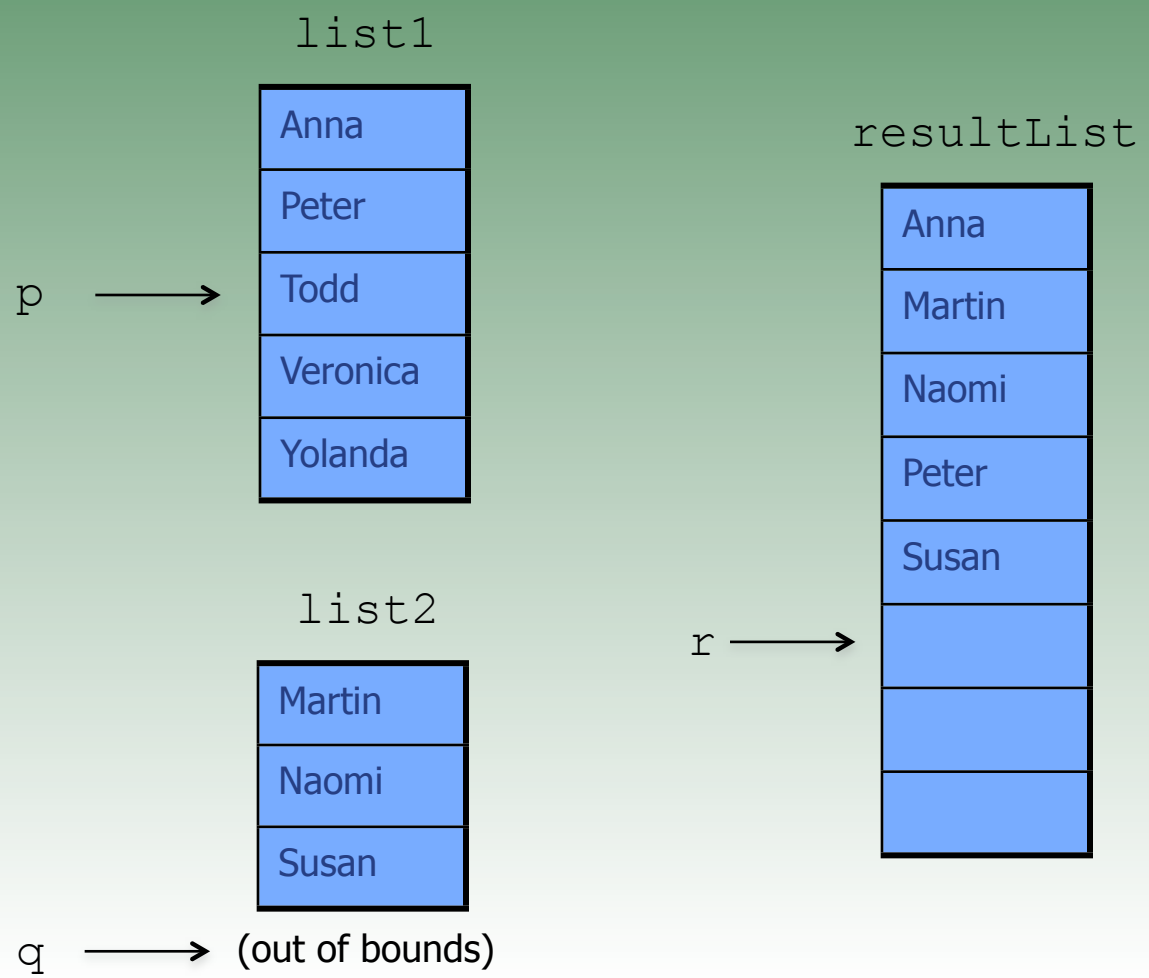
Merging Sorted Arrays

Step 11: We move `q` and `r` to the next position



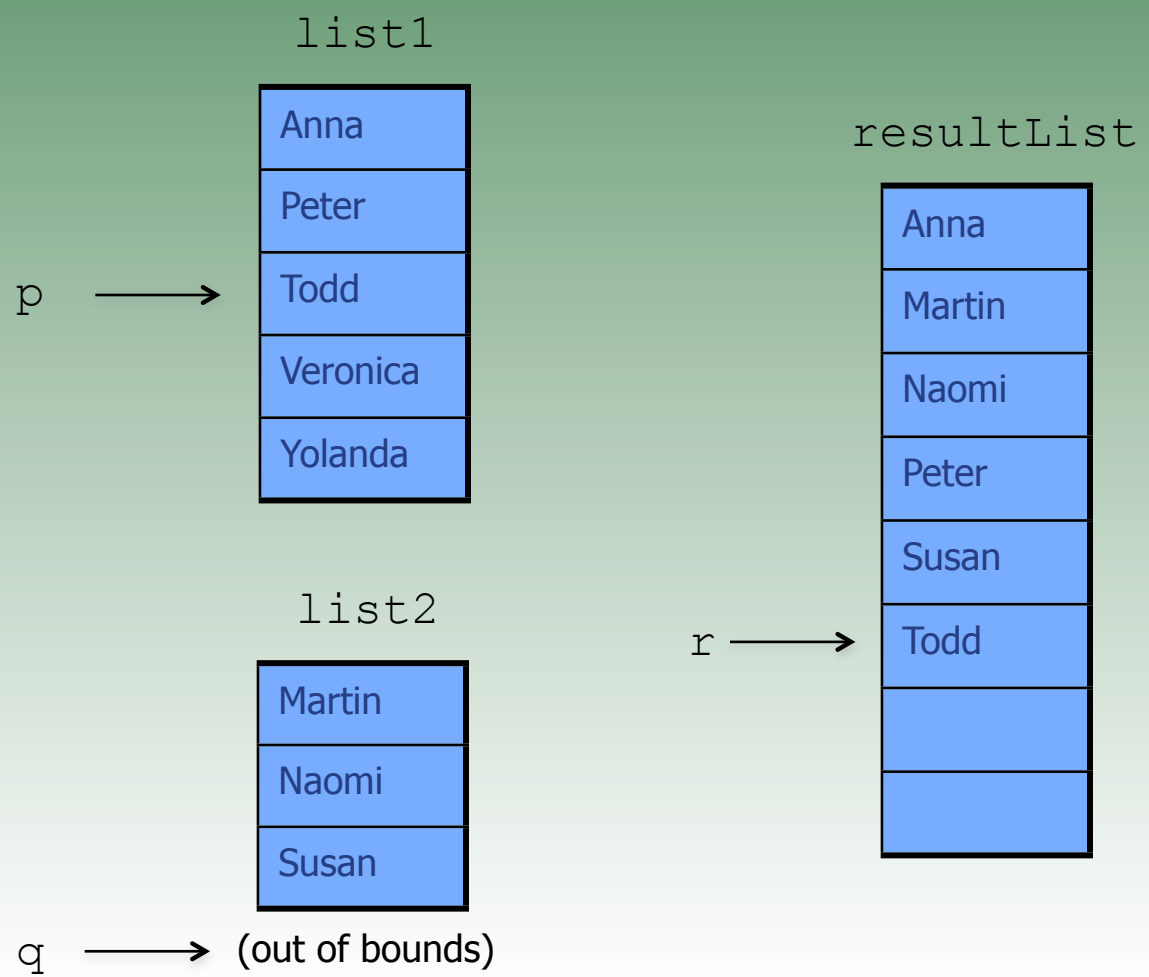
Merging Sorted Arrays

Step 12: We cannot compare `list2[q]` because there are no more elements left to compare in that list. Therefore, the remaining elements of `list1` are copied to `resultList`



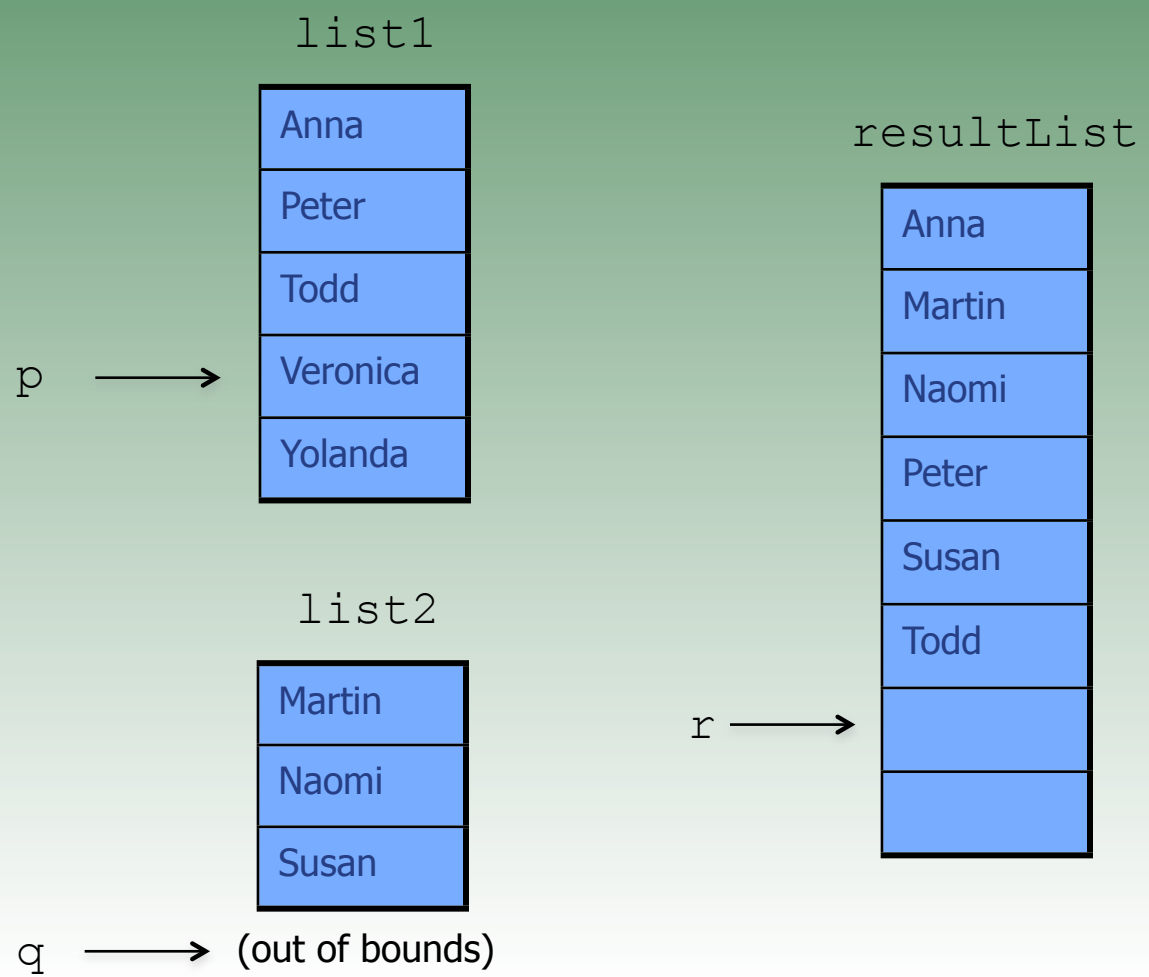
Merging Sorted Arrays

Step 13: Copy `list1[p]` to `resultList[r]`



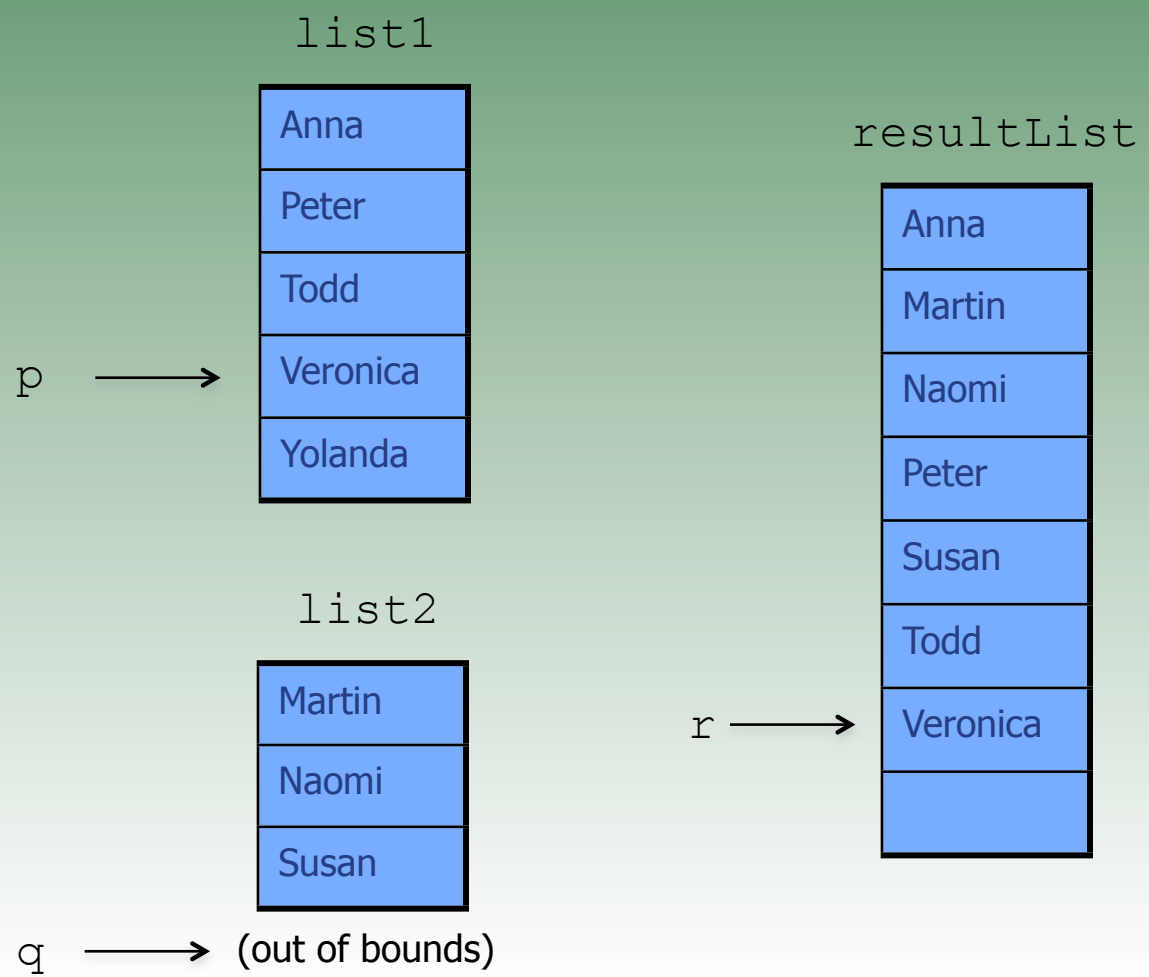
Merging Sorted Arrays

Step 14: Move `p` and `r` to the next position



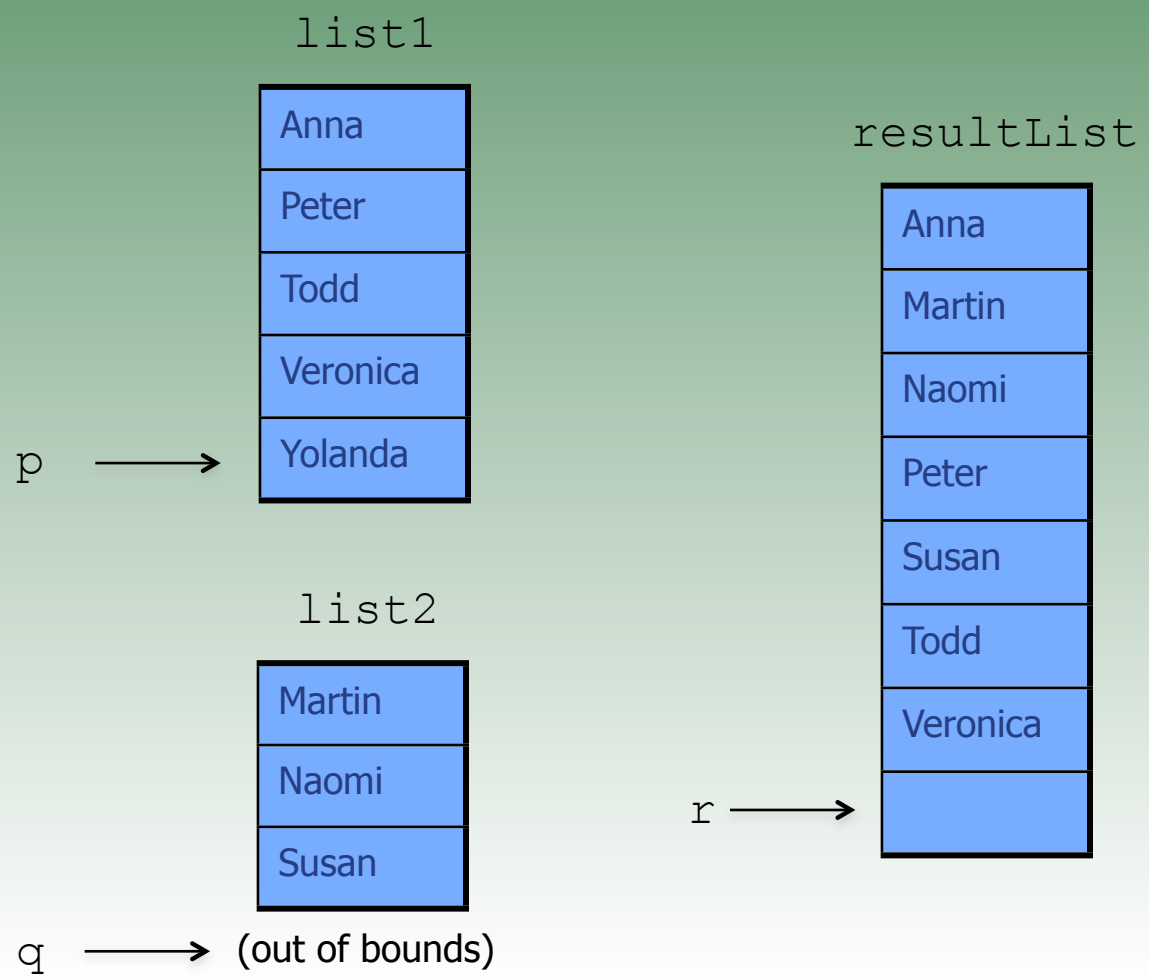
Merging Sorted Arrays

Step 15: Copy `list1[p]` to `resultList[r]`



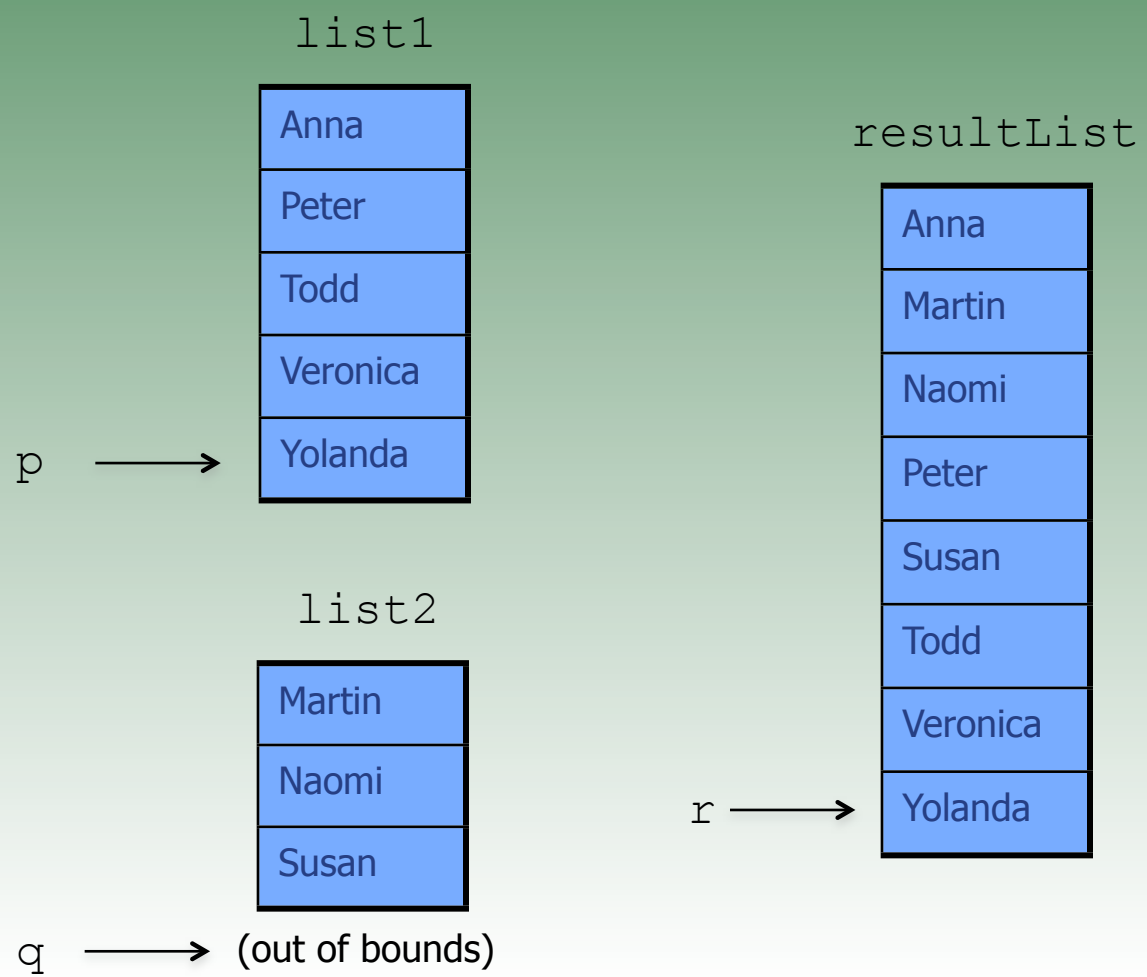
Merging Sorted Arrays

Step 16: Move `p` and `r` to the next position



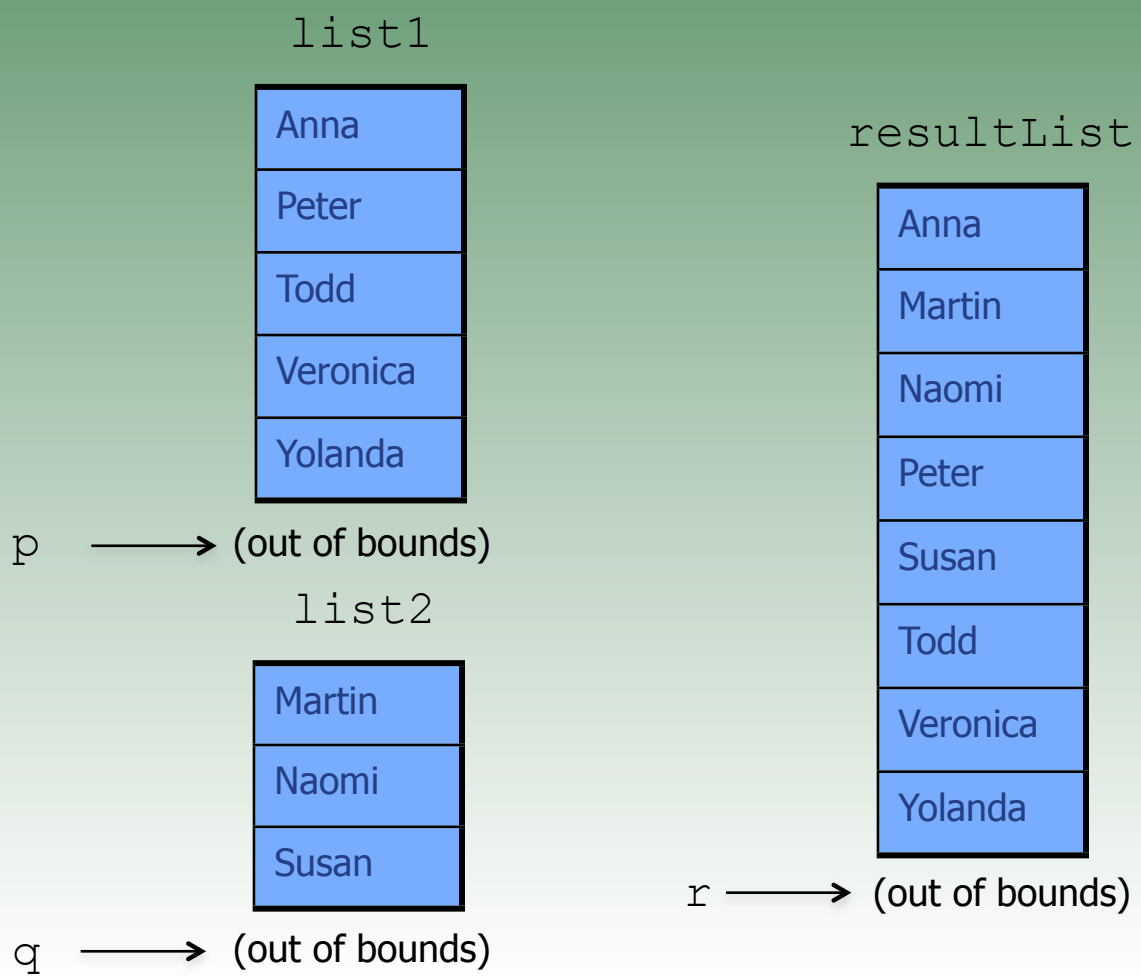
Merging Sorted Arrays

Step 17: Copy `list1[p]` to `resultList[r]`



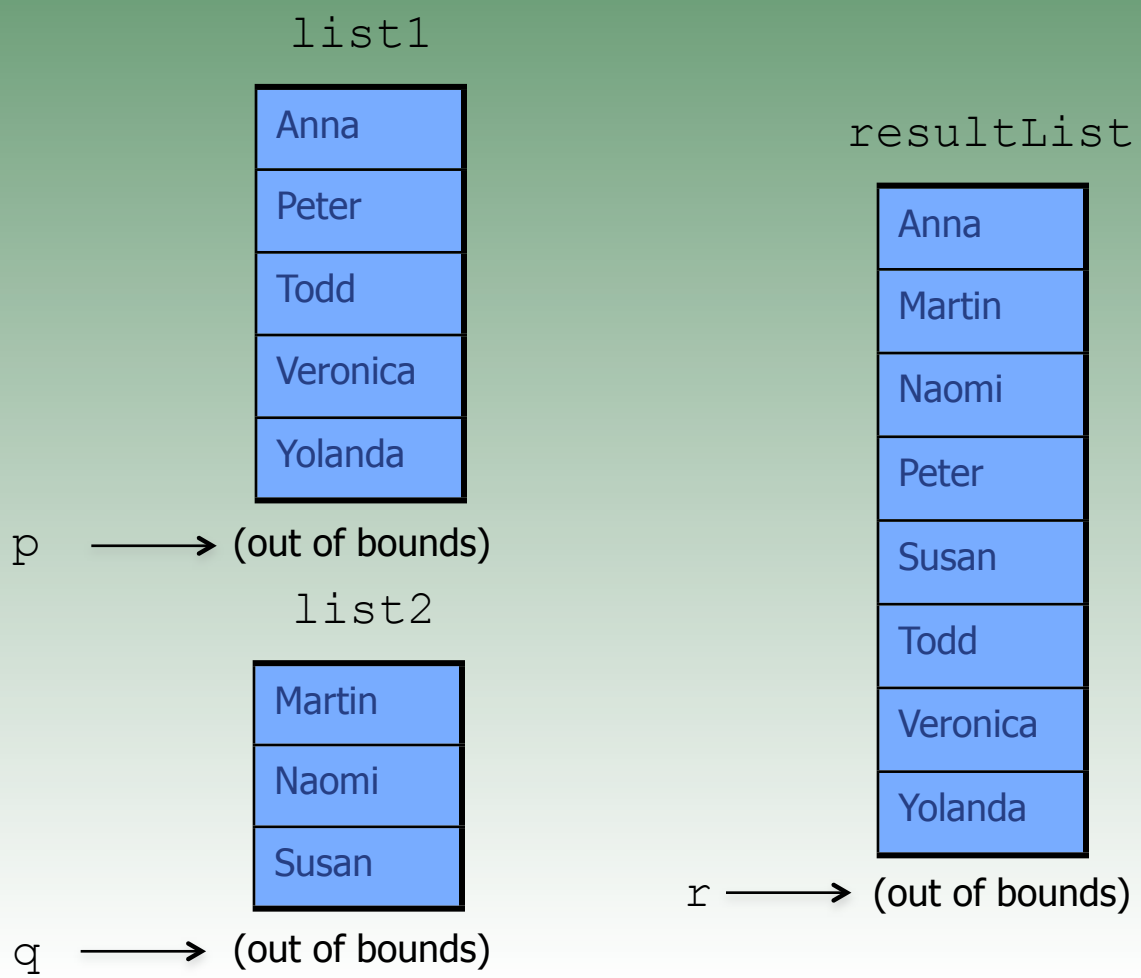
Merging Sorted Arrays

Step 18: Move `p` and `r` to the next position



Merging Sorted Arrays

Step 19: Return `resultList`



Merging Sorted Arrays

When writing a Java program, the process of merging needs to happen if and only if there is at least one element in one of the lists to merge

The cases to consider are:

1. The first list is empty or its end is reached: All remaining elements of the second list go to the result list

2. The second list is empty or its end is reached: All remaining elements of the first list go to the result list

3. The element to be copied to the result list comes from the list that currently has the lower (or equal) value

Exercise 4

Write a Java method with the signature

```
public static String[] mergeLists(String[] list1, String[] list2)
```

that merges two sorted lists: `list1` and `list2`, and returns the resulting list

Use the algorithm of the next slide

Include your method in the class `SearchAndSort`

Exercise 4 - Continued

algorithm

```
set p to first index location of list1
set q to first index location of list2
set r to first index location of resultList

while (p or q are within their lists) {

    if p has reached the end of list1 then
        copy the q element of list2 to resultList[r]
        advance q to the next index position in list2
    else if q has reached the end of list2 then
        copy the p element of list1 to resultList[r]
        advance p to the next index position in list1
    else if the p element of list1 is less or equal to the q element of list2
        copy the p element of list1 to resultList[r]
        advance p to the next index position in list1
    else
        copy the q element of list2 to resultList[r]
        advance q to the next index position in list2

    advance r to next index position in resultList
}
return resultList
```

Selected Solutions

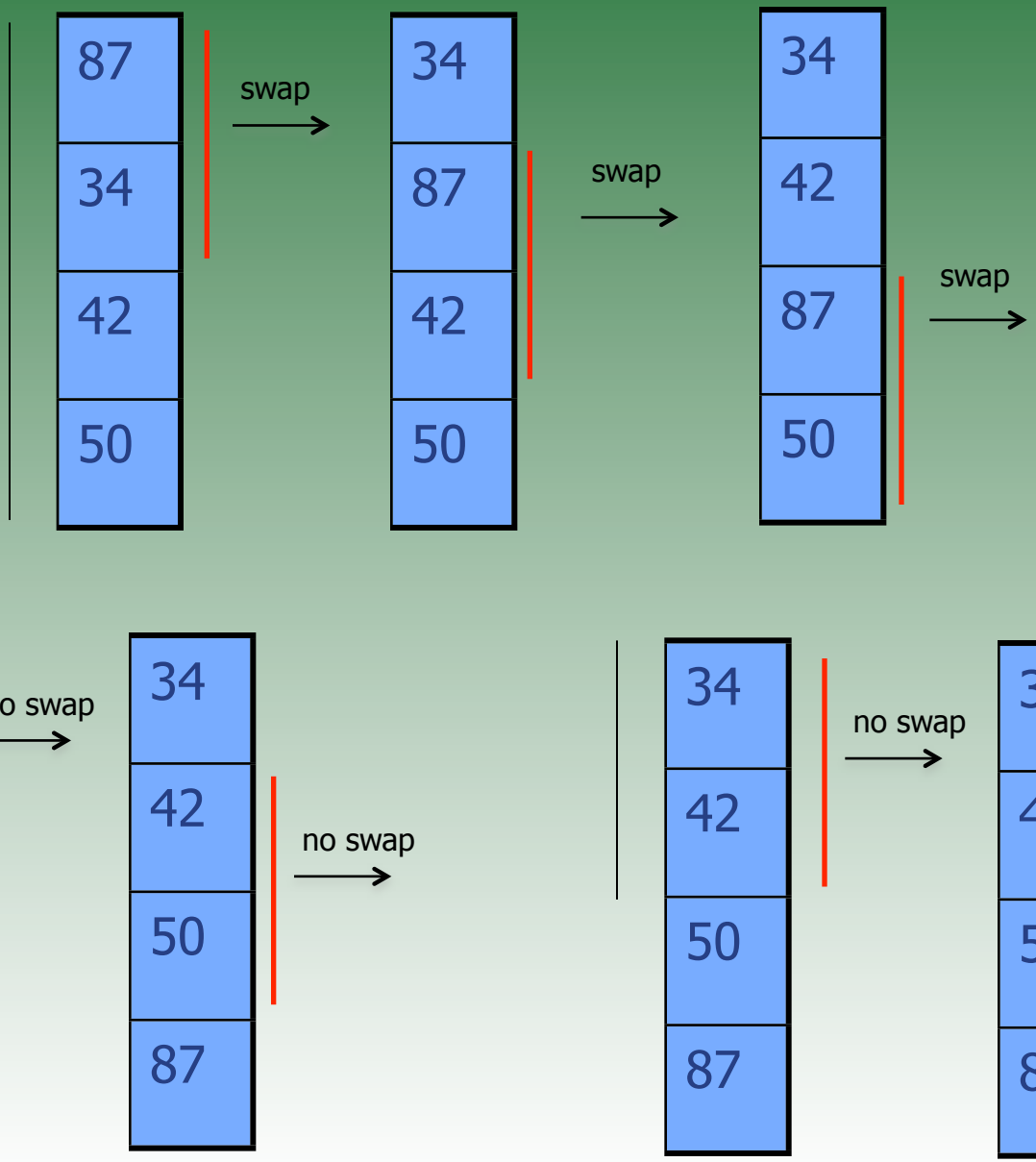
Exercise 1c, d, e Solutions

```
// implementation of selectionSort()

public class SearchAndSort {
    public static int indexOfLargest(int[] list, int end) {
        int indexOfLargestValue = 0;

        for (int i = 0; i <= end; i++) {
            if (list[i] > list[indexOfLargestValue]) {
                indexOfLargestValue = i;
            }
        }
        return indexOfLargestValue;
    }

    public static void swapNumbers(int[] numbers, int i, int j) {
        int tmp = numbers[i];
        numbers[i] = numbers[j];
        numbers[j] = tmp;
    }

    public static void selectionSort(int[] integers) {
        for (int i = integers.length - 1; i >= 1; i--) {
            int k = indexOfLargest(integers, i);
            swapNumbers(integers, k, i);
        }
    }
}
```

Extension of Exercise 1: Selection Sort with Strings

```
// implementation of selectionSort() with String type
// we add these methods to the SearchAndSort class of the previous slide
// overloading will allow for methods that have the same name

public static int indexOfLargest(String[] list, int end) {
    int indexOfLargestValue = 0;

    for (int i = 0; i <= end; i++) {
        if (list[i].compareTo(list[indexOfLargestValue]) > 0) {
            indexOfLargestValue = i;
        }
    }
    return indexOfLargestValue;
}

public static void swap(String[] list, int i, int j) {
    String tmp = list[i];
    list[i] = list[j];
    list[j] = tmp;
}

public static void selectionSort(String[] list) {
    for (int i = list.length - 1; i >= 1; i--) {
        int k = indexOfLargest(list, i);
        swap(list, k, i);
    }
}
```

Exercise 2a-i Solution



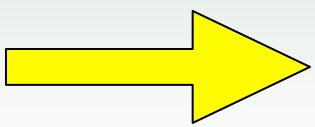
Exercise 2d Solution

```
// perform a bubble sort on an array of integers
// use a boolean variable to stop sorting if there are no swappings
// during a complete pass of comparisons
// this would mean that the array has been fully sorted
// method returns void

public static void bubbleSort(int[] items) {
    boolean swap = true;  // guarantee entry into the loop

    for (int limit = items.length - 1; limit >= 1 && swap; limit--) {

        swap = false;   // assume no swapping has happened

        for (int i = 0; i < limit; i++) {
            if (items[i] > items[i + 1]) {
                int tmp = items[i];
                items[i] = items[i + 1];
                items[i + 1] = tmp;
                swap = true; // set to true because swapping happened
            }
        }
    }
}
```

Exercise 2e Solution

Scan	List size	Comparisons
1	n	n - 1
2	n - 1	n - 2
3	n - 2	n - 3
4	n - 3	n - 4
5	n - 4	n - 5
…	…	…
n - 1	n - [(n - 1) - 1] = 2	n - (n - 1) = 1

Next slide



Exercise 2e Solution Continued

Bubble Sort

Using the arithmetic series formula $S_N = \frac{N}{2}(t_1 + t_N)$ the total comparisons are

$$1+2+3+4+\ldots+(n-1) = \frac{(n-1)}{2}(1+n-1) = \frac{n^2-n}{2} \approx n^2$$

The number of comparisons performed is directly proportional to the square of the number of elements.
The relation is quadratic

Complexity of Bubble Sort algorithm is $O(n^2)$