3.1 Sorting: Introduction

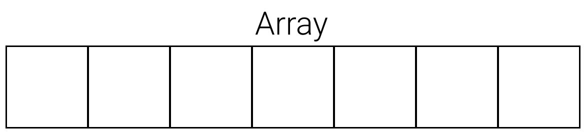
Sorting is the process of putting a collection of elements into ascending (or descending) order. Ex: Given the numbers array [17, 3, 44, 6, 9], the array after sorting is [3, 6, 9, 17, 44]. Strings and other data types can also be sorted. Ex: Given the string array ["pear", "apple", "grape", "banana"], the array after sorting is ["apple", "banana", "grape", "pear"].

The challenge of sorting is that a program can't "see" the entire array to know where to move an element. Instead, a program is limited to simpler steps, typically observing or swapping just two elements at a time. So sorting just by swapping values is an important part of sorting algorithms.

PARTICIPATION ACTIVITY 3.1.1: Sort by swapping tool.

Sort the numbers from smallest on left to largest on right. Select two numbers then click "Swap values".

Start



Swap

PARTICIPATION ACTIVITY 3.1.2: Sorted elements.

1) The array is sorted into ascending order:
[3, 9, 44, 18, 76]

- True
- False

2) The array is sorted into descending order:
[20, 15, 10, 5, 0]

- True
- False

3) The array is sorted into descending order:
[99.87, 99.02, 67.93, 44.10]

- ○ True
- ○ False

4) The array is sorted into descending order:
['F', 'D', 'C', 'B', 'A']

- ○ True
- ○ False

5) The array is sorted into ascending order:
["chopsticks", "forks", "knives", "spork"]

- ○ True
- ○ False

6) The array is sorted into ascending order:
["great", "greater", "greatest"]

- ○ True
- ○ False

3.2 Selection sort

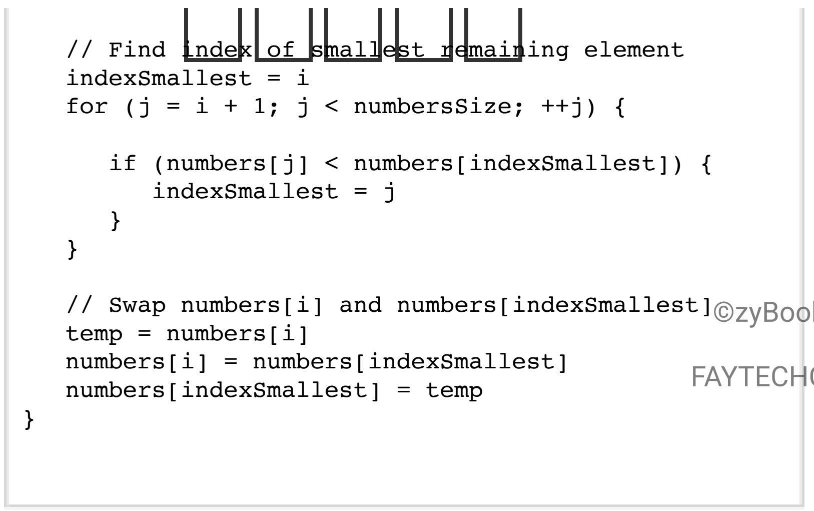
Selection sort

Selection sort is a sorting algorithm that treats the input as two parts, a sorted part and an unsorted part, and repeatedly selects the minimum value to move from the unsorted part to the end of the sorted part.

The term "selection" comes from the fact that for each iteration of the outer loop, a value is selected for position i.

PARTICIPATION ACTIVITY | 3.2.1: Selection sort.

☐ Unsorted ☐ Sorted



```
   // Find index of smallest remaining element
   indexSmallest = i
   for (j = i + 1; j < numbersSize; ++j) {

      if (numbers[j] < numbers[indexSmallest]) {
         indexSmallest = j
      }
   }

   // Swap numbers[i] and numbers[indexSmallest]
   temp = numbers[i]
   numbers[i] = numbers[indexSmallest]
   numbers[indexSmallest] = temp
}
```

Animation content:

Static figure: A code block and an array with elements 7, 9, 3, 18, and 8 are displayed.
Begin pseudocode:

```
for (i = 0; i < numbersSize - 1; ++i) {

   // Find index of smallest remaining element
   indexSmallest = i
   for (j = i + 1; j < numbersSize; ++j) {

      if (numbers[j] < numbers[indexSmallest]) {
         indexSmallest = j
      }
   }

   // Swap numbers[i] and numbers[indexSmallest]
   temp = numbers[i]
   numbers[i] = numbers[indexSmallest]
   numbers[indexSmallest] = temp
}
```

End pseudocode.

Step 1: Selection sort treats the input as two parts, a sorted and unsorted part. Variables i and j keep track of the two parts. Index variables i and j appear below the array, with accompanying arrows pointing to array elements. i's arrow points to the element at index 0 (7) and j's arrow points to the element at index 1 (9).
Step 2: The selection sort algorithm searches the unsorted part of the array for the smallest element; indexSmallest stores the index of the smallest element found. The lines of code, for (j = i + 1; j < numbersSize; ++j) {, if (numbers[j] < numbers[indexSmallest]) {, indexSmallest = j, }, }, are highlighted. indexSmallest variable appears above array, with accompanying arrow pointing to the element at index 0 (7). j advances through indices 1,2, 3, and 4. Each comparison of element pairs is shown to the left:
9 < 7 (No)
3 < 7 (Yes)
18 < 3 (No)
8 < 3 (No)
indexSmallest variable advances to index 2.
Step 3: Elements at i and indexSmallest are swapped. The lines of code, temp = numbers[i], numbers[i] = numbers[indexSmallest], numbers[indexSmallest] = temp, are highlighted. Elements at index i and indexSmallest are swapped, yielding the array: 3, 9, 7, 18, 8.
Step 4: Indices for the sorted and unsorted parts are updated. Index i advances to 1. Index j moves back to index 2.
Step 5: The unsorted part is searched again, swapping the smallest element with the element at i. The lines of code, indexSmallest = i , for (j = i + 1; j < numbersSize; ++j) {, if (numbers[j] < numbers[indexSmallest]) {, indexSmallest = j, }, }, are highlighted, and index j advances through indices 2, 3, and 4. Each comparison of element pairs is shown to the left:
7 < 9 (Yes)
18 < 7 (No)
8 < 7 (No)
Index variables before the swap are i = 1, j = 4, and indexSmallest = 2. Then elements at index i and indexSmallest are swapped, yielding the array: 3, 7, 9, 18, 8.
Step 6: The process repeats until all elements are sorted. Execution continues, swapping pairs at indices 2 and 4, then 3 and 4. The result is a sorted array: 3, 7, 8, 9, 28.

Animation captions:

1. Selection sort treats the input as two parts, a sorted and unsorted part. Variables i and j keep track of the two parts.
2. The selection sort algorithm searches the unsorted part of the array for the smallest element; indexSmallest stores the index of the smallest element found.
3. Elements at i and indexSmallest are swapped.
4. Indices for the sorted and unsorted parts are updated.
5. The unsorted part is searched again, swapping the smallest element with the element at i.
6. The process repeats until all elements are sorted.

PARTICIPATION ACTIVITY | 3.2.2: Selection sort algorithm execution.

Assume selection sort's goal is to sort in ascending order.

1) Given array [9, 8, 7, 6, 5], what element is at index 0 after the first pass over the outer loop (i = 0)?

Check **Show answer**

2) Given array [9, 8, 7, 6, 5], how many swaps occur during the first pass of the outer loop (i = 0)?

Check **Show answer**

3) Given array [5, 9, 8, 7, 6], what is the array after completing the second outer loop iteration (i = 1)? Type answer as: 1, 2, 3

Check **Show answer**

Selection sort runtime

Selection sort has the advantage of being easy to code, involving one loop nested within another loop, as shown below.

Figure 3.2.1: Selection sort algorithm.

```
SelectionSort(numbers, numbersSize) {
   i = 0
   j = 0
   temp = 0  // Temporary variable for swap

   for (i = 0; i < numbersSize - 1; ++i) {

      // Find index of smallest remaining element
      indexSmallest = i
      for (j = i + 1; j < numbersSize; ++j) {

         if ( numbers[j] < numbers[indexSmallest] ) {
            indexSmallest = j
         }
      }

      // Swap numbers[i] and numbers[indexSmallest]
      temp = numbers[i]
      numbers[i] = numbers[indexSmallest]
      numbers[indexSmallest] = temp
   }
}

main() {
   numbers[] = { 10, 2, 78, 4, 45, 32, 7, 11 }
   NUMBERS_SIZE = 8
   i = 0

   print("UNSORTED: ")
   for (i = 0; i < NUMBERS_SIZE; ++i) {
      print(numbers[i] + " ")
   }
   printLine()

   SelectionSort(numbers, NUMBERS_SIZE)

   print("SORTED: ")
   for (i = 0; i < NUMBERS_SIZE; ++i) {
      print(numbers[i] + " ")
   }
   printLine()
}
```

```
UNSORTED: 10 2 78 4 45 32 7 11
SORTED: 2 4 7 10 11 32 45 78
```

Selection sort may require a large number of comparisons. The selection sort algorithm runtime is $O(N^2)$. If an array has N elements, the outer loop executes N - 1 times. For each of those N - 1 outer loop executions, the inner loop executes an average of $\frac{N}{2}$ times. So the total number of comparisons is proportional to $(N-1) \cdot \frac{N}{2}$, or $O(N^2)$. Other sorting algorithms involve more complex algorithms but have faster execution times.

PARTICIPATION ACTIVITY 3.2.3: Selection sort runtime.

Enter an integer value for each answer.

1) When sorting an array with 50 elements, `indexSmallest` is assigned to a minimum of ____ times.

Check **Show answer**

2) Sorting 2X elements takes about how many times longer than X elements?

Check **Show answer**

3) Sorting 10X elements takes about how many times longer than X elements?

Check **Show answer**

CHALLENGE ACTIVITY 3.2.1: Selection sort.

704586.1152092.qx3zqy7

Start

When using selection sort to sort an array with 11 elements, what is the minimum number of assignments to `indexSmallest` once the outer loop starts?

Ex: 4

1	2	3	4

Check Next

View solution (Instructors only)

3.3 C++: Selection sort

Selection sort

Selection sort's outer loop uses the variable i to hold the index position that will be sorted next in the array. The inner loop uses the variable j to examine all indices from i+1 to the end of the array. When the j loop finishes, the variable indexSmallest stores the index position of the smallest item in the array from i onward. The final step is to swap the values at position i and indexSmallest.

Figure 3.3.1: Selection sort algorithm.

```
#include <iostream>
#include <string>
using namespace std;

void SelectionSort(int* numbers, int numbersSize) {
   for (int i = 0; i < numbersSize - 1; i++) {
      // Find index of smallest remaining element
      int indexSmallest = i;
      for (int j = i + 1; j < numbersSize; j++) {
         if (numbers[j] < numbers[indexSmallest]) {
            indexSmallest = j;
         }
      }

      // Swap numbers[i] and numbers[indexSmallest]
      int temp = numbers[i];
      numbers[i] = numbers[indexSmallest];
      numbers[indexSmallest] = temp;
   }
}

string ArrayToString(int* array, int arraySize) {
   // Special case for empty array
   if (0 == arraySize) {
      return string("[]");
   }

   // Start the string with the opening '[' and element 0
   string result = "[" + to_string(array[0]);

   // For each remaining element, append comma, space, and element
   for (int i = 1; i < arraySize; i++) {
      result += ", ";
      result += to_string(array[i]);
   }

   // Add closing ']' and return result
   result += "]";
   return result;
}

int main() {
   // Create an array of numbers to sort
   int numbers[] = { 10, 2, 78, 4, 45, 32, 7, 11 };
   int numbersSize = sizeof(numbers) / sizeof(numbers[0]);

   // Display the contents of the array
   cout << "UNSORTED: " << ArrayToString(numbers, numbersSize) << endl;

   // Call the SelectionSort function
   SelectionSort(numbers, numbersSize);

   // Display the sorted contents of the array
   cout << "SORTED:   " << ArrayToString(numbers, numbersSize) << endl;
}
```

```
UNSORTED: [10, 2, 78, 4, 45, 32, 7, 11]
```

```
SORTED:   [2, 4, 7, 10, 11, 32, 45, 78]
```

PARTICIPATION ACTIVITY 3.3.1: Selection sort algorithm.

Suppose the SelectionSort() function is executed with the array [3, 12, 7, 2, 9, 14, 8].

1) When i is assigned with 0 in the outer loop, what is j assigned with in the first pass through the inner loop?

- ○ 0
- ○ 1
- ○ 3

2) Variable i's last assigned value is _____.

- ○ 5
- ○ 6
- ○ 7

3) Which statement best describes the following code fragment taken from the SelectionSort() function?

```
int temp = numbers[i];
numbers[i] =
numbers[indexSmallest];
numbers[indexSmallest] = temp;
```

- ○ Shifts the value at position indexSmallest one space to the right in the array.
- ○ Tests to see if the value at numbers[indexSmallest] is smaller than the value at numbers[i].
- ○ Swaps the values located at indices i and indexSmallest.

zyDE 3.3.1: Selection sort algorithm.

The program below uses the selection sort algorithm to sort various arrays. The code has been modified to also calculate how many comparisons and swaps occur.

SelectionSort() is called three times to sort three distinct arrays. The first is unsorted, the second is sorted in ascending order, and the third is sorted in descending order. Try running the program with different arrays to see how the total number of comparisons and swaps changes (or doesn't change).

Current file: **SelectionSortDemo.cpp** ▾ **Load default template...**

```
#include <iostream>
#include <string>
#include "SortTracker.h"
using namespace std;

void SelectionSort(int* numbers, int numbersSize, SortTracker& tracker) {
   for (int i = 0; i < numbersSize - 1; i++) {
      // Find index of smallest remaining element
      int indexSmallest = i;
      for (int j = i + 1; j < numbersSize; j++) {
         if (tracker.IsFirstLTSecond(numbers, j, indexSmallest)) {
            indexSmallest = j;
         }
      }

      // Swap numbers[i] and numbers[indexSmallest]
      tracker.Swap(numbers, i, indexSmallest);
   }
}

string ArrayToString(const int* array, int arraySize) {
   // Special case for empty array
   if (0 == arraySize) {
      return string("[]");
   }

   // Start the string with the opening '[' and element 0
   string result = "[" + to_string(array[0]);

   // For each remaining element, append comma, space, and element
   for (int i = 1; i < arraySize; i++) {
      result += ", ";
      result += to_string(array[i]);
   }

   // Add closing ']' and return result
   result += "]";
   return result;
}
```

```
}

void SelectionSortDemo(int* numbersArray, int arrayLength) {
   // Display the contents of the array
   cout << "Before sorting:     " << ArrayToString(numbersArray, arrayLength);
   cout << endl;

   // Sort the numbers array using the selection sort algorithm
   SortTracker tracker;
   SelectionSort(numbersArray, arrayLength, tracker);

   // Display the sorted contents of the array
   cout << "After sorting:      " << ArrayToString(numbersArray, arrayLength);
   cout << endl;

   // Display stats
   cout << "Total comparisons: " << tracker.GetComparisonCount() << endl;
   cout << "Total swaps:        " << tracker.GetSwapCount() << endl;
}

int main() {
   // Create arrays:
   // - array1 is unsorted
   // - array2 is sorted in ascending order
   // - array3 is sorted in descending order
   int array1[] = { 33, 18, 78, 64, 45, 32, 70, 11, 27 };
   int array1Length = sizeof(array1) / sizeof(array1[0]);
   int array2[] = { 10, 20, 30, 40, 50, 60, 70, 80, 90 };
   int array2Length = sizeof(array2) / sizeof(array2[0]);
   int array3[] = { 99, 88, 77, 66, 55, 44, 33, 22, 11 };
   int array3Length = sizeof(array3) / sizeof(array3[0]);

   cout << "Demo 1 - Unsorted array:" << endl;
   SelectionSortDemo(array1, array1Length);
   cout << endl << "Demo 2 - Array sorted in ascending order:" << endl;
   SelectionSortDemo(array2, array2Length);
   cout << endl << "Demo 3 - Array sorted in descending order:" << endl;
   SelectionSortDemo(array3, array3Length);

   return 0;
}
```

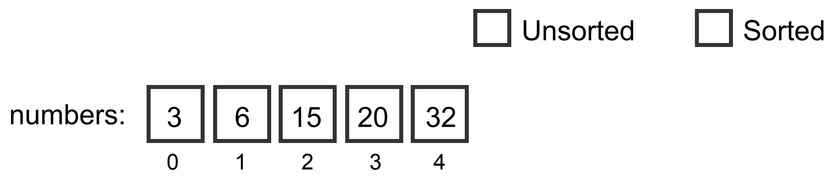
Run

3.4 Insertion sort

Insertion sort algorithm

Insertion sort is a sorting algorithm that treats the input as two parts, a sorted part and an unsorted part, and repeatedly inserts the next value from the unsorted part into the correct location in the sorted part.

PARTICIPATION ACTIVITY 3.4.1: Insertion sort.



```
for (i = 1; i < numbersSize; ++i) {
   j = i
   // Insert numbers[i] into sorted part
   // stopping once numbers[i] in correct position
   while (j > 0 && numbers[j] < numbers[j - 1]) {

      // Swap numbers[j] and numbers[j - 1]
      temp = numbers[j]
      numbers[j] = numbers[j - 1]
      numbers[j - 1] = temp
      j -= 1
   }
}
```

Animation content:

Static figure: An unsorted array named numbers with elements 3, 6, 15, 20, 32.
Begin code:

```
for (i = 1; i < numbersSize; ++i) {
  j = i
  // Insert numbers[i] into sorted part
  // stopping once numbers[i] in correct position
  while (j > 0 && numbers[j] < numbers[j - 1]) {

    // Swap numbers[j] and numbers[j - 1]
    temp = numbers[j]
```

```
      numbers[j] = numbers[j - 1]
      numbers[j - 1] = temp
      j -= 1
   }
}
```

End code.

Step 1: Variable i is the index of the first unsorted element. Since the element at index 0 is already sorted, i starts at 1. Array is shown: 32, 6, 15, 3, 20. Index variables i and j appear below array, with accompanying arrows pointing to array elements. Both i's arrow and j's arrow point to the element at index 1 (element 6). The following lines of code from the selection sort implementation are highlighted:

```
i = 1;
j = i
while (j > 0 && numbers[j] < numbers[j - 1]) {
```

Step 2: Variable j keeps track of the index of the current element being inserted into the sorted part. If the current element is less than the element to the left, the values are swapped. The lines of code

```
while (j > 0 && numbers[j] < numbers[j - 1]) {
      // Swap numbers[j] and numbers[j - 1]
      temp = numbers[j]
      numbers[j] = numbers[j - 1]
      numbers[j - 1] = temp
      j -= 1
```

are highlighted. Comparison of elements at j and j - 1 is shown:

6 < 32 (Yes)

Array elements at indices 0 and 1 are swapped, yielding the array: 6, 32, 15, 3, 20. Then j is decremented to 0.

Step 3: Once the current element is inserted in the correct location in the sorted part, i is incremented to the next element in the unsorted part. The next iteration of the i loop is highlighted. First, variables i and j are each assigned with 2. j decrements to 1 during the inner while loop. Element comparisons that occur are shown:

15 < 32 (Yes)
15 < 6 (No)

Elements at indices 1 and 2 are swapped, yielding the array: 6, 15, 32, 3, 20.

Step 4: If the current element being inserted is smaller than all elements in the sorted part, that element will be repeatedly swapped with each sorted element until index 0 is reached. The next iteration of the i loop is highlighted. Comparisons that occur:

3 < 32 (Yes)
3 < 15 (Yes)
3 < 6 (Yes)

Three swaps occur to move element 3 to index 0, yielding the array: 3, 6, 15, 32, 20.

Step 5: Once all elements in the unsorted part are inserted in the sorted part, the array is sorted. The outer i loop's final iteration occurs, swapping only elements 23 and 20. The final array is sorted:

3, 6, 15, 20, 32.

Animation captions:

1. Variable i is the index of the first unsorted element. Since a single-element array is already sorted, i starts at 1.
2. Variable j keeps track of the index of the current element being inserted into the sorted part. If the current element is less than the element to the left, the values are swapped.
3. Once the current element is inserted in the correct location in the sorted part, i is incremented to the next element in the unsorted part.
4. If the current element being inserted is smaller than all elements in the sorted part, that element will be repeatedly swapped with each sorted element until index 0 is reached.
5. Once all elements in the unsorted part are inserted in the sorted part, the array is sorted.

The index variable i denotes the starting position of the current element in the unsorted part. Initially, the element at index 0 is assumed to be sorted, so the outer for loop initializes i to 1. The inner while loop inserts the current element into the sorted part by repeatedly swapping the current element with the elements in the sorted part that are larger. Once a smaller or equal element is found in the sorted part, the current element has been inserted in the correct location and the while loop terminates.

Figure 3.4.1: Insertion sort algorithm.

```
InsertionSort(numbers, numbersSize) {
   i = 0
   j = 0
   temp = 0  // Temporary variable for swap

   for (i = 1; i < numbersSize; ++i) {
      j = i
      // Insert numbers[i] into sorted part
      // stopping once numbers[i] in correct position
      while (j > 0 && numbers[j] < numbers[j - 1]) {

         // Swap numbers[j] and numbers[j - 1]
         temp = numbers[j]
         numbers[j] = numbers[j - 1]
         numbers[j - 1] = temp
         --j
      }
   }
}

main() {
   numbers = { 10, 2, 78, 4, 45, 32, 7, 11 }
   NUMBERS_SIZE = 8
   i = 0

   print("UNSORTED: ")
   for(i = 0; i < NUMBERS_SIZE; ++i) {
      print(numbers[i] + " ")
   }
   printLine()

   InsertionSort(numbers, NUMBERS_SIZE)

   print("SORTED: ")
   for(i = 0; i < NUMBERS_SIZE; ++i) {
      print(numbers[i] + " ")
   }
   printLine()
}
```

```
UNSORTED: 10 2 78 4 45 32 7 11
SORTED: 2 4 7 10 11 32 45 78
```

PARTICIPATION ACTIVITY 3.4.2: Insertion sort algorithm execution.

Assume insertion sort's goal is to sort in ascending order.

Given the array [20, 14, 85, 3, 9], what is the element at index 0 after the first iteration of the outer loop (i = 1)?

Check Show answer

Insertion sort runtime

Insertion sort's typical runtime is $O(N^2)$. If an array has N elements, the outer loop executes N - 1 times. For each outer loop execution, the inner loop may need to examine all elements in the sorted part. Thus, the inner loop executes on average $\frac{N}{2}$ times. So the total number of comparisons is proportional to $(N-1) \cdot (\frac{N}{2})$, or $O(N^2)$. Other sorting algorithms involve more complex algorithms but faster execution.

PARTICIPATION ACTIVITY 3.4.3: Insertion sort runtime.

1) In the worst case, assuming each comparison takes 1 µs, how long will insertion sort algorithm take to sort 10 elements?

µs

Check Show answer

2) Using the Big O runtime complexity, how many times longer will sorting 20 elements take compared to sorting 10 elements?

Check Show answer

Nearly sorted arrays

Some sorting algorithms like insertion sort run faster when the inputs are sorted or nearly sorted. A ***nearly sorted*** array only contains a few elements not in sorted order. Ex: [4, 5, 17, 25, 89, 14] is nearly sorted, having only one element not in sorted position.

PARTICIPATION

ACTIVITY 3.4.4: Nearly sorted inputs.

Determine if each of the following arrays is unsorted, sorted, or nearly sorted. Assume ascending order.

1) [6, 14, 85, 102, 102, 151]

- ○ Unsorted
- ○ Sorted
- ○ Nearly sorted

2) [23, 24, 36, 48, 19, 50, 101]

- ○ Unsorted
- ○ Sorted
- ○ Nearly sorted

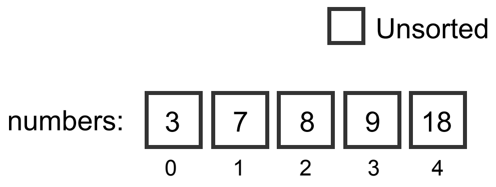
3) [15, 19, 21, 24, 2, 3, 6, 11]

- ○ Unsorted
- ○ Sorted
- ○ Nearly sorted

Insertion sort runtime for nearly sorted input

For each outer loop execution, if the element is already in sorted position, only a single comparison is made. Each element not in sorted position requires at most N comparisons. If a constant number, C, of unsorted elements exist, sorting the N - C sorted elements requires one comparison each, and sorting the C unsorted elements requires at most N comparisons each. The runtime for nearly sorted inputs is O((N - C) * 1 + C * N) = O(N).

PARTICIPATION ACTIVITY 3.4.5: Using insertion sort for nearly sorted inputs.



Animation content:

Static figure: A sorted
array named numbers with elements 3, 7, 8, 9, 18.

Step 1: The sorted part initially contains the first element.
The first element, 3, is highlighted and labeled as sorted.

Step 2: An element already in sorted position only requires a single comparison, which is O(1) complexity.
3 is compared with the next element 7. 3 is less than 7 so the element 7 is labeled as sorted. 7 is compared with the next element 9. 7 is less than 9 so the element 9 is labeled as sorted. 9 is compared with the next element 18. 9 is less than 18 so the element 18 is labeled as sorted.

Step 3: An element not in sorted position requires O(N) comparisons. For nearly sorted inputs, insertion sort's runtime is O(N).
18 is compared with the next element 8. 18 is not less than 8, so the element 8 swaps places with 18. 8 is compared with the next previous element 9. 8 is less than 9, so the element 8 swaps places with 9. 8 is compared with the next previous element 7. 8 is greater than 7. The array is sorted.

Animation captions:

1. The sorted part initially contains the first element.
2. An element already in sorted position only requires a single comparison, which is O(1) complexity.
3. An element not in sorted position requires O(N) comparisons. For nearly sorted inputs, insertion sort's runtime is O(N).

PARTICIPATION ACTIVITY 3.4.6: Insertion sort algorithm execution for nearly sorted input.

Assume insertion sort's goal is to sort in ascending order.

1) Given the array [10, 11, 12, 13, 14, 15], how many comparisons will be

made during the third outer loop execution (i = 3)?

Check **Show answer**

2) Given the array [10, 11, 12, 13, 14, 7], how many comparisons will be made during the final outer loop execution (i = 5)?

Check **Show answer**

3) Given the array [18, 23, 34, 75, 3], how many total comparisons will insertion sort require?

Check **Show answer**

CHALLENGE ACTIVITY 3.4.1: Insertion sort.

704586.1152092.qx3zqy7

Start

Given array [28, 41, 47, 57, 36, 42, 49], when using insertion sort, what is the value of i when the first swap executes?

Ex: 1

1	2	3	4	5

Check Next

View solution *(Instructors only)*

3.5 C++: Insertion sort

Insertion sort algorithm

InsertionSort()'s index variable `i` denotes the starting position of the current element in the unsorted part. Initially, the first element (i.e., element at index 0) is assumed to be sorted, so the outer for loop assigns `i` with 1 to begin. The inner while loop inserts the current element into the sorted part by repeatedly swapping the current element with the elements in the array's sorted part that are larger. Once a smaller or equal element is found in the array's sorted part, the current element has been inserted in the correct location and the while loop terminates.

Figure 3.5.1: Insertion sort algorithm.

```
#include <iostream>
#include <string>
using namespace std;

void InsertionSort(int* numbers, int numbersSize) {
   for (int i = 1; i < numbersSize; i++) {
      int j = i;
      while (j > 0 && numbers[j] < numbers[j - 1]) {
         // Swap numbers[j] and numbers [j - 1]
         int temp = numbers[j];
         numbers[j] = numbers[j - 1];
         numbers[j - 1] = temp;
         j--;
      }
   }
}

string ArrayToString(int* array, int arraySize) {
   // Special case for empty array
   if (0 == arraySize) {
      return string("[]");
   }

   // Start the string with the opening '[' and element 0
   string result = "[" + to_string(array[0]);

   // For each remaining element, append comma, space, and element
   for (int i = 1; i < arraySize; i++) {
      result += ", ";
      result += to_string(array[i]);
   }

   // Add closing ']' and return result
   result += "]";
   return result;
}

int main() {
   // Create an array of numbers to sort
   int numbers[] = { 10, 2, 78, 4, 45, 32, 7, 11 };
   int numbersSize = sizeof(numbers) / sizeof(numbers[0]);

   // Display the contents of the array
   cout << "UNSORTED: " << ArrayToString(numbers, numbersSize) << endl;

   // Call the InsertionSort function
   InsertionSort(numbers, numbersSize);

   // Display the sorted contents of the array
   cout << "SORTED:   " << ArrayToString(numbers, numbersSize) << endl;
}
```

```
UNSORTED: [10, 2, 78, 4, 45, 32, 7, 11]
SORTED:   [2, 4, 7, 10, 11, 32, 45, 78]
```

PARTICIPATION ACTIVITY 3.5.1: Insertion sort algorithm implementation.

1) Which index variable denotes the starting position of the current element in the unsorted part?

- ○ i
- ○ j
- ○ k

2) InsertionSort()'s `j` variable is always _____ equal to `i`.

- ○ greater than or
- ○ less than or
- ○ not

3) If the numbers array has 4 elements in descending order, then the InsertionSort() function does _____ swaps.

- ○ 4
- ○ 6
- ○ 16

zyDE 3.5.1: Insertion sort algorithm.

The program below uses the insertion sort algorithm to sort an array. The code has been modified to also calculate how many item comparisons occur. Try running the program with different arrays to see how the total number of comparisons changes (or doesn't change).

Current file: **InsertionSortDemo.cpp** **Load default template...**

```
#include <iostream>
#include <string>
#include "SortTracker.h"
using namespace std;

void InsertionSort(int* numbers, int numbersSize, SortTracker& tracker) {
   for (int i = 1; i < numbersSize; i++) {
      int j = i;
      while (j > 0 && tracker.IsFirstLTSecond(numbers, j, j - 1)) {
         // Swap numbers[j] and numbers[j - 1]
         tracker.Swap(numbers, j, j - 1);

         // Decrement j in preparation for next comparison
         j--;
      }
   }
}

string ArrayToString(const int* array, int arraySize) {
   // Special case for empty array
   if (0 == arraySize) {
      return string("[]");
   }

   // Start the string with the opening '[' and element 0
   string result = "[" + to_string(array[0]);

   // For each remaining element, append comma, space, and element
   for (int i = 1; i < arraySize; i++) {
      result += ", ";
      result += to_string(array[i]);
   }

   // Add closing ']' and return result
   result += "]";
   return result;
}

void InsertionSortDemo(int* numbersArray, int arrayLength) {
   // Display the contents of the array
   cout << "Before sorting:    " << ArrayToString(numbersArray, arrayLength);
   cout << endl;

   // Sort the numbers array using the insertion sort algorithm
   SortTracker tracker;
   InsertionSort(numbersArray, arrayLength, tracker);

   // Display the sorted contents of the array
   cout << "After sorting:     " << ArrayToString(numbersArray, arrayLength);
   cout << endl;

   // Display stats
   cout << "Total comparisons: " << tracker.GetComparisonCount() << endl;
   cout << "Total swaps:       " << tracker.GetSwapCount() << endl;
}

int main() {
   // Create arrays:
   // - array1 is unsorted
   // - array2 is sorted in ascending order
   // - array3 is sorted in descending order
   int array1[] = { 33, 18, 78, 64, 45, 32, 70, 11, 27 };
   int array1Length = sizeof(array1) / sizeof(array1[0]);
   int array2[] = { 10, 20, 30, 40, 50, 60, 70, 80, 90 };
   int array2Length = sizeof(array2) / sizeof(array2[0]);
   int array3[] = { 99, 88, 77, 66, 55, 44, 33, 22, 11 };
   int array3Length = sizeof(array3) / sizeof(array3[0]);

   cout << "Demo 1 - Unsorted array:" << endl;
   InsertionSortDemo(array1, array1Length);
   cout << endl << "Demo 2 - Array sorted in ascending order:" << endl;
   InsertionSortDemo(array2, array2Length);
   cout << endl << "Demo 3 - Array sorted in descending order:" << endl;
   InsertionSortDemo(array3, array3Length);

   return 0;
}
```

Run

3.6 Shell sort

Shell sort's interleaved arrays

Shell sort is a sorting algorithm that treats the input as a collection of interleaved arrays, and sorts each array individually with a variant of the insertion sort algorithm. Shell sort uses gap values to determine the number of interleaved arrays. A ***gap value*** is a positive integer representing the distance between elements in an interleaved array. For each interleaved array, if an element is at index i, the next element is at index i + gap value.

Shell sort begins by choosing a gap value K and sorting K interleaved arrays in place. Shell sort finishes by performing a standard insertion sort on the entire array. Because the interleaved parts have already been sorted, smaller elements will be close to the array's beginning and larger elements towards the end. Insertion sort can then quickly sort the nearly-sorted array.

Any positive integer gap value can be chosen. In the case that the array size is not evenly divisible by the gap value, some interleaved arrays will have fewer elements than others.

PARTICIPATION ACTIVITY 3.6.1: Sorting interleaved arrays with shell sort speeds up insertion sort.

12 36 42 56 75 77 82 91 93
Original, insertion sort: 17 swaps
12 36 42 56 75 77 82 91 93
Shell sort (gap=3): 7 swaps

Animation content:

Static Figure:
Two 9 value arrays are shown.
The first array: [12, 36, 42, 56, 75, 77, 82, 91, 93]. The text "Original, insertion sort: 17 swaps" is shown below the first array.
The second array: [12, 42, 36, 56, 77, 82, 75, 91, 93]. The text "Shell sort (gap=3): 7 swaps" is shown below the second array.

Step 1: If a gap value of 3 is chosen, shell sort views the array as 3 interleaved arrays. 56, 12, and 75 make up the first array, 42, 77, and 91 the second, and 93, 82, and 36 the third.
Original array: [56, 42, 93, 12, 77, 82, 75, 91, 36]. Color coding indicates interleaved arrays. Elements 56, 12, and 75 are yellow. Elements 42, 77, and 91 are green. Elements 93, 82, and 36 are blue.

Step 2: Shell sort sorts each of the 3 arrays with insertion sort.

The three arrays are separated visually and sorted individually with the insertion sort algorithm.

Step 3: The result is not a sorted array, but is closer to sorted than the original. Ex: The 3 smallest elements, 12, 42, and 36, are the first 3 elements in the array.
Arrays visually recombine back into one array: [12, 42, 36, 56, 77, 82, 75, 91, 93].

Step 4: Sorting the original array with insertion sort requires 17 swaps.
Regular insertion sort algorithm is shown running on the original array, requiring 17 swaps total.

Step 5: Sorting the interleaved arrays required 4 swaps. Running insertion sort on the resulting array requires 3 more. 7 total swaps occur, far less than insertion sort on the original array.
Regular insertion sort algorithm is shown running on the array from step 3, requiring only 3 additional swaps to make a sorted array.

Animation captions:

1. If a gap value of 3 is chosen, shell sort views the array as 3 interleaved arrays. 56, 12, and 75 make up the first array, 42, 77, and 91 the second, and 93, 82, and 36 the third.
2. Shell sort sorts each of the 3 arrays with insertion sort.
3. The result is not a sorted array, but is closer to sorted than the original. Ex: The 3 smallest elements, 12, 42, and 36, are the first 3 elements in the array.
4. Sorting the original array with insertion sort requires 17 swaps.
5. Sorting the interleaved arrays required 4 swaps. Running insertion sort on the resulting array requires 3 more. 7 total swaps occur, far less than insertion sort on the original array.

PARTICIPATION ACTIVITY 3.6.2: Shell sort's interleaved arrays.

For each question, assume an array with 6 elements.

1) With a gap value of 3, how many interleaved arrays will be sorted?

- ○ 1
- ○ 2
- ○ 3
- ○ 6

2) With a gap value of 3, how many elements will be in each interleaved array?

- ○ 1

- ○ 2
- ○ 3
- ○ 6

3) If a gap value of 2 is chosen, how many interleaved arrays will be sorted?

- ○ 1
- ○ 2
- ○ 3
- ○ 6

4) If a gap value of 4 is chosen, how many interleaved arrays will be sorted?

- ○ A gap value of 4 cannot be used on an array with 6 elements.
- ○ 2
- ○ 3
- ○ 4

Insertion sort for interleaved arrays

If a gap value of K is chosen, creating K entirely new arrays would be computationally expensive. So instead of creating new arrays, shell sort sorts interleaved arrays in-place with a variant of the insertion sort algorithm. The variant redefines the concept of "next" and "previous" elements. For an element at index X, the next element is at X + K, instead of X + 1, and the previous element is at X - K instead of X - 1.

PARTICIPATION ACTIVITY | 3.6.3: Interleaved insertion sort.

```
InsertionSortInterleaved(numbers, numbersSize, startIndex, gap) {
   i = 0
   j = 0
   temp = 0  // Temporary variable for swap

   for (i = startIndex + gap; i < numbersSize; i = i + gap) {
      j = i
      while (j - gap >= startIndex && numbers[j] < numbers[j - gap]) {
         temp = numbers[j]
         numbers[j] = numbers[j - gap]
         numbers[j - gap] = temp
         j = j - gap
```

```
        }
    }
}
```

Animation content:

Static figure:
Begin pseudocode:

```
InsertionSortInterleaved(numbers, numbersSize, startIndex, gap) {
  i = 0
  j = 0
  temp = 0  // Temporary variable for swap

  for (i = startIndex + gap; i < numbersSize; i = i + gap) {
    j = i
    while (j - gap >= startIndex && numbers[j] < numbers[j - gap]) {
      temp = numbers[j]
      numbers[j] = numbers[j - gap]
      numbers[j - gap] = temp
      j = j - gap
    }
  }
}
```

End pseudocode.
Three 9 element arrays are shown.
The first array, labeled, "Original numbers array:": [88, 67, 91, 45, 14, 68 ,71 ,26, 64].
The first array points to 3 code lines: "InsertionSortInterleaved(array, 9, 0, 3)", "InsertionSortInterleaved(array, 9, 1, 3)", and "InsertionSortInterleaved(array, 9, 2, 3)".
The three code lines point to the second array: [45, 14, 64, 71, 26, 68, 88, 67, 91].

The second array points to 1 code line: "InsertionSortInterleaved(array, 9, 0, 1)".
The 1 code line points two the third array: [14, 26, 45, 64, 67, 68, 71, 88, 91]

Step 1: Calling InsertionSortInterleaved() with a start index of 0 and a gap of 3 sorts the interleaved array with elements at indices 0, 3, and 6. i and j are first assigned with 3.
Original numbers array is shown as: [88, 67, 91, 45, 14, 68, 71, 26, 64]. First function call occurs: InsertionSortInterleaved(list, 9, 0, 3).

Step 2: When swapping elements 45 and 88, 45 jumps the gap and moves towards the front more quickly compared to the regular insertion sort.
Highlight shows code execution inside the InsertionSortInterleaved(list, 9, 0, 3) function call. The first iteration of the nested while loop swaps elements at indices 0 and 3, yielding the array: [45, 67, 91, 88, 14, 68, 71, 26, 64].

Step 3: The sort continues, putting 45, 71, and 88 in the correct order.
Next iteration of nested while loops swaps elements at indices 3 and 6, yielding the array: [45, 67, 91, 71, 14, 68, 88, 26, 64].

Step 4: Only one of three interleaved arrays has been sorted. Two more InsertionSortInterleaved() calls are needed, with starting indices of 1 and 2.
Next two function calls appear: InsertionSortInterleaved(list, 9, 1, 3) and InsertionSortInterleaved(list, 9, 2, 3). Execution of InsertionSortInterleaved(list, 9, 1, 3) swaps elements and indices 1 and 4, then 4 and 7. Execution of InsertionSortInterleaved(list, 9, 2, 3) swaps elements at indices 2 and 5, then 5 and 8, then 2 and 5. Resulting array is: [45, 14, 64, 71, 26, 68, 88, 67, 91].

Step 5: Calling InsertionSortInterleaved() with a starting index of 0 and a gap of 1 is equivalent to the regular insertion sort, and finishes sorting the array.
Final function call appears: InsertionSortInterleaved(list, 9, 0, 1). The standard insertion sort algorithm moves: element 14 down to index 0, element 26 down to index 1, element 68 down to index 4, element 67 down to index 4. Resulting array is sorted: [14, 26, 45, 64, 67, 68, 71, 88, 91].

Animation captions:

1. Calling InsertionSortInterleaved() with a start index of 0 and a gap of 3 sorts the interleaved array with elements at indices 0, 3, and 6. i and j are first assigned with 3.
2. When swapping elements 45 and 88, 45 jumps the gap and moves towards the front more quickly compared to the regular insertion sort.
3. The sort continues, putting 45, 71, and 88 in the correct order.
4. Only one of three interleaved arrays has been sorted. Two more InsertionSortInterleaved() calls are needed, with starting indices of 1 and 2.
5. Calling InsertionSortInterleaved() with a starting index of 0 and a gap of 1 is equivalent to the regular insertion sort, and finishes sorting the array.

PARTICIPATION ACTIVITY | 3.6.4: Insertion sort variant.

For each call to `InsertionSortInterleaved(numbersArray, X, ...)`, assume that numbersArray is an array of length X. Each question can be answered without knowing the contents of numbersArray.

1) `InsertionSortInterleaved(numbersArray, 10, 0, 5)` is called. What are the indices of the first two elements compared?

- ○ 1 and 5
- ○ 1 and 6
- ○ 0 and 4
- ○ 0 and 5

2) `InsertionSortInterleaved(numbersArray, 4, 1, 4)` is called. What is the value of the loop variable i first initialized to?

- ○ 1
- ○ 4
- ○ 5

3) `InsertionSortInterleaved(numbersArray, 4, 1, 4)` causes an out of bounds array access.

- ○ True
- ○ False

4) If a gap value of 2 is chosen, then the following two function calls will fully sort numbersArray:

```
InsertionSortInterleaved(numbersArray,
9, 0, 2)
InsertionSortInterleaved(numbersArray,
9, 1, 2)
```

- ○ True
- ○ False

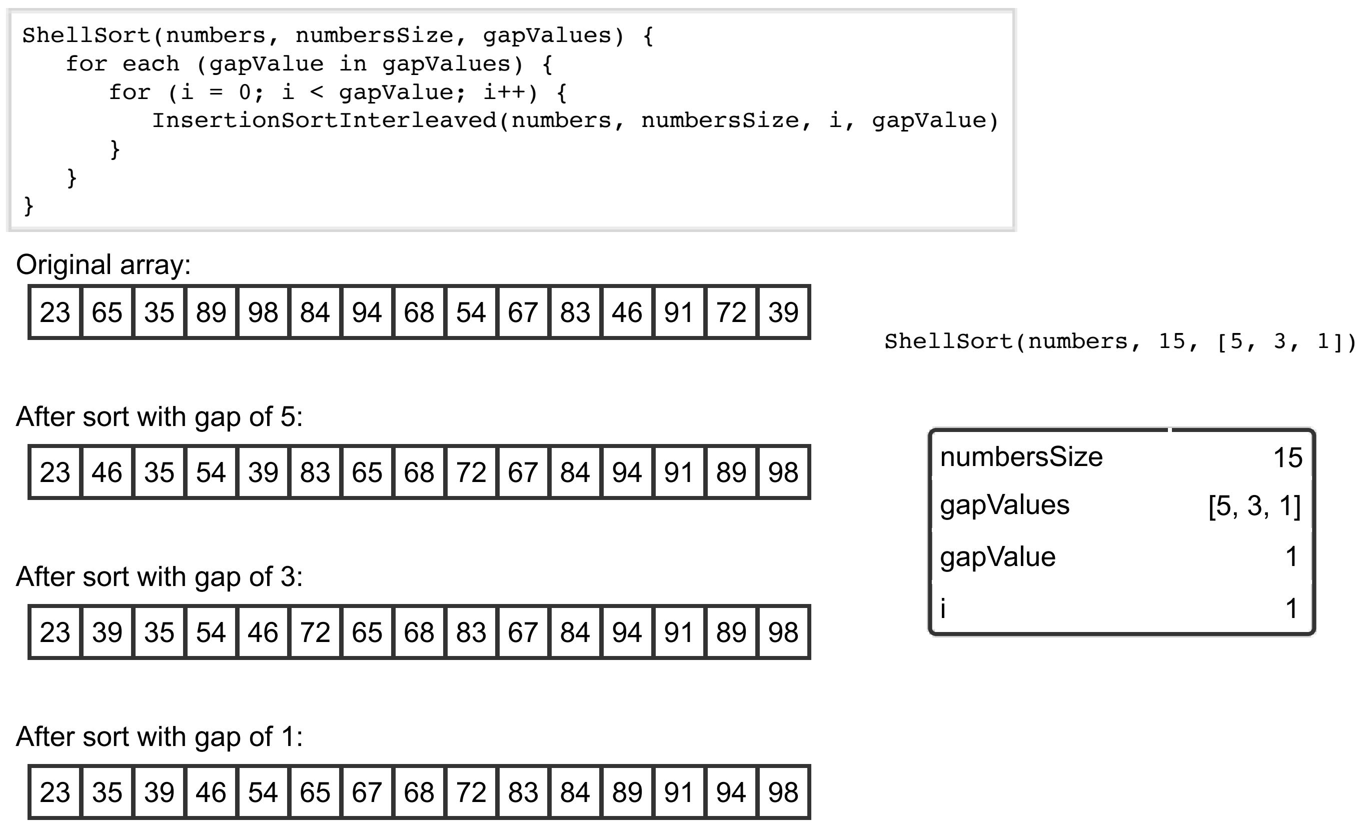
Shell sort algorithm

Shell sort begins by picking an arbitrary collection of gap values. For each gap value K, K calls are made to the insertion sort variant function to sort K interleaved arrays. Shell sort ends with a final gap value of 1, to finish with the regular insertion sort.

Shell sort tends to perform well when choosing gap values in descending order. A common option is to choose powers of 2 minus 1, in descending order. Ex: For an array with 100 elements, gap values would be 63, 31, 15, 7, 3, and 1. This gap selection technique results in shell sort's time complexity being no worse than $O(N^{3/2})$.

Using gap values that are powers of 2 or in descending order is not required. Shell sort will correctly sort arrays using any positive integer gap values in any order, provided a gap value of 1 is included.

PARTICIPATION ACTIVITY 3.6.5: Shell sort algorithm.



```
ShellSort(numbers, numbersSize, gapValues) {
   for each (gapValue in gapValues) {
      for (i = 0; i < gapValue; i++) {
         InsertionSortInterleaved(numbers, numbersSize, i, gapValue)
      }
   }
}
```

Original array:

23	65	35	89	98	84	94	68	54	67	83	46	91	72	39

`ShellSort(numbers, 15, [5, 3, 1])`

After sort with gap of 5:

23	46	35	54	39	83	65	68	72	67	84	94	91	89	98

numbersSize	15
gapValues	[5, 3, 1]
gapValue	1
i	1

After sort with gap of 3:

23	39	35	54	46	72	65	68	83	67	84	94	91	89	98

After sort with gap of 1:

23	35	39	46	54	65	67	68	72	83	84	89	91	94	98

Animation content:

Static Figure:
Begin pseudocode:
ShellSort(numbers, numbersSize, gapValues) {
  for each (gapValue in gapValues) {

```
      for (i = 0; i < gapValue; i++) {
         InsertionSortInterleaved(numbers, numbersSize, i, gapValue)
      }
   }
}
```

End pseudocode.
Four 15-element arrays are shown.
The first array, labeled, "Original array": [23, 65, 35, 89, 98, 84, 94, 68, 54, 67, 83, 46, 91, 72, 39].
The second array, labeled, "After sort with gap of 5": [23, 46, 35, 54, 39, 83, 65, 68, 72, 67, 84, 94, 91, 89, 98].
The third array, labeled, "After sort with gap of 3": [23, 39, 35, 54, 46, 72, 65, 68, 83, 67, 84, 94, 91, 89, 98].
The fourth array, labeled, "After sort with gap of 1": [23, 35, 39, 46, 54, 65, 67, 68, 72, 83, 84, 89, 91, 94, 98].
The code line "ShellSort(numbers, 15, gaps, numGaps)" is shown. A table containing execution flow data is shown:
numbersSize value is 15
gapValues values is [5, 3, 1]
gapValue value is 1
i value is 1

Step 1: The first gap value of 5 causes 5 interleaved arrays to be sorted. The inner for loop iterates over the start indices for the interleaved array. So, i is initialized with the first array's starting index, or 0.
ShellSort() function appears along with the call to sort an array of numbers: ShellSort(numbers, 15, [5, 3, 1]). Original array appears as: [23, 65, 35, 89, 98, 84, 94, 68, 54, 67, 83, 46, 91, 72, 39]. Code highlighter begins stepping through code, executing the first InsertionSortInterleaved() call with i=0 and gapValue=5. Array elements at indices 5 and 10 are swapped, yielding the array: [23, 65, 35, 89, 98, 83, 94, 68, 54, 67, 84, 46, 91, 72, 39].

Step 2: The second for loop iteration sorts the interleaved array starting at index 1 with the same gap value of 5.
Execution of loop for i = 1 is shown. Elements at indices 6 and 11 are swapped, then elements at indices 1 and 6. Resulting array: [23, 46, 35, 89, 98, 83, 65, 68, 54, 67, 84, 94, 91, 72, 39].

Step 3: For the gap value 5, the remaining interleaved arrays at start indices 2, 3, and 4 are sorted. Remaining iterations of i loop are shown for the gap value of 5. Element swaps occur at index pairs: 3 and 8, 8 and 13, 4 and 9, 9 and 14, and 4 and 9. Resulting array: [23, 46, 35, 54, 39, 83, 65, 68, 72, 67, 84, 94, 91, 89, 98].

Step 4: The next gap value of 3 causes 3 interleaved arrays to be sorted. Few swaps are needed because the array is already partially sorted.

Iterations of i loop are shown for the gap value of 3. Element swaps occur at index pairs: 1 and 4 and 5 and 8. Resulting array: [23, 39, 35, 54, 46, 72, 65, 68, 83, 67, 84, 94, 91, 89, 98].

Step 5: The final gap value of 1 finishes sorting.
Execution of final InsertionSortInterleaved() call occurs, producing a sorted array: [23, 35, 39, 46, 54, 65, 67, 68, 72, 83, 84, 89, 91, 94, 98].

Animation captions:

1. The first gap value of 5 causes 5 interleaved arrays to be sorted. The inner for loop iterates over the start indices for the interleaved array. So, i is initialized with the first array's starting index, or 0.
2. The second for loop iteration sorts the interleaved array starting at index 1 with the same gap value of 5.
3. For the gap value 5, the remaining interleaved arrays at start indices 2, 3, and 4 are sorted.
4. The next gap value of 3 causes 3 interleaved arrays to be sorted. Few swaps are needed because the array is already partially sorted.
5. The final gap value of 1 finishes sorting.

PARTICIPATION ACTIVITY 3.6.6: Shell sort algorithm.

1) ShellSort() properly sorts an array using any collection of gap values, provided the collection contains 1.

- True
- False

2) Calling ShellSort() with gaps [7, 3, 1] vs. [3, 7, 1] produces the same result with no difference in efficiency.

- True
- False

3) How many times is InsertionSortInterleaved() called if ShellSort() is called with gap array [10, 2, 1]?

- 3
- 13

○ 20

CHALLENGE ACTIVITY | 3.6.1: Shell sort.

704586.1152092.qx3zqy7

Start

The following array is given: [67, 97, 21, 56, 37, 60, 17, 48]
A gap value of 2 is used, so the array is treated as 2 interleaved arrays.

What is the first interleaved array?

[Ex: 1, 2, 3]
(comma between values)

What is the second interleaved array?

[ ]

1	2	3

Check Next

View solution (Instructors only)

3.7 C++: Shell sort

Sorting interleaved arrays for shell sort

Shell sort uses a variation of the insertion sort algorithm to sort subsets of an input array. Instead of comparing elements that are immediately adjacent to each other, the insertion sort variant compares elements that are at a fixed distance apart, known as a gap space. Shell sort repeats this variation of insertion sort using different gap sizes and starting points within the input array.

Figure 3.7.1: Interleaved insertion sort algorithm.

```
void InsertionSortInterleaved(int* numbers, int numbersSize, int startIndex, int
gap) {
   for (int i = startIndex + gap; i < numbersSize; i += gap) {
      int j = i;
      while (j - gap >= startIndex && numbers[j] < numbers[j - gap]) {
         // Swap numbers[j] and numbers [j - gap]
         int temp = numbers[j];
         numbers[j] = numbers[j - gap];
         numbers[j - gap] = temp;
         j -= gap;
      }
   }
}
```

PARTICIPATION ACTIVITY 3.7.1: Sorting interleaved arrays.

1) InsertionSort() is the same as InsertionSortInterleaved() with a gap space of 1.

- ○ True
- ○ False

2) If the InsertionSortInterleaved() function runs with an array of size 10, startIndex assigned with 2, and gap assigned with 3, then the loop variable i will be assigned with the values 2, 5, and 8.

- ○ True
- ○ False

Shell sort algorithm

The ShellSort() function calls the InsertionSortInterleaved() function repeatedly using different gap sizes and start indices. Ex: If a gapValue is 3, then ShellSort() will execute:

- `InsertionSortInterleaved(numbers, numbersSize, 0, 3)`
- `InsertionSortInterleaved(numbers, numbersSize, 1, 3)`
- `InsertionSortInterleaved(numbers, numbersSize, 2, 3)`

All values from zero to gap - 1 are used as startIndex. This process repeats for all gap values. The ShellSort() function takes as parameters the array to be sorted, and the array of gap values to be used.

Figure 3.7.2: Shell sort algorithm in C++.

```
void ShellSort(int* numbers, int numbersSize, int* gapValues, int gapValuesSize) {
   for (int g = 0; g < gapValuesSize; g++) {
      for (int i = 0; i < gapValues[g]; i++) {
         InsertionSortInterleaved(numbers, numbersSize, i, gapValues[g]);
      }
   }
}
```

Choosing good gap values will minimize the total number of swap operations. The only requirement for the shell sort algorithm is that one of the gap values (usually the last one) is 1. Gap values are often chosen in decreasing order. A gap value of 1 is equivalent to the regular insertion sort algorithm. Thus, if larger gap values are previously used, the final insertion sort will do less work than if the regular insertion sort is done throughout.

PARTICIPATION ACTIVITY 3.7.2: Shell sort.

1) Which is a reasonable choice for gap values, given an input array of size 75?

- ○ 1, 3, 7, 15, 31
- ○ 31, 15, 7, 3, 1
- ○ 1, 2, 3

2) If ShellSort() is run with an input array of size 20 and a gap values array of [15, 7, 3, 1], how many times will InsertionSortInterleaved() be called?

- ○ 80
- ○ 4
- ○ 26

zyDE 3.7.1: Shell sort algorithm.

The program below uses the shell sort algorithm to sort various arrays. The code has been modified to also calculate how many comparisons and swaps occur.

ShellSort() is called three times to sort three distinct arrays. The first is unsorted, the second is sorted in ascending order, and the third is sorted in descending order. Try running the program with different arrays to see how the total number of comparisons and swaps changes (or doesn't change).

Current file: **ShellSortDemo.cpp** ▾ **Load default template...**

```
#include <iostream>
#include <string>
#include "SortTracker.h"
using namespace std;

int InsertionSortInterleaved(int* numbers, int numbersSize, int startIndex,
   int gap, SortTracker& tracker) {
   int swapsBefore = tracker.GetSwapCount();

   for (int i = startIndex + gap; i < numbersSize; i += gap) {
      int j = i;
      while (j - gap >= startIndex &&
         tracker.IsFirstLTSecond(numbers, j, j - gap)) {
         // Swap numbers[j] and numbers[j - gap]
         tracker.Swap(numbers, j, j - gap);

         // Decrease j in preparation for next comparison
         j -= gap;
      }
   }

   // Return the number of swaps that occurred during this function call
   return tracker.GetSwapCount() - swapsBefore;
}

int ShellSort(int* numbers, int numbersSize, int* gapValues, int gapValuesSize
   SortTracker& tracker) {
   int totalSwaps = 0;
   for (int g = 0; g < gapValuesSize; g++) {
      int swapsForGap = 0;
      for (int i = 0; i < gapValues[g]; i++) {
         swapsForGap += InsertionSortInterleaved(numbers, numbersSize, i,
            gapValues[g], tracker);
      }
      tracker.PrintArray(numbers, numbersSize, cout, ", ",
         "                  [", "]");
      cout << " (after " << swapsForGap;
      cout << " swap" << (swapsForGap == 1 ? "" : "s") << " with gap=";
      cout << gapValues[g] << ")" << endl;
      totalSwaps += swapsForGap;
   }
   return totalSwaps;
}

void ShellSortDemo(int* numbersArray, int arrayLength) {
   SortTracker tracker;
   int gapValues[] = { 4, 2, 1 };
   int gapValuesLength = sizeof(gapValues) / sizeof(gapValues[0]);

   // Display the contents of the array
   tracker.PrintArray(numbersArray, arrayLength, cout, ", ",
      "Before sorting:    [", "]\n");

   // Sort the numbers array using the shell sort algorithm
   ShellSort(numbersArray, arrayLength, gapValues, gapValuesLength, tracker);

   // Display the sorted contents of the array
   tracker.PrintArray(numbersArray, arrayLength, cout, ", ",
      "After sorting:     [", "]\n");

   // Display stats
   cout << "Total comparisons: " << tracker.GetComparisonCount() << endl;
   cout << "Total swaps:       " << tracker.GetSwapCount() << endl;
}

int main() {
   // Create arrays:
   // - array1 is unsorted
   // - array2 is sorted in ascending order
   // - array3 is sorted in descending order
   int array1[] = { 33, 18, 78, 64, 45, 32, 70, 11, 27 };
   int array1Length = sizeof(array1) / sizeof(array1[0]);
   int array2[] = { 10, 20, 30, 40, 50, 60, 70, 80, 90 };
   int array2Length = sizeof(array2) / sizeof(array2[0]);
   int array3[] = { 99, 88, 77, 66, 55, 44, 33, 22, 11 };
   int array3Length = sizeof(array3) / sizeof(array3[0]);

   cout << "Demo 1 - Unsorted array:" << endl;
   ShellSortDemo(array1, array1Length);
   cout << endl << "Demo 2 - Array sorted in ascending order:" << endl;
   ShellSortDemo(array2, array2Length);
   cout << endl << "Demo 3 - Array sorted in descending order:" << endl;
   ShellSortDemo(array3, array3Length);

   return 0;
}
```

Run

3.8 Quicksort

Overview and partitioning algorithm

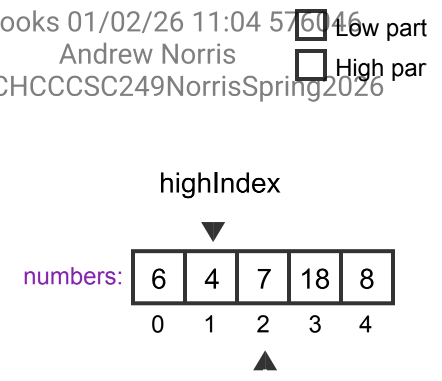
Quicksort is a sorting algorithm that repeatedly partitions the input into low and high parts (each part unsorted), and then recursively sorts each part. To partition, quicksort chooses a pivot to divide the data into low and high parts. The ***pivot*** can be any value within the array being sorted and is commonly the middle element's value. Ex: For the array [4, 34, 10, 25, 1], the middle element is located at index 2 (the middle of indices 0 and 4) and has a value of 10.

Once the pivot is chosen, the quicksort algorithm divides the array into two parts, referred to as the low partition and the high partition. All values in the low partition are less than or equal to the pivot value. All values in the high partition are greater than or equal to the pivot value. The values in each partition are not necessarily sorted. Ex: Partitioning [4, 34, 10, 25, 1] with a pivot value of 10 results in a low partition of [4, 1, 10] and a high partition of [25, 34]. Values equal to the pivot may appear in either or both of the partitions.

The partitioning algorithm uses two index variables, a low index and a high index, initialized to the left and right sides of the current elements being sorted. While the element at the low index is less than the pivot, the low index is incremented, because that element should remain in the low partition. Likewise, while the element at the high index is greater than the pivot, the high index is decremented, because that element should remain in the high partition. Then, if lowIndex >= highIndex, all elements have been partitioned, and the partitioning algorithm returns highIndex, which is the index of the low partition's last element. Otherwise, the elements at the low and high indices are swapped to move those elements to the correct partitions. The algorithm then increments lowIndex, decrements highIndex, and repeats.

PARTICIPATION ACTIVITY 3.8.1: Quicksort partitions data into a low partition with values ≤ pivot and a high partition with values ≥ pivot.

```
Partition(numbers, lowIndex, highIndex) {
   // Pick middle element as pivot
   midpoint = lowIndex + (highIndex - lowIndex) / 2
   pivot = numbers[midpoint]

   done = false
   while (!done) {
      while (numbers[lowIndex] < pivot) {
         lowIndex += 1
      }
      while (pivot < numbers[highIndex]) {
         highIndex -= 1
      }
```



```
      // If zero or one elements remain, then all numbers are
      // partitioned
      if (lowIndex >= highIndex) {
         done = true
      }
      else {
         // Swap numbers[lowIndex] and numbers[highIndex]
         temp = numbers[lowIndex]
         numbers[lowIndex] = numbers[highIndex]
         numbers[highIndex] = temp

         // Update lowIndex and highIndex
         lowIndex += 1
         highIndex -= 1
      }
   }
   return highIndex
}
```

Animation content:

Begin pseudocode:

```
Partition(numbers, lowIndex, highIndex) {
  // Pick middle element as pivot
  midpoint = lowIndex + (highIndex - lowIndex) / 2
  pivot = numbers[midpoint]

  done = false
  while (!done) {
    while (numbers[lowIndex] < pivot) {
      lowIndex += 1
    }
    while (pivot < numbers[highIndex]) {
      highIndex -= 1
    }

    // If zero or one elements remain, then all numbers are
    // partitioned
    if (lowIndex >= highIndex) {
      done = true
    }
    else {
      // Swap numbers[lowIndex] and numbers[highIndex]
      temp = numbers[lowIndex]
      numbers[lowIndex] = numbers[highIndex]
```

```
        numbers[highIndex] = temp

        // Update lowIndex and highIndex
        lowIndex += 1
        highIndex -= 1
      }
    }
    return highIndex
}
```

End pseudocode.

Step 1: Array is shown as [7, 4, 6, 18, 8]. Labels lowIndex and highIndex point to array indices 0 and 4, respectively. Code execution begins within the Partition function, and the calculation of the midpoint is shown as:
lowIndex + (highIndex - lowIndex) / 2
= 0 + (4 - 0) / 2
= 2
The pivot variable is then assigned with numbers[midpoint], or 6.

Step 2: The first nested while loop executes. Since the condition 7 < 6 is false, lowIndex is not incremented and remains 0.

Step 3: The second nested while loop executes. Conditions 6 < 8 and 6 < 18 are true, so highIndex is decremented twice to become 2. The condition 6 < 6 is false, so the second nested while loop then ends.

Step 4: Elements at indices 0 and 2 (lowIndex and highIndex) are swapped, yielding the array: [6, 4, 7, 18, 8].

Step 5: Execution continues, changing lowIndex and highIndex to 1. The next iteration of the outermost loop begins. lowIndex is incremented to 2, since 4 < 6. The condition 7 < 6 is false, so lowIndex is not incremented further. The second nested while loop does not decrement highIndex, since the condition 6 < 4 is false. The following if statement's condition of lowIndex >= highIndex is now true, so the variable done is assigned with true.

Step 6: The Partition() function's execution ends, returning highIndex's value of 1.

Animation captions:

1. The pivot value is the value of the middle element.
2. lowIndex is incremented until a value greater than or equal to the pivot is found.
3. highIndex is decremented until a value less than or equal to the pivot is found.
4. Elements at indices lowIndex and highIndex are swapped, moving those elements to the

correct partitions.

5. The partition process repeats until indices lowIndex and highIndex reach or pass each other, indicating all elements have been partitioned.
6. Once partitioned, the algorithm returns highIndex, which is the highest index of the low partition. The partitions are not yet sorted.

PARTICIPATION ACTIVITY 3.8.2: Quicksort pivot location and value.

Determine the midpoint and pivot values.

1) numbers = [1, 2, 3, 4, 5], lowIndex = 0, highIndex = 4

`midpoint =`

Check Show answer

2) numbers = [1, 2, 3, 4, 5], lowIndex = 0, highIndex = 4

`pivot =`

Check Show answer

3) numbers = [200, 11, 38, 9], lowIndex = 0, highIndex = 3

`midpoint =`

Check Show answer

4) numbers = [200, 11, 38, 9], lowIndex = 0, highIndex = 3

`pivot =`

Check Show answer

5) numbers = [55, 7, 81, 26, 0, 34, 68, 125], lowIndex = 3, highIndex = 7

`midpoint =`

Check Show answer

6) numbers = [55, 7, 81, 26, 0, 34, 68, 125], lowIndex = 3, highIndex = 7

`pivot =` ______

Check Show answer

PARTICIPATION ACTIVITY | 3.8.3: Low and high partitions.

Determine if the low and high partitions are correct given highIndex and pivot.

1) pivot = 35, highIndex = 2

1	4	35	62	98

○ Correct

○ Incorrect

2) pivot = 65, highIndex = 3

29	17	65	39	84

○ Correct

○ Incorrect

3) pivot = 5, highIndex = 1

5	13	16	77	84	20	19

○ Correct

○ Incorrect

4) pivot = 8, highIndex = 2

8	3	8	41	57	8

○ Correct

○ Incorrect

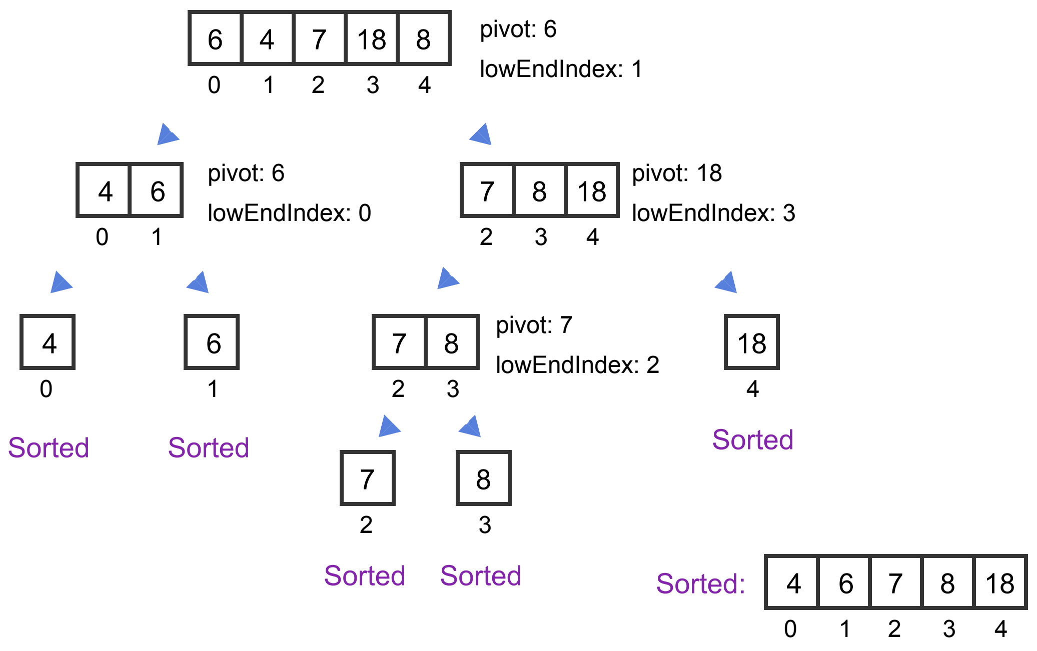
Recursively sorting partitions

Once partitioned, each partition needs to be sorted. Quicksort is typically implemented as a recursive algorithm using calls to quicksort() to sort the low and high partitions. This recursive sorting process continues until a partition has one or zero elements, and thus is already sorted.

PARTICIPATION ACTIVITY 3.8.4: Quicksort.

```
Quicksort(numbers, lowIndex, highIndex) {
   if (highIndex <= lowIndex) {
      return
   }

   lowEndIndex = Partition(numbers, lowIndex, highIndex)
   Quicksort(numbers, lowIndex, lowEndIndex)
   Quicksort(numbers, lowEndIndex + 1, highIndex)
}
```

☐ Low partition
☐ High partition



Animation content:

Static Figure:
Begin pseudocode:
Quicksort(numbers, lowIndex, highIndex) {
  if (highIndex <= lowIndex) {
    return
  }

  lowEndIndex = Partition(numbers, lowIndex, highIndex)
  Quicksort(numbers, lowIndex, lowEndIndex)

```
   Quicksort(numbers, lowEndIndex + 1, highIndex)
}
```

End pseudocode.
A 5 element array is shown. The array contains 5 values, 6, 4, 7, 18, and 8 consecutively. The indexes start from 0 to 4. Values 6 and 4 have a background of gray indicating the values as low partition. Values 7, 18, and 8 have a background of orange indicating the values as high partition. The pivot value of 6 and the lowEndIndex value of 1 is shown.
The 5 element array is broken into two separate arrays. The first broken array contains the values 4, of index 0 and a background color of gray and 6, of index 1 and a background color of orange. The pivot value of 6 and the lowEndIndex value of 0 is shown. This array is broken into two separate arrays again showing that each element has been sorted. 4 at index 0 and 6 at index 1.

The second broken array contains the values 7, of index 2 and a background color of gray, 8, of index 3 and a background color of gray, and 18, of index 4 with a background color of orange. The pivot value of 18 and the lowEndIndex value of 3 is shown. This array is broken into two separate arrays again. The first broken array contains the values 7, of index 2 and a background color of gray and 8, of index 3 and a background color of orange. The pivot value of 7 and the lowEndIndex value of 2 is shown. This array is broken into two separate arrays again showing that each element has been sorted. 7 at index 2 and 8 at index 3. The second broken array shows that 18 has been sorted. 18 at index 4.

The full sorted array is shown at the bottom; 4, 6, 7, 8, 18 with the corresponding indexes, 0, 1, 2, 3, 4.

Step 1: The array from low index 0 to high index 4 has more than one element, so Partition() is called.
First call to Quicksort() occurs. Full array is shown: [6, 4, 7, 18, 8].

Step 2: Quicksort() is called recursively to sort the low and high partitions.
Low and high partitions are shown for the two recursive calls. Low partition: [6, 4]. High partition: [7, 18, 8].

Step 3: The low partition has more than one element. Partition() is called for the low partition, followed by recursive calls to Quicksort().
Partitioning of [6, 4] occurs, producing [4, 6]. Arrays for the two recursive Quicksort() calls are shown: [4] and [6].

Step 4: Each partition that has one element is already sorted.
Calls to sort [4] and [6] return in Quicksort()'s first if statement, since each has only one element.

Step 5: The high partition has more than one element and thus is partitioned and recursively

sorted.
Partitioning of [7, 18, 8] occurs, yielding [7, 8, 18]. Arrays for the two recursive Quicksort() calls are shown: [7, 8] and [18].

Step 6: The low partition with two elements is partitioned and recursively sorted.
Recursive sorting of [7, 8] begins. Arrays for the two recursive Quicksort() calls are shown: [7] and [8].

Step 7: Each remaining partition with only one element is already sorted.
Partitions [7] and [8] each have only one element and so are already sorted.

Step 8: All elements are sorted.
All single-element partitions are combined visually to yield the sorted array: [4, 6, 7, 8, 18].

Animation captions:

1. The array from low index 0 to high index 4 has more than one element, so Partition() is called.
2. Quicksort() is called recursively to sort the low and high partitions.
3. The low partition has more than one element. Partition() is called for the low partition, followed by recursive calls to Quicksort().
4. Each partition that has one element is already sorted.
5. The high partition has more than one element and thus is partitioned and recursively sorted.
6. The low partition with two elements is partitioned and recursively sorted.
7. Each remaining partition with only one element is already sorted.
8. All elements are sorted.

Figure 3.8.1: Quicksort algorithm.

```
Partition(numbers, lowIndex, highIndex) {
   // Pick middle element as pivot
   midpoint = lowIndex + (highIndex - lowIndex) / 2
   pivot = numbers[midpoint]

   done = false
   while (!done) {
      // Increment lowIndex while numbers[lowIndex] < pivot
      while (numbers[lowIndex] < pivot) {
         lowIndex += 1
      }

      // Decrement highIndex while pivot < numbers[highIndex]
      while (pivot < numbers[highIndex]) {
         highIndex -= 1
      }

      // If zero or one elements remain, then all numbers are
      // partitioned. Return highIndex.
      if (lowIndex >= highIndex) {
         done = true
      }
      else {
         // Swap numbers[lowIndex] and numbers[highIndex]
         temp = numbers[lowIndex]
         numbers[lowIndex] = numbers[highIndex]
         numbers[highIndex] = temp

         // Update lowIndex and highIndex
         lowIndex += 1
         highIndex -= 1
      }
   }

   return highIndex
}

Quicksort(numbers, lowIndex, highIndex) {
   // Base case: If the partition size is 1 or zero
   // elements, then the partition is already sorted
   if (highIndex <= lowIndex) {
      return
   }

   // Partition the data within the array. Value lowEndIndex
   // returned from partitioning is the index of the low
   // partition's last element.
   lowEndIndex = Partition(numbers, lowIndex, highIndex)

   // Recursively sort low partition (lowIndex to lowEndIndex)
   // and high partition (lowEndIndex + 1 to highIndex)
   Quicksort(numbers, lowIndex, lowEndIndex)
   Quicksort(numbers, lowEndIndex + 1, highIndex)
}

main() {
   numbers[] = { 10, 2, 78, 4, 45, 32, 7, 11 }
   NUMBERS_SIZE = 8
   i = 0

   print("UNSORTED: ")
   for (i = 0; i < NUMBERS_SIZE; ++i) {
      print(numbers[i] + " ")
   }
   printLine()

   // Initial call to quicksort
   Quicksort(numbers, 0, NUMBERS_SIZE - 1)

   print("SORTED: ")
   for (i = 0; i < NUMBERS_SIZE; ++i) {
      print(numbers[i] + " ")
   }
   printLine()
}
```

```
UNSORTED: 10 2 78 4 45 32 7 11
SORTED: 2 4 7 10 11 32 45 78
```

PARTICIPATION ACTIVITY 3.8.5: Quicksort algorithm.

1) Quicksort() makes recursive calls if the partition size is > 0.
   - ○ True
   - ○ False

2) Partitioning must occur before the recursive calls.
   - ○ True
   - ○ False

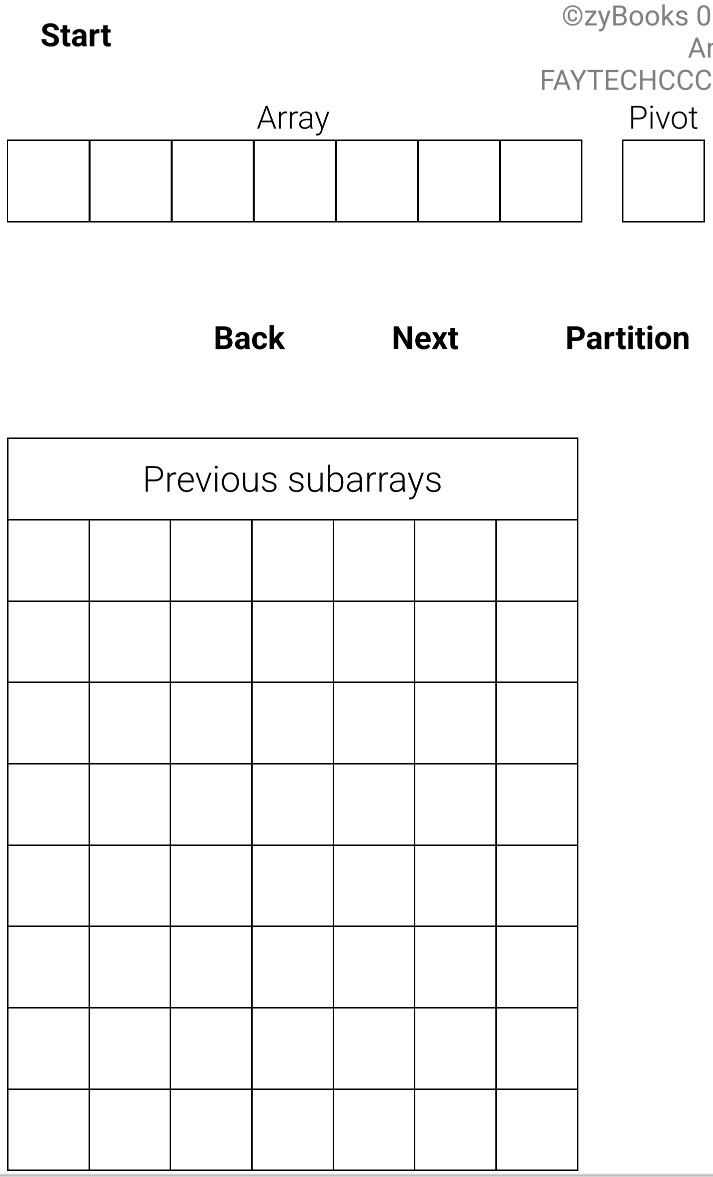
3) The low partition must be recursively sorted before the high partition.
   - ○ True
   - ○ False

Quicksort activity

The following activity helps build intuition as to how partitioning an array into two unsorted parts, one part <= a pivot value and the other part >= a pivot value, and then recursively sorting each part, ultimately leads to a sorted array.

PARTICIPATION ACTIVITY | 3.8.6: Quicksort tool.

Select all values in the unshaded subarray



Quicksort runtime

Quicksort's average and best case runtimes are both O(N log N). Quicksort has several partitioning levels, the first level dividing the input into 2 parts, the second into 4 parts, the third into 8 parts, etc. At each level, the algorithm does at most N comparisons moving the lowIndex and highIndex indices. If the pivot yields two equal-sized parts, then log N levels will exist, requiring the N * log N comparisons.

PARTICIPATION ACTIVITY | 3.8.7: Quicksort runtime.

Assume quicksort always chooses a pivot that divides the elements into two equal parts.

1) How many partitioning levels are required for an array of 8 elements?

Check Show answer

2) How many partitioning levels are required for an array of 1024 elements?

Check Show answer

3) How many total comparisons are required to sort an array of 1024 elements?

Check Show answer

Worst case runtime

For typical unsorted data, partitioning often yields similarly sized parts. However, partitioning may yield unequally sized parts in some cases. If the pivot is the smallest or largest element, one partition will have just 1 element, and the other partition will have all other elements. If unequal partitioning happens at every level, N - 1 levels will exist, yielding N + N-1 + N-2 + ... + 2 + 1 = $(N+1) \cdot (N/2)$, which is O(N^2). So the worst case runtime for the quicksort algorithm is O(N^2). Fortunately, the worst case runtime rarely occurs.

PARTICIPATION ACTIVITY 3.8.8: Worst case quicksort runtime.

Assume quicksort always chooses the smallest element as the pivot.

1) Given numbers = [7, 4, 2, 25, 19], lowIndex = 0, and highIndex = 4, what are the contents of the low partition? Type answer as: 1, 2, 3

Check Show answer

2) How many partitioning levels are required for an array of 5 elements?

Check Show answer

3) How many partitioning levels are required for an array of 1024 elements?

Check Show answer

4) How many total calls to Quicksort() are made to sort an array of 1024 elements?

Check Show answer

CHALLENGE ACTIVITY 3.8.1: Quicksort.

704586.1152092.qx3zqy7

Start

Given numbers = [16, 56, 35, 93, 79], lowIndex = 0, highIndex = 4

When quicksort is used:

what is the midpoint? Ex: 9

what is the pivot?

1	2	3	4	5	6

Check Next

View solution (Instructors only)

3.9 C++: Quicksort

The Partition() function

The quicksort algorithm works by partitioning a section of the unsorted array into a left part and a right part, based on a chosen element within the array called the pivot. The Partition() function has three parameters: the unsorted array, the low index, and the high index. When the function completes, the elements between the low and high indices are reorganized so that the elements in the left part are less than or equal to the pivot and the elements in the right part are greater than or equal to the pivot. The left and right parts may be different sizes, and the function returns the index of the last item in the left part. The left and right parts are not, themselves, sorted.

Figure 3.9.1: The Partition() function used by quicksort.

```
int Partition(int* numbers, int lowIndex, int highIndex) {
   // Pick middle element as the pivot
   int midpoint = lowIndex + (highIndex - lowIndex) / 2;
   int pivot = numbers[midpoint];

   bool done = false;
   while (!done) {
      // Increment lowIndex while numbers[lowIndex] < pivot
      while (numbers[lowIndex] < pivot) {
         lowIndex++;
      }

      // Decrement highIndex while pivot < numbers[highIndex]
      while (pivot < numbers[highIndex]) {
         highIndex--;
      }

      // If lowIndex and highIndex have met or crossed each other, then
      // partitioning is done
      if (lowIndex >= highIndex) {
         done = true;
      }
      else {
         // Swap array elements at lowIndex and highIndex
         int temp = numbers[lowIndex];
         numbers[lowIndex] = numbers[highIndex];
         numbers[highIndex] = temp;

         // Update lowIndex and highIndex
         lowIndex++;
         highIndex--;
      }
   }

   // highIndex is the last index in the left partition
   return highIndex;
}
```

PARTICIPATION ACTIVITY 3.9.1: Array partitioning.

1) The value 15 would be selected as the pivot in the array:
[5, 3, 15, 72, 14, 41, 32, 18].

- True
- False

2) After partitioning, the left and right parts are sorted.

- ○ True
- ○ False

3) The following array is properly partitioned with pivot 17:
[3, 1, 14, 12, 19, 17, 22]

- ○ True
- ○ False

4) The partition function has $O(N)$ runtime complexity.

- ○ True
- ○ False

The Quicksort() function

The Quicksort() function uses recursion to sort the two parts of the array, thus sorting the full array. The function has three parameters: the unsorted array, the start index, and the end index. Quicksort() starts by calling Partition() to partition the array into left (low) and right (high) parts. Quicksort() then calls itself, using recursion to sort the two array parts.

A program can sort an array by calling Quicksort() and specifying startIndex as 0 and endIndex as the index of the last item in the array.

Figure 3.9.2: The Quicksort() algorithm.

```
void Quicksort(int* numbers, int startIndex, int endIndex) {
   // Only sort if at least 2 elements exist
   if (endIndex <= startIndex) {
      return;
   }

   // Partition the array
   int high = Partition(numbers, startIndex, endIndex);

   // Recursively sort the left partition
   Quicksort(numbers, startIndex, high);

   // Recursively sort the right partition
   Quicksort(numbers, high + 1, endIndex);
}
```

PARTICIPATION ACTIVITY 3.9.2: The quicksort algorithm.

1) Quicksort()'s third parameter is the array's length.

   ○ True

   ○ False

2) Suppose the following code is executed:

```
int numbers[] = { 8, 2, 7, 6, 1,
4, 3, 5 };
Quicksort(numbers, 3, 6);
```

   After this code finishes, the numbers array is:
   [8, 2, 7, 1, 3, 4, 6, 5].

   ○ True

   ○ False

zyDE 3.9.1: Quicksort algorithm.

The program below uses the quicksort algorithm to sort various arrays. The code has been modified to calculate how many comparisons and swaps occur.

Quicksort() is called three times to sort three distinct arrays. The first is unsorted, the second is sorted in ascending order, and the third is sorted in descending order. Try running the program with different arrays to see how the total number of comparisons and swaps changes (or doesn't change).

Current file: QuicksortDemo.cpp Load default template...

```
#include <iostream>
#include <string>
#include "SortTracker.h"
using namespace std;

int Partition(int* numbers, int lowIndex, int highIndex, SortTracker& tracker)
   // Pick middle element as the pivot
   int midpoint = lowIndex + (highIndex - lowIndex) / 2;
   int pivot = numbers[midpoint];

   bool done = false;
   while (!done) {
      // Increment lowIndex while numbers[lowIndex] < pivot
      while (tracker.IsLT(numbers[lowIndex], pivot)) {
         lowIndex++;
      }

      // Decrement highIndex while pivot < numbers[highIndex]
      while (tracker.IsLT(pivot, numbers[highIndex])) {
         highIndex--;
      }

      // If lowIndex and highIndex have met or crossed each other, then
      // partitioning is done
      if (lowIndex >= highIndex) {
         done = true;
      }
      else {
         // Swap array elements at lowIndex and highIndex
         tracker.Swap(numbers, lowIndex, highIndex);

         // Update lowIndex and highIndex
         lowIndex++;
         highIndex--;
      }
   }

   // highIndex is the last index in the left partition
   return highIndex;
   return highIndex;
}

void Quicksort(int* numbers, int lowIndex, int highIndex, SortTracker& tracker
   // Only sort if at least 2 elements exist
   if (highIndex <= lowIndex) {
      return;
   }

   // Partition the array
   int lowEndIndex = Partition(numbers, lowIndex, highIndex, tracker);

   // Recursively sort the left partition
   Quicksort(numbers, lowIndex, lowEndIndex, tracker);

   // Recursively sort the right partition
   Quicksort(numbers, lowEndIndex + 1, highIndex, tracker);
}

void QuicksortDemo(int* numbersArray, int arrayLength) {
   SortTracker tracker;

   // Display the contents of the array
   tracker.PrintArray(numbersArray, arrayLength, cout, ", ",
      "Before sorting:    [", "]\n");

   // Sort the numbers array using the quicksort algorithm
   Quicksort(numbersArray, 0, arrayLength - 1, tracker);

   // Display the sorted contents of the array
   tracker.PrintArray(numbersArray, arrayLength, cout, ", ",
      "After  sorting:    [", "]\n");

   // Display stats
   cout << "Total comparisons: " << tracker.GetComparisonCount() << endl;
   cout << "Total swaps:       " << tracker.GetSwapCount() << endl;
}

int main() {
   // Create arrays:
   // - array1 is unsorted
   // - array2 is sorted in ascending order
   // - array3 is sorted in descending order
   int array1[] = { 33, 18, 78, 64, 45, 32, 70, 11, 27 };
   int array1Length = sizeof(array1) / sizeof(array1[0]);
   int array2[] = { 10, 20, 30, 40, 50, 60, 70, 80, 90 };
   int array2Length = sizeof(array2) / sizeof(array2[0]);
   int array3[] = { 99, 88, 77, 66, 55, 44, 33, 22, 11 };
   int array3Length = sizeof(array3) / sizeof(array3[0]);

   cout << "Demo 1 - Unsorted array:" << endl;
   QuicksortDemo(array1, array1Length);
   cout << endl << "Demo 2 - Array sorted in ascending order:" << endl;
   QuicksortDemo(array2, array2Length);
   cout << endl << "Demo 3 - Array sorted in descending order:" << endl;
   QuicksortDemo(array3, array3Length);

   return 0;
}
```

Run

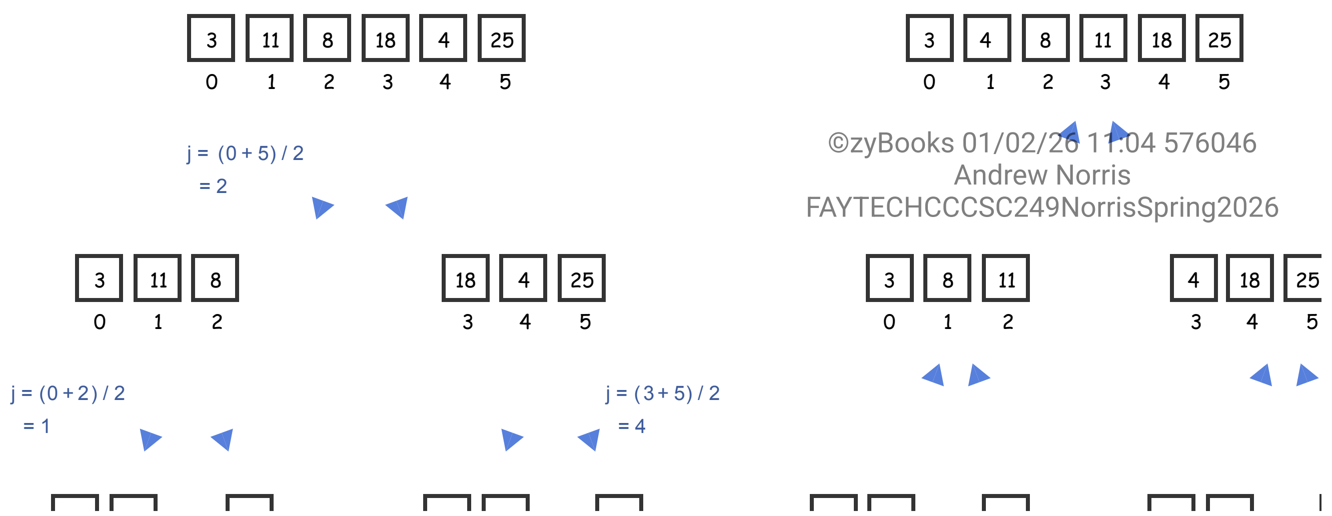
3.10 Merge sort

Merge sort overview and partitioning

Merge sort is a sorting algorithm that divides an array into two halves, recursively sorts each half, and then merges the sorted halves to produce a sorted array. The recursive partitioning continues until an array of 1 element is reached, as an array of 1 element is already sorted.

The algorithm uses three index variables to keep track of the elements to sort for each recursive function call. Two indices, i and k, store the index of the partition's first and last element, respectively. A third index, computed as j = (i + k) / 2, is computed to divide the partition into two halves. Elements from i to j are in the left half, and elements from j + 1 to k are in the right half.

PARTICIPATION ACTIVITY | 3.10.1: Merge sort recursively divides the input into two halves, sorts each half, and merges the halves together.



Animation content:

Static Figure:
Two array sections depicting how Merge sort splits and merges together are shown. The first section splits the original array, and the second section merges the split array into the final array.

In the split section, the original array is shown: [3, 11, 8, 18, 4, 25] with index from 0 to 5. Array is split into two parts: [3, 11, 8] and [18, 4, 25]. The split equation is shown: j = (0+5)/2 = 2. The split array, [3, 11, 8] with index from 0 to 2, is split into two parts: [3, 11] and [8]. The split equation is shown: j = (0+2)/2 = 1. The other split array, [18, 4, 25] with index from 3 to 5, is split into two parts: [18, 4] and [25]. The split equation is shown: j = (3+5)/2 = 4. The split array, [3, 11] with index 0 and 1, is split into two parts: [3] with index 0 and [11] with index 1. The split equation is shown: j = (0+1)/2 = 0. The other split array, [18, 4] with index 3 and 4, is split into two parts: [18] with index 3 and [4] with index 4. The split equation is shown: j = (3+4)/2 = 3.

In the merge section, the split array is shown, [3] index 0, [11] index 1, [8] index 2, [18] index 3, [4] index 4, and [25] index 5. The arrays [3] and [11] merge to [3, 11]. The arrays [3, 11] and [8] merge to [3, 8, 11]. The arrays [18] and [4] merge to [4, 18]. The arrays [4, 18] and [25] merge to [4, 18, 25]. The arrays [3, 8, 11] and [4, 18, 25] merge to the final array [3, 4, 8, 11, 1,8, 25] with index from 0 to 5.

Step 1: MergeSort recursively divides the array into two halves. The index j of the left half's last element is computed as ((low index) + (high index)) / 2.
Original array is shown: [3, 11, 8, 18, 4, 25]. Array is split into two parts: [3, 11, 8] and [18, 4, 25].

Step 2: The array is divided until only one element remains.
Splitting continues. Array [3, 11, 8] is split into [3, 11] and [8]. Array [3, 11] is split into [3] and [11]. Array [18, 4, 25] is split into [18, 4] and [25]. Array [18, 4] is split into [18] and [4].

Step 3: An array of 1 element is already sorted.
Each array with one element is highlighted: [3], [11], [8], [18], [4], and [25]. Each is sorted.

Step 4: At each level, the sorted arrays are merged together while maintaining the sorted order.

Sorted arrays merge. [3] and [11] merge to [3, 11]. [3, 11] and [8] merge to [3, 8, 11]. [18] and [4] merge to [4, 18]. [4, 18] and [25] merge to [4, 18, 25]. [3, 8, 11] and [4, 18, 25] merge to [3, 4, 8, 11, 1,8, 25].

Animation captions:

1. MergeSort recursively divides the array into two halves. The index j of the left half's last element is computed as ((low index) + (high index)) / 2.
2. The array is divided until only one element remains.
3. An array of 1 element is already sorted.
4. At each level, the sorted arrays are merged together while maintaining the sorted order.

PARTICIPATION ACTIVITY 3.10.2: Merge sort partitioning.

1) numbers = [11, 22, 33, 44, 55]

i = 0

k = 4

What is the index of the left partition's last element?

- 0
- 2
- 4

2) numbers = [11, 22, 33, 44, 55]

i = 0

k = 4

What are the left and right partitions?

- Left partition = [11, 22, 33], right partition = [33, 44, 55]
- Left partition = [11, 22, 33], right partition = [44, 55]
- Left partition = [11, 22], right partition = [33, 44, 55]

3) numbers = [34, 78, 14, 23, 8, 35]

i = 3

k = 5

What is the index of the left partition's last element?

- 2
- 3
- 4

4) numbers = [34, 78, 14, 23, 8, 35]

i = 3

k = 5

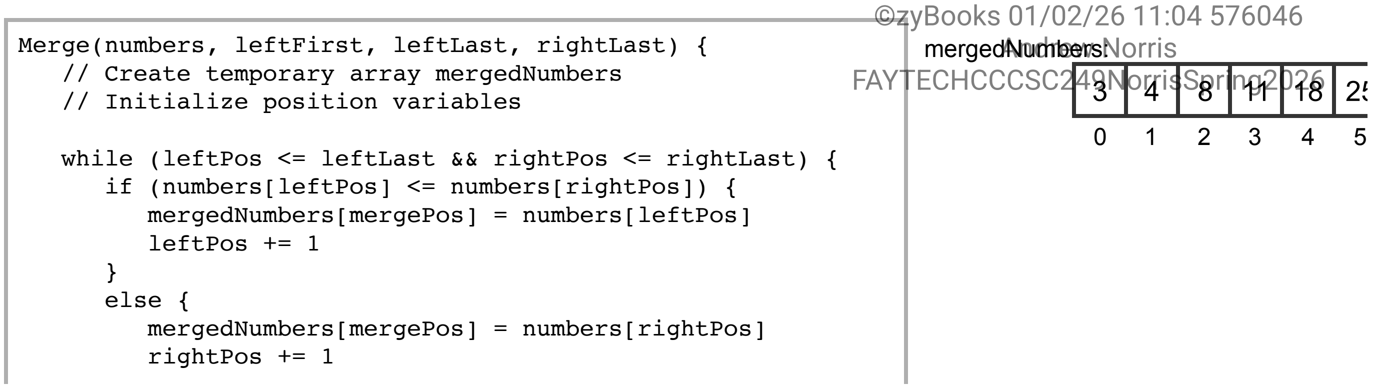
What are the left and right partitions?

- Left partition = [34, 78, 14, 23, 8], right partition = [35]
- Left partition = [23, 8], right partition = [35]
- Left partition = [23, 8, 35], right partition = []

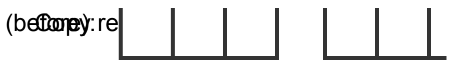
Merge sort algorithm

Merge sort merges the two sorted partitions into a single array by repeatedly selecting the smallest element from either the left or right partition and adding that element to a temporary merged array. Once fully merged, the elements in the temporary merged array are copied back to the original array.

PARTICIPATION ACTIVITY 3.10.3: Merge sort's merging algorithm.



```
Merge(numbers, leftFirst, leftLast, rightLast) {
   // Create temporary array mergedNumbers
   // Initialize position variables

   while (leftPos <= leftLast && rightPos <= rightLast) {
      if (numbers[leftPos] <= numbers[rightPos]) {
         mergedNumbers[mergePos] = numbers[leftPos]
         leftPos += 1
      }
      else {
         mergedNumbers[mergePos] = numbers[rightPos]
         rightPos += 1
```



```
      }
      mergePos += 1
   }

   // If left partition not empty, add remaining elements
   while (leftPos <= leftLast) {
      mergedNumbers[mergePos] = numbers[leftPos]
      leftPos += 1
      mergePos += 1
   }

   // If right partition not empty, add remaining elements
   while (rightPos <= rightLast) {
      mergedNumbers[mergePos] = numbers[rightPos]
      rightPos += 1
      mergePos += 1
   }

   // Copy mergedNumbers back to numbers
   for (mergePos = 0; mergePos < mergedSize; mergePos += 1) {
      numbers[leftFirst + mergePos] = mergedNumbers[mergePos]
   }
}
```

Animation content:

Static figure:
Merge() function code is shown. Three arrays are shown to the right:
mergedNumbers: [3, 4, 8, 11, 18, 25]
numbers: [3, 4, 8, 11, 18, 25]
numbers (before): [3, 8, 11, 4, 18, 25]

Begin pseudocode:

```
Merge(numbers, leftFirst, leftLast, rightLast) {
  // Create temporary array mergedNumbers
  // Initialize position variables

  while (leftPos <= leftLast && rightPos <= rightLast) {
    if (numbers[leftPos] <= numbers[rightPos]) {
      mergedNumbers[mergePos] = numbers[leftPos]
      leftPos += 1
    }
    else {
      mergedNumbers[mergePos] = numbers[rightPos]
      rightPos += 1
    }
    mergePos += 1
  }
```

```
   // If left partition not empty, add remaining elements
   while (leftPos <= leftLast) {
      mergedNumbers[mergePos] = numbers[leftPos]
      leftPos += 1
      mergePos += 1
   }

   // If right partition not empty, add remaining elements
   while (rightPos <= rightLast) {
      mergedNumbers[mergePos] = numbers[rightPos]
      rightPos += 1
      mergePos += 1
   }

   // Copy mergedNumbers back to numbers
   for (mergePos = 0; mergePos < mergedSize; mergePos += 1) {
      numbers[leftFirst + mergePos] = mergedNumbers[mergePos]
   }
}
```

End pseudocode.
Two arrays are shown, mergedNumbers and numbers. mergedNumbers array is the final merged numbers array: [3, 4, 8, 11, 18, 25]. numbers array is an unsorted array: [3, 8, 11, 4, 18, 25].
Comparison equations based on the code execution are shown:
3 <= 4 is true Add 3 to mergedNumbers
8 <= 4 is false Add 4 to mergedNumbers
8 <= 18 is true Add 8 to mergedNumbers
11 <= 18 is true Add 11 to mergedNumbers

Three text labels are shown at the bottom: "Left partition empty", "Right partition not empty", and "Copy remaining".

Step 1: A temporary array is created for merged numbers. Variables mergePos, leftPos, and rightPos are initialized to the first element of each of the corresponding array.
Two arrays shown, mergedNumbers and numbers. mergedNumbers has an allocated size of 6, with no contents shown. numbers also has an allocated size of 6, but contents are shown: [3, 8, 11, 4, 18, 25]. Code execution begins, initializing index variables as follows: mergePos = 0 and points to element 0 in mergedNumbers, leftPos = 0 and points to element 0 in numbers, and rightPos = 3 and points to element 3 in numbers.

Step 2: Elements in the left and right partitions are compared. The smaller is added to the temporary array and the relevant indices are updated.

The first iteration of the algorithm's first while loops executes. 3 <= 4 is true, so element 3 is copied from index 0, leftPos's value, in numbers, to index 0, mergePos's value, in mergedNumbers. Index variables mergePos and leftPos are both incremented to 1.

Step 3: Element comparisons continue until one of the partitions is empty.
Remaining iterations of the algorithm's first while loop execute, each copying one element and incrementing two indices.
8 <= 4 is false, so mergedNumbers[mergePos] is assigned with numbers[rightPos] (mergedNumbers[1] is assigned with numbers[3]). mergePos is incremented to 2 and rightPos is incremented to 4.
8 <= 18 is true, so mergedNumbers[mergePos] is assigned with numbers[leftPos] (mergedNumbers[2] is assigned with numbers[1]). mergePos is incremented to 3 and leftPos is incremented to 2.
11 <= 18 is true, so mergedNumbers[mergePos] is assigned with numbers[leftPos] (mergedNumbers[3] is assigned with numbers[2]). mergePos is incremented to 4 and leftPos is incremented to 3.

Step 4: The left partition is empty. Remaining elements from the right partition are copied to mergedNumbers. Copied elements are already in sorted order.
The algorithm's next while loop executes 0 iterations, since leftPos <= leftLast (3 <= 2) is false. The algorithm's last while loop copies remaining elements from right partition to mergedNumbers. mergedNumbers becomes [3, 4, 8, 11, 18, 25]. Index variable values are then: mergePos = 6, leftPos = 3, rightPos = 6.

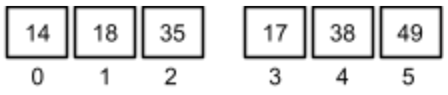
Step 5: Lastly, all elements in mergedNumbers are copied back to the original numbers array.
The final for loop copies elements from mergedNumbers back to numbers. So the numbers array becomes [3, 4, 8, 11, 18, 25].

Animation captions:

1. A temporary array is created for merged numbers. Variables mergePos, leftPos, and rightPos are initialized to the first element of each of the corresponding array.
2. Elements in the left and right partitions are compared. The smaller is added to the temporary array and the relevant indices are updated.
3. Element comparisons continue until one of the partitions is empty.
4. The left partition is empty. Remaining elements from the right partition are copied to mergedNumbers. Copied elements are already in sorted order.
5. Lastly, all elements in mergedNumbers are copied back to the original numbers array.

PARTICIPATION ACTIVITY 3.10.4: Tracing the merge operation.

Trace the merge operation by determining the next value added to mergedNumbers.



1) leftPos = 0, rightPos = 3

Check Show answer

2) leftPos = 1, rightPos = 3

Check Show answer

3) leftPos = 1, rightPos = 4

Check Show answer

4) leftPos = 2, rightPos = 4

Check Show answer

5) leftPos = 3, rightPos = 4

Check Show answer

6) leftPos = 3, rightPos = 5

Check Show answer

Figure 3.10.1: Merge sort algorithm.

```
Merge(numbers, i, j, k) {
   mergedSize = k - i + 1                // Size of merged partition
   mergePos = 0                          // Position to insert merged number
   leftPos = 0                           // Position of elements in left partition
   rightPos = 0                          // Position of elements in right
partition
   mergedNumbers = new int[mergedSize]   // Dynamically allocates temporary array
                                         // for merged numbers

   leftPos = i                           // Initialize left partition position
   rightPos = j + 1                      // Initialize right partition position

   // Add smallest element from left or right partition to merged numbers
   while (leftPos <= j && rightPos <= k) {
      if (numbers[leftPos] <= numbers[rightPos]) {
         mergedNumbers[mergePos] = numbers[leftPos]
         ++leftPos
      }
      else {
         mergedNumbers[mergePos] = numbers[rightPos]
         ++rightPos

      }
      ++mergePos
   }

   // If left partition is not empty, add remaining elements to merged numbers
   while (leftPos <= j) {
      mergedNumbers[mergePos] = numbers[leftPos]
      ++leftPos
      ++mergePos
   }

   // If right partition is not empty, add remaining elements to merged numbers
   while (rightPos <= k) {
      mergedNumbers[mergePos] = numbers[rightPos]
      ++rightPos
      ++mergePos
   }

   // Copy merge number back to numbers
   for (mergePos = 0; mergePos < mergedSize; ++mergePos) {
      numbers[i + mergePos] = mergedNumbers[mergePos]
   }
}

MergeSort(numbers, i, k) {
   j = 0

   if (i < k) {
      j = (i + k) / 2  // Find the midpoint in the partition

      // Recursively sort left and right partitions
      MergeSort(numbers, i, j)
      MergeSort(numbers, j + 1, k)

      // Merge left and right partition in sorted order
      Merge(numbers, i, j, k)
   }
}

main() {
   numbers = { 10, 2, 78, 4, 45, 32, 7, 11 }
   NUMBERS_SIZE = 8
   i = 0

   print("UNSORTED: ")
   for(i = 0; i < NUMBERS_SIZE; ++i) {
      print(numbers[i] + " ")
   }
   printLine()

   MergeSort(numbers, 0, NUMBERS_SIZE - 1)

   print("SORTED: ")
   for(i = 0; i < NUMBERS_SIZE; ++i) {
      print(numbers[i] + " ")
   }
   printLine()
}
```

```
UNSORTED: 10 2 78 4 45 32 7 11
SORTED: 2 4 7 10 11 32 45 78
```

Merge sort runtime

The merge sort algorithm's runtime is O(N log N). Merge sort divides the input in half until an array of 1 element is reached, which requires log N partitioning levels. At each level, the algorithm does about N comparisons selecting and copying elements from the left and right partitions, yielding N * log N comparisons.

Merge sort requires O(N) additional memory elements for the temporary array of merged elements. For the final merge operation, the temporary array has the same number of elements as the input. Some sorting algorithms sort the array elements in place and require no additional memory, but are more complex to write and understand.

PARTICIPATION ACTIVITY 3.10.5: Merge sort runtime and memory complexity.

1) How many recursive partitioning levels are required for an array of 8 elements?

Check Show answer

2) How many recursive partitioning levels are required for an array of 2048 elements?

Check Show answer

3) How many elements will the temporary merge array have for merging two partitions with 250 elements each?

Check Show answer

CHALLENGE ACTIVITY 3.10.1: Merge sort.

704586.1152092.qx3zqy7

Start

numbers:

84	92	44	64	82	43	12	94

What arguments should be provided to MergeSort() to sort the numbers array?

MergeSort(numbers, Ex: 1 ,)

1	2	3	4	5

Check Next

View solution (Instructors only)

3.11 C++: Merge sort

Merge sort algorithm

Merge sort is a sorting algorithm that divides an array into two halves, recursively sorts each half, and merges the sorted halves to produce a sorted array. Merge sort uses two functions: Merge() and MergeSort(). The Merge() function merges two sequential, sorted partitions within an array and has four parameters:

1. The array of numbers containing the two sorted partitions to merge
2. The start index of the first sorted partition
3. The end index of the first sorted partition
4. The end index of the second sorted partition

The MergeSort() function sorts a partition in an array and has three parameters:

1. The array containing the partition to sort
2. The start index of the partition to sort
3. The end index of the partition to sort

If the partition size is greater than 1, the MergeSort() function recursively sorts the left and right halves of the partition, then merges the sorted halves together. When the start index is 0 and the end index is the array length minus 1, MergeSort() sorts the entire array.

Figure 3.11.1: Merge sort algorithm.

```
#include <iostream>
#include <string>
using namespace std;

void Merge(int* numbers, int leftFirst, int leftLast, int rightLast) {
   int mergedSize = rightLast - leftFirst + 1;       // Size of merged partition
   int* mergedNumbers = new int[mergedSize]; // Dynamically allocates temporary
                                             // array for merged numbers
   int mergePos = 0;                         // Position to insert merged number
   int leftPos = leftFirst;                  // Initialize left partition position
   int rightPos = leftLast + 1;              // Initialize right partition
position

   // Add smallest element from left or right partition to merged numbers
   while (leftPos <= leftLast && rightPos <= rightLast) {
      if (numbers[leftPos] <= numbers[rightPos]) {
         mergedNumbers[mergePos] = numbers[leftPos];
         leftPos++;
      }
      else {
         mergedNumbers[mergePos] = numbers[rightPos];
         rightPos++;
      }
      mergePos++;
   }

   // If left partition is not empty, add remaining elements to merged numbers
   while (leftPos <= leftLast) {
      mergedNumbers[mergePos] = numbers[leftPos];
      leftPos++;
      mergePos++;
   }

   // If right partition is not empty, add remaining elements to merged numbers
   while (rightPos <= rightLast) {
      mergedNumbers[mergePos] = numbers[rightPos];
      rightPos++;
      mergePos++;
   }

   // Copy merged numbers back to numbers
   for (mergePos = 0; mergePos < mergedSize; mergePos++) {
      numbers[leftFirst + mergePos] = mergedNumbers[mergePos];
   }

   // Free temporary array
   delete[] mergedNumbers;
}

void MergeSort(int* numbers, int startIndex, int endIndex) {
   if (startIndex < endIndex) {
      // Find the midpoint in the partition
      int mid = (startIndex + endIndex) / 2;

      // Recursively sort left and right partitions
      MergeSort(numbers, startIndex, mid);
      MergeSort(numbers, mid + 1, endIndex);
      // Merge left and right partition in sorted order
      Merge(numbers, startIndex, mid, endIndex);
   }
}

string ArrayToString(int* array, int arraySize) {
   // Special case for empty array
   if (0 == arraySize) {
      return string("[]");
   }

   // Start the string with the opening '[' and element 0
   string result = "[" + to_string(array[0]);

   // For each remaining element, append comma, space, and element
   for (int i = 1; i < arraySize; i++) {
      result += ", ";
      result += to_string(array[i]);
   }

   // Add closing ']' and return result
   result += "]";
   return result;
}

int main() {
   // Create an array of numbers to sort
   int numbers[] = { 61, 76, 19, 4, 94, 32, 27, 83, 58 };
   int numbersSize = sizeof(numbers) / sizeof(numbers[0]);

   // Display the contents of the array
   cout << "UNSORTED: " << ArrayToString(numbers, numbersSize) << endl;

   // Call the MergeSort function
   MergeSort(numbers, 0, numbersSize - 1);

   // Display the sorted contents of the array
   cout << "SORTED:   " << ArrayToString(numbers, numbersSize) << endl;
}
```

```
UNSORTED: [61, 76, 19, 4, 94, 32, 27, 83, 58]
SORTED:   [4, 19, 27, 32, 58, 61, 76, 83, 94]
```

PARTICIPATION ACTIVITY 3.11.1: Merge sort in C++.

1) To sort an array with 10 elements, the arguments passed to MergeSort() will be: array, 0, 10.

○ True

- ○ False

2) If MergeSort() is called with startIndex = 0 and endIndex = 4, what is the size of the partition that will be sorted?

- ○ 3
- ○ 4
- ○ 5

3) If MergeSort() is called with startIndex = 0 and endIndex = 5, what is the value of mid after the following line executes?

```
mid = (startIndex + endIndex)
/ 2;
```

- ○ 2
- ○ 2.5

zyDE 3.11.1: Merge sort algorithm.

The program below uses the merge sort algorithm to sort various arrays. The code has been modified to also calculate how many comparisons occur.

MergeSort() is called three times to sort three distinct arrays. The first is unsorted, the second is sorted in ascending order, and the third is sorted in descending order. Try running the program with different arrays to see how the total number of comparisons changes (or doesn't change).

Current file: MergeSortDemo.cpp Load default template...

```
#include <iostream>
#include <string>
#include "SortTracker.h"
using namespace std;

void Merge(int* numbers, int leftFirst, int leftLast, int rightLast,
   SortTracker& tracker) {
   int mergedSize = rightLast - leftFirst + 1;
   int* mergedNumbers = new int[mergedSize];
   int mergePos = 0;
   int leftPos = leftFirst;
   int rightPos = leftLast + 1;

   // Add smallest element from left or right partition to merged numbers
   while (leftPos <= leftLast && rightPos <= rightLast) {
      if (tracker.IsFirstLTESecond(numbers, leftPos, rightPos)) {
         mergedNumbers[mergePos] = numbers[leftPos];
         leftPos++;
      }
      else {
         mergedNumbers[mergePos] = numbers[rightPos];
         rightPos++;
      }
      mergePos++;
   }

   // If left partition is not empty, add remaining elements to merged number
   while (leftPos <= leftLast) {
      mergedNumbers[mergePos] = numbers[leftPos];
      leftPos++;
      mergePos++;
   }

   // If right partition is not empty, add remaining elements to merged numbe
   while (rightPos <= rightLast) {
      mergedNumbers[mergePos] = numbers[rightPos];
      rightPos++;
      mergePos++;
   }

   // Copy merged numbers back to numbers
   for (mergePos = 0; mergePos < mergedSize; mergePos++) {
      numbers[leftFirst + mergePos] = mergedNumbers[mergePos];
   }

   // Free temporary array
   delete[] mergedNumbers;
}

void MergeSort(int* numbers, int startIndex, int endIndex,
   SortTracker& tracker) {
   if (startIndex < endIndex) {
      // Find the midpoint in the partition
      int mid = (startIndex + endIndex) / 2;

      // Recursively sort left and right partitions
      MergeSort(numbers, startIndex, mid, tracker);
      MergeSort(numbers, mid + 1, endIndex, tracker);

      // Merge left and right partition in sorted order
      Merge(numbers, startIndex, mid, endIndex, tracker);
   }
}

void MergeSortDemo(int* numbersArray, int arrayLength) {
   SortTracker tracker;

   // Display the contents of the array
   tracker.PrintArray(numbersArray, arrayLength, cout, ", ",
      "Before sorting:    [", "]\n");

   // Sort the numbers array using the merge sort algorithm
   MergeSort(numbersArray, 0, arrayLength - 1, tracker);

   // Display the sorted contents of the array
   tracker.PrintArray(numbersArray, arrayLength, cout, ", ",
      "After  sorting:    [", "]\n");

   // Display stats
   cout << "Total comparisons: " << tracker.GetComparisonCount() << endl;
}

int main() {
   // Create arrays:
   // - array1 is unsorted
   // - array2 is sorted in ascending order
   // - array3 is sorted in descending order
   int array1[] = { 33, 18, 78, 64, 45, 32, 70, 11, 27 };
   int array1Length = sizeof(array1) / sizeof(array1[0]);
   int array2[] = { 10, 20, 30, 40, 50, 60, 70, 80, 90 };
```

```
    int array2Length = sizeof(array2) / sizeof(array2[0]);
    int array3[] = { 99, 88, 77, 66, 55, 44, 33, 22, 11 };
    int array3Length = sizeof(array3) / sizeof(array3[0]);

    cout << "Demo 1 - Unsorted array:" << endl;
    MergeSortDemo(array1, array1Length);
    cout << endl << "Demo 2 - Array sorted in ascending order:" << endl;
    MergeSortDemo(array2, array2Length);
    cout << endl << "Demo 3 - Array sorted in descending order:" << endl;
    MergeSortDemo(array3, array3Length);

    return 0;
}
```

Run

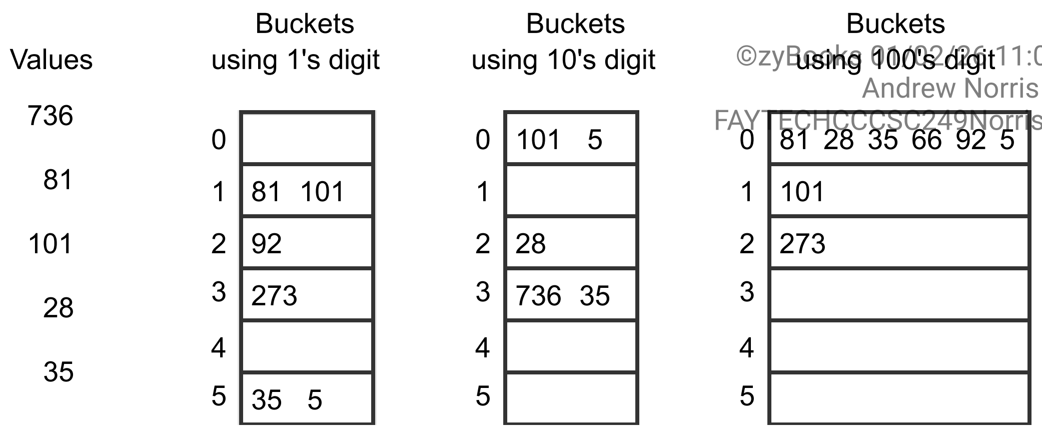
3.12 Radix sort

Buckets

Radix sort is a sorting algorithm designed specifically for integers. The algorithm makes use of a concept called buckets and is a type of bucket sort.

Any array of integer values can be subdivided into buckets by using the integer values' digits. A ***bucket*** is a collection of integer values that all share a particular digit value. Ex: Values 57, 97, 77, and 17 all have a 7 as the 1's digit, and would all be placed into bucket 7 when subdividing by the 1's digit.

PARTICIPATION ACTIVITY 3.12.1: A particular single digit in an integer can determine the integer's bucket.



Animation content:

Static figure:
Tables with four columns are shown. The first column is labeled, “Values” and contains number values: 736, 81, 101, 28, 35, 66, 92, 5, 273, consecutively. The number values in the 1’s place are highlighted yellow. The number values in the 10’s place are highlighted blue. The number values in the 100’s place are highlighted green.
The second column is labeled “Buckets using 1's digit” and contains a table with 10 buckets labeled from 0 – 9. The table has a yellow background. buckets 0, 4 7 and 9 are empty. Bucket 1 contains number values, 81 and 101. Bucket 2 contains the number value 92. Bucket 3 contains the number value 273. Bucket 5 contains number values, 35 and 5. Bucket 6 contains number values, 736 and 66. Bucket 8 contains the number value, 28.
The third column is labeled “Buckets using 10's digit” and contains a table with 10 buckets labeled from 0 – 9. The table has a blue background. Buckets 1, 4, and 5 are empty. Bucket 0 contains number values, 101 and 5. Bucket 2 contains the number value 28. Bucket 3 contains number values 736 and 35. Bucket 6 contains the number value, 66. Bucket 7 contains the number value, 273. Bucket 8 contains the number value, 81. Bucket 9 contains the number value, 92.
The fourth column is labeled “Buckets using 100's digit” and contains a table with 10 buckets labeled from 0 – 9. The table has a green background. Buckets 3, 4, 5, 6, 8, and 9 are empty. Bucket 1 contains number values, 81, 28, 35, 66, 92, and 5. Bucket 1 contains the number value 101. Bucket 2 contains the number value 273. Bucket 7 contains the number value, 736.

Step 1: Using only the 1's digit, each integer can be put into a bucket. 736 is put into bucket 6, 81 into bucket 1, 101 into bucket 1, and so on.
Table with four columns is shown. Number values appear stacked vertically in the leftmost column: 736, 81, 101, 28, 35, 66, 92, 5, 273. Next column is titled "Buckets using 1's digit", and has a table within with buckets 0 - 9. Next column is titled "Buckets using 10's digit", and has a table within with buckets 0 - 9. Last column is titled "Buckets using 100's digit", and has a table within with buckets 0 - 9.
A yellow highlight appears, highlighting the ones digit of each number: 6, 1, 1, 8, 5, 6, 2, 5, 3. A copy of each number moves to a bucket based on the 1's digit: 736 to bucket 6, 81 to bucket 1, 101 to bucket 1, 28 to bucket 8, 35 to bucket 5, 66 to bucket 6, 92 to bucket 2, 5 to bucket 5, and 273 to bucket 3.

Step 2: Using only the 10's digit, each integer can be put into a bucket. 736 is put into bucket 3, 81 into bucket 8, and so on. 5 is like 05 so is put into bucket 0.
A gray '0' is prepended to 5, making "05" in the leftmost column. Blue highlight then appears for 10's digits in leftmost column: 3, 8, 0, 2, 3, 6, 9, 0, 7. A copy of each number moves to a bucket based on the 10's digit: 736 to bucket 3, 81 to bucket 8, 101 to bucket 0, 28 to bucket 2, 35 to bucket 3, 66 to bucket 6, 92 to bucket 9, 5 to bucket 0, and 273 to bucket 7.

Step 3: Using only the 100's digit, each integer can be put into a bucket. 736 is put into bucket 7, 81 is like 081 so is put into bucket 0, and so on.
Additional gray '0' digits are prepended to make each number in the leftmost column three digits: 736, 081, 101, 028, 035, 066, 092, 005, 273. Blue highlight then appears for 100's digits in leftmost column: 7, 0, 1, 0, 0, 0, 0, 0, 2. A copy of each number moves to a bucket based on the 100's digit: 736 to bucket 7, 81 to bucket 0, 101 to bucket 1, 28 to bucket 0, 35 to bucket 0, 66 to bucket 0, 92 to bucket 0, 5 to bucket 0, and 273 to bucket 2.

Animation captions:

1. Using only the 1's digit, each integer can be put into a bucket. 736 is put into bucket 6, 81 into bucket 1, 101 into bucket 1, and so on.
2. Using only the 10's digit, each integer can be put into a bucket. 736 is put into bucket 3, 81 into bucket 8, and so on. 5 is like 05 so is put into bucket 0.
3. Using only the 100's digit, each integer can be put into a bucket. 736 is put into bucket 7, 81 is like 081 so is put into bucket 0, and so on.

PARTICIPATION ACTIVITY 3.12.2: Using the 1's digit, place each integer in the correct bucket.

How to use this tool

Mouse: Drag/drop
Keyboard: Grab/release **Spacebar** (or **Enter**). Move ↑ ↓ ← →. Cancel **Esc**

7 **50** **74** **49**

Bucket 0

Bucket 4

Bucket 7

Bucket 9

Reset

PARTICIPATION ACTIVITY 3.12.3: Using the 10's digit, place each integer in the correct bucket.

How to use this tool

Mouse: Drag/drop
Keyboard: Grab/release `Spacebar` (or `Enter`). Move `↑` `↓` `←` `→`. Cancel `Esc`

86 **7** **74** **50**

Bucket 0

Bucket 5

Bucket 7

Bucket 8

Reset

PARTICIPATION ACTIVITY 3.12.4: Bucket concepts.

1) Integers will be placed into buckets based on the 1's digit. More buckets are needed for an array with one thousand integers than for an array with one hundred integers.

- True
- False

2) Consider integers X and Y, such that X < Y. X will always be in a lower bucket than Y.

- True
- False

3) All integers from an array could be placed into the same bucket, even if the array has no duplicates.

- True
- False

PARTICIPATION ACTIVITY 3.12.5: Assigning integers to buckets.

For each question, consider the array of integers: [51, 47, 96, 52, 27].

1) When placing integers using the 1's digit, how many integers will be in bucket 7?

- 0
- 1
- 2

2) When placing integers using the 1's digit, how many integers will be in bucket 5?

- 0
- 1
- 2

3) When placing integers using the 10's digit, how many will be in bucket 9?

- 0
- 1
- 2

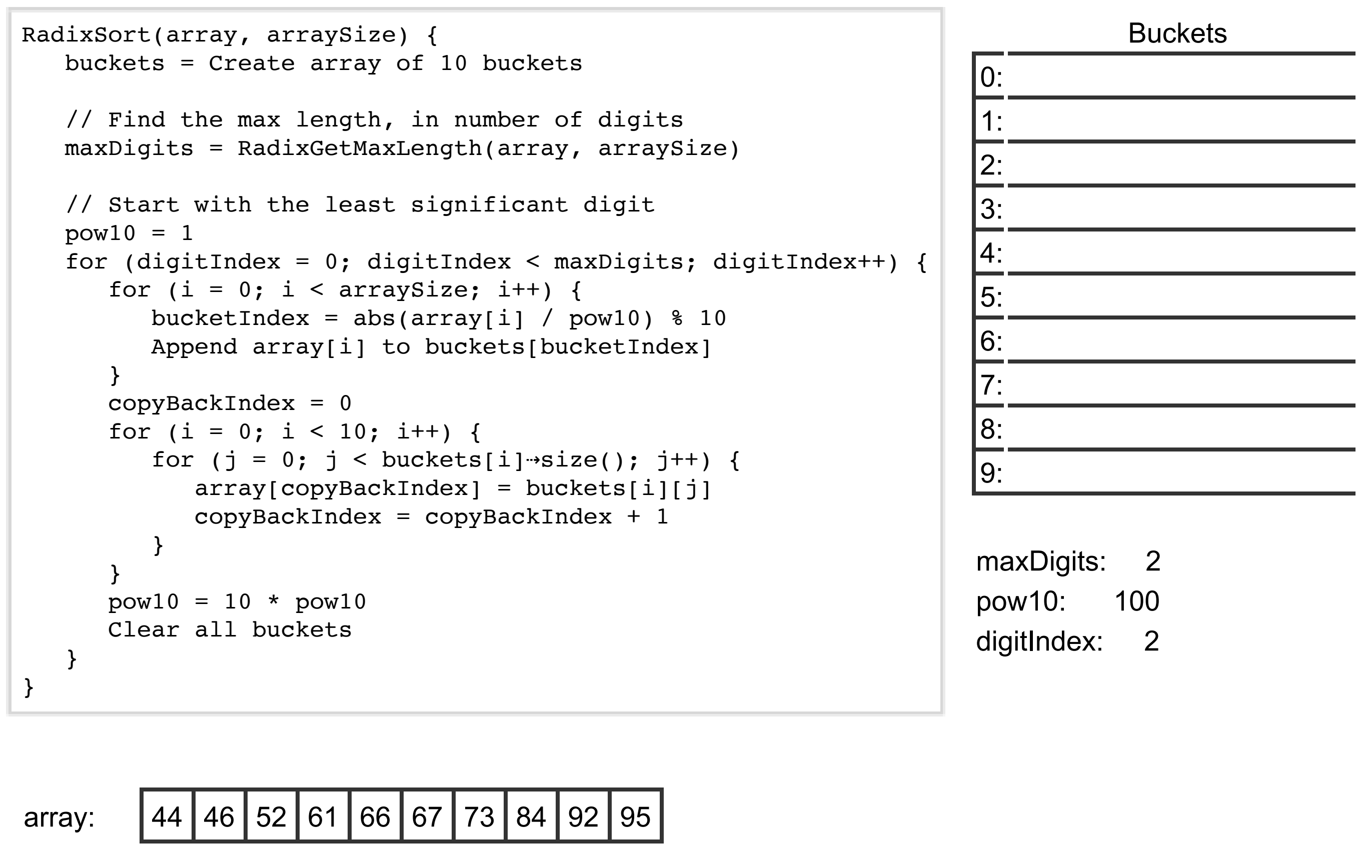
4) All integers would be in bucket 0 if using the 100's digit.

- True
- False

Radix sort algorithm

Radix sort is a sorting algorithm specifically for an array of *integers*: The algorithm processes one digit at a time starting with the least significant digit and ending with the most significant. Two steps are needed for each digit. First, all array elements are placed into buckets based on the current digit's value. Then, the array is rebuilt by removing all elements from buckets, in order from lowest bucket to highest.

PARTICIPATION ACTIVITY 3.12.6: Radix sort algorithm (for non-negative integers).



```
RadixSort(array, arraySize) {
   buckets = Create array of 10 buckets

   // Find the max length, in number of digits
   maxDigits = RadixGetMaxLength(array, arraySize)

   // Start with the least significant digit
   pow10 = 1
   for (digitIndex = 0; digitIndex < maxDigits; digitIndex++) {
      for (i = 0; i < arraySize; i++) {
         bucketIndex = abs(array[i] / pow10) % 10
         Append array[i] to buckets[bucketIndex]
      }
      copyBackIndex = 0
      for (i = 0; i < 10; i++) {
         for (j = 0; j < buckets[i]⇢size(); j++) {
            array[copyBackIndex] = buckets[i][j]
            copyBackIndex = copyBackIndex + 1
         }
      }
      pow10 = 10 * pow10
      Clear all buckets
   }
}
```

Animation content:

Static figure:
RadixSort() function implementation on the left. 10 empty buckeys on the right.
Begin pseudocode:
RadixSort(array, arraySize) {

```
  buckets = Create array of 10 buckets

  // Find the max length, in number of digits
  maxDigits = RadixGetMaxLength(array, arraySize)

  // Start with the least significant digit
  pow10 = 1
  for (digitIndex = 0; digitIndex < maxDigits; digitIndex++) {
    for (i = 0; i < arraySize; i++) {
      bucketIndex = abs(array[i] / pow10) % 10
      Append array[i] to buckets[bucketIndex]
    }
    copyBackIndex = 0
    for (i = 0; i < 10; i++) {
      for (j = 0; j < buckets[i]⇢size(); j++) {
        array[copyBackIndex] = buckets[i][j]
        copyBackIndex = copyBackIndex + 1
      }
    }
    pow10 = 10 * pow10
    Clear all buckets
  }
}
```

End pseudocode.

Step 1: Array is shown: [92, 66, 95, 61, 44, 84, 73, 46, 52, 67]. Execution of RadixSort() function begins. Variable maxDigits is assigned 2, the return value from calling RadixGetMaxLength(array, arraySize). Variable pow10 is assigned with 1. The outer for loop begins, assigning digitIndex with 0. The first nested for loop puts array elements in buckets: 92 in bucket 2, 66 in bucket 6, 95 in bucket 5, 61 in bucket 1, 44 in bucket 4, 84 in bucket 4, 73 in bucket 3, 46 in bucket 6, 52 in bucket 2, and 67 in bucket 7.

Step 2: The second nested for loop executes, copying elements back to the array: [61, 92, 52, 73, 44, 84, 95, 66, 46, 67].

Step 3: Variable pow10 is assigned with 10 and the next iteration of the outer for loop begins. The first nested for loop puts elements in buckets: 61 in bucket 6, 92 in bucket 9, 52 in bucket 5, 73 in bucket 7, 44 in bucket 4, 84 in bucket 8, 95 in bucket 9, 66 in bucket 6, 46 in bucket 4, and 67 in bucket 6.

Step 4: The second nested loop copies elements from buckets back to the array, completing the sort: [44, 46, 52, 61, 66, 67, 73, 84, 92, 95].

Animation captions:

1. Radix sort begins by allocating 10 buckets and putting each number in a bucket based on the 1's digit.
2. Numbers are taken out of buckets, in order from lowest bucket to highest, rebuilding the array.
3. The process is repeated for the 10's digit. First, the array numbers are placed into buckets based on the 10's digit.
4. The items are copied from buckets back into the array. Since all digits have been processed, the result is a sorted array.

Figure 3.12.1: RadixGetMaxLength() and RadixGetLength() functions.

```
// Returns the maximum length, in number of digits, out of all elements in the
array
RadixGetMaxLength(array, arraySize) {
   maxDigits = 0
   for (i = 0; i < arraySize; i++) {
      digitCount = RadixGetLength(array[i])
      if (digitCount > maxDigits)
         maxDigits = digitCount
   }
   return maxDigits
}

// Returns the length, in number of digits, of value
RadixGetLength(value) {
   if (value == 0)
      return 1

   digits = 0
   while (value != 0) {
      digits = digits + 1
      value = value / 10
   }
   return digits
}
```

PARTICIPATION ACTIVITY 3.12.7: Radix sort algorithm.

1) RadixGetLength(17) returns _____.

Check **Show answer**

2) RadixGetMaxLength() returns _____ when the array is [17, 1024, 101].

Check Show answer

3) When sorting the array [57, 5, 501] with RadixSort(), what is the largest number of integers that will be in bucket 5 at any given moment?

Check Show answer

PARTICIPATION ACTIVITY 3.12.8: Radix sort algorithm analysis.

1) When sorting an array of N 3-digit integers, RadixSort()'s worst-case time complexity is O(N).

- True
- False

2) When sorting an array with N elements, the maximum number of elements that RadixSort() may put in a bucket is N.

- True
- False

3) RadixSort() has a space complexity of O(1).

- True
- False

4) The RadixSort() function shown above also works on an array of floating-point values.

- True

○ False

Sorting signed integers

The above radix sort algorithm correctly sorts arrays of non-negative integers. But if the array contains negative integers, the above algorithm would sort by absolute value, so the integers are not correctly sorted. A small extension to the algorithm correctly handles negative integers.

The extended algorithm has more code after the primary for loop ends. First, two buckets are allocated, one for negative integers and the other for non-negative integers. Next, the array is iterated through in order. Each negative integer is placed in the negative bucket and each non-negative integer in the non-negative bucket. Then the negative bucket's order is reversed and the non-negative bucket is concatenated, yielding a sorted array. The completed radix sort algorithm follows.

Figure 3.12.2: RadixSort algorithm (for negative and non-negative integers).

```
RadixSort(array, arraySize) {
   buckets = create array of 10 buckets

   // Find the max length, in number of digits
   maxDigits = RadixGetMaxLength(array, arraySize)

   pow10 = 1
   for (digitIndex = 0; digitIndex < maxDigits; digitIndex++) {
      for (i = 0; i < arraySize; i++) {
         bucketIndex = abs(array[i] / pow10) % 10
         Append array[i] to buckets[bucketIndex]
      }
      arrayIndex = 0
      for (i = 0; i < 10; i++) {
         for (j = 0; j < buckets[i]⇢size(); j++) {
            array[arrayIndex] = buckets[i][j]
            arrayIndex = arrayIndex + 1
         }
      }
      pow10 = pow10 * 10
      Clear all buckets
   }

   negatives = all negative values from array
   nonNegatives = all non-negative values from array
   Reverse order of negatives
   Concatenate negatives and nonNegatives into array
}
```

PARTICIPATION ACTIVITY 3.12.9: Sorting signed integers.

For each question, assume radix sort has sorted integers by absolute value to produce the array [-12, 23, -42, 73, -78], and is about to build the negative and non-negative buckets to complete the sort.

1) What integers will be placed into the negative bucket? Type answer as: 15, 42, 98

[ ]

Check **Show answer**

2) What integers will be placed into the non-negative bucket? Type answer as: 15, 42, 98

[ ]

Check **Show answer**

3) After reversal, what integers are in the negative bucket? Type answer as: 15, 42, 98

[ ]

Check **Show answer**

4) What is the final array after RadixSort() concatenates the two buckets? Type answer as: 15, 42, 98

[ ]

Check **Show answer**

Radix sort with different bases

This section presents radix sort with base 10, but other bases can be used as well. Ex: Using base 2 is another common approach, where only 2 buckets are required instead of 10.

CHALLENGE ACTIVITY | 3.12.1: Radix sort.

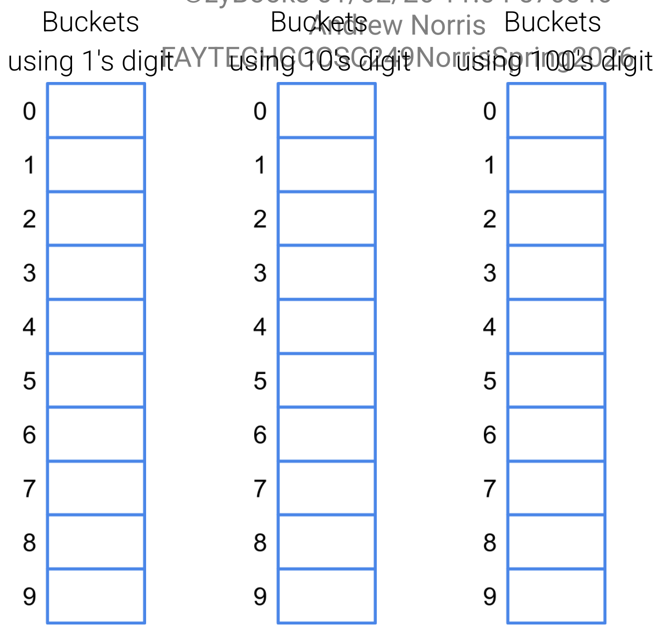
704586.1152092.qx3zqy7

Start

Using only the 1's digit,
what is the correct bucket for 959? Ex: 5
what is the correct bucket for 87?

Using only the 10's digit,
what is the correct bucket for 959?
what is the correct bucket for 87?

Using only the 100's digit,
what is the correct bucket for 959?
what is the correct bucket for 87?



1 2 3 4

Check Next

View solution (Instructors only)

3.13 C++: Radix sort

Radix sort algorithm

Radix sort can be implemented in C++ using a vector of ten `vector<int>*` objects for the buckets. The main loop iterates through each relevant digit. The first of two nested loops copies each array element to the appropriate bucket. The second nested loop concatenates buckets in order and copies elements back to the array.

When the main loop completes, the array is sorted by absolute value. Two additional vectors are then built from the array, one with all negative elements, the other with all non-negative elements. Then, the reversed negative vector is concatenated with the non-negative vector into the array to produce the sorted result.

Figure 3.13.1: Radix sort algorithm.

```
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int RadixGetLength(int value) {
   if (value == 0) {
      return 1;
   }

   int digits = 0;
   while (value != 0) {
      digits++;
      value /= 10;
   }
   return digits;
}

// Returns the maximum length, in number of digits, out of all array elements
int RadixGetMaxLength(int* numbers, int numbersSize) {
   int maxDigits = 0;
   for (int i = 0; i < numbersSize; i++) {
      int digitCount = RadixGetLength(numbers[i]);
      if (digitCount > maxDigits) {
         maxDigits = digitCount;
      }
   }
   return maxDigits;
}

void RadixSort(int* numbers, int numbersSize) {
   vector<vector<int>*> buckets;
   for (int i = 0; i < 10; i++) {
      vector<int>* bucket = new vector<int>();
      buckets.push_back(bucket);
   }

   int copyBackIndex = 0;

   // Find the max length, in number of digits
   int maxDigits = RadixGetMaxLength(numbers, numbersSize);

   int pow10 = 1;
   for (int digitIndex = 0; digitIndex < maxDigits; digitIndex++) {
      // Put numbers into buckets
      for (int i = 0; i < numbersSize; i++) {
         int num = numbers[i];
         int bucketIndex = (abs(num) / pow10) % 10;
         buckets[bucketIndex]->push_back(num);
      }

      // Copy buckets back into numbers array
      copyBackIndex = 0;
      for (int i = 0; i < 10; i++) {
         vector<int>& bucket = *buckets[i];
         for (int j = 0; j < bucket.size(); j++) {
            numbers[copyBackIndex] = bucket[j];
            copyBackIndex++;
         }
         bucket.clear();
      }

      pow10 *= 10;
   }

   vector<int> negatives;
   vector<int> nonNegatives;
   for (int i = 0; i < numbersSize; i++) {
      int num = numbers[i];
      if (num < 0) {
         negatives.push_back(num);
      }
      else {
         nonNegatives.push_back(num);
      }
   }

   // Copy sorted content to array - negatives in reverse, then non-negatives
   copyBackIndex = 0;
   for (int i = negatives.size() - 1; i >= 0; i--) {
      numbers[copyBackIndex] = negatives[i];
      copyBackIndex++;
   }
   for (int i = 0; i < nonNegatives.size(); i++) {
      numbers[copyBackIndex] = nonNegatives[i];
      copyBackIndex++;
   }

   // Free each dynamically allocated bucket
   for (int i = 0; i < 10; i++) {
      delete buckets[i];
   }
}

string ArrayToString(int* array, int arraySize) {
   // Special case for empty array
   if (0 == arraySize) {
      return string("[]");
   }

   // Start the string with the opening '[' and element 0
   string result = "[" + to_string(array[0]);

   // For each remaining element, append comma, space, and element
   for (int i = 1; i < arraySize; i++) {
      result += ", ";
      result += to_string(array[i]);
   }

   // Add closing ']' and return result
   result += "]";
   return result;
}
```

```
int main() {
   // Create an array of numbers to sort
   int numbers[] = { -9, 47, 81, 101, -5, 38, -99, 96, 51, -999, -11, 64 };
   int numbersSize = sizeof(numbers) / sizeof(numbers[0]);

   // Display the contents of the array
   cout << "UNSORTED: " << ArrayToString(numbers, numbersSize) << endl;

   // Call the RadixSort function
   RadixSort(numbers, numbersSize);

   // Display the sorted contents of the array
   cout << "SORTED:   " << ArrayToString(numbers, numbersSize) << endl;
}
```

```
UNSORTED: [-9, 47, 81, 101, -5, 38, -99, 96, 51, -999, -11, 64]
SORTED:   [-999, -99, -11, -9, -5, 38, 47, 51, 64, 81, 96, 101]
```

PARTICIPATION ACTIVITY 3.13.1: Radix sort algorithm implementation.

1) If the array { 56, 19, 2, 101, 70 } is passed to RadixSort(), after the line `int maxDigits = RadixGetMaxLength(numbers)`, what is the value of maxDigits?

- 1
- 3
- 5

2) RadixSort() reallocates the numbers array during each iteration of the digit-index loop.

- True
- False

3) If the array { 0, 10, -20, 30, -40 } is passed to RadixSort(), then before the function's last two for loops execute, the contents of `negatives` are _____.

- 0, -20, -40
- -20, -40

○ -40, -20

zyDE 3.13.1: Radix sort in C++.

RadixSortDemo.cpp

Load default template...

```
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int RadixGetLength(int value) {
   if (value == 0) {
      return 1;
   }

   int digits = 0;
   while (value != 0) {
      digits++;
      value /= 10;
   }
   return digits;
}

// Returns the maximum length, in number of digits, out of all array elements
int RadixGetMaxLength(int* numbers, int numbersSize) {
   int maxDigits = 0;
   for (int i = 0; i < numbersSize; i++) {
      int digitCount = RadixGetLength(numbers[i]);
      if (digitCount > maxDigits) {
         maxDigits = digitCount;
      }
   }
   return maxDigits;
}

void RadixSort(int* numbers, int numbersSize) {
   vector<vector<int>*> buckets;
   for (int i = 0; i < 10; i++) {
      vector<int>* bucket = new vector<int>();
      buckets.push_back(bucket);
   }

   int copyBackIndex = 0;

   // Find the max length, in number of digits
   int maxDigits = RadixGetMaxLength(numbers, numbersSize);

   int pow10 = 1;
   for (int digitIndex = 0; digitIndex < maxDigits; digitIndex++) {
      // Put numbers into buckets
      for (int i = 0; i < numbersSize; i++) {
         int num = numbers[i];
         int bucketIndex = (abs(num) / pow10) % 10;
         buckets[bucketIndex]->push_back(num);
      }

      // Copy buckets back into numbers array
      copyBackIndex = 0;
      for (int i = 0; i < 10; i++) {
         vector<int>& bucket = *buckets[i];
         for (int j = 0; j < bucket.size(); j++) {
            numbers[copyBackIndex] = bucket[j];
            copyBackIndex++;
         }
         bucket.clear();
      }

      pow10 *= 10;
   }

   vector<int> negatives;
   vector<int> nonNegatives;
   for (int i = 0; i < numbersSize; i++) {
      int num = numbers[i];
      if (num < 0) {
         negatives.push_back(num);
      }
      else {
         nonNegatives.push_back(num);
      }
   }

   // Copy sorted content to array - negatives in reverse, then non-negatives
   copyBackIndex = 0;
   for (int i = negatives.size() - 1; i >= 0; i--) {
      numbers[copyBackIndex] = negatives[i];
      copyBackIndex++;
   }
   for (int i = 0; i < nonNegatives.size(); i++) {
      numbers[copyBackIndex] = nonNegatives[i];
      copyBackIndex++;
   }

   // Free each dynamically allocated bucket
   for (int i = 0; i < 10; i++) {
      delete buckets[i];
   }
}

string ArrayToString(int* array, int arraySize) {
   // Special case for empty array
   if (0 == arraySize) {
      return string("[]");
   }

   // Start the string with the opening '[' and element 0
   string result = "[" + to_string(array[0]);

   // For each remaining element, append comma, space, and element
   for (int i = 1; i < arraySize; i++) {
      result += ", ";
      result += to_string(array[i]);
   }

   // Add closing ']' and return result
   result += "]";
   return result;
}

int main() {
   // Create an array of numbers to sort
   int numbers[] = { -9, 47, 81, 101, -5, 38, -99, 96, 51, -999, -11, 64 };
   int numbersSize = sizeof(numbers) / sizeof(numbers[0]);

   // Display the contents of the array
   cout << "UNSORTED: " << ArrayToString(numbers, numbersSize) << endl;

   // Call the RadixSort function
   RadixSort(numbers, numbersSize);

   // Display the sorted contents of the array
   cout << "SORTED:   " << ArrayToString(numbers, numbersSize) << endl;
}
```

Run

3.14 Overview of fast sorting algorithms

Fast sorting algorithm

A ***fast sorting algorithm*** is a sorting algorithm that has an average runtime complexity of $O(N \log N)$ or better. The table below shows average runtime complexities for several sorting algorithms.

Table 3.14.1: Sorting algorithms' average runtime complexity.

Sorting algorithm	Average case runtime complexity	Fast?
Selection sort	$O(N^2)$	No
Insertion sort	$O(N^2)$	No
Shell sort	$O(N^{1.5})$	No
Quicksort	$O(N \log N)$	Yes
Merge sort	$O(N \log N)$	Yes
Heap sort	$O(N \log N)$	Yes
Radix sort	$O(N)$	Yes

PARTICIPATION ACTIVITY 3.14.1: Fast sorting algorithms.

1) Insertion sort is a fast sorting algorithm.
- True
- False

2) Merge sort is a fast sorting algorithm.
- True
- False

3) Radix sort is a fast sorting algorithm.
- True
- False

Comparison sorting

An ***element comparison sorting algorithm*** is a sorting algorithm that operates on an array of elements that can be compared to each other. Ex: An array of strings can be sorted with a comparison sorting algorithm, since two strings can be compared to determine if the one string is less than, equal to, or greater than another string. Selection sort, insertion sort, shell sort, quicksort, merge sort, and heap sort are all comparison sorting algorithms. Radix sort, in contrast, subdivides each array element into integer digits and is not a comparison sorting algorithm.

Table 3.14.2: Identifying comparison sorting algorithms.

Sorting algorithm	Comparison?
Selection sort	Yes
Insertion sort	Yes
Shell sort	Yes
Quicksort	Yes
Merge sort	Yes
Heap sort	Yes
Radix sort	No

PARTICIPATION ACTIVITY 3.14.2: Comparison sorting algorithms.

1) Selection sort can be used to sort an array of strings.

- True
- False

2) The fastest average runtime complexity of a comparison sorting algorithm is $O(N \log N)$.

- True
- False

Best and worst case runtime complexity

A fast sorting algorithm's best or worst case runtime complexity may differ from the average runtime complexity. Ex: The best and average case runtime complexity for quicksort is $O(N \log N)$, but the worst case is $O(N^2)$.

Table 3.14.3: Fast sorting algorithm's best, average, and worst case runtime complexity.

Sorting algorithm	Best case runtime complexity	Average case runtime complexity	Worst case runtime complexity
Quicksort	$O(N \log N)$	$O(N \log N)$	$O(N^2)$
Merge sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
Heap sort	$O(N)$	$O(N \log N)$	$O(N \log N)$
Radix sort	$O(N)$	$O(N)$	$O(N)$

PARTICIPATION ACTIVITY 3.14.3: Runtime complexity.

1) A fast sorting algorithm's worst case runtime complexity must be $O(N \log N)$ or better.

- True
- False

2) Which fast sorting algorithm's worst case runtime complexity is worse than $O(N \log N)$?

- Quicksort
- Heap sort
- Radix sort

3.15 C++: Sorting with different operators

Sorting a vector with std::sort()

A vector can be sorted using the std::sort() function, passing the vector's begin and end iterators as the first and second arguments, respectively. The type of item in the vector must support comparison using the < operator.

Figure 3.15.1: Using std::sort() to sort vectors of ints, doubles, and strings.

```
#include <algorithm>
#include <iostream>
#include <sstream>
#include <vector>
using namespace std;

template<typename T>
string VectorToString(vector<T>& items) {
   // Special case for empty vector
   if (items.size() < 1) {
      return string("[]");
   }

   // Use a string stream to build the string
   ostringstream stringStream;
   stringStream << "[" << items[0];
   for (int i = 1; i < items.size(); i++) {
      stringStream << ", " << items[i];
   }
   stringStream << "]";
   return stringStream.str();
}

int main() {
   // Create and sort a vector of ints
   vector<int> integers = { 3, 7, 2, 8, 12, 4, 9, 5 };
   cout << "Unsorted vector: " << VectorToString(integers) << endl;
   sort(integers.begin(), integers.end());
   cout << "Sorted vector:   " << VectorToString(integers) << endl << endl;

   // Create and sort a vector of doubles
   vector<double> doubles = { 71.2, 63.4, 99.9, 33.0, 84.75 };
   cout << "Unsorted vector: " << VectorToString(doubles) << endl;
   sort(doubles.begin(), doubles.end());
   cout << "Sorted vector:   " << VectorToString(doubles) << endl << endl;

   // Create and sort a vector of strings
   vector<string> fruits = { string("grape"), string("banana"), string("apple"),
      string("strawberry"), string("blueberry") };
   cout << "Unsorted vector: " << VectorToString(fruits) << endl;
   sort(fruits.begin(), fruits.end());
   cout << "Sorted vector:   " << VectorToString(fruits) << endl;

   return 0;
}
```

```
Unsorted vector: [3, 7, 2, 8, 12, 4, 9, 5]
Sorted vector:   [2, 3, 4, 5, 7, 8, 9, 12]

Unsorted vector: [71.2, 63.4, 99.9, 33, 84.75]
Sorted vector:   [33, 63.4, 71.2, 84.75, 99.9]

Unsorted vector: [grape, banana, apple, strawberry, blueberry]
Sorted vector:   [apple, banana, blueberry, grape, strawberry]
```

PARTICIPATION ACTIVITY | 3.15.1: Sorting a vector with std::sort().

1) std::sort() has one parameter for the vector to sort.

- True
- False

2) std::sort() sorts in ascending order.

- True
- False

3) std::sort() can only sort vectors of ints, doubles, or strings.

- True
- False

Sorting with a user-defined type

A vector with items of a user-defined type (struct or class) can be sorted with std::sort(), provided the user-defined type overloads the < operator.

Ex: Code in the figure below declares an InventoryItem class with a string name, a double cost, and an integer stock count. The class overloads the < operator to compare items by name. So a vector of InventoryItem objects can be sorted by name using std::sort().

Figure 3.15.2: InventoryItem.h.

```
#ifndef INVENTORYITEM_H
#define INVENTORYITEM_H

#include <iomanip>
#include <sstream>
#include <string>

class InventoryItem {
public:
   std::string name;
   double price;
   int numberInStock;

   InventoryItem(const std::string& itemName, double itemPrice, int
numberOfItemsInStock) {
      name = itemName;
      price = itemPrice;
      numberInStock = numberOfItemsInStock;
   }

   bool operator<(const InventoryItem& rhs) const {
      return name < rhs.name;
   }

   std::string ToString() const {
      std::ostringstream stream;
      stream << std::fixed << std::setprecision(2);
      stream << name << " - $" << price << " (" << numberInStock;
      stream << " in stock)";
      return stream.str();
   }
};

#endif
```

Figure 3.15.3: Using std::sort() to sort a vector of InventoryItem objects by name.

```
#include <algorithm>
#include <iostream>
#include <vector>
#include "InventoryItem.h"
using namespace std;

int main() {
   // Create a vector of InventoryItem objects
   vector<InventoryItem> items;
   items.push_back(InventoryItem("Toothpaste", 5.00, 250));
   items.push_back(InventoryItem("Toothbrush", 7.00, 500));
   items.push_back(InventoryItem("Gum", 1.50, 100));
   items.push_back(InventoryItem("Mints", 2.50, 50));
   items.push_back(InventoryItem("Kettle chips", 3.00, 40));
   cout << "Unsorted vector:" << endl;
   for (InventoryItem& item : items) {
      cout << item.ToString() << endl;
   }
   cout << endl;

   // Sort the vector, then show items
   sort(items.begin(), items.end());
   cout << "Vector sorted by item name:" << endl;
   for (InventoryItem& item : items) {
      cout << item.ToString() << endl;
   }

   return 0;
}
```

```
Unsorted vector:
Toothpaste - $5.00 (250 in stock)
Toothbrush - $7.00 (500 in stock)
Gum - $1.50 (100 in stock)
Mints - $2.50 (50 in stock)
Kettle chips - $3.00 (40 in stock)

Vector sorted by item name:
Gum - $1.50 (100 in stock)
Kettle chips - $3.00 (40 in stock)
Mints - $2.50 (50 in stock)
Toothbrush - $7.00 (500 in stock)
Toothpaste - $5.00 (250 in stock)
```

PARTICIPATION ACTIVITY 3.15.2: Sorting a vector of InventoryItem objects

1) Changing InventoryItem's < operator implementation from `return name < rhs.name;` to `return name > rhs.name;` causes items to be ____.

- ○ sorted ascending by name
- ○ sorted descending by name
- ○ left in their original order

2) Changing InventoryItem's < operator implementation to **`return price < rhs.price;`** causes items to be ____.

- ○ sorted ascending by name
- ○ sorted ascending by price
- ○ sorted ascending by number in stock

3) When using **`std::sort(items.begin(), items.end())`**, code in InventoryItem's < operator implementation must change in order to change the sorting criteria.

- ○ True
- ○ False

Sorting using a comparison function

The std::sort() function has an optional third parameter for an item comparison function. The comparison function has two parameters, each an array element. True is returned if the first element is less than the second, otherwise false is returned.

Using a comparison function allows two InventoryItem objects to be compared by price or number in stock, even when InventoryItem's < operator only compares item names. Also, the comparison function can cause a descending sort by returning true when the first element is greater than, instead of less than, the second element.

Figure 3.15.4: Sorting using a comparison function.

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <vector>
#include "InventoryItem.h"
using namespace std;

bool PriceAsc(const InventoryItem& a, const InventoryItem& b) {
   return a.price < b.price;
}

bool StockDesc(const InventoryItem& a, const InventoryItem& b) {
   return a.numberInStock > b.numberInStock;
}

int main() {
   // Create a vector of InventoryItem objects
   vector<InventoryItem> items1;
   items1.push_back(InventoryItem("Toothpaste", 5.00, 250));
   items1.push_back(InventoryItem("Toothbrush", 7.00, 500));
   items1.push_back(InventoryItem("Gum", 1.50, 100));
   items1.push_back(InventoryItem("Mints", 2.50, 50));
   items1.push_back(InventoryItem("Kettle chips", 3.00, 40));
   cout << "Unsorted:" << endl;
   for (InventoryItem& item : items1) {
      cout << item.ToString() << endl;
   }
   cout << endl;

   // Make a copy of the vector before sorting, so that each sort
   // starts from the same state
   vector<InventoryItem> items2(items1);

   // Sort the first vector ascending by price
   sort(items1.begin(), items1.end(), PriceAsc);
   cout << "Sorted ascending by price:" << endl;
   for (InventoryItem& item : items1) {
      cout << item.ToString() << endl;
   }
   cout << endl;

   // Sort the second vector descending by number in stock
   sort(items2.begin(), items2.end(), StockDesc);
   cout << "Sorted descending by number in stock:" << endl;
   for (InventoryItem& item : items2) {
      cout << item.ToString() << endl;
   }
   cout << endl;

   return 0;
}
```

```
Unsorted:
Toothpaste - $5.00 (250 in stock)
Toothbrush - $7.00 (500 in stock)
Gum - $1.50 (100 in stock)
Mints - $2.50 (50 in stock)
```

```
Kettle chips - $3.00 (40 in stock)

Sorted ascending by price:
Gum - $1.50 (100 in stock)
Mints - $2.50 (50 in stock)
Kettle chips - $3.00 (40 in stock)
Toothpaste - $5.00 (250 in stock)
Toothbrush - $7.00 (500 in stock)

Sorted descending by number in stock:
Toothbrush - $7.00 (500 in stock)
Toothpaste - $5.00 (250 in stock)
Gum - $1.50 (100 in stock)
Mints - $2.50 (50 in stock)
Kettle chips - $3.00 (40 in stock)
```

PARTICIPATION ACTIVITY 3.15.3: Using a comparison function with std::sort().

1) The comparison function returns true if the two items are equal.

- True
- False

2) When given a comparison function, std::sort() does not use InventoryItem's < operator.

- True
- False

3) A comparison function can use at most 1 of the 3 as sorting criteria: name, price, number in stock.

- True
- False

Additional sorting criteria

A comparison function can use multiple class members for sorting criteria. Ex: A comparison function first compares prices and returns true if the first item's price is greater than the second, or false if the first item's price is less. But when prices are equal, item names are compared as the secondary sort criterion. Using such a comparison function sorts items descending by price and ascending by name for groups of items that have the same price.

zyDE 3.15.1: Sorting descending by price, ascending by name for items with equal prices.

Current file: main.cpp ▾ **Load default template...**

```cpp
#include <algorithm>
#include <functional>
#include <iostream>
#include <vector>
#include "InventoryItem.h"
using namespace std;

bool PriceDescThenNameAsc(const InventoryItem& a, const InventoryItem& b) {
   // If prices are equal, use names as secondary criteria
   if (a.price == b.price) {
      return a.name < b.name;
   }
   return a.price > b.price;
}

int main() {
   // Create a vector of InventoryItem objects
   vector<InventoryItem> items;
   items.push_back(InventoryItem("Toothpaste", 5.00, 250));
   items.push_back(InventoryItem("Toothbrush", 7.00, 500));
   items.push_back(InventoryItem("Vitamins", 5.00, 500));
   items.push_back(InventoryItem("Gum", 1.50, 100));
   items.push_back(InventoryItem("Candy", 1.50, 200));
   items.push_back(InventoryItem("Strawberry smoothie", 5.00, 20));
   items.push_back(InventoryItem("Potato chips", 3.00, 40));
   items.push_back(InventoryItem("Mints", 2.50, 50));
   items.push_back(InventoryItem("Energy bar", 5.00, 30));
   items.push_back(InventoryItem("Eggs", 2.50, 40));
   cout << "Unsorted:" << endl;
   for (InventoryItem& item : items) {
      cout << item.ToString() << endl;
   }
   cout << endl;

   // Sort the vector, then show items
   sort(items.begin(), items.end(), PriceDescThenNameAsc);
   cout << "Sorted descending by price, ascending by name for equal prices:";
   cout << endl;
   for (InventoryItem& item : items) {
      cout << item.ToString() << endl;
   }
   cout << endl;

   return 0;
}
```

Run

PARTICIPATION ACTIVITY 3.15.4: Additional sorting criteria.

Refer to the code above.

1) What is `result` after executing the code below?

```
InventoryItem brandA("Coffee
brand A", 4.00, 100);
InventoryItem brandB("Coffee
brand B", 4.00, 100);
bool result =
PriceDescThenNameAsc(brandA,
brandB);
```

- true
- false
- Not enough information to determine

2) If InventoryItem("Milk", 3.00, 200) were also in the vector, where would the item be in the sorted output?

- Between Eggs and Mints
- Between Vitamins and Potato chips
- Between Potato chips and Eggs

3) If PriceDescThenNameAsc() compares two items with equal names and equal prices, then what is returned?

- true
- false
- Depends on if the number in stock is also equal
- Not enough information to determine

Sorting arrays with std::sort()

std::sort() can also be used on arrays. The std::begin() and std::end() functions return iterators to the beginning and end of an array, respectively. The < operator is used to compare array elements when a comparison function is not provided.

zyDE 3.15.2: Sorting arrays with std::sort().

Current file: main.cpp ▾ Load default template...

```
#include <algorithm>
#include <iostream>
#include <sstream>
#include "InventoryItem.h"
using namespace std;

bool CompareItemsByStock(const InventoryItem& a, const InventoryItem& b) {
   return a.numberInStock < b.numberInStock;
}

int main() {
   // Create and sort an array of integers
   int numbers[] = { 3, 7, 2, 8, 12, 4, 9, 5 };
   int numbersSize = sizeof(numbers) / sizeof(numbers[0]);

   cout << "Unsorted integer array: [" << numbers[0];
   for (int i = 1; i < numbersSize; i++) {
      cout << ", " << numbers[i];
   }
   cout << "]" << endl;

   std::sort(std::begin(numbers), std::end(numbers));
   cout << "Sorted integer array:   [" << numbers[0];
   for (int i = 1; i < numbersSize; i++) {
      cout << ", " << numbers[i];
   }
   cout << "]" << endl << endl;

   // Create and sort an array of InventoryItem objects
   InventoryItem items[] = {
      InventoryItem("Toothpaste", 5.00, 250),
      InventoryItem("Toothbrush", 7.00, 500),
      InventoryItem("Gum", 1.50, 100),
      InventoryItem("Mints", 2.50, 50),
      InventoryItem("Kettle chips", 3.00, 40)
   };
   int itemsSize = sizeof(items) / sizeof(items[0]);

   cout << "Unsorted InventoryItem array:" << endl;
   for (int i = 0; i < itemsSize; i++) {
      cout << items[i].ToString() << endl;
   }
   cout << endl;

   std::sort(std::begin(items), std::end(items), CompareItemsByStock);
   cout << "InventoryItem array sorted by number in stock:" << endl;
   for (int i = 0; i < itemsSize; i++) {
      cout << items[i].ToString() << endl;
   }
   cout << endl;

   return 0;
}
```

Run

PARTICIPATION ACTIVITY 3.15.5: Calling std::sort() to sort an array.

What two-argument call to std::sort() will sort the given array?

float grades[] = { 89.5, 92.1, 77.7, 64.3, 98.6, 73.0 };

```
std::sort(
);
```

Check Show answer

PARTICIPATION ACTIVITY 3.15.6: Sorting arrays with std::sort().

1) The following comparison function can be used to sort an array of integers in descending order.

```
bool CompareIntsDescending(const
int& a, const int& b) {
   return a > b;
}
```

- True
- False

2) A comparison function that works to sort a vector of InventoryItem objects can also be used to sort an array of InventoryItem objects.

○ True
○ False

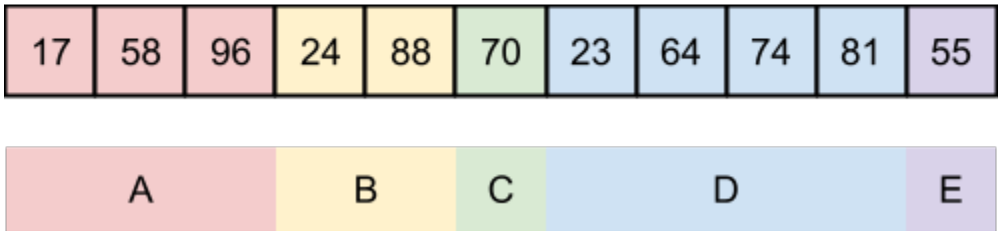
3.16 LAB: Natural merge sort

LAB ACTIVITY | 3.16.1: LAB: Natural merge sort | Full screen | 0 / 10

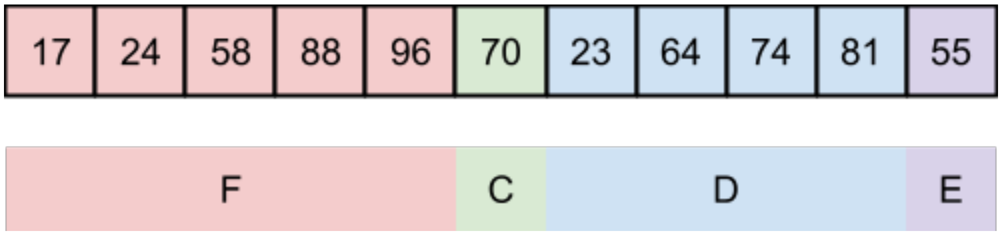
Natural merge sort overview

The merge sort algorithm recursively divides the array in half until an array with one element is reached. A variant of merge sort, called natural merge sort, instead finds already-sorted runs of elements and merges the runs together.

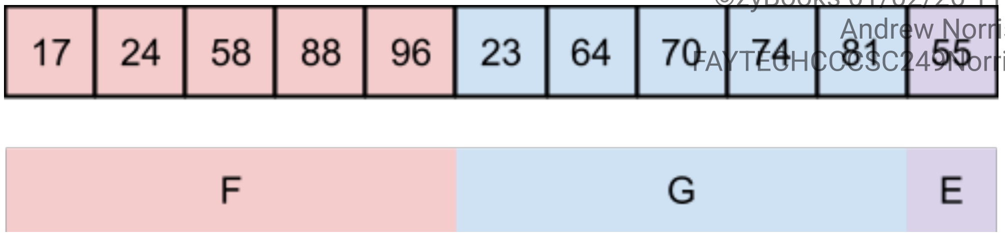
Ex: The unsorted array below has five sorted runs.



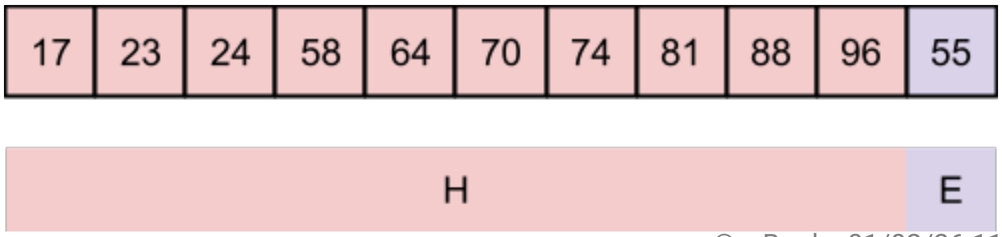
Natural merge sort starts at index 0 and finds sorted runs A and B. Runs A and B are merged, using the same merging algorithm as merge sort, to make run F.



Next, the algorithm starts after the merged part F, again looking for two sequential, sorted runs. Runs C and D are found and merged to make run G.

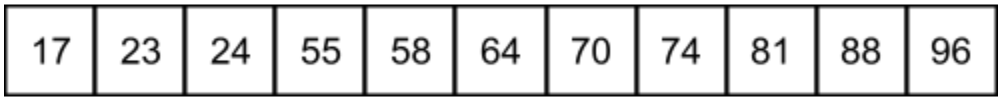


The algorithm then starts after the merged portion G. Only one run exists, run E, until the end of the array is reached. So the algorithm starts back at index 0, finds runs F and G, and merges to make run H.



Again, a single run is found after the just-merged part, so the search starts back at index 0. Runs H and E are found and merged.

One last search for a sorted run occurs, finding a sorted run length equal to the array's length. So the array is sorted and the algorithm terminates.



Step 1: Implement the GetSortedRunLength() member function

Implement the GetSortedRunLength() member function in NaturalMergeSorter.h. GetSortedRunLength() has three parameters:

- `array`: A pointer to an array of integers
- `arrayLength`: An integer for the array's length
- `startIndex`: An integer for the run's starting index

Return the number of array elements sorted in ascending order, starting at startIndex and ending either at the end of the sorted run, or the end of the array, whichever comes first. Return 0 if startIndex is out of bounds.

File main.cpp has several test cases for GetSortedRunLength() that can be run by clicking the **Run** button. One test case also exists for NaturalMergeSort(), but that can be ignored until step two is completed.

The program's output does not affect grading.

Submit for grading to ensure that the GetSortedRunLength() unit tests pass before proceeding.

Step 2: Implement the NaturalMergeSort() member function

Implement the NaturalMergeSort() member function in NaturalMergeSorter.h. NaturalMergeSort() must:

1. Start at index `i=0`
2. Get the length of the first sorted run, starting at `i`
    1. Return if the first run's length equals the array length
    2. If the first run ends at the array's end, reassign i=0 and repeat step 2

3. Get the length of the second sorted run, starting immediately after the first
4. Merge the two runs with the provided Merge() function
5. Reassign `i` with the first index after the second run, or 0 if the second run ends at the array's end
6. Go to step 2

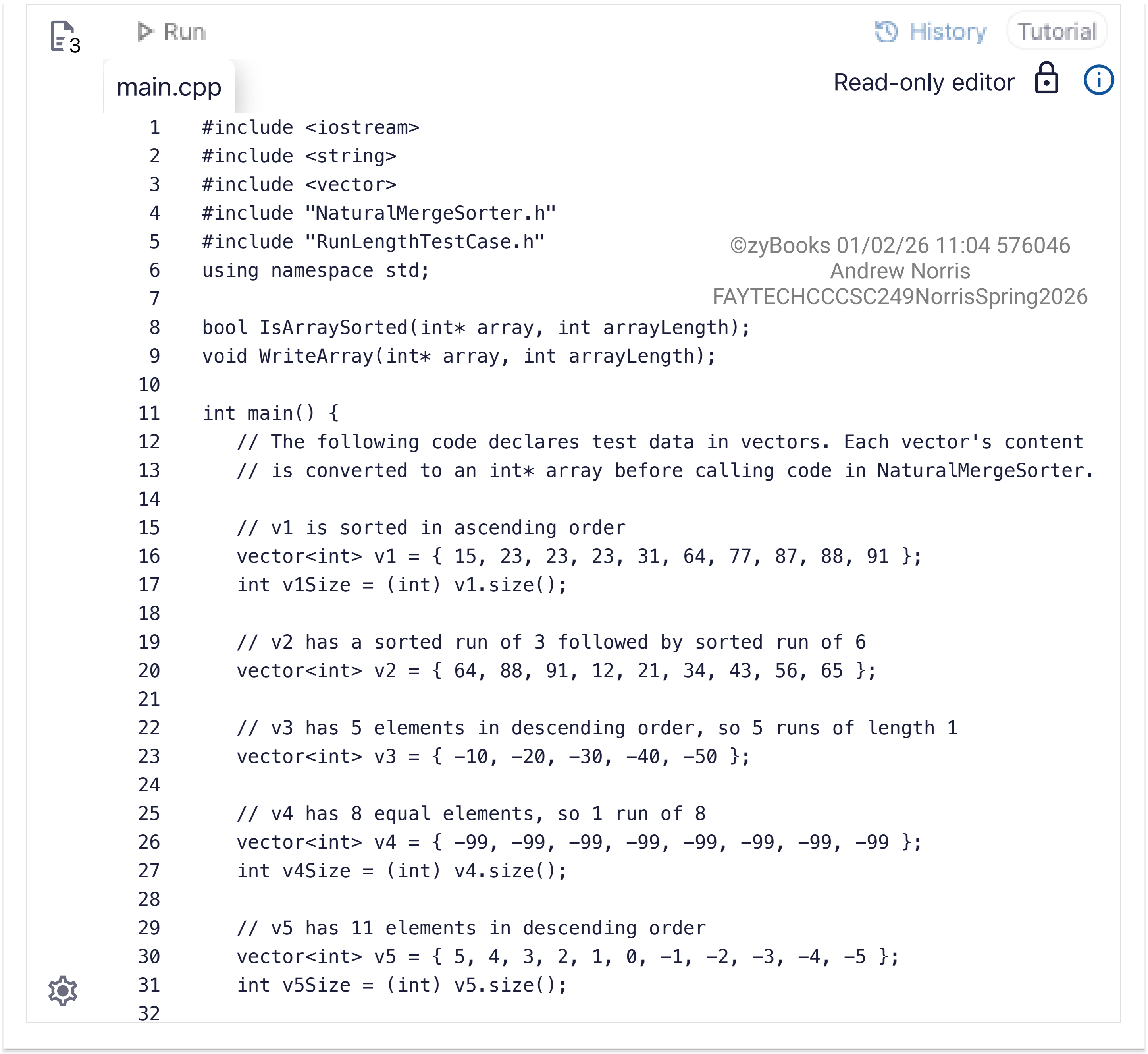
Open new tab Dock

3 Run History Tutorial

```
#include <iostream>
#include <string>
#include <vector>
#include "NaturalMergeSorter.h"
#include "RunLengthTestCase.h"
using namespace std;

bool IsArraySorted(int* array, int arrayLength);
void WriteArray(int* array, int arrayLength);

int main() {
   // The following code declares test data in vectors. Each vector's content
   // is converted to an int* array before calling code in NaturalMergeSorter.

   // v1 is sorted in ascending order
   vector<int> v1 = { 15, 23, 23, 23, 31, 64, 77, 87, 88, 91 };
   int v1Size = (int) v1.size();

   // v2 has a sorted run of 3 followed by sorted run of 6
   vector<int> v2 = { 64, 88, 91, 12, 21, 34, 43, 56, 65 };

   // v3 has 5 elements in descending order, so 5 runs of length 1
```

DESKTOP CONSOLE

Model Solution



```
#include <iostream>
#include <string>
#include <vector>
#include "NaturalMergeSorter.h"
#include "RunLengthTestCase.h"
using namespace std;

bool IsArraySorted(int* array, int arrayLength);
void WriteArray(int* array, int arrayLength);

int main() {
   // The following code declares test data in vectors. Each vector's content
   // is converted to an int* array before calling code in NaturalMergeSorter.

   // v1 is sorted in ascending order
   vector<int> v1 = { 15, 23, 23, 23, 31, 64, 77, 87, 88, 91 };
   int v1Size = (int) v1.size();

   // v2 has a sorted run of 3 followed by sorted run of 6
   vector<int> v2 = { 64, 88, 91, 12, 21, 34, 43, 56, 65 };

   // v3 has 5 elements in descending order, so 5 runs of length 1
   vector<int> v3 = { -10, -20, -30, -40, -50 };

   // v4 has 8 equal elements, so 1 run of 8
   vector<int> v4 = { -99, -99, -99, -99, -99, -99, -99, -99 };
   int v4Size = (int) v4.size();

   // v5 has 11 elements in descending order
   vector<int> v5 = { 5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5 };
   int v5Size = (int) v5.size();

```