Lecture 27:
Searching and Sorting

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
May 12th, 2010

19.2 Searching Algorithms

- Looking up a phone number, accessing a website and checking the definition of a word in a dictionary all involve searching large amounts of data.
- Searching algorithms all accomplish the same goal—finding an element that matches a given search key, if such an element does, in fact, exist.
- The major difference is the amount of effort they require to complete the search.
- One way to describe this effort is with Big O notation.
  - For searching and sorting algorithms, this is particularly dependent on the number of data elements.

19.3 Searching Algorithms (cont.)

- In Chapter 7, we discussed the linear search algorithm, which is a simple and easy-to-implement searching algorithm.
- We'll now discuss the efficiency of the linear search algorithm as measured by Big O notation.
- Then, we'll introduce a searching algorithm that is relatively efficient but more complex to implement.

19.3.1 Efficiency of Linear Search

- Suppose an algorithm simply tests whether the first element of a vector is equal to the second element of the vector.
- If the vector has 10 elements, this algorithm requires only one comparison.
- If the vector has 1000 elements, the algorithm still requires only one comparison.
- In fact, the algorithm is independent of the number of vector elements.
- This algorithm is said to have a constant runtime, which is represented in Big O notation as O(1).
- An algorithm that is O(1) does not necessarily require only one comparison.
- O(1) just means that the number of comparisons is constant—it does not grow as the size of the vector increases.
- An algorithm that tests whether the first element of a vector is equal to any of the next three elements will always require three comparisons, but in Big O notation it's still considered O(1).

19.3.1 Efficiency of Linear Search (cont.)

- O(1) is often pronounced "on the order of 1" or more simply "order 1."
- An algorithm that tests whether the first element of a vector is equal to any of the other elements of the vector requires at most n – 1 comparisons, where n is the number of elements in the vector.
- If the vector has 10 elements, the algorithm requires up to nine comparisons.
- If the vector has 1000 elements, the algorithm requires up to 999 comparisons.
- As *n* grows larger, the *n* part of the expression "dominates," and subtracting one becomes inconsequential.
- Big O is designed to highlight these dominant terms and ignore terms that become unimportant as *n* grows.

19.3.1 Efficiency of Linear Search (cont.)

- An algorithm that requires a total of $n - 1$ comparisons is said to be O(n).
- An O(n) algorithm is referred to as having a linear runtime.
- O(n) is often pronounced "on the order of n" or more simply "order n."

19.3.1 Efficiency of Linear Search (cont.)

- Now suppose you have an algorithm that tests whether any element of a vector is duplicated elsewhere in the vector.
- The first element must be compared with every other element in the vector.
- The second element must be compared with every other element except the first (it was already compared to the first).
- The third element must be compared with every other element except the first two.
- In the end, this algorithm will end up making $(n - 1) + (n - 2) + \ldots + 2 + 1$ or $n^2/2 - n/2$ *comparisons*.
- As n increases, the n^2 term dominates and the n term becomes inconsequential.
- Again, Big O notation highlights the n^2 term, leaving $n^2/2$.

19.3.1 Efficiency of Linear Search (cont.)

- Big O is concerned with how an algorithm's runtime grows in relation to the number of items processed.
- Suppose an algorithm requires n^2 comparisons.
- With four elements, the algorithm will require 16 comparisons; with eight elements, 64 comparisons.
- With this algorithm, doubling the number of elements quadruples the number of comparisons.
- Consider a similar algorithm requiring $n^2/2$ comparisons.
- With four elements, the algorithm will require eight comparisons; with eight elements, 32 comparisons.
- Again, doubling the number of elements quadruples the number of comparisons.
- Both of these algorithms grow as the square of *n*, so Big O ignores the constant, and both algorithms are considered to be $O(n^2)$, which is referred to as quadratic runtime and pronounced "on the order of n-squared" or more simply "order n-squared."

19.3.1 Efficiency of Linear Search (cont.)

- When n is small, O(n^2) algorithms will not noticeably affect performance.
- As *n* grows, you'll start to notice the performance degradation.
- An O(n^2) algorithm running on a million-element vector would require a trillion "operations" (where each could actually require several machine instructions to execute).
  - This could require a few hours to execute.
- A billion-element vector would require a quintillion operations, a number so large that the algorithm could take decades! Unfortunately, O(n^2) algorithms tend to be easy to write.

19.3.1 Efficiency of Linear Search (cont.)

- In this chapter, you'll see algorithms with more favorable Big O measures.
- These efficient algorithms often take a bit more cleverness and effort to create, but their superior performance can be worth the extra effort, especially as *n* gets large and as algorithms are compounded into larger programs.

19.3.1 Efficiency of Linear Search (cont.)

- The linear search algorithm runs in *O(n)* time.
- The worst case in this algorithm is that every element must be checked to determine whether the search key exists in the vector.
- If the size of the vector is doubled, the number of comparisons that the algorithm must perform is also doubled.
- Linear search can provide outstanding performance if the element matching the search key happens to be at or near the front of the vector.
- But we seek algorithms that perform well, on average, across all searches, including those where the element matching the search key is near the end of the vector.
- If a program needs to perform many searches on large vectors, it may be better to implement a different, more efficient algorithm, such as the binary search which we present in the next section.

Performance Tip 19.1

Sometimes the simplest algorithms perform poorly. Their virtue is that they're easy to program, test and debug. Sometimes more complex algorithms are required to realize maximum performance.

19.3.2 Binary Search

- The binary search algorithm is more efficient than the linear search algorithm, but it requires that the vector first be sorted.
- This is only worthwhile when the vector, once sorted, will be searched a great many times—or when the searching application has stringent performance requirements.
- The first iteration of this algorithm tests the middle element in the vector.
- If this matches the search key, the algorithm ends.

19.3.2 Binary Search (cont.)

- Assuming the vector is sorted in ascending order, then if the search key is less than the middle element, the search key cannot match any element in the second half of the vector and the algorithm continues with only the first half of the vector (i.e., the first element up to, but not including, the middle element).
- If the search key is greater than the middle element, the search key cannot match any element in the first half of the vector and the algorithm continues with only the second half of the vector (i.e., the element after the middle element through the last element).
- Each iteration tests the middle value of the remaining portion of the vector.
- If the element does not match the search key, the algorithm eliminates half of the remaining elements.
- The algorithm ends either by finding an element that matches the search key or by reducing the subvector to zero size.

```
1  // Fig 19.2: BinarySearch.h
2  // Class that contains a vector of random integers and a function
3  // that uses binary search to find an integer.
4  #include <vector>
5  using namespace std;
6
7  class BinarySearch
8  {
9  public:
10    BinarySearch( int ); // constructor initializes vector
11    int binarySearch( int ) const; // perform a binary search on vector
12    void displayElements() const; // display vector elements
13 private:
14    int size; // vector size
15    vector< int > data; // vector of ints
16    void displaySubElements( int, int ) const; // display range of values
17 }; // end class BinarySearch
```

Fig. 19.2 | `BinarySearch` class definition.

```
1  // Fig 19.3: BinarySearch.cpp
2  // BinarySearch class member-function definition.
3  #include <iostream>
4  #include <cstdlib> // prototypes for functions srand and rand
5  #include <ctime> // prototype for function time
6  #include <algorithm> // prototype for sort function
7  #include "BinarySearch.h" // class BinarySearch definition
8  using namespace std;
9
10 // constructor initializes vector with random ints and sorts the vector
11 BinarySearch::BinarySearch( int vectorSize )
12 {
13    size = ( vectorSize > 0 ? vectorSize : 10 ); // validate vectorSize
14    srand( time( 0 ) ); // seed using current time
15
16    // fill vector with random ints in range 10-99
17    for ( int i = 0; i < size; i++ )
18       data.push_back( 10 + rand() % 90 ); // 10-99
19
20    std::sort( data.begin(), data.end() ); // sort the data
21 } // end BinarySearch constructor
22
```

Fig. 19.3 | `BinarySearch` class member-function definition. (Part 1 of 4.)

```
23  // perform a binary search on the data
24  int BinarySearch::binarySearch( int searchElement ) const
25  {
26     int low = 0; // low end of the search area
27     int high = size - 1; // high end of the search area
28     int middle = ( low + high + 1 ) / 2; // middle element
29     int location = -1; // return value; -1 if not found
30
31     do // loop to search for element
32     {
33        // print remaining elements of vector to be searched
34        displaySubElements( low, high );
35
36        // output spaces for alignment
37        for ( int i = 0; i < middle; i++ )
38           cout << "   ";
39
40        cout << " * " << endl; // indicate current middle
41
42        // if the element is found at the middle
43        if ( searchElement == data[ middle ] )
44           location = middle; // location is the current middle
45        else if ( searchElement < data[ middle ] ) // middle is too high
46           high = middle - 1; // eliminate the higher half
```

Fig. 19.3 | `BinarySearch` class member-function definition. (Part 2 of 4.)

```
47          else // middle element is too low
48             low = middle + 1; // eliminate the lower half
49
50          middle = ( low + high + 1 ) / 2; // recalculate the middle
51       } while ( ( low <= high ) && ( location == -1 ) );
52
53       return location; // return location of search key
54    } // end function binarySearch
55
56    // display values in vector
57    void BinarySearch::displayElements() const
58    {
59       displaySubElements( 0, size - 1 );
60    } // end function displayElements
61
62    // display certain values in vector
63    void BinarySearch::displaySubElements( int low, int high ) const
64    {
65       for ( int i = 0; i < low; i++ ) // output spaces for alignment
66          cout << "   ";
67
68       for ( int i = low; i <= high; i++ ) // output elements left in vector
69          cout << data[ i ] << " ";
```

Fig. 19.3 | `BinarySearch` class member-function definition. (Part 3 of 4.)

```
70
71       cout << endl;
72    } // end function displaySubElements
```

Fig. 19.3 | `BinarySearch` class member-function definition. (Part 4 of 4.)

19.3.2 Binary Search (cont.)

- Line 29 initializes the `location` of the found element to `-1`—the value that will be returned if the search key is not found.
- Lines 31–51 loop until `low` is greater than `high` (this occurs when the element is not found) or `location` does not equal `-1` (indicating that the search key was found).
- Line 43 tests whether the value in the `middle` element is equal to `searchElement`.
  - If so, line 44 assigns `middle` to `location`.
  - Then the loop terminates and `location` is returned to the caller.
- Each iteration of the loop tests a single value (line 43) and eliminates half of the remaining values in the vector (line 46 or 48).

```
1  // Fig 19.4: Fig19_04.cpp
2  // BinarySearch test program.
3  #include <iostream>
4  #include "BinarySearch.h" // class BinarySearch definition
5  using namespace std;
6
7  int main()
8  {
9     int searchInt; // search key
10    int position; // location of search key in vector
11
12    // create vector and output it
13    BinarySearch searchVector ( 15 );
14    searchVector.displayElements();
15
16    // get input from user
17    cout << "\nPlease enter an integer value (-1 to quit): ";
18    cin >> searchInt; // read an int from user
19    cout << endl;
20
21    // repeatedly input an integer; -1 terminates the program
22    while ( searchInt != -1 )
23    {
```

Fig. 19.4 | `BinarySearch` test program. (Part 1 of 4.)

```
24          // use binary search to try to find integer
25          position = searchVector.binarySearch( searchInt );
26
27          // return value of -1 indicates integer was not found
28          if ( position == -1 )
29             cout << "The integer " << searchInt << " was not found.\n";
30          else
31             cout << "The integer " << searchInt
32                << " was found in position " << position << ".\n";
33
34          // get input from user
35          cout << "\n\nPlease enter an integer value (-1 to quit): ";
36          cin >> searchInt; // read an int from user
37          cout << endl;
38       } // end while
39    } // end main
```

Fig. 19.4 | `BinarySearch` test program. (Part 2 of 4.)

```
26 31 33 38 47 49 49 67 73 74 82 89 90 91 95

Please enter an integer value (-1 to quit): 38

26 31 33 38 47 49 49 67 73 74 82 89 90 91 95
                        *
26 31 33 38 47 49 49
          *
The integer 38 was found in position 3.


Please enter an integer value (-1 to quit): 91

26 31 33 38 47 49 49 67 73 74 82 89 90 91 95
                        *
                          73 74 82 89 90 91 95
                                    *
                                      90 91 95
                                          *
The integer 91 was found in position 13.
```

Fig. 19.4 | `BinarySearch` test program. (Part 3 of 4.)

```
Please enter an integer value (-1 to quit): 25

26 31 33 38 47 49 49 67 73 74 82 89 90 91 95
                        *
26 31 33 38 47 49 49
          *
26 31 33
    *
26
 *
The integer 25 was not found.


Please enter an integer value (-1 to quit): -1
```

Fig. 19.4 | `BinarySearch` test program. (Part 4 of 4.)

19.3.2 Binary Search (cont.)

- In the worst-case scenario, searching a sorted vector of 1023 elements will take only 10 comparisons when using a binary search.
- Repeatedly dividing 1023 by 2 (because, after each comparison, we can eliminate from consideration half of the remaining vector) and rounding down (because we also remove the middle element) yields the values 511, 255, 127, 63, 31, 15, 7, 3, 1 and 0.
- The number 1023 ($2^{10} – 1$) is divided by 2 only 10 times to get the value 0, which indicates that there are no more elements to test.

19.3.2 Binary Search (cont.)

- Dividing by 2 is equivalent to one comparison in the binary search algorithm.
- Thus, a vector of 1,048,575 ($2^{20} – 1$) elements takes a maximum of 20 comparisons to find the key, and a vector of about one billion elements takes a maximum of 30 comparisons to find the key.
- This is a tremendous improvement in performance over the linear search.
- For a one-billion-element vector, this is a difference between an average of 500 million comparisons for the linear search and a maximum of only 30 comparisons for the binary search! The maximum number of comparisons needed for the binary search of any sorted vector is the exponent of the first power of 2 greater than the number of elements in the vector, which is represented as $\log_2 n$.

19.3.2 Binary Search (cont.)

- All logarithms grow at roughly the same rate, so in Big O notation the base can be omitted.
- This results in a Big O of O(log *n*) for a binary search, which is also known as logarithmic runtime and pronounced "on the order of log n" or more simply "order log *n*."

19.4 Sorting Algorithms

- Sorting data (i.e., placing the data into some particular order, such as ascending or descending) is one of the most important computing applications.
- The choice of algorithm affects only the runtime and memory use of the program.
- The next section examines the efficiency of selection sort and insertion sort algorithms using Big O notation.
- The last algorithm—merge sort, which we introduce in this chapter—is much faster but is more difficult to implement.

19.4.1 Efficiency of Selection Sort

- Selection sort is an easy-to-implement, but inefficient, sorting algorithm.
  - The first iteration of the algorithm selects the smallest element in the vector and swaps it with the first element.
  - The second iteration selects the second-smallest element (which is the smallest element of the remaining elements) and swaps it with the second element.
  - The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last element, leaving the largest element in the last index.
  - After the i^{th} iteration, the smallest i elements of the vector will be sorted into increasing order in the first i elements of the vector.

19.4.1 Efficiency of Selection Sort (cont.)

- The selection sort algorithm iterates $n - 1$ times, each time swapping the smallest remaining element into its sorted position.
- Locating the smallest remaining element requires $n - 1$ comparisons during the first iteration, $n - 2$ during the second iteration, then $n - 3, \ldots, 3, 2, 1$.
- This results in a total of $n(n - 1)/2$ or $(n^2 - n)/2$ comparisons.
- In Big O notation, smaller terms drop out and constants are ignored, leaving a final Big O of $O(n^2)$.

Insertion sort is another simple, but inefficient, sorting algorithm.

  - The algorithm's first iteration takes the second element in the vector and, if it's less than the first element, swaps it with the first element.
  - The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements, so all three elements are in order.
  - At the i^{th} iteration of this algorithm, the first i elements in the original vector will be sorted.
- Insertion sort iterates $n - 1$ times, inserting an element into the appropriate position in the elements sorted so far.

19.4.2 Efficiency of Insertion Sort (cont.)

- For each iteration, determining where to insert the element can require comparing the element to each of the preceding elements—*n* – 1 comparisons in the worst case.
- Each individual repetition statement runs in *O(n)* time.
- For determining Big O notation, nested statements mean that you must multiply the number of comparisons.
- For each iteration of an outer loop, there will be a certain number of iterations of the inner loop.
- In this algorithm, for each *O(n)* iteration of the outer loop, there will be *O(n)* iterations of the inner loop, resulting in a Big O of *O(n * n)* or $O(n^2)$.

19.4.3 Merge Sort (A Recursive Implementation)

- Merge sort is an efficient sorting algorithm but is conceptually more complex than selection sort and insertion sort.
- The merge sort algorithm sorts a vector by splitting it into two equal-sized subvectors, sorting each subvector then merging them into one larger vector.
- Merge sort performs the merge by looking at the first element in each vector, which is also the smallest element in the vector.
- Merge sort takes the smallest of these and places it in the first element of the larger, sorted vector.
- If there are still elements in the subvector, merge sort looks at the second element in that subvector (which is now the smallest element remaining) and compares it to the first element in the other subvector.
- Merge sort continues this process until the larger vector is filled.

19.4.3 Merge Sort (A Recursive Implementation) (cont.)

- The implementation of merge sort in this example is recursive.
- The base case is a vector with one element.
- A one-element vector is, of course, sorted, so merge sort immediately returns when it's called with a one-element vector.
- The recursion step splits a vector of two or more elements into two equal-sized subvectors, recursively sorts each subvector, then merges them into one larger, sorted vector.
  - If there is an odd number of elements, one subvector is one element larger than the other.

19.4.3 Merge Sort (A Recursive Implementation) (cont.)

- Figure 19.5 defines class `MergeSort`, and lines 22–25 of Fig. 19.6 define the `sort` function.
- Line 24 calls function `sortSubVector` with `0` and `size - 1` as the arguments.
  - These arguments correspond to the beginning and ending indices of the vector to be sorted, causing `sortSubVector` to operate on the entire vector.
  - Function `sortSubVector` is defined in lines 28–52.
  - Line 31 tests the base case.
  - If the size of the vector is `0`, the vector is already sorted, so the function simply returns immediately.
  - If the size of the vector is greater than or equal to `1`, the function splits the vector in two, recursively calls function `sortSubVector` to sort the two subvectors, then merges them.

19.4.3 Merge Sort (A Recursive Implementation) (cont.)

- Line 46 recursively calls function `sortSubVector` on the first half of the vector, and line 47 recursively calls function `sortSubVector` on the second half of the vector.
- When these two function calls return, each half of the vector has been sorted.
- Line 50 calls function `merge` (lines 55–99) on the two halves of the vector to combine the two sorted vectors into one larger sorted vector.

```
1  // Fig 19.5: MergeSort.h
2  // Class that creates a vector filled with random integers.
3  // Provides a function to sort the vector with merge sort.
4  #include <vector>
5  using namespace std;
6
7  // MergeSort class definition
8  class MergeSort
9  {
10 public:
11    MergeSort( int ); // constructor initializes vector
12    void sort(); // sort vector using merge sort
13    void displayElements() const; // display vector elements
14 private:
15    int size; // vector size
16    vector< int > data; // vector of ints
17    void sortSubVector( int, int ); // sort subvector
18    void merge( int, int, int, int ); // merge two sorted vectors
19    void displaySubVector( int, int ) const; // display subvector
20 }; // end class SelectionSort
```

Fig. 19.5 | `MergeSort` class definition.

```
1  // Fig 19.6: MergeSort.cpp
2  // Class MergeSort member-function definition.
3  #include <iostream>
4  #include <vector>
5  #include <cstdlib> // prototypes for functions srand and rand
6  #include <ctime> // prototype for function time
7  #include "MergeSort.h" // class MergeSort definition
8  using namespace std;
9
10 // constructor fill vector with random integers
11 MergeSort::MergeSort( int vectorSize )
12 {
13    size = ( vectorSize > 0 ? vectorSize : 10 ); // validate vectorSize
14    srand( time( 0 ) ); // seed random number generator using current time
15
16    // fill vector with random ints in range 10-99
17    for ( int i = 0; i < size; i++ )
18       data.push_back( 10 + rand() % 90 );
19 } // end MergeSort constructor
20
21 // split vector, sort subvectors and merge subvectors into sorted vector
22 void MergeSort::sort()
23 {
```

Fig. 19.6 | `MergeSort` class member-function definition. (Part 1 of 5.)

```
24       sortSubVector( 0, size - 1 ); // recursively sort entire vector
25    } // end function sort
26
27    // recursive function to sort subvectors
28    void MergeSort::sortSubVector( int low, int high )
29    {
30       // test base case; size of vector equals 1
31       if ( ( high - low ) >= 1 ) // if not base case
32       {
33          int middle1 = ( low + high ) / 2; // calculate middle of vector
34          int middle2 = middle1 + 1; // calculate next element over
35
36          // output split step
37          cout << "split:   ";
38          displaySubVector( low, high );
39          cout << endl << "         ";
40          displaySubVector( low, middle1 );
41          cout << endl << "         ";
42          displaySubVector( middle2, high );
43          cout << endl << endl;
44
45          // split vector in half; sort each half (recursive calls)
46          sortSubVector( low, middle1 ); // first half of vector
47          sortSubVector( middle2, high ); // second half of vector
```

Fig. 19.6 | `MergeSort` class member-function definition. (Part 2 of 5.)

```
48
49          // merge two sorted vectors after split calls return
50          merge( low, middle1, middle2, high );
51       } // end if
52    } // end function sortSubVector
53
54    // merge two sorted subvectors into one sorted subvector
55    void MergeSort::merge( int left, int middle1, int middle2, int right )
56    {
57       int leftIndex = left; // index into left subvector
58       int rightIndex = middle2; // index into right subvector
59       int combinedIndex = left; // index into temporary working vector
60       vector< int > combined( size ); // working vector
61
62       // output two subvectors before merging
63       cout << "merge:   ";
64       displaySubVector( left, middle1 );
65       cout << endl << "         ";
66       displaySubVector( middle2, right );
67       cout << endl;
68
69       // merge vectors until reaching end of either
70       while ( leftIndex <= middle1 && rightIndex <= right )
71       {
```

Fig. 19.6 | `MergeSort` class member-function definition. (Part 3 of 5.)

```
72          // place smaller of two current elements into result
73          // and move to next space in vector
74          if ( data[ leftIndex ] <= data[ rightIndex ] )
75             combined[ combinedIndex++ ] = data[ leftIndex++ ];
76          else
77             combined[ combinedIndex++ ] = data[ rightIndex++ ];
78       } // end while
79
80       if ( leftIndex == middle2 ) // if at end of left vector
81       {
82          while ( rightIndex <= right ) // copy in rest of right vector
83             combined[ combinedIndex++ ] = data[ rightIndex++ ];
84       } // end if
85       else // at end of right vector
86       {
87          while ( leftIndex <= middle1 ) // copy in rest of left vector
88             combined[ combinedIndex++ ] = data[ leftIndex++ ];
89       } // end else
90
91       // copy values back into original vector
92       for ( int i = left; i <= right; i++ )
93          data[ i ] = combined[ i ];
94
```

Fig. 19.6 | `MergeSort` class member-function definition. (Part 4 of 5.)

```
95       // output merged vector
96       cout << "          ";
97       displaySubVector( left, right );
98       cout << endl << endl;
99    } // end function merge
100
101   // display elements in vector
102   void MergeSort::displayElements() const
103   {
104      displaySubVector( 0, size - 1 );
105   } // end function displayElements
106
107   // display certain values in vector
108   void MergeSort::displaySubVector( int low, int high ) const
109   {
110      // output spaces for alignment
111      for ( int i = 0; i < low; i++ )
112         cout << "   ";
113
114      // output elements left in vector
115      for ( int i = low; i <= high; i++ )
116         cout << " " << data[ i ];
117   } // end function displaySubVector
```

Fig. 19.6 | `MergeSort` class member-function definition. (Part 5 of 5.)

19.4.3 Merge Sort (A Recursive Implementation) (cont.)

- Lines 70–78 in function `merge` loop until the program reaches the end of either subvector.
- Line 74 tests which element at the beginning of the vectors is smaller.
- If the element in the left vector is smaller, line 75 places it in position in the combined vector.
- If the element in the right vector is smaller, line 77 places it in position in the combined vector.
- When the `while` loop has completed (line 78), one entire subvector is placed in the combined vector, but the other subvector still contains data.
- Line 80 tests whether the left vector has reached the end.
- If so, lines 82–83 fill the combined vector with the elements of the right vector.

19.4.3 Merge Sort (A Recursive Implementation) (cont.)

- If the left vector has not reached the end, then the right vector must have reached the end, and lines 87–88 fill the combined vector with the elements of the left vector.
- Finally, lines 92–93 copy the combined vector into the original vector.
- Figure 19.7 creates and uses a `MergeSort` object.
- The output from this program displays the splits and merges performed by merge sort, showing the progress of the sort at each step of the algorithm.

```
1  // Fig 19.7: Fig19_07.cpp
2  // MergeSort test program.
3  #include <iostream>
4  #include "MergeSort.h" // class MergeSort definition
5  using namespace std;
6
7  int main()
8  {
9     // create object to perform merge sort
10    MergeSort sortVector( 10 );
11
12    cout << "Unsorted vector:" << endl;
13    sortVector.displayElements(); // print unsorted vector
14    cout << endl << endl;
15
16    sortVector.sort(); // sort vector
17
18    cout << "Sorted vector:" << endl;
19    sortVector.displayElements(); // print sorted vector
20    cout << endl;
21 } // end main
```

Fig. 19.7 | MergeSort test program. (Part 1 of 5.)

```
Unsorted vector:
 30 47 22 67 79 18 60 78 26 54

split:    30 47 22 67 79 18 60 78 26 54
          30 47 22 67 79
                         18 60 78 26 54

split:    30 47 22 67 79
          30 47 22
                   67 79

split:    30 47 22
          30 47
                22

split:    30 47
          30
             47

merge:    30
             47
          30 47
```

Fig. 19.7 | `MergeSort` test program. (Part 2 of 5.)

```
merge:    30 47
                22
          22 30 47

split:              67 79
                    67
                       79

merge:              67
                       79
                    67 79

merge:    22 30 47
                    67 79
          22 30 47 67 79

split:                     18 60 78 26 54
                           18 60 78
                                    26 54

split:                     18 60 78
                           18 60
                                 78
```

Fig. 19.7 | `MergeSort` test program. (Part 3 of 5.)

```
split:                    18 60
                          18
                             60

merge:                    18
                             60
                          18 60

merge:                    18 60
                                78
                          18 60 78

split:                                26 54
                                      26
                                         54

merge:                                26
                                         54
                                      26 54

merge:                    18 60 78
                                      26 54
                          18 26 54 60 78
```

Fig. 19.7 | `MergeSort` test program. (Part 4 of 5.)

```
merge:    22 30 47 67 79
                       18 26 54 60 78
          18 22 26 30 47 54 60 67 78 79

Sorted vector:
 18 22 26 30 47 54 60 67 78 79
```

Fig. 19.7 | `MergeSort` test program. (Part 5 of 5.)

19.4.3 Merge Sort (A Recursive Implementation) (cont.)

- Merge sort is a far more efficient algorithm than either insertion sort or selection sort.
- Consider the first (nonrecursive) call to function `sortSubVector` (line 24).
- This results in two recursive calls to function `sortSubVector` with subvectors each approximately half the size of the original vector, and a single call to function `merge`.
- This call to function `merge` requires, at worst, $n - 1$ comparisons to fill the original vector, which is O(n).
- The two calls to function `sortSubVector` result in four more recursive calls to function `sortSubVector`—each with a subvector approximately one-quarter the size of the original vector—and two calls to function `merge`.
- These two calls to function `merge` each require, at worst, $n/2 - 1$ comparisons, for a total number of comparisons of *O(n)*.

19.4.3 Merge Sort (A Recursive Implementation) (cont.)

- This process continues, each call to `sortSubVector` generating two additional calls to `sortSubVector` and a call to `merge`, until the algorithm has split the vector into one-element subvectors.
- At each level, *O(n)* comparisons are required to merge the subvectors.
- Each level splits the size of the vectors in half, so doubling the size of the vector requires one more level.
- Quadrupling the size of the vector requires two more levels.
- This pattern is logarithmic and results in $\log_2 n$ *levels*.
- This results in a total efficiency of $O(n \log n)$.

19.4.3 Merge Sort (A Recursive Implementation) (cont.)

- Figure 19.8 summarizes the searching and sorting algorithms we cover in this book and lists the Big O for each.
- Figure 19.9 lists the Big O categories we've covered in this chapter along with a number of values for *n* to highlight the differences in the growth rates.

Algorithm	Location	Big O
Searching Algorithms		
Linear search	Section 7.7	$O(n)$
Binary search	Section 19.2.2	$O(\log n)$
Recursive linear search	Exercise 19.8	$O(n)$
Recursive binary search	Exercise 19.9	$O(\log n)$
Sorting Algorithms		
Insertion sort	Section 7.8	$O(n^2)$
Selection sort	Section 8.6	$O(n^2)$
Merge sort	Section 19.3.3	$O(n \log n)$
Bubble sort	Exercises 19.5 and 19.6	$O(n^2)$
Quicksort	Exercise 19.10	Worst case: $O(n^2)$ Average case: $O(n \log n)$

Fig. 19.8 | Searching and sorting algorithms with Big O values.

n	Approximate decimal value	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
2^{10}	1000	10	2^{10}	$10 \cdot 2^{10}$	2^{20}
2^{20}	1,000,000	20	2^{20}	$20 \cdot 2^{20}$	2^{40}
2^{30}	1,000,000,000	30	2^{30}	$30 \cdot 2^{30}$	2^{60}

Fig. 19.9 | Approximate number of comparisons for common Big O notations.

Questions