Searching and Big O

© Copyright 1992-2015 by Pearson Education, Inc. All Rights Reserved.

19.1 Introduction

- Searching data involves determining whether a value (referred to as the search key) is present in the data and, if so, finding its location.
  - Two popular search algorithms are the simple linear search and the faster but more complex binary search.
- Sorting places data in ascending or descending order, based on one or more sort keys.
  - This chapter introduces two simple sorting algorithms, the selection sort and the insertion sort, along with the more efficient but more complex merge sort.

© Copyright 1992-2015 by Pearson Education, Inc. All Rights Reserved.

19.2 Linear Search

- The linear search algorithm searches each element in an array sequentially.
  - If the search key does not match an element in the array, the algorithm tests each element, and when the end of the array is reached, informs the user that the search key is not present.
  - If the search key is in the array, the algorithm tests each element until it finds one that matches the search key and returns the index of that element.

© Copyright 1992-2015 by Pearson Education, Inc. All Rights Reserved.

```java
// Fig. 19.2: LinearSearchTest.java
// Sequentially searching an array for an item.
import java.security.SecureRandom;
import java.util.Arrays;
import java.util.Scanner;

public class LinearSearchTest
{
   // perform a linear search on the data
   public static int linearSearch(int data[], int searchKey)
   {
      // loop through array sequentially
      for (int index = 0; index < data.length; index++)
         if (data[index] == searchKey)
            return index; // return index of integer

      return -1; // integer was not found
   } // end method linearSearch

   public static void main(String[] args)
   {
      Scanner input = new Scanner(System.in);
      SecureRandom generator = new SecureRandom();

```

Fig. 19.2 | Sequentially searching an array for an item. (Part 1 of 3.)

© Copyright 1992-2015 by Pearson Education, Inc. All Rights Reserved.

```
      int[] data = new int[10]; // create array

      for (int i = 0; i < data.length; i++) // populate array
         data[i] = 10 + generator.nextInt(90);

      System.out.printf("%s%n%n", Arrays.toString(data)); // display array

      // get input from user
      System.out.print("Please enter an integer value (-1 to quit): ");
      int searchInt = input.nextInt();

      // repeatedly input an integer; -1 terminates the program
      while (searchInt != -1)
      {
         int position = linearSearch(data, searchInt); // perform search

         if (position == -1) // not found
            System.out.printf("%d was not found%n%n", searchInt);
         else // found
            System.out.printf("%d was found in position %d%n%n",
               searchInt, position);

```

Fig. 19.2 | Sequentially searching an array for an item. (Part 2 of 3.)

© Copyright 1992-2015 by Pearson Education, Inc. All Rights Reserved.

```java
            // get input from user
            System.out.print("Please enter an integer value (-1 to quit): ");
            searchInt = input.nextInt();
         }
      } // end main
   } // end class LinearSearchTest
```

```
[59, 97, 34, 90, 79, 56, 24, 51, 30, 69]

Please enter an integer value (-1 to quit): 79
79 was found in position 4

Please enter an integer value (-1 to quit): 61
61 was not found

Please enter an integer value (-1 to quit): 51
51 was found in position 7

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

Fig. 19.2 | Sequentially searching an array for an item. (Part 3 of 3.)

© Copyright 1992-2015 by Pearson Education, Inc. All Rights Reserved.

19.3 Big O Notation

- Searching algorithms all accomplish the *same* goal—finding an element (or elements) 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.*
- Big O notation indicates how hard an algorithm may have to work to solve a problem.
  - For searching and sorting algorithms, this depends particularly on how many data elements there are.

© Copyright 1992-2015 by Pearson Education, Inc. All Rights Reserved.

19.3.1 O(1) Algorithms

- If an algorithm is completely independent of the number of elements in the array, it is said to have a constant run time, which is represented in Big O notation as O(1) and pronounced as "order one."
  - An algorithm that's *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 array increases.

© Copyright 1992-2015 by Pearson Education, Inc. All Rights Reserved.

19.3.2 O(*n*) Algorithms

- 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 run time.
  - $O(n)$ is often pronounced "on the order of n" or simply "order n."

© Copyright 1992-2015 by Pearson Education, Inc. All Rights Reserved.

19.3.3 O(n^2) Algorithms

- Constant factors are omitted in Big O notation.
- Big O is concerned with how an algorithm's run time grows in relation to the number of items processed.
- **O(n²) is referred to as quadratic run time and pronounced "on the order of *n-squared" or more simply "order n-squared."***
  - When *n is small, O(n^2) algorithms (running on today's computers) will not noticeably affect performance.*
  - But as *n grows, you'll start to notice the performance degradation.*
  - An *O(n^2) algorithm running on a million-element array would require a trillion "operations" (where each could actually require several machine instructions to execute).*
  - A billion-element array would require a quintillion operations.
- You'll also see algorithms with more favorable Big O measures.

© Copyright 1992-2015 by Pearson Education, Inc. All Rights Reserved.

19.3.4 Big O of the Linear Search

- 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 item exists in the array.
  - If the size of the array 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 array.
  - 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 array.
- If a program needs to perform many searches on large arrays, it’s better to implement a more efficient algorithm, such as the binary search.

© Copyright 1992-2015 by Pearson Education, Inc. All Rights Reserved.

19.4 Binary Search

- The binary search algorithm is more efficient than linear search, but it requires that the array be sorted.
  - The first iteration tests the middle element in the array. If this matches the search key, the algorithm ends.
  - If the search key is less than the *middle* element, the algorithm continues with only the first half of the array.
  - If the search key is *greater than* the middle element, the algorithm continues with only the second half.
  - Each iteration tests the middle value of the remaining portion of the array.
  - If the search key does not match the element, the algorithm eliminates half of the remaining elements.
  - The algorithm ends by either finding an element that matches the search key or reducing the subarray to zero size.

© Copyright 1992-2015 by Pearson Education, Inc. All Rights Reserved.

```
[13, 18, 29, 36, 42, 47, 56, 57, 63, 68, 80, 81, 82, 88, 88]

Please enter an integer value (-1 to quit): 18
13 18 29 36 42 47 56 57 63 68 80 81 82 88 88
                      *

13 18 29 36 42 47 56
          *

13 18 29
    *

18 was found in position 1

Please enter an integer value (-1 to quit): 82
13 18 29 36 42 47 56 57 63 68 80 81 82 88 88
                      *

                        63 68 80 81 82 88 88
                                  *

                                    82 88 88
                                        *

                                    82
                                     *

82 was found in position 12
```

Fig. 19.3 | Use binary search to locate an item in an array (the * in the output marks the middle element). (Part 6 of 7.)

© Copyright 1992-2015 by Pearson Education, Inc. All Rights Reserved.

```
Please enter an integer value (-1 to quit): 69
13 18 29 36 42 47 56 57 63 68 80 81 82 88 88
                     *
                        63 68 80 81 82 88 88
                                 *
                        63 68 80
                           *
                              80
                               *
69 was not found

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

Fig. 19.3 | Use binary search to locate an item in an array (the * in the output marks the middle element). (Part 7 of 7.)

© Copyright 1992–2015 by Pearson Education, Inc. All Rights Reserved.

19.4.2 Efficiency of the Binary Search

- In the worst-case scenario, searching a sorted array of 1023 elements takes *only 10 comparisons* when using a binary search.
  - 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.
  - Dividing by 2 is equivalent to one comparison in the binary search algorithm.
- Thus, an array of 1,048,575 (2^{20} – 1) elements takes a *maximum of 20 comparisons* to find the key, and an array of over one billion elements takes a maximum of 30 comparisons to find the key.
  - A difference between an average of 500 million comparisons for the linear search and a *maximum of only 30 comparisons* for the binary search!

© Copyright 1992-2015 by Pearson Education, Inc. All Rights Reserved.

19.4.2 Efficiency of the Binary Search (cont.)

- The maximum number of comparisons needed for the binary search of any sorted array is the exponent of the first power of 2 greater than the number of elements in the array, which is represented as $\log_2 n$.
- 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 run time and pronounced as “order log n.”

© Copyright 1992-2015 by Pearson Education, Inc. All Rights Reserved.

Recursion

© Copyright 1992-2015 by Pearson Education, Inc. All Rights Reserved.

18.2 Recursion Concepts

- When a recursive method is called to solve a problem, it actually is capable of solving only the *simplest case(s)*, or base case(s).
  - If the method is called with a *base case*, it returns a result.
- If the method is called with a more complex problem, it divides the problem into two conceptual pieces
  - a piece that the method knows how to do and
  - a piece that it does not know how to do.
- To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or smaller version of it.
- Because this new problem resembles the original problem, the method calls a fresh copy of itself to work on the smaller problem
  - this is a recursive call
  - also called the recursion step

© Copyright 1992-2015 by Pearson Education, Inc. All Rights Reserved.

18.2 Recursion Concepts (cont.)

- The recursion step normally includes a `return` statement, because its result will be combined with the portion of the problem the method knew how to solve to form a result that will be passed back to the original caller.
- The recursion step executes while the original method call is still active.
- For recursion to eventually terminate, each time the method calls itself with a simpler version of the original problem, the sequence of smaller and smaller problems must converge on a base case.
    - When the method recognizes the base case, it returns a result to the previous copy of the method.
    - A sequence of returns ensues until the original method call returns the final result to the caller.

© Copyright 1992-2015 by Pearson Education, Inc. All Rights Reserved.

18.2 Recursion Concepts (cont.)

- A recursive method may call another method, which may in turn make a call back to the recursive method.
  - This is known as an indirect recursive call or indirect recursion.

© Copyright 1992-2015 by Pearson Education, Inc. All Rights Reserved.

18.7 Recursion vs. Iteration

- Both iteration and recursion are *based on a control statement*:
  - Iteration uses a repetition statement (e.g., `for`, `while` or `do`…`while`)
  - Recursion uses a selection statement (e.g., `if`, `if`…`else` or `switch`)
- Both iteration and recursion involve *repetition*:
  - Iteration explicitly uses a repetition statement
  - Recursion achieves repetition through repeated method calls
- Iteration and recursion each involve a *termination test*:
  - Iteration terminates when the loop-continuation condition fails
  - Recursion terminates when a base case is reached.

© Copyright 1992-2015 by Pearson Education, Inc. All Rights Reserved.

18.7 Recursion vs. Iteration (cont.)

- Both iteration and recursion can occur infinitely:
  - An infinite loop occurs with iteration if the loop-continuation test never becomes false
  - Infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case, or if the base case is not tested.

© Copyright 1992-2015 by Pearson Education, Inc. All Rights Reserved.

18.7 Recursion vs. Iteration (cont.)

- Recursion repeatedly invokes the mechanism, and consequently the overhead, of method calls.
  - Can be expensive in terms of both processor time and memory space.
- Each recursive call causes another copy of the method (actually, only the method's variables, stored in the activation record) to be created
  - this set of copies can consume considerable memory space.
- Since iteration occurs within a method, repeated method calls and extra memory assignment are avoided.

© Copyright 1992-2015 by Pearson Education, Inc. All Rights Reserved.