1 **Array Searching and Sorting**

CST242

2 **Multiple-Dimensional Arrays**

- Elements of arrays may have *more than one* index, e.g.
  mileage[1][3]
- In the example above, it is as though the elements are in two-dimensions such a table with rows and columns
- Additional dimensions that go beyond two may be included as is needed

4 **Accessing Array Elements**

- An array element is accessed by using row and column integers in two sets of brackets
- Example:
  mileage[1][3]
- Integer variables may be used to replace either or both of the indexes, e.g.
  mileage[start][end]

5 **Declaring a 2-Dimensional Array**

- The new array object is created using *two* sets of brackets in the array declaration
- First bracketed value is conceptually the number of *rows*
- Second bracketed value is conceptually the number of *columns*
- Example:
  int[][] mileage = new int[4][4];

6 **Initializing a 2-Dimensional Array in the Declaration**

- Each "row" is assigned in a *separate subset* of command-delimited brackets
- Example:

```
int[][] mileage =
{
   {  0, 160, 390,  40},
   {160,   0, 240, 200},
   {390, 240,   0, 440},
   { 40, 200, 440,   0}
};
```

8 **Searching and Sorting**

- 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

9 **The Arrays Class**

- Class Arrays contains methods for manipulating arrays (such as sorting and searching)

- Also contains a static methods that allow arrays to be viewed as lists
- Found in the “java.util” package, e.g.
  import java.util.Arrays;

10 **The sort() Method of Class Arrays**

- The sort method is a static method of the Arrays class that sorts the elements in an array in ascending order by default
- Overloaded version lets sort order to be changed
- Format:
  Arrays.sort(*arrayObject*);
- Example:
  Arrays.sort(values);

11 **The toString() Method of Class Arrays**

- The toString method is a static method of class Arrays that displays the array object as a *list* (without iterating through it)
- Format:
  Arrays.toString(*arrayObject*);
- Example:
  System.out.println(Arrays.toString(values));

13 **Linear Search (Page 1)**

- The linear search algorithm searches each element in an array *sequentially* attempting to match elements to a *search key*
  1. The algorithm tests each element against the search key until it *finds one that matches* the search key and returns the *index* of that element
  2. 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

14 **Linear Search (Page 2)**

- If there are *duplicate values* in the array, the linear search returns the index of the *first element* in the array that matches the search key

15 **Big O Notation (Page 1)**

- Searching algorithms all accomplish the same goal—finding an element that matches a given search key, if such an element does exist
- Major difference is the *amount of effort* they require to complete the search

16 **Big O Notation (Page 2)**

- Big O notation indicates the worst-case run time for an algorithm—that is 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

17 **Big O Notation (Page 3)**

- 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)
  - 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 array increases

18 **Big O Notation (Page 4)**

- 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*"

19 **Big O Notation (Page 5)**

- 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

20 **Big O Notation (Page 6)**

- $O(n^2)$ 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

21 **Big O Notation (Page 6)**

- You will also see algorithms with more favorable Big O measures

22 **Big O for a Linear Search (Page 1)**

- 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

23 **Big O for a Linear Search (Page 2)**

- A linear search can provide good performance if the element matching the search key happens to be at or near the front of the array
  - 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

24 **Big O for a Linear Search (Page 3)**

- If a program needs to perform many searches on large arrays, it is better to implement a more efficient algorithm, such as the binary search

26 **Binary Search (Page 1)**

- 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

27 **Binary Search (Page 2)**

- The binary search algorithm (*con.*)
  - 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

30 **Big O for a Binary Search (Page 1)**

- 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

31 **Big O for a Binary Search (Page 2)**

- Thus:
  - An array of 1,048,575 ($2^{20} – 1$) elements takes a maximum of 20 comparisons to find the key
  - An array of 1,073,741,824 (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

32 **Big O for a Binary Search (Page 3)**

- 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, represented as $\log_2 n$.

33 **Big O for a Binary Search (Page 4)**

- 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, also known as logarithmic run time

35 **The binarySearch() Method of Class Arrays (Page 1)**

- The binarySearch() method searches an array object for the specified value (search key) using the binary search algorithm
- Returns the index of the matching element, or returns -1 if the search fails
- The array must be sorted before making the call

36 **The binarySearch() Method of Class Arrays (Page 2)**

- Format:
  Arrays.binarySearch(*arrayObject*, *searchKey*)
- Example:
  int index = Arrays.binarySearch(values, lookup);

38 **Sorting Algorithms**

- Sorting data (e.g., placing the data into some particular *order*, ascending or descending) is one of the most important computing applications
- An important item to understand about sorting is that the sorted array will *be the same* no matter which algorithm you use to sort the array
- The choice of algorithm affects only the *run time* and *memory use* of the program

39 **Selection Sort**

- The selection sort is a simple, but inefficient, sorting algorithm
- Its first iteration selects the smallest element in the array and swaps it with the first element.
- The second iteration selects the second-smallest item (which is the smallest item of the remaining elements) and swaps it with the second element

40 **Selection Sort**

- The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last index, leaving the largest element in the last index
- After the i^{th} iteration, the smallest i items of the array will be sorted into increasing order in the first i elements of the array:
  - After the first iteration, the smallest element is in first position
  - After second iteration, the two smallest elements are in order in first two positions, etc.

41 **Selection Sort**

- After the first iteration, the smallest element is in first position; after second iteration, the two smallest elements are in order in first two positions, etc.
- The selection sort algorithm runs in $O(n^2)$ time.

45 **Insertion Sort**

- The insertion sort is another simple although inefficient, sorting algorithm
- The first iteration takes the second element in the array 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, so all three elements are in order

- At the *i*th iteration of this algorithm, the first *i* elements in the original array will be sorted

50 **Merge Sort** **(Page 1)**

- The merge sort is an efficient sorting algorithm that conceptually is more complex than selection sort and insertion sorts
- Sorts an array by splitting it into two equal-sized sub-arrays, sorting each sub-array, then merging them into one larger array

51 **Merge Sort** **(Page 2)**

- The implementation of the merge sort in this example is recursive
  - The base case is an array with one element (which of course is sorted already), so the merge sort immediately returns in this case
  - Recursion step splits array into two approximately equal pieces, recursively sorts them, then merges the two sorted arrays into one larger, sorted array