ECET11 – Object Oriented Programming LAB activity 4

Objective: Pointers, Recursive functions, 2D Arrays, String Manipulation

Source: Deitel & Deitel Edition 5 – Chapter 8 Exercises

1. (Quicksort) A recursive sorting technique is called Quicksort. The basic algorithm for a single-subscripted array of values is as follows:

   a. *Partitioning Step*: Take the first element of the unsorted array and determine its final location in the sorted array (i.e., all values to the left of the element in the array are less than the element, and all values to the right of the element in the array are greater than the element). We now have one element in its proper location and two unsorted subarrays.

   b. *Recursive Step*: Perform Step 1 on each unsorted subarray.

   Each time Step 1 is performed on a subarray, another element is placed in its final location of the sorted array, and two unsorted subarrays are created. When a subarray consists of one element, that subarray must be sorted; therefore, that element is in its final location.

   The basic algorithm seems simple enough, but how do we determine the final position of the first element of each subarray? As an example, consider the following set of values (the element in bold is the partitioning element it will be placed in its final location in the sorted array):

   37 2 6 4 89 8 10 12 68 45

   a. Starting from the rightmost element of the array, compare each element with 37 until an element less than **37** is found. Then swap **37** and that element. The first element less than **37** is **12**, so **37** and **12** are swapped. The values now reside in the array as follows:

12   2   6   4   89   8   10   37   68   45

Element *12* was just swapped with ***37***.

b. Starting from the left of the array, but beginning with the element after **12**, compare each element with **37** until an element greater than **37** is found. Then swap **37** and that element. The first element greater than **37** is **89**, so **37** and **89** are swapped. The values now reside in the array as follows:

12   2   6   4   37   8   10   89   68   45

c. Starting from the right, but beginning with the element before **89**, each element is compared with **37** until an element less than **37** is found. Then swap **37** and that element. The first element less than **37** is **10**, so **37** and **10** are swapped. The values now reside in the array as follows:

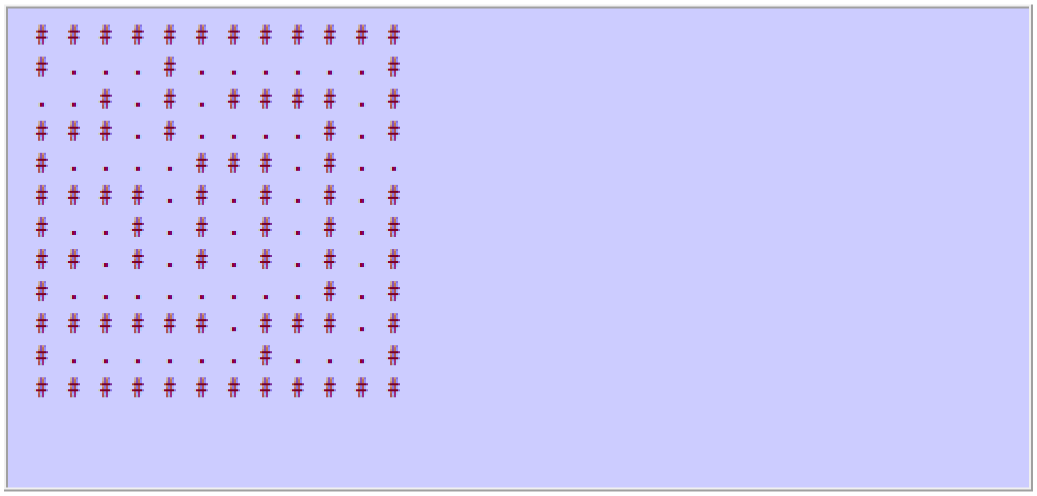
12   2   6   4   10   8   37   89   68   45

d. Starting from the left, but beginning with the element after 10, each element is compared with **37** until an element greater than **37** is found. Then swap **37** and that element. There are no more elements greater than **37**, so when we compare **37** with itself, we know that **37** has been placed in its final location of the sorted array.

Once the partition has been applied to the array, there are two unsorted subarrays. The subarray with values less than **37** contains **12**, **2**, **6**, **4**, **10** and **8**. The subarray with values greater than **37** contains **89**, **68** and **45**. The sort continues with both subarrays being partitioned in the same manner as the original array.

Based on the preceding discussion, write recursive function *quickSort* to sort a single-subscripted integer array. The function should receive as arguments an integer array, a starting subscript and an ending subscript. Function partition should be called by *quickSort* to perform the partitioning step.

2. (Maze Traversal) The grid of hashes (#) and dots (.) in the figure below is a two-dimensional array representation of a maze. In the two-dimensional array, the hashes represent the walls of the maze and the dots represent squares in the possible paths through the maze. Moves can be made only to a location in the array that contains a dot.



```
# # # # # # # # # # # #
# . . . # . . . . . . #
. . # . # . # # # # . #
# # # . # . . . . # . #
# . . . . # # # . # . .
# # # # . # . # . # . #
# . . # . # . # . # . #
# # . # . # . # . # . #
# . . . . . . . . # . #
# # # # # # . # # # . #
# . . . . . . # . . . #
# # # # # # # # # # # #
```

There is a simple algorithm for walking through a maze that guarantees finding the exit (assuming that there is an exit). If an exit is not provided, you will arrive at the starting location again. Place your right hand on the wall to your right and begin walking forward. Never remove your hand from the wall. If the maze turns to the right, you follow the wall to the right. As long as you do not remove your hand from the wall, eventually you will arrive at the exit of the maze. There may be a shorter path than the one you have taken, but you are guaranteed to get out of the maze if you follow the algorithm.

Write function *mazeTraverse* to walk through the maze. The function should receive arguments that include a 12-by-12 character array representing the maze and the starting location of the maze. As *mazeTraverse* attempts to locate the exit from the maze, it should place the character X in each square in the path. The function should display the maze after each move, so the user can watch the maze being solved.

3. (Check Protection) Computers are frequently employed in check-writing systems such as payroll and accounts-payable applications. Many strange stories circulate regarding weekly paychecks being printed (by mistake) for amounts in excess of $1 million. Weird amounts are printed by computerized check-writing systems, because of human error or machine failure. Systems designers build controls into their systems to prevent such erroneous checks from being issued.

Another serious problem is the intentional alteration of a check amount by someone who intends to cash a check fraudulently. To prevent a dollar amount from being altered, most computerized check-writing systems employ a technique called check protection.

One common security method requires that the check amount be both written in numbers and "spelled out" in words. Even if someone is able to alter the numerical amount of the check, it is extremely difficult to change the amount in words. Write a program that inputs a numeric check amount and writes the word equivalent of the amount. Your program only needs to handle check amounts up to $999.99.

For example, the amount 112.43 should be written as

ONE HUNDRED TWELVE and 43/100