**Sofia University**
**Department of Mathematics and Informatics**

**Course : OO Programming with C#.NET**
**Date: October 24, 2010**
**Student Name:**

# Lab No. 3b

**Submit the all C#.NET files developed to solve the problems listed below. Use comments and** Modified-Hungarian notation.

## Problem No. 1

**Дадени** са **52** карти с:

a) **сила** (faces) – **"Ace", "Deuce", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King"**

b) **цвят** (suit)- **"Hearts", "Diamonds", "Clubs" и " "Spades"**

**Модифицирайте** приложението от **Fig. 8.9- 10** (приложен към заданието с файл **SampleCodeFig8.rar** ) даден в ***лекция 4***, така че да **раздава** по **5 карти** при игра на покер.

**Модифицирайте също** class ***DeckOfCards*** от **Fig. 8.10** да включва **методи за определяне** дали **петте карти от едно раздаване** съдържат

a) чифт (2 карти с еднаква сила)

b) два чифта

c) терца (например, три Аса)

d) четири карти с еднаква сила (например, четири Аса)

e) пет карти от една боя

f) последователност (т.е., пет карти с поредни стойности на сила)

g) "пълна къща" (т.е., две карти от една боя и три карти от друга боя)

[***Препоръка***: *Добавете методи* ***getFace*** и ***getSuit*** към ***class Card*** даден на **Fig. 8.9**., а също и *метод* ***private int[] totalHand( Card hand[] )*** към ***class DeckOfCards*** който връща **масив от 13 елемента**, всеки които е броя на картите със съответна сила в дадено раздаване ***Card hand[]*** – например, първият елемент на този масив ще дава броя на асата в раздаването, десетият елемент ще е броят на десетките в в раздаването и т.н. Посредством ***totalHand()*** лесно може да се отговори на всеки от въпросите ***a- g*** ]

**Напишете конзолно приложение** за тестване н аметодите ( a – g)

## Задача 2

Да се напише клас **SortUtils**, който има следните методи :
а) **public static void InitArray(int[] a)**, който да инициализира едномерен масив със стойности въведени от клавиатурата;
б) **public static void PrintArray(int[] a)**, който да отпечатва на екрана стойностите на едномерен масив във вид на редица, където елементите са разделени със запетаи, а редицата в квадратни скоби;
в) **public static void SortArray(int[] a**), който да сортира едномерен масив по "метода на мехурчето";
д) **public static int[] MergeSort(int[] a, int[] b)**, който да слива два

предварително сортирани във възходящ ред масива и да връща получения при сливането масив, който също да е сортиран във възходящ ред

Напишете **конзолно приложение** за тестване на методите клас **`SortUtils`**

**Problem No. 4**

Computers are playing an increasing role in education. Write a **WPF application** that will help an elementary school student learn multiplication. Use the **next** method from an object of type **`Random`** to produce two positive one-digit integers. It should display a question, such as **How much is 6 times 7?** The student should then type the answer into a **`TextBox`**. Your **C#. NET** program should check the student's answer. If it is correct, display **"Very good!"** in a **Label**, then ask another multiplication question.

If the answer is incorrect, display **"No. Please try again."** in the same **Label**, then let the student try the same question again until the student finally gets it right. A separate method should be used to generate each new question. This method should be called once when the program begins execution and then each time the user answers a question correctly.

**Problem No.5**

Write a **WPF application** that simulates coin tossing. Let the program toss the coin each time the user presses the **Toss** button. Count the number of times each side of the coin appears. Display the results. The program should call a separate method **Flip**, which takes no arguments and returns **False** for tails and **True** for heads. [*Note*: If the program simulates the coin tossing realistically, each side of the coin should appear approximately half the time.]

**Problem No. 6**

*(Quicksor*t) We considered the **bubble sort** in one of the earlier problems. Consider now the recursive sorting technique called **Quicksort**. The basic algorithm for a one-dimensional 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, it must be sorted; therefore, that element is in its final location. The basic algorithm seems simple, but how do we determine the final position of the first element of each subarray? 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 to **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 new array is

***12*** 2 6 4 89 8 10 **37** 68 45

Element 12 is italicized+ bold to indicate that it was just swapped with **3**7.

b) Starting from the left of the array, but beginning with the element after 12, compare each element to **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 new array is

12 2 6 4 **37** 8 10 ***89*** 68 45

c) Starting from the right, but beginning with the element before 89, compare each element to **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 new array is

12 2 6 4 ***10*** 8 **37** 89 68 45

d) Starting from the left, but beginning with the element after 10, compare each element to **37** until an element greater than **37** is found, then swap **37** and that element. There are no more elements greater than **3**7, so when we compare **37** to itself, we know that **37** has been placed in its final location of the sorted array.

Once the partition has been applied to the above 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. The algorithm has the following recursive structure:

```
procedure quicksort(l, r: integer);

var i;
begin
if r>l then
begin
      i:=:partition(1, r)
      quicksort (1, i- 1) ;
      quicksort(i+l, r);
end
end
```

Using the preceding discussion, **write recursive** procedure **QuickSort** to sort a one-dimensional **Integer** array. The procedure should receive as arguments an **Integer** array, a starting index and an ending index. Procedure **Partition** should be called by **QuickSort** to perform the partitioning step**. Write also a WPF application** with appropriate GUI to test the **QuickSort with an array of integer numbers of an arbitrary.**

**Problem No. 7**

(**Selection Sort**) A selection sort searches an array looking for the smallest element in the array, then swaps that element with the first element of the array. The process is repeated for the sub array beginning with the second element. Each pass of the array places one element in its proper location. For an array of n- elements, **n- 1** passes must be made, and for each sub array, n- 1 comparisons must be made to find the smallest value. When the sub array being processed contains one element, the array is sorted. Write a **recursive** method selectionSort to perform this algorithm. **Write also a WPF application** with appropriate GUI to test the **Selection sort with an array of integer numbers of an arbitrary dimension (***allow the user to add the numbers from a single line Textbox to a multiline TextBox by clicking a button, then use another button to selectSort the numbers displayed in this Textbox and display them in an adjacent multiline Textbox***).**

**Problem No. 8**

(**Merge Sort**) A merge sort considers an array as split in two parts (of equal size, in particular), which are assumed to be sorted. In case the array consists of one element or it is empty, then the array is already sorted. Otherwise, to sort of the array it remains to **merge** the left and right part of the array, so that the whole array becomes sorted. The merge operation starts by comparing the left most elements of the sorted subarrays of the given array and writing these elements in a temporary array (of size equal to the sum of the sizes of both subarrays) in an increasing order. In case, there remain elements in one of the subarrays after all of the elements of the other subarray have been written to the temporary subarray, such elements are being appended to the end of the temporary array without changing their order. Write a **recursive** method **MergeSort** to perform this algorithm. It should have the form

```
void MergeSort(int [] a, int from, int to) {
        if (from <to) {
                int midde = (to + from )/2;
                MergeSort(a, from, mid);
                MergeSort(a, mid + 1,  to);
                Merge(a, from, to); // remains to be encoded
          }
}
```

**Write also a WPF application** with appropriate GUI to test the **Merge sort with an array of integer numbers of an arbitrary dimension.**