Practical Task 3

Due date: 09:00pm Friday, August 3, 2018

(Very strict, late submission is not allowed)

General Instructions

1. Read the general instructions, do not skip them.
2. Make sure that Microsoft Visual Studio 2015 is installed on your machine. Community version can be found on the SIT221 course web-page in CloudDeakin in Resources → Course materials → Visual Studio Community 2015 Installation Package. Your submission will be tested in this environment later by the markers.

   If you use Mac, you may install Visual Studio for Mac from https://e5.onthehub.com/WebStore/ProductsByMajorVersionList.aspx?ws=a8de52ee-a68b-e011-969d-0030487d8897&vsro=8, or use Xamarin Studio.

   If you are an on-campus student, this is your responsibility to check that your program is compilable and runs perfectly not just on your personal machine. You may always test your program in a classroom prior to submission to make sure that everything works properly in the MS Visual Studio 2015 environment. Markers generally have no obligations to work extra to make your program runnable; hence verify your code carefully before submission.
3. Read Chapter 7 of SIT221 Workbook available in CloudDeakin in Resources → Course materials → SIT221 Workbook. It starts with general explanation of algorithm complexity and describes Bubble Sort, Insertion Sort, and Selection Sort algorithms which you need to implement as part of this practical task. Investigate the suggested examples and follow the pseudocodes as you progress with coding of the algorithms. The chapter ends with the description of Quick Sort algorithm and you should peruse it. However, it offers the "median three" variant as a pivot point selection rule. Instead of it, in this task, you must apply a random selection policy and select a pivot point uniformly at random. This policy is briefly described in our course book. Furthermore, read chapter 8.1.2 of SIT221 Workbook to find out how to encode Binary Search for the last exercise of the practical task.
4. Read Chapter 12 of the SIT221 course book "Data structures and algorithms in Java" (2014) by M. Goodrich, R. Tamassia, and M. Goldwasser, in particular Section 12.2 about the Quick Sort algorithm. You may access the book on-line for free from the reading list application in CloudDeakin available in Resources → Course materials → Course Book: Data structures and algorithms in Java. Explore thoroughly section 12.2.2 on additional optimisation of Quick Sort and the attached pseudocode 12.6 as later your code must follow this scheme and implement the "in-place" version of the algorithm. As a pivot point selection rule, choose uniformly at random an element from a range of elements available in every recursive call of the method. Picking pivots at random is explained in 12.1.1. This should replace the existing line 6 of the pseudocode 12.6 with a randomized selection function (consider Random class of the .NET Framework).

5. You may need to read Chapter 11 of SIT232 Workbook available in Resources → Course materials → SIT232 Workbook to remind / study Generics and their application in object-oriented programming. You may also need to read Chapter 6 of SIT232 Workbook about Polymorphism and Interfaces as you need excellent understanding of these topics to progress well through the practical tasks of the course. Make sure you are proficient with them as they form a basis to design and develop programming modules in this and all the subsequent tasks. You may find other important topics required to complete the task, like exceptions handling, in other chapters of the workbook.
6. Download the zip file called "Practical Task 3 (template)" attached to this task. It contains a template for the program that you need to work on. Open the solution file in Microsoft Visual Studio. There are two connected projects inside: DataStructures_Algorithms and Runner. The former project has now the official working solution built-in for the vector class you developed in Practical Task 1 (Vector.cs file in Week01 subfolder). You will need to further modify the vector class according to the specification. You are allowed to change it as you want unless you meet all the requirements in terms of functionality and signatures of requested methods and properties.

   There are currently no files in the Week03 subfolder and the objective of the practical task is to add several new classes and interfaces. There are no constraints on how to design and implement the new modules internally, but their exterior must meet all the requirements of the task in terms of functionality accessible by a user.

   The second project has Runner03_Task1.cs, which is new. It implements IRunner interface and acts as a main method in case of this practical task by means of the Run(string[] args) method. Make sure that you set the right runner class in Program.cs. You may modify this class as you need as its main purpose is to test the implementation of the vector class and sorting operations. However, it has already a good structure to check the sorting algorithms that you are required to realize. Therefore, you should first explore the content of the method and see whether it satisfies your goals in terms of testing. Some lines are commented out to make the current version compilable, so make sure you make them active as you progress with different algorithms. Furthermore, some comments must be substituted by calls of relevant sorting algorithms on the vector type data structure. In general, you should use this class and its method to thoroughly test the vector and related classes providing sorting operations. Your aim here is to cover all potential logical issues and runtime errors. This (testing) part of the task is as important as writing the code. We will later on replace your runner by our special versions in order to test functionality of your code for common mistakes.

Objectives

1. Study implementation of common sorting and searching algorithms

Specification

Main task

Task 1. Designing aggregation relationship between the data structure and an object performing sorting operations on it

To start this task, you first need to figure out the way how particular sorting algorithms represented via separated classes can relate to the vector class you have completed in the Practical Task 1. The idea is to allow an instance of the vector class to switch between possible algorithms easily when a user wants to change the sorting technique in use. An easy solution to organise this is to create an interface further implemented by all the classes providing sorting algorithms, then allocate a private variable in the vector class to refer to one of these classes. The later reference implements so-called aggregation in terms of object-oriented programming design, where the selected algorithm (implemented as a class) becomes "a part of" the vector class. To remind yourself about class relations, read Chapter 4 of the SIT232 Workbook (page 99). Thus, your first step is to build a new interface called "ISorter" in the Week03 subfolder, which implies the following generic method.

void Sort<T>(T[] array, IComparer<T> comparer)	Sorts the elements in the entire one-dimensional array of generic type T using the specified comparer. If comparer is provided, the elements of the Vector <T> are sorted using the specified IComparer<T> implementation. Otherwise, if null, the default comparer Comparer<T>.Default is applied.

As we want to protect the program from unexpected runtime exceptions, we enforce here that type T must implement the ***IComparable<T>*** interface. Thus, its future instances must be comparable and can be handled by the ***Sort*** method. This requires to impose the following constraint on class T in the declaration part of the ***Sort*** method:

```
void Sort<T>(T[] array, IComparer<T> comparer) where T: IComparable<T>;
```

When the interface is ready, add "Sorter", a new property of type ISorter, to the vector class and make it public accessible. One may read and write to this property; that is, to get a reference to the object serving as a sorter for an instance of the vector class or to set a new sorter instead of the existing one.

The next step is to link the existing ***Sort()*** and ***Sort(IComparer<T> comparer)*** methods to ***Sorter*** variable. Implementation of the both methods were expected as part of the Practical Task 2. However, this time, we make them working a bit different due to the link to ***Sorter***. It is easy to build this connection as you only need to call

- ***Sorter.Sort(data, null)*** for ***Sort()*** and
- ***Sorter.Sort(data, comparer)*** for ***Sort(IComparer<T> comparer)***

methods, where ***data*** represents the array of type T internal to the vector class.

Now, have a look at (and uncomment) the ***DefaultSorter*** internal class, which is a part of the vector class since Practical Task 3. ***DefaultSorter*** implements ***ISorter*** interface. Specifically, this class delegates calls for sorting to the ***Array.Sort()*** method and does the same work which the Practical Task 2 asked you to do in its part one. This is now just a way around.

The only last thing to do is to connect the ***Sorter*** variable to the ***DefaultSorter*** object in the both constructors of ***Vector***. This is trivial and needs a single line of code only:

```
Sorter = new DefaultSorter();
```

By now, you should be able to use ***Sort*** and ***Sort(IComparer<T> comparer)*** methods of the vector class. Move to the ***Run()*** method of the ***Runner03_Task1*** class and find the section entitled "Default Sort". Remove the first commented line in that section, and try to sort integer values stored in ***vector*** variable in ascending order.

Task 2. Implementation of Comparer objects for integer data types

Similarly to the part three of Task 2, you are asked here to create two new classes called "IntAscendingComparer" and "IntDescendingComparer", respectively, in Week03 subfolder. They both must implement IComparer<int> interface. Check the following link:

https://msdn.microsoft.com/en-us/library/8ehhxeaf(v=vs.110).aspx

to remind how it works and which method must be defined for the both classes. The former class must provide comparison mechanism to sort integers in ascending order. The later proceeds similarly, but will be used to sort integers in descending order.

Task 3. Implementation of the Bubble Sort, Insertion Sort sorting, Selection Sort, and Randomized Quick algorithms

In Week03 folder, add new classes called "BubbleSort", "InsertionSort", "SelectionSort", and "RandomizedQuickSort", respectively. All the classes must implement the ***ISorter*** interface and provide particular solution for the inherited ***Sort<T>(…)*** method of the interface. They must have only default constructors. You may add any extra private methods and attributes if necessary. Again, note that ***RandomizedQuickSort*** must realize the random pivot point selection rule and do sorting operations in-place as stated in point 4 of the general instructions.

Complete all these algorithms and check their work in the ***Runner03_Task1*** runner. Confirm correctness of the algorithms by replacing comments in the corresponding sections of the **Run()** method with relevant **Sort()** method invocations. To make a particular sorting algorithm active in the scope of the ***vector*** object, you must uncomment the lines assigning an object implementing one of the algorithms to the ***Sorter*** property of the ***vector***, for example

```
vector.Sorter = new BubbleSort();
```

This command actually implies the aggregation mechanism you constructed in part 1 of this practical task.

Run computational experiments on various datasets by managing their sizes via the value of ***problem_size*** variable positioned in the beginning of the ***Run()*** method. Compare the running times and make conclusions about performance of the algorithms.

Task 4. Implementation of Binary Search algorithm

In this task, you must further extend the vector class to support searching operations for an element in a sequence of generic data type. You must add the following functionality for the class:

int BinarySearch(T item)	Searches the entire sorted Vector<T> for an element using the default comparer and returns the zero-based index of the element. This method uses the default comparer Comparer<T>. Default for type T to determine the order of list elements.
int BinarySearch(T item, IComparer<T> comparer)	Searches the entire sorted Vector<T> for an element using the specified comparer and returns the zero-based index of the element, if item is found; otherwise, a negative number.

Note that you are not allowed to delegate the binary search operation to ***Array*** class or any other collection classes. You must implement the algorithm from scratch. To learn the similarity with the standard ***List<T>*** class of .NET Framework, have a look at the material published on

https://msdn.microsoft.com/en-us/library/w4e7fxsh(v=vs.110).aspx

Especially, focus on remarks and complexity issues of this ***BinarySearch()*** method. Before you start searching, make sure that your data is sorted in ascending order according to the utilised comparer. Finally, alter the ***Run()*** method of the ***Runner03_Task1*** class to test correctness of your solution.

Submission instructions

Submit your program code as an answer to the main task via the CloudDeakin System by the deadline. You should zip your Vector.cs, ISorter.cs, IntAscendingComparer.cs, IntDescendingComparer.cs, BubbleSort.cs, InsertionSort.cs, SelectionSort.cs, and RandomizedQuickSort.cs files only and name the zip file as PracticalTask3.zip. Please, make sure the file name is correct. You must not submit the whole MS Visual Studio project / solution files.

Rank the sorting algorithms in ascending order of their speed based on the results of computational experiments you performed in task 3. Do you think this order is correct with regard to the theoretical insights you got during our lectures? Enter your answers into the submission textbox in the CloudDeakin system.

Marking scheme

You will get 1 mark for outstanding work, 0.5 mark only for reasonable effort, and zero for unsatisfactory work or any compilation errors. Note that the exercises from the section of **additional tasks will be marked**, too.