Practical Task 2

Due date: 09:00pm Friday, July 27, 2018

(Very strict, late submission is not allowed)

General Instructions

1. 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.

2. Read Chapter 1.7 of SIT221 Workbook available in CloudDeakin in Resources → Course materials → SIT221 Workbook. It provides a short discussion on IComparable<T> interface and implementation of collections.

3. 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.

4. Download the zip file called “Practical Task 2 (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 the default Vector.cs file in Week01 subfolder. You need to replace it with your version of the vector class, which you should have completed by now. If you struggle with implementation of this class and are not able to finish the Practical Task 1, then official working solution will be available right after the submission deadline for Task 1; that is, at 10:00pm Friday, July 20, 2018. You may then import that solution and continue your work on Practical Task 2. However, in this case, you have to wait till the release date and have only one week ahead to complete Practical Task 2. So, do your best to complete Task 1 as soon as possible to potentially get full scores and start the next task earlier.

You will need to extend 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. In the Week02 subfolder, you will find two more class files required to complete Task 2. Again, you are free to change them to meet all the requirements of the task.

The second project has two new files: Runner02_Task1.cs and Runner02_Task1.cs. They both implement IRunner interface and can act as main methods 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 to switch between different logic described in the specification below. You are allowed to modify the both classes as you need as their main purpose is to test the implementation of the vector class and sorting operations. You should use these classes and their methods to thoroughly test the vector and related classes aiming on covering 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 runners by our special versions in order to test functionality of your code for common mistakes.

In addition, the Runner project has the folder called “Data”, which contains test data for the first practical task. You must use it again. This is a dataset of three files: 1H.txt, 1T.txt, and 10T.txt with 100, 1000 and, 10,000 integer values, respectively. The Runner reads the data via the argument of the command line. You may change parameter of the command line when debugging in MS Visual Studio 2015 following the order Project → DataStructures_Algorithms Properties → Debug → Start Options → Command line arguments. You then need to replace the line by a new one, for example “..\..\Data\Week01\1T.txt”. This line sets the path to a data file.

5. Your task this time is to implement several interfaces in order to allow your vector class to sort collections of generic data types. Read carefully about each interface and how it is involved in the process of data sorting. This task does not ask you to write lots of code, but rather allows you to understand the work of the interfaces required to connect your code to generic collections and methods. This knowledge is crucial for you as a programmer.

Objectives

1. Study functionality and interfaces required to support sorting operations in .NET Framework
2. Understand the work of IComparer<T> and IComparable<T> interfaces and difference between them

Specification

Main task

Task 1. Extend the vector class to support default sorting

In this task, you are asked to extend the vector class implemented within the Practical Task 1 to support sorting operations for elements in a sequence of generic data type. You must add the following functionality for the class:

void Sort()	Sorts the elements in the entire Vector<T> using the default comparer. This method uses the Array.Sort method, which applies the introspective sort.
void Sort(IComparer<T> comparer)	Sorts the elements in the entire Vector<T> using the specified comparer. If comparer is provided, the elements of the Vector <T> are sorted using the specified IComparer<T> implementation. This method uses the Array.Sort method, which applies the introspective sort.

Note that this time you do not need to implement any searching algorithms and may simply delegate calls to the corresponding methods of the ***Array*** class of the .NET Framework. Therefore, you need to apply either the default ***Sort*** method from the ***Array*** class to your internal data structure, or that one of the ***Array*** class that expects a comparer object as an argument. Read more about the standard .NET Framework array class methods following the link

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

Again, you may compare your solution with functionality of the standard List<T> class. Check the remarks for the List<T> class

https://msdn.microsoft.com/en-us/library/b0zbh7b6(v=vs.110).aspx

and note the following:

- Methods you delegate to use either the default comparer ***Comparer<T>.Default*** for type T to determine the order of elements, or a specified ***IComparer<T>*** implementation.
- For each of the methods, the default comparer ***Comparer<T>.Default*** checks whether type T implements the ***IComparable<T>*** generic interface and uses that implementation, if available. If not, ***Comparer<T>.Default*** checks whether type T implements the ***IComparable*** interface. If type T does not implement either interface, ***Comparer<T>.Default*** throws an ***InvalidOperationException***. For ***Sort(IComparer<T> comparer)***, this check is performed if the provided input comparer argument for Sort method is null.

Make sure you understand how the introspective sort of the ***Array.Sort*** method performs. Specifically, it acts as follows:

- If the partition size is fewer than 16 elements, it uses an insertion sort algorithm.
- If the number of partitions exceeds 2 * LogN, where N is the range of the input array, it uses a Heapsort algorithm.
- Otherwise, it uses a Quicksort algorithm.

This implementation performs an ***unstable*** sort; that is, if two elements are equal, their order might not be preserved. In contrast, a ***stable*** sort preserves the order of elements that are equal. On average, this method is an O(*n* log *n*) operation, where *n* is Count; in the worst case it is an O(n^2) operation.

Now, use the Runner02_Task1 class to load (already implemented in the given code) the data from 1H.txt (a file of 100 elements). Alter the class to correctly sort the data and record the sorted data to "..\\..\\Data\\Week01\\1Hsorted.txt". Furthermore, add a path to the output file (i.e. "S_1H.txt") as another input argument to your solution in the same way as you did it for the input argument.

HINT: Before applying the sorting methods of the array class to the internal data structure in the vector class, think what the scope of elements it is applied to is and how you can manage it. Otherwise, it may lead your sorting functions to undesired results.

Task 2. Implement IComparable<T> interface to sort objects

Have a look at the ***Part*** class in the Week02 directory of the project. Your task is to modify this class so that the vector class can run sorting operations on a set of objects of this type. As the first step, enforce the Main method of the program to create and run an instance of the Runner02_Task2 class. Then modify the Runner02_Task2 class so that it creates a vector of Part class objects and sorts them using the methods of the vector class. Finally, record the data to a file. You will likely get a compilation error as the result. Can you explain the nature of this error?

Note that this error occurs at the compilation stage just because you impose the "***where T : IComparable<T>***" constraint on the ***Vector<T>*** class. Otherwise, a runtime ***InvalidOperationException*** is to be generated as discussed in the first part of the task. So, think about these two facts and about the outcome you would generally prefer: compilation error versus runtime error (exception).

Now, modify the Part class to implement ***IComparable<Part>*** interface. This will require realization of a specific method which is a part of the interface. Read the document about this interface following the link

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

and make sure you understand the remarks. After extension, the ***Part*** class should support sorting of its objects in ascending order according to the ***PartId*** attribute. Complete the work and try to run your project again.

Task 3. Implement IComparer<T> interface for a class to sort objects

In this task, you must create a new class called "PartComparer" in Week02 subfolder. This class must implement ***IComparer<Part>*** interface, which you may learn from the following link:

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

Similar to the ***IComparable<Part>*** interface, this interface requires a method to be implemented by you. This method compares two objects and returns a value indicating whether one is less than, equal to, or greater than the other. See

https://msdn.microsoft.com/en-us/library/xh5ks3b3(v=vs.110).aspx

for details. Compare the required methods in both interfaces. Finally, complete your code for Task 3 and check the correctness of the sorting operation, i.e. that elements appear in the descending order of their PartIds.

Additional Tasks

1. Answer whether the following statements are right or wrong.
   - $n^2 + 10^6 n = O(n^2)$
   - $n \log n = O(n)$
   - $n \log n = \Omega(n)$
   - $\log n = o(n)$
2. Clearly explain your decision in a couple of sentences.
   - Is it true that if $f(n) = \theta(g(n))$ and $g(n) = \theta(h(n))$, then $h(n) = \theta(f(n))$?
   - Is it true that if $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $h(n) = \Omega(f(n))$?
3. Is it true that $\theta(n^2)$ algorithm always takes longer to run than an $\theta(\log n)$ algorithm? Provide a short explanation.
4. For each pair of functions given below, point out the asymptotic relationships that apply: $f = O(g)$, $f = \theta(g)$, and $f = \Omega(g)$.
   - $f(n) = \sqrt{n}$ and $g(n) = log\, n$
   - $f(n) = 1$ and $g(n) = 2$
   - $f(n) = 1000 \cdot 2^n$ and $g(n) = 3^n$
   - $f(n) = 4^{n+4}$ and $g(n) = 2^{2n+2}$
   - $f(n) = 5n \cdot log\, n$ and $g(n) = n \cdot log\, 5n$
   - $f(n) = n!$ and $g(n) = (n+1)!$

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, Part.cs and PartComparer.cs files only and name the zip file as PracticalTask2.zip. Please, **make sure the file name is correct**. You must not submit the whole MS Visual Studio project / solution files. You **must answer all additional questions**

and write down your solutions as a text document. Submit the document (PDF version is preferred) via the CloudDeakin system along with the zip file.

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.