Programming Project 1

Due date: 09:00pm Friday, August 24, 2018

General Instructions

1. **Read** and **follow** the general instructions.
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. Download the zip file called "Programming Project 1 (template)" attached to this task. It contains a template for the project 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 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 extend the vector class according to the specification in part 4.1. However, it does not have those changes that are to be finished as part of Practical task 3, mainly with regard to its part 1. Therefore, you have two options: either to complete the Practical Task 3 as soon as possible to start your work on this assignment, or, if you struggle with extension of Vector class and are not able to finish the Practical Task 3, wait till the release date of the official solutions, which is 10:00pm Friday, August 3, 2018. You may then import official results of Practical Task 3 and continue your work on the project. However, in this case, you have to wait till the release date and will get less time ahead to complete this task. So, do your best to complete Task 3 quickly. Alternatively, you may first start with the parts 5.1 and 5.2, which are challenging but independent of the weekly practical tasks you do. In this case, you may safely wait till the release of our solutions to get feedback and verify your answers first.
4. Through these tasks, you will have to read some chapters of the SIT221 course book "Data structures and algorithms in Java" (2014) by M. Goodrich, R. Tamassia, and M. Goldwasser. If you do not have your own book, 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.

5. You may also 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.

Objectives

1. Study implementation of recursive sorting and selection algorithms

2. Compare the most common sorting techniques

3. Get important problem solving skills

4. Train in explanation of algorithmic concepts

Specification

Task 1. Implementation of the Merge Sort algorithm

Part 1.1 Read Section 12.1 of the SIT221 course book "Data structures and algorithms in Java" (2014) by M. Goodrich, R. Tamassia, and M. Goldwasser on the Merge Sort algorithm. This material repeats what you have already learned from the lectures in weeks 2 and 3. Make sure that you understand algorithmic concept of this method with regard to its recursive nature. Use the general description of Merge Sort in 12.1.1 and the code fragments 12.1 and 12.2 to implement its array-based top-down recursive version. The corresponding sorting function must be encapsulated within a new class called "MergeSortTopDown" and implement the ***ISorter*** interface (explained in the Practical Task 3), thus introduce specific implementation for the inherited

```
void Sort<T>(T[] array, IComparer<T> comparer)
```

method of the interface. It must have a default constructor only. You are allowed to add any extra private methods and attributes if necessary. The new class should be placed into the Project01 subfolder of the attached .NET framework project.

Part 1.2 As an additional task to the previous part, show by the Master Theorem that the recursive Merge Sort has $\theta(n \log n)$ worst-case runtime complexity. Present your deduction in the report.

Part 1.3 Now, proceed with Section 12.1.5 and investigate the bottom-up version of Merge Sort. Make sure that the issue related to temporary memory usage, which makes two versions different not only from implementation point of view, but also from efficiency prospective, is clear for you. Similarly, in folder Project01, create a new class named "MergeSortBottomUp" implementing ***ISorter*** and encode this algorithm.

Part 1.4 Complete the both versions of the algorithm and check their work in the **RunnerPr01_01** runner. Proceed similarly to the Practical Task 3 to make your algorithms active. Specifically, alter the **Run()** method to ensure that an algorithm is assigned to the ***Sorter*** property of object of ***Vector*** class (completed earlier within your practical tasks), for example

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

Run computational experiments on various datasets by managing their dimensions 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 2. Additional Optimization for Quick Sort

Part 2.1 In Practical Task 3, you were asked to implement the randomized Quick Sort algorithm performing sorting operations in-place, i.e. using the same input data structure without allocating auxiliary memory to store intermediate results for the sorting process. To succeed in this task, read Section 12.2.2 of the course book again. It concludes with the discussion on memory efficiency. Particularly, it states that the Quick Sort algorithm needs space for a program stack proportional to the depth of the recursion tree despite the proposed implementation, which divides the sequence of values into two parts in-place in each step. Your task is to do a small research on this issue and write a short essay (400-600 words in total) focusing on the following questions:

1) Why do you think that, in the worst case, the size of the program stack affected by the recursive algorithm might take up to $n-1$ elements?
2) Why is the stack depth of $O(\log n)$ desired and expected?
3) How should the randomized Quick Sort algorithm you have made be modified to guarantee the stack size of $O(\log n)$?

Answering these questions, you should not just adopt the text from the book, but must find relevant material (it is better to rely on several different sources) providing solution to this problem. When use some references in your essay, do not forget to mention them and cite. Alternatively, you may propose your own solution if you believe it is to be correct and matching the requirements. In addition, prepare a pseudocode of the Quick Sort algorithm satisfying the $O(\log n)$ memory space complexity for the stack and present it along with your essay.

Finally, implement your solution as a new class with name "QuickSortOptimized". The class must implement the ***ISorter*** interface (explained in the Practical Task 3), thus introduce specific implementation for the inherited

```
void Sort<T>(T[] array, IComparer<T> comparer)
```

method of the interface. Again, it must have a default constructor only. You may add any extra private methods and attributes if necessary. The new class should be placed in folder Project01 of the attached .NET framework project.

Refine the algorithm and check its work in the **RunnerPr01_01** runner. Do the same operations as for MergeSortTopDown and MergeSortBottomUp classes' objects in section 1.4 of this project.

Task 3. Comparing Sorting Algorithms

Part 3.1 Read Chapter 12.4 and consider all the algorithms to sort an n-element sequence that you have studied so far in this project and in Practical Task 3. Write a short essay (600-900 words in total) summarizing their pros and cons. Your answer should be concise and explain the most important facts related to their use in real practice. You must focus on the most significant facts and keep your answer detailed, therefore rather than use general words, describe the most impactful features that form the trade-off in choice of particular sorting technique for certain data. Obviously, you need to discuss runtime and space complexity issues, but also mention other important features like stability, in-place processing, possible application for dynamic sorting and etc.

Be careful, **do not copy** answers directly from the book or any other resources. You should use this material, but discuss this topic entirely in your own words. If you copy from somewhere, it might be identified as **plagiarism** that normally leads to zero mark for the whole assignment. We generally expect your answer to be a bit beyond of the discussion provided in Chapter 12.4, mainly focusing on extra important features of the algorithms and practical applications. Therefore, do a proper research work and make sure you read several different sources to support your discussion. In your text, do not put all ideas into one bucket. Talk about all the algorithms sequentially making references to other algorithms mainly to compare with them.

Task 4. Implementation of the Randomized Quick-Select algorithm

Part 4.1 As the Chapter 12.5. of the course book states, sorting is not the only interesting problem dealing with a total order relation on a set of elements. There are a number of applications in which we are interested in identifying a single element in terms of its rank relative to the sorted order of the entire set. One of the examples occurred in the general order-statistic problem is selection of the k^{th} smallest element from an unsorted collection of n comparable elements. While this problem can be solved by sorting the collection and then indexing into the sorted sequence at index $k-1$, this approach would take $O(n \log n)$ time with the best comparison-based sorting algorithms. Clearly, this guarantees bad performance for the cases when $k=1$ or $k=n$ because we can easily solve the selection problem for these values of k in $O(n)$ time with the simple linear search algorithm. That is why we are interested in an algorithm that can achieve an $O(n)$ running time for arbitrary values of k.

Read thoroughly the chapter and its Section 12.5.2 in particular. Your task is to further extend the vector class to support a new method to select the k^{th} smallest element in a sequence of generic data type with the following signature:

T Min(int k);	Searches for the kth smallest element in the entire unsorted Vector<T> using the default Comparer<T>.Default comparer and returns the element. If the source sequence is empty, it returns value which is default for the type T.

Finish your code for the algorithm and check its work in the **RunnerPr01_01** runner on arbitrary data.

Task 5. Problem solving skills

Part 5.1 Assume you are given sorted lists of customer information. Each customer has an age, a weight, and a name. Your task is to determine the basis on which they were sorted. There is an assumption that the customers were sorted based on alphabetical order of names, ascending order of ages, and descending order of weights. One of these fields was the primary field used to initially sort the customers. Then one of these fields was used as the secondary field to break ties between customers that were indistinguishable based on the primary field. And finally the remaining field was used to break ties between customers that had the same values for their primary and secondary fields.

First of all, create a class named "SortingOrder" in Project01 subfolder of the project. It must have a public function

```
string Solve(string[] name, int[] age, int[] weight)
```

which, when given a set of names, ages, and weights as arrays of size n, returns a string that tells what sort was used. The result of the method must have one of the following values:

– NAW	name was the primary, age secondary, weight the final field
– NWA	name was the primary, weight secondary, age final
– ANW	age was primary, name was secondary, weight final
– AWN	age was primary, weight was secondary, name final
– WAN	weight primary, age secondary, name final
– WNA	weight primary, name secondary, age final
– IND	indeterminate: more than one of the above is possible
– NOT	none of the above sorting methods was used

Note that the tuple (***name[i]***, ***age[i]***, ***weight[i]***) gives the attributes of customer i. Thus, the first elements of ***name***, ***age***, and ***weight*** correspond to the first customer in the sorted list of customers, the second elements to the second customer, etc. The general constraints imposed on the data are as follows:

- ***name*** will contain between 1 and 50 elements inclusive
- ***age*** and ***weight*** will contain the same number of elements as ***name***
- each element in ***name*** will contain between 1 and 50 characters inclusive
- each character in each element of ***name*** will contain only uppercase letters 'A'-'Z'
- each element of age and weight will be between 1 and 300 inclusive

Consider the following examples:

4) For arrays ["BOB", "BOB", "DAVE", "JOE"], [22, 35, 35, 30], and [122, 122, 195, 200] the solution is "IND". The ages are not in ascending order and the weights are not in descending order so the primary field must be name. The tie between the 2 BOB's could have been broken by increasing age, leaving the weight field as final. But it is also possible that the weight field was secondary, leaving the 2 BOB tie to be resolved by age. So we cannot determine which of those two sorts was used.
5) For arrays ["BOB", "BOB", "DAVE", "DAVE"], [22, 35, 35, 30], and [122, 122, 195, 200] the method should return "NOT". The ages are not in ascending order and the weights are not in descending order so the primary field must be name. There is a tie between the 2 BOB's and between the 2 DAVE's. If the secondary field were age, then the DAVE's would have been placed in the other order. That is also true if weight were the secondary field. So none of the sorts could have been used.
6) For arrays ["BOB", "BOB", "DAVE", "DAVE"], [22, 35, 35, 30], and [122, 122, 195, 190] the expected result is "NWA". The ages are not in ascending order and the weights are not in descending order so the primary field must be name. Weight as secondary field properly orders the 2 DAVE's and leaves the ordering of the 2 BOB's up to the final field.

Second, check the Project01 subfolder in Data directory of the Runner project. The class ***SortingOrder_Generator*** there was written for you to provide a benchmark suite of test instances. Its static ***Count()*** method returns the total number of the instances it may create. The public static method

```
string Generate(int k, out string[] a1, out int[] a2, out int[] a3)
```

of this class can create instance k (this value is taken as an input argument) and return an array of names as ***a1*** parameter, an array of ages as ***a2***, and an array of weights as ***a3***. Its return value provides string containing the correct answer for this instance k. Therefore, you may extract data calling the ***Generate(…)*** method on the class ***SortingOrder_Generator*** as it is already done in ***RunnerPr01_02*** class.

Finally, develop an algorithm to solve the problem and make sure you pass all the tests in reasonable time. There is no particular guideline and you are allowed to apply any algorithmic technique that you find relevant. The same is valid in terms of coding; that is, you may implement your algorithm as you wish and use any private methods and attributes unless the prescribed ***Solve(…)*** method returns correct answer to the problem.

HINT: To solve this problem, think how sequential sorting operations can help you. It is important to read Section 12.3 of the course book, especially about the Radix Sort. You may not use the Radix Sort itself, but you should focus on some very important features it has and on the discussion about it. Note that the Radix Sort is not only the algorithm that has those features, so you may think how other techniques can be applied. It is possible to solve the problem in no more than ~40 lines of code.

Part 5.2 A *Deque* is a data structure which supports $O(1)$ time insertion and deletion operations at both the front and back ends. In this task, you are given an array of integers. A set of deques may help with sorting these numbers according to the following policy. You must process each number in a data array in the order they are given. For each number x in the array, you may only do exactly one of the following:

- 1. Push x onto the front end of an existing deque.
- 2. Push x onto the back end of an existing deque.
- 3. Create a new deque with x as its only element.

It is not permissible to skip a number temporarily and process it at a later time. It is also not permissible to insert a number into the middle of an existing deque; only front and back insertions are allowed. To make things easier, data will not contain duplicate elements. Once all the numbers have been processed, if you have built the deques wisely, you should be able to create a single, sorted list by placing the resulting deques on top of each other in an order of your choice.

Create a new class named “MinDequesNumber” in Project01 subfolder of the project. It must have a public function

```
int Solve(int[] data)
```

which accepts data as an array of integer numbers. You must implement an algorithm within this method which returns the least number of deque data structures required for this sorting to be possible. The general constraints imposed on the data are as follows.

- ***data*** will contain between 3 and 50 elements, inclusive.
- Each element of ***data*** will be between -1000 and 1000, inclusive.
- ***data*** will not contain duplicate elements.

Consider the following examples:

1) For array [50, 45, 55, 60, 65, 40, 70, 35, 30, 75] the algorithm must return 1. Only one deque is necessary to sort this list. The first element, 50, starts the deque. For each successive element encountered, if that element is less than 50, push it onto the front of the deque. Otherwise, if it is greater than 50, push it onto the back of the deque. Once all the data has been processed, the deque will contain all the elements and be in sorted order.
2) For array [3, 6, 0, 9, 5, 4] the expected answer is 2. Your algorithm could process the numbers in the following way:
   - Create a new deque d1 = {3}.
   - Create a new deque d2 = {6}.
   - Push 0 onto the front of d1; d1 = {0, 3}
   - Push 9 onto the back of d2; d2 = {6, 9}
   - Push 5 onto the front of d2; d2 = {5, 6, 9}
   - Push 4 onto the front of d2; d2 = {4, 5, 6, 9}

   We can now combine the deques together by putting d2 after d1, resulting in the sorted list {0, 3, 4, 5, 6, 9}. Two deques were used (and it is impossible to succeed using any less than two), so the method returns 2.
3) For array [0, 2, 1, 4, 3, 6, 5, 8, 7, 9] the result must be 5. The five deques will be {0,1}, {2,3}, {4,5}, {6,7}, and {8,9}. It is impossible to use fewer than five deques and still be able to combine them into a single, sorted list at the end.

Inspect ***MinDequesNumber_Generator*** class placed in the Project01 subfolder in Data directory of the Runner project, which acts as a generator of test instances for this problem.

Similarly to the previous task, static ***Count()*** method returns the total number of the instances it may create. The public static method

```
int Generate(int k, out int[] a1)
```

gives instance k (integer k is accepted as an input argument) and returns an array of integers associated with k as the second argument. Its return integer-type value is the correct answer for the instance k. Therefore, you may extract data calling the ***Generate(…)*** method of the class name ***MinDequesNumber_Generator*** as it is already done in ***RunnerPr01_03*** class.

Develop an algorithm able to solve the problem and make sure you pass all the tests in reasonable time. There is no particular guideline and you may use any algorithmic technique which you find applicable to the problem. The implementation part of the algorithm is also up to you. You must only return the correct answer to each of the instances through calls to the ***Solve(…)*** method.

HINT: To solve this problem, think how a sorting algorithm, for example the Bubble Sort, can be applied to determine the least number of required deque data structures. It is possible to solve the problem in no more than ~20 lines of code.

Submission instructions

Submit your program code as answers to the parts 1.1, 1.3, 2.1, 4.1, 5.1, and 5.2 via the CloudDeakin System by the deadline. The normal Deakin University policy is applied for late submissions. You **must zip** your MergeSortTopDown.cs, MergeSortBottomUp.cs, QuickSortOptimized.cs, Vector.cs, SortingOrder.cs, and MinDequesNumber.cs files only and name the zip file as Project1.zip. Please, make sure the file names are correct. You must not submit the whole MS Visual Studio project / solution files.

Submit solutions to the parts 1.2, 2.1, 3.1 in the form of essay in a Microsoft Word document and name the file as Project1.docx. **Do not put it inside the Project1.zip and do not archive it.**

This is your responsibility to keep file names correct as some of them will be processed automatically by scripts. If they are in wrong format, the scripts will not mark them and you may not get marks.

Marking scheme

You may get the following scores for this project:

- In parts 1.1 and 1.3, you will get up to 8 scores in total (either 4 scores or 0 for each algorithm) if your MergeSortTopDown and MergeSortBottomUp **run properly and sort generic data correctly**. You will get 0 scores for an algorithm if it **does not work**, **causes compilation or runtime errors, has no code commenting** (markers will use this to verify your algorithm), or **does not sort generic data as prescribed**.
- In part 1.2, you will get either 2 scores for the right solution presented in your text report, or 0 scores if the **solution occurs to be incorrect**.

- In part 2.1, you will get either 4 scores if your QuickSortOptimized **runs properly and sorts generic data correctly according to the idea presented in your essay**, or 0 scores if it **does not work**, **causes compilation or runtime errors**, **has no code commenting** (markers will use this to verify your algorithm), or **does not sort generic data as prescribed**.
- You may get up to 7 scores for your essay as part of 2.1. Specifically, 2 scores can be granted for a correct pseudocode and 5 scores for an outstanding essay answering all the questions **exactly and explaining the nature of the issue with great details**. This should be a precise answer to the question. You will get 3 scores if ideas are **generally clear, but lack of details**. This normally happens when markers have to guess whether you really understand the matter and how to resolve it. So, your task is to convince the markers that you got exceptional knowledge in this exercise. Otherwise, you will get zero scores.
- In part 3.1, you may get up to 9 scores for an **outstanding essay definitely going beyond the scope of the chapter of the course book**, mainly due to description of extra features of the sorting algorithms. You will get 6 scores if you **summarize and compare your algorithms well, but all the facts mainly come from the lecture notes and that chapter of the course book**. This means that your research was not really complete. You will get 3 points only if your **discussion remains limited to the runtime and memory complexity of the sorting algorithms**. Do not think that this is a hard task, you mainly need to do a good survey and read various articles from different sources. This topic is already in the web for ages. **If nothing is done or the effort is evaluated as insufficient**, you will get zero scores.
- In part 4.1, you will get either 4 scores if your Min(int k) method as part of the vector class **runs properly and does find the kth smallest element**. Otherwise, your mark might be zero when it **does not work**, **causes compilation or runtime errors**, **has no code commenting** (markers will use this to verify your algorithm), or **does not return the element as prescribed**.
- In parts 5.1 and 5.2, you will get s**cores proportional to the number of passed test instances**. You are given only 50% of instances we have, so the rest 50% are kept as a secret and will be used (and released after the deadline) to verify your algorithms automatically. Thus, part 5.1 and 5.2 have 27 and 105 tests in total and your result will be a fraction scaled to 12 scores (maximum for each of the parts). According to this, you may get up to 24 scores. Note that we will have a look at your code to avoid cheating; that is, the cases when your functions just simply return expected answers because you know them. We do not treat this as a real solution and you will get zero for it.

General remarks: We give either 4 or 0 scores for parts on sorting and selection algorithms, therefore no partial marks are available. This is because in real life an algorithmic solution either works or not; no one pays for partially correct solutions. We do give you partial marks for algorithms where solutions are challenging. This is mainly to motivate you to tackle those problems.

In the best case, you may get up to 58 scores. Your scores will be capped by 50. So, you may potentially skip some problems. However, we strongly encourage you to attempt all the exercises as (i) this content is examinable and you most likely get similar questions on your final exam, and (ii) you never know where you may lose some marks on other questions. Your marks will be finally scaled to 15% of the final grade for this unit.

Good luck!