# 6G5Z0024

**Assessed Exercise 3**

**This assessed exercise has 3 assessed tasks: Task A, Task B and Task C**

**Task A**

**Calculate** the **Big O notation** for the two segments of C# code shown below. When you submit your assignment, add in the “Task A” folder a document (eg a word or text file) which shows the result of the calculations.

To achieve full marks, you must show your full working out and the notation for each segment of code provided below.

Segment 1:

int n = Int32.Parse(Console.ReadLine());

LinkedList<int> sumSequence = new LinkedList<int>();

            sumSequence.AddLast(0);

            sumSequence.AddLast(1);

            for (int i = 2; i < n; i++)

            {

                int currentSum = 0;

                for (int j = i - 1; j > 0; j--)

                {

                    currentSum += sumSequence.ElementAt(j);

                }

                sumSequence.AddLast(currentSum);

            }

            for (int i = 0; i < sumSequence.Count(); i++)

            {

                Console.WriteLine(sumSequence.ElementAt(i));

            }

Segment 2:

LinkedList<string> names = new LinkedList<string>();

            string input = "";

            Console.WriteLine("Please enter who is attending!");

            Console.WriteLine("Type '!' to finish name entry...");

            Console.WriteLine("---");

            while (!input.Equals("!")){

                Console.WriteLine("Please enter name: ");

                input = Console.ReadLine();

                names.AddLast(input);

            }

            Console.WriteLine("The following are attending: ");

            for(int i = 0; i < names.Count(); i++)

            {

                Console.WriteLine(names.ElementAt(i));

            }

*Hints to complete the task: We have seen examples of Big O notation calculations in the lecture on Complexity, so please consult the slides on Moodle when attempting this Task.*

**Task B**

The code below implements the *Selection Sort* (seen in the lecture)for an array of int

static public void SelectionSort(int[] a)

{

for (int i = 0; i < a.Length -1 ; i++)

{

int smallest = i;

for (int j = i + 1; j < a.Length; j++)

{

if (a[j] < a[smallest])

smallest = j;

}

swap(ref a[i], ref a[smallest]);

}

}

where *swap(ref int x, ref int y)* is the method that exchanges the values of x and y

static void swap(ref int x, ref int y)

{

int temp = x;

x = y;

y = temp;

}

It is very useful to implement *generic* functions that can be used to sort different defined data types without the need to reimplement familiar code.

Modify the code above to implement in C# **Selection Sort** to be a **generic** function *SelecSortGen* that can sort an array of ***any*** IComparable objects.

Implement a class *Book* with members as *Title* (string), *Author* (string) and *Publication\_Year*(int). Show that you can use the implemented function *SelecSortGen* to sort an **array of int** and an **array of Book** objects (in this case you can choose to sort by any of the member variables of Book).

*Hints: Do not implement different versions of selection sort but only one, generic, that can be used to sort* an array of *any* *IComparable* objects.

*Start by creating a Console Application in C# and implement the class Book*.

*Modify the code of the Selection Sort (above) to develop the code of SelecSortGen that uses a generic type T (and that works for arrays of any IComparable objects).*

*Notice that the same “where T: IComparable” we have seen for classes can be also done for static methods:*

*Add (and complete) the generic function (below) to the class Program:*

*E.g.,* static public void SelectSortGen<T>(T[] a) where T : IComparable

{

}

*You will also need to do a generic version of the swap method.*

*Then, test SelectSortGen on an array of int and on an array of Book objects.*

**Task C**

In preparation for the next academic year, you are considering attending a number of extra-curricular activities for additional credits. However, you are only able to attend a single event at a time, and you must complete it in full from start to finish.

In C#, implement an efficient application (with user interface) that allows for the following:

* **Insert** a new activity to be stored within a collection. Each activity is composed of:
  + **ID** of the activity (assume that each activity has a unique ID, which is an **integer** value).
  + **Starting** and **Finishing Times** for the activity (e.g. an activity could consist of: ID = “1234”, starting\_time = 12.00, finishing\_time = 17.00).
* **Display** the **largest possible set of requests** that can be satisfied.

The application should allow a user to freely insert new requests, which may not necessarily be in order of starting or finishing time.

The application can either be a GUI (Windows Forms Application) or a menu driven interface (Console Application).

**Example can be found on the next page…**

Example

Given the following available activities in a day…

ID = 1, starting time = 12:00, finishing time = 12:15

ID = 2, starting time = 15:00, finishing time = 15:45

ID = 3, starting time = 12:30, finishing time = 16:00

ID = 4, starting time = 13:45, finishing time = 15:00

ID = 5, starting time = 12:00, finishing time = 12:45

ID = 6, starting time = 12:30, finishing time = 13:30

ID = 7, starting time = 13:00, finishing time = 14:30

It is possible that some requests entered into the application will have overlapping times with each other (See activities with IDs 1 & 5)

However, as it is only possible for us to attend one activity at a time, the largest set of activities we can attend here consists of 4 activities, which are:

ID = 1, ID = 6, ID = 4, ID = 2

*Hints to complete the task: You could create a class Activity (that contains ID, starting and finishing time of an activity). For simplicity, assume that it is not necessary to consider the date of the activity but only the starting and finishing time. Do not implement an algorithm which tries all the possibilities (that would be very inefficient!) but think to a Greedy algorithm seen in the Lecture.*

*You may want to consider utilizing the generic sorting algorithm that you implemented as part of the previous task whilst you implement a greedy algorithm here.*