WinForms/Avalonia and .NET
Cpt S 321 Homework Assignment
Washington State University

Submission Instructions:

- Create a branch called “Branch_HW2” and work in this branch for this assignment.
- Once you are done and before the deadline, tag the version that you would want us to grade with the assignment number. For example, “HW1”, “HW2”, etc.
- On Canvas -> Assignments -> Submit the link to your repository (a link to the tag or the branch works) by the HW deadline.
- **IMPORTANT: The HW must be tagged by the due date and a link to that tag needs to be submitted via Canvas in order to receive a grade.**

Assignment Instructions:

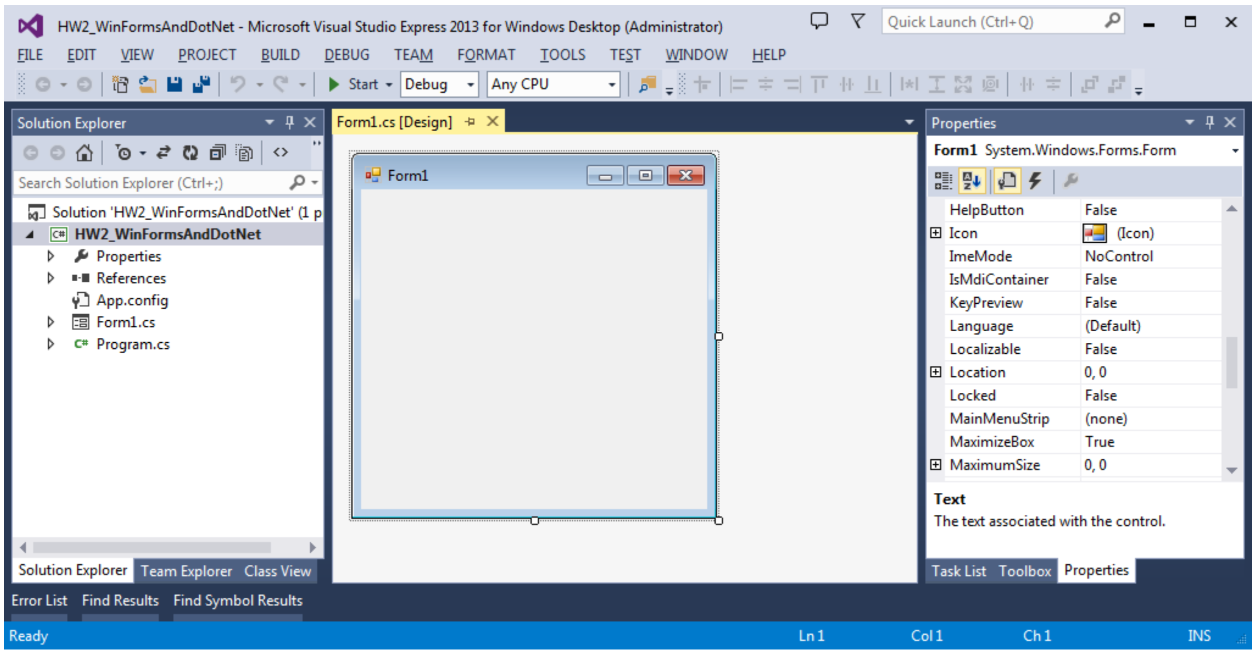
Read all the instructions *carefully* before you write any code.

A. If you are using WINFORMS, follow the instructions below:

Using Visual Studio, create a new WinForms application:

- In VS create a new project (Right click on the Solution -> Add -> New Project)
- Make sure you have Visual C# -> Windows Desktop selected in the left pane.
- Select “Windows Forms App” (for older versions you might only see “Windows Forms App (.NET Framework)” which is what you want)

You will see the WinForms designer appear and it will look something like the following:



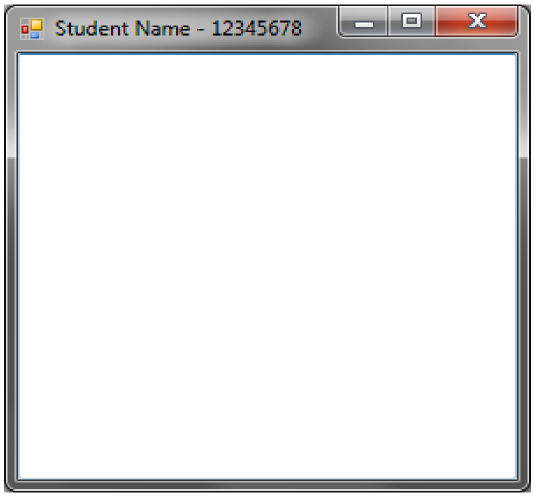
Form1 System.Windows.Forms.Form

HelpButton	False
Icon	(Icon)
ImeMode	NoControl
IsMdiContainer	False
KeyPreview	False
Language	(Default)
Localizable	False
Location	0, 0
Locked	False
MainMenuStrip	(none)
MaximizeBox	True
MaximumSize	0, 0

Text
The text associated with the control.

In this application you will just be doing computations with simple text output for the results, but instead of displaying such results in the console/terminal window, you'll be displaying them in a text box within your interface. Add a TextBox control from the Toolbox View (View -> Toolbox if you can't see it) into your interface – you do this by dragging and dropping from the toolbox to the form (the middle view in the above screenshot). Using the Property View (the right view in the above screenshot), set the Multiline property to true and the Dock property to "Fill"; if you do not see the multiline property you might not have the textbox selected so make sure you select it first. Also set the Text property of your form to your name and ID #, so that it appears in the title bar; again here make sure you select the form and then look in the Properties View.

After constructing the interface but before writing code, you can run the app to see what it looks like. Your interface should look something like the following at this point:



Next, create a method in the form that you would want to be executed after the form loads. This is where you will be calling the methods for the processing tasks described below. Note that we say "calling the methods" which means that we expect you to organize your code into classes and methods using object-oriented design principles where methods and classes are cohesive elements.

Visual Studio will flip over to your code at this point and your form class will look something like the following:

```
public partial class Form1 : Form
{
        public Form1()
        {
            InitializeComponent();
            RunDistinctIntegers();

        }

        private static void RunDistinctIntegers() // this is your method
        {

        }
}
```

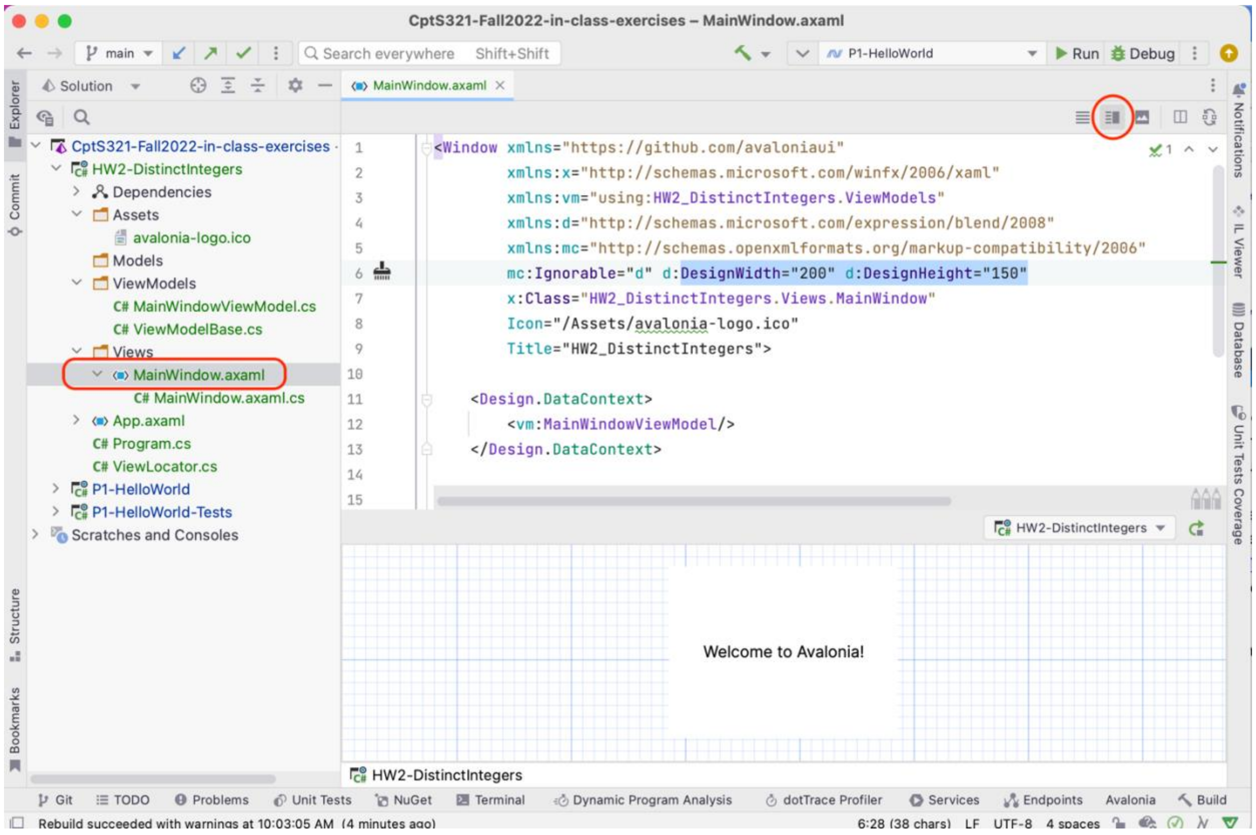
B. If you are using AVALONIA, follow the instructions below:

Install and setup Avalonia in JetBrains Rider: https://docs.avaloniaui.net/docs/getting-started/ide-support/jetbrains-rider-setup

Using Rider, create a new Avalonia application:

- In Rider create a new project (Right click on the Solution -> Add -> New Project)
- Select "Avalonia .NET Core MVVM App"

You will see the new application appear and it will look something like the following:



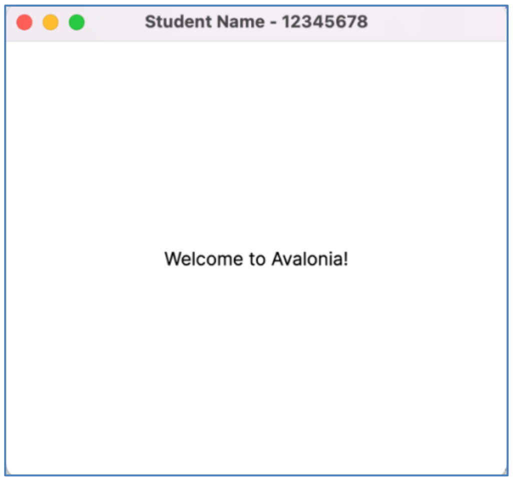
```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:vm="using:HW2_DistinctIntegers.ViewModels"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        mc:Ignorable="d" d:DesignWidth="200" d:DesignHeight="150"
        x:Class="HW2_DistinctIntegers.Views.MainWindow"
        Icon="/Assets/avalonia-logo.ico"
        Title="HW2_DistinctIntegers">

    <Design.DataContext>
        <vm:MainWindowViewModel/>
    </Design.DataContext>
```

Rebuild succeeded with warnings at 10:03:05 AM (4 minutes ago)

In this application you will just be doing computations with simple text output for the results, but instead of displaying such results in the console/terminal window, you'll be displaying them in a text box within your interface. By default Avalonia has created a TextBlock into your interface – where the text "Welcome to Avalonia" is displayed. In the MainWindow.axaml file, set the text of the title property of the main window to your name and ID #, so that it appears in the title bar.

After constructing the interface but before writing code, you can run the app to see what it looks like. Your interface should look something like the following at this point:



You can remove the horizontal and vertical center alignment for the text block and place a small padding:

```
<TextBlock Text="{Binding Greeting}" Padding="10"/>
```

Next, create a method in the form that you would want to be executed after the form loads. This is where you will be calling the methods for the processing tasks described below. Note that we say "calling the methods" which means that we expect you to organize your code into classes and methods using object-oriented design principles where methods and classes are cohesive elements.

If you open your MainWindowViewModel.cs, adapt your code to look something like the following:

```
public class MainWindowViewModel : ViewModelBase
{
    public MainWindowViewModel()
    {
        Greeting = RunDistinctIntegers();
    }

    private string RunDistinctIntegers() // this is your method
    {

    }

    public string Greeting { get; set;}
}
```

Once you have completed A or B above, follow the instructions below.

Use the Random class to create a list (System.Collections.Generic.List) with 10,000 random integers in the range [0, 20,000] (give or take a few hundred in that range is fine). Then determine how many distinct integers are in the list with 3 different approaches. Also, have them run in the order listed below. Do not change the order.

To be clear: You are not allowed to use ready implementations of Distinct. A note on what you're doing: you're taking an array of numbers and conceptually just removing duplicates and counting how many are left. If the input array was {1,1,3,5,6,6,7,7,7,9} then the distinct number set is {1,3,5,6,7,9}, implying 6 distinct numbers.

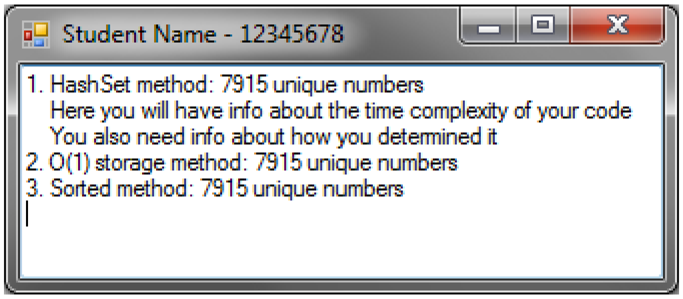
1. Do not alter the list in any way and use a hash set to determine the number of distinct integers in the list. The result will be included in the output, which is discussed more below. Also, include in the output information about the time complexity of this method. Be careful with this and describe how you determined it. Use a good amount of detail, as a too sparse explanation might not be worth full points. Use the MSDN documentation to assist you.
2. Do not alter the list in any way and determine the number of distinct items it contains while keeping the storage complexity (auxiliary) at O(1). This means that you cannot dynamically allocate any memory. If you have an algorithm that would require more storage if the list had

more items, then it is not O(1) storage complexity. Moreover, you cannot allocate additional lists, arrays, or containers of any sort. There is some leniency on the time complexity requirements for this part. Due to the strict memory requirements your method may end up being quite slow next to the other 2, but it still should execute in a matter of a few seconds on any computer made within the last 5 years.

3. Sort the list (use built-in sorting functionality) and then use a new algorithm to determine the number of distinct items with O(1) storage, no dynamic memory allocation, and O(n) time complexity. Do not alter the list further after sorting it. Determine the number of distinct items in O(n) time (not including the sorting time, which you can ignore), where n is the number of items in the list.

Output:

Use either a string or StringBuilder object to compile text results from all of your methods. Put this string in the text box after completion, so that it shows up when the app runs. Examine your results before submitting your code and make sure they make sense. Obviously all 3 methods should be giving you the same answer. It should look something like the following (but of course with more details filled in where I have neglected to give you the answer) when you're finished:



```
1. HashSet method: 7915 unique numbers
   Here you will have info about the time complexity of your code
   You also need info about how you determined it
2. O(1) storage method: 7915 unique numbers
3. Sorted method: 7915 unique numbers
```

Point breakdown (the assignment is worth 10 points):

- 1 points for correct implementation of part 1
- 2 points for correct implementation of part 2
- 2 points for correct implementation of part 3

And as usual:

- 1 point: For a "healthy" version control history, i.e., 1) the HW assignment should be built iteratively, 2) every commit should be a cohesive functionality, 3) the commit message should concisely describe what is being committed, 4) you should follow TDD – i.e., write and commit tests first and then implement and commit the functionality.
- 1 point: Code is clean, efficient and well organized.
- 1 point: Quality of identifiers.
- 1 point: Existence and quality of comments.
- 1 point: Existence and quality of test cases.

General Homework Requirements	
Quality of Version Control	● Homework should be built iteratively (i.e., one feature at a time, not in one huge commit). ● Each commit should have cohesive functionality. ● Commit messages should concisely describe what is being committed. ● TDD should be used (i.e, write and commit tests first and then implement and commit functionality). ● Include “TDD” in all commit messages with tests that are written before the functionality is implemented. ● Use of a .gitignore. ● Commenting is done as the homework is built (i.e, there is commenting added in each commit, not done all at once at the end).
Quality of Code	● Each file should only contain one public class. ● Correct use of access modifiers. ● Classes are cohesive. ● Namespaces make sense. ● Code is easy to follow. ● StyleCop is installed and configured correctly for all projects in the solution and all warnings are resolved. If any warnings are suppressed, a good reason must be provided. ● Use of appropriate design patterns and software principles seen in class.
Quality of Identifiers	● No underscores in names of classes, attributes, and properties. ● No numbers in names of classes or tests. ● Identifiers should be descriptive. ● Project names should make sense. ● Class names and method names use PascalCasing. ● Method arguments and local variables use camelCasing. ● No Linguistic Antipatterns or Lexicon Bad Smells.
Existence and Quality of Comments	● Every method, attribute, type, and test case has a comment block with a minimum of <summary>, <returns>, <param>, and <exception> filled in as applicable. ● All comment blocks use the format that is generated when typing “///” on the line above each entity. ● There is useful inline commenting <u>in addition to comment blocks</u> that explains how the algorithm is implemented.
Existence and Quality of Tests	● Normal, boundary, and overflow/error cases should be

	tested for each feature. ● Test cases should be modularized (i.e, you should have a separate test case for each thing you test - do not combine them into one large test case). ● *Note: In assignments with a GUI, we do not require testing of the GUI itself.*