Assignment 5

Software Development 2022
Department of Computer Science
University of Copenhagen

Nicolas Damgaard Frisch <nfri@di.ku.dk>
Lucas Falch Sørensen <luso@di.ku.dk>
Sofie Sylvest Aastrup <ssa@di.ku.dk>

Version 1.1: March 4th

Due: 11th of March at 15:00

This assignment is individual.
Remember to read the entire assignment text before beginning to code.

When developing software, many programmers tend to focus primarily on the beauty and performance of their code. This is important, but it is equally, if not more important, to pay attention to the correctness of the code. Without proper, reproducible tests, it is difficult to have confidence that the software works as intended. This concern is reinforced as the software changes and grows; both with respect to complexity and size.

"Unit testing" refers to the act of testing of individual "units" of source code. A "unit" is a small, testable part of an application. It is best to design your application in terms of such units, indeed to enable unit testing.

Hopefully you got a taste of this in the previous assignments. We will proceed to explore this approach to development in this assignment.

The exercise set is heavily inspired by Chapter 7, Beautiful Tests, in Beautiful Code, by Andy Oram and Greg Wilson, O'Reilly Media, 2007.

1 Comparison Of Abstract Data Types

In the object oriented programming paradigm, we are modelling the real world using abstract data types. In this part of the assignment, we will look at comparing abstract data types.

An abstract data type might not be naturally comparable; which is why you as the developer need to explicitly define how they are compared. (How would you for example compare one student in `DIKUDebate` to another student from A2 to each other? By `Intellect`? `MaxIntellect`? `StrengthOfArgument`? `Name`? Or a combination?) In the .NET framework, the `IComparable` interface is implemented to allow comparison between abstract data types. Various methods in the .NET framework rely on this interface to be implemented e.g. `Array.Sort` and `ArrayList.Sort`.

Classes that implement the `IComparable` interface must implement the following method:

`int CompareTo(object other)`

> Compares **this** object to the **other**.
>
> Returns less than 0, if **this** is less than the **other**; 0 if they are equal; and greater than 0, if **this** is greater than the **other**.
> In addition, it should adhere to the following specifications in the documentation. [1]

In the handout you will find a `Person` class, which implements the `IComparable` interface, such that we are able to sort a list of Person instances based on their age in decreasing order. Since people are not implicitly comparable in the same way that ints, floats etc. are, the class implements the `IComparable` interface. This marks that the type is comparable for methods like `Array.Sort` and forced us to implement the `CompareTo(object other)` method defining how instances of our class should be sorted when compared to each other. Thus can utilise this already implemented sorting functionality on our own classes like `Person`. For now, you should only have to download the handout. Later you will use the Person class in testing the searching algorithms described below.

2 Binary Search

To delve further into testing, we will consider the *binary search* algorithm. This algorithm is an amazingly fast searching algorithm. Given a sorted, indexed collection of comparable elements and a target element, it searches the collection to find the index of the target element. If the target element is not in the collection, it returns a sentinel index (typically, -1), indicating that it was not found.

In the description above, there is an important ambiguity regarding the return value. When you use binary search the same index as other searching algorithms is not always returned when there are identical elements (or elements which have the same place in the sorting order). Think about why this is. You should be aware of this when you test, since it can cause problems.

If you need your memory refreshed on how *binary search* works, Wikipedia has an excellent article on the subject.[2]

[1] `https://docs.microsoft.com/en-us/dotnet/api/system.icomparable.compareto?view=net-6.0#notes-to-implementers`
[2] `https://en.wikipedia.org/wiki/Binary_search_algorithm`

2.1 Sample Implementation

In the hand-out alongside the assignment text, we provide a BinarySearch folder, containing a sample implementation of the binary search algorithm for arrays of `IComparable` objects. The implementation is incomplete. It is your task to finish it.

From the hand-out, proceed with the following tasks:

2.1. Open the BinarySearch folder.

2.2. You will find 3 projects in the solution: A text-user interface (TUI), a Test project with a single test, and a library project.

The library hosts:

- an implementation of binary search in `Search.cs`;
- a method to show the contents of an array in `Show.cs`; and
- a random `IComparable` (`int`) array generator in `Generator.cs`.

The `Main` method in the TUI project uses all of these library features to conduct a handful `Console.WriteLine`-style tests of the binary search implementation.

2.3. Navigate into the TUI project through the terminal, and Run the TUI project to see that it works.
Note that the Library project is a library much in the same sense as you may know it from F#. Thus you will be able to build it, but not run it directly.

3 Testing Binary Search with NUnit

Littering a program with print statements is not a very robust way to perform tests. We will therefore again be using NUnit to test our solution. Sometimes you might write tests that fail. Usually this is indicator that something is wrong with the implementation. You will be tasked to fix it later and it is okay to move on from a failing test to write more tests so you have more test cases to understand how it fails.

3.1. A test project has already been added for you, but it currently only holds a single test that does nothing.

3.2. Remove the dummy test.

3.3. Add 3 test methods:

3.3.1. `TestTooLow`, testing that the binary search yields -1 for values less than any given value in the array.

3.3.2. `TestTooHigh`, testing that the binary search yields -1 for values greater than any given value in the array.

3.3.3. `TestElement`, testing that the binary search yields a value other than -1 for any value in the array.

Hint: NUnit has both `Assert.AreEqual` and `Assert.AreNotEqual`.

3.4. Create a new member function `TestEmptyArray` (don't forget the `[Test]` attribute). Create an empty array. Create a for loop that iterates over the integers $[-100, 100]$. For each loop iteration, assert that a call to *binary search* with the given integer returns -1.

3.5. Run the tests. Like previously; use `dotnet test` to run them. All your tests should pass in the end, but some of them might fail right now. Don't craft your tests to fit the implementation. The tests should help you verify the implementation and fix its issues.

3.6. Come up with a more exhaustive list of cases that might fail and write tests for those to make sure that the implementation is correct.

3.7. Convinced that the Binary-search works on integer values, instantiate an array of type Person and search for a specific person.

3.8. Create more tests to reassure yourself that your implementation also works for abstract data types such as Person.

4 Modifications

The following sections ask you to consider a number of explicit problems with our implementation, which might have eluded your attention thus far.

4.1 Arithmetic Overflow

A central part of doing solid unit testing is to test that the code performs well under extreme circumstances. This is also called border case testing. The first thing we will optimize within the *binary search* algorithm is taking into account arithmetic overflow.
When the handed out implementation is run against a sufficiently large array, the following line will generate an arithmetic overflow:

```
        var mid = (high + low) / 2;
```

4.1. Find a solution for the problem and implement it instead of the shown line above.

4.2 Out-of-Bounds

When searching through a *sorted* array, it can be checked whether or not the target element is in the array without running through the algorithm. This can be done by checking at most 2 elements in the array.

4.2. Add code to the implementation to conduct these checks before a binary search is commenced. The test-cases `TestTooLow` and `TestTooHigh` from above should still pass.

4.3 Running Time

You might recall from your DMA course that *binary search* has a "logarithmic" running time: By continuously halving the array when searching through it, the algorithm will perform at most $\lceil 1 + \log_2 n \rceil$ look-ups for an array of size n. Thus, *binary search* has an asymptotic running time of $O(\log_2 n)$.

It is a good idea to verify this using a test. This is where the use of the `IComparable` type comes in handy.

4.3. Declare a new class, `Library.ComparisonCountedInt`.

The class should have one constructor, taking in an `int` value. This value is to remain constant throughout the life-time of the object.

The class should implement the interface `IComparable`, and so implement the method `CompareTo`, as mentioned above.

Voluntary part: Implement tests that ensure that your implementation adheres to the implementation notes: `https://docs.microsoft.com/en-us/dotnet/api/system.icomparable.compareto?view=net-6.0#notes-to-implementers`

`CompareTo` should increment a private instance variable for the number of comparisons (initialized to 0 in the constructor). Other than that it should refer to the implementation of `CompareTo`, already available for `int`.

The class should also expose a read-only property `ComparisonCount`, returning the number of comparisons performed on the object.

4.4. Declare a public static method in `Library.ComparisonCountedInt`:

`int CountComparisons(ComparisonCountedInt[] array)`
Returns the sum over the comparison counts in the given array.

4.5. Declare test-cases with the prefix `TestRunningTime(arraySize)` with arrays of different sizes, checking that the algorithm does not overstep the logarithmic bound.

Hints:

- You can use the provided `Generator` class to generate a sorted array of a given size and magnitude. It is already initialised in your `[SetUp()]` method of your NUnit test-class.

  You will need to extend `Generator` with a method to generate an array of the proper type (i.e., `ComparisonCountedInt[]`).

- Use `[TestCase(`*`arraySize`*`)]` instead of `[Test]` above the test to generate different array sizes (Multiple `[TestCase()]` can be placed on consecutive lines to run the test multiple times with different inputs.)
- Use `((int)Math.Ceiling(Math.Log(n, 2.0))` to get the base 2 logarithm of `n`.
- Consider the difference between the asymptotic running time, and the actual running time. Do we need to adjust for this difference in our test?
- Use `Assert.LessOrEqual` or `Assert.GreaterOrEqual`.

4.4 Comparing with Linear Search

Linear search[3] is the simplest search algorithm. We simply go through the array from one end to the other and see if it matches. It therefore makes at most n comparisons.

4.6. Declare a public static method `Library.Search.Linear`:

`int Linear(IComparable[] array, IComparable target)`
Performs a linear search of array for the least index of the target element. Returns -1 if no such index exists.

4.7. Assert that Linear search and Binary search return the same index when searching through the same array. If they do not, why do you think that is (HINT: this is the ambiguity from earlier)? Fix your implementation of Binary search such that it works.

4.8. Declare at least one test-case with the prefix `TestLinearVsBinary`, checking that *Binary Search* performs fewer comparisons than *Linear Search*, on average.

5 Cleaning up your code

To keep your TA happy, and receive more valuable feedback, you should:

- Remove commented out code.
- Make sure that your files and folders have the correct names.
- Make sure the code compiles without errors and warnings.
- Make the code comprehensible (perform adequate renaming, separate long methods into several methods, add comments where appropriate).
- Make sure the code follows our style guide[4].

[3] `https://en.wikipedia.org/wiki/Linear_search`
[4] See `https://github.com/diku-dk/su-guides/blob/main/guides/CSharpStyle.md`.

- Make sure to make a clean build: `dotnet clean`.
- Remove `obj/` and `bin/` directories as well as any auto-completion data or tagging databases that VS Code.

6 Submission

Your work must be submitted through Absalon. You should submit a single file:

- Your code (Search library, tests and person) as a zip-file. Zip the entire directory of your project, including the `.sln` and `.proj` files. The zip file should be named firstName-A5.zip.

When submitting code make sure that you only submit what is required to run the code. In C#, that is usually the `.csproj`, `.sln` as well as any `.cs` files. Exclude any OS specific files. When in doubt, attempt to simulate running the code from the zip file, i.e. copy and extract and run.