Assignment 1: CAB301

Shridhar Thorat: n10817239

April 8, 2023

Contents

1 Introduction

The goal of this assignment was to design 3 Abstract Data types (ADTs) in a C# environment. This report aims to design pseudocode algorithms for the three methods in *Scheduler*, and implement them in C#. As well as determining their theoretical efficiency and describing the effects during implementation. It will also include a test plan for the three ADT's designed.

2 Algorithm Design and Analysis

For the simple purposes of this assignment and selection having a fairly simple implementation in C#, *Scheduler* implements the *selection* sort algorithm for all three methods in a similar method to the pseudocode in page 98—100(Levitin, 2012). The Selection sort algorithm can be directly translated with some minor changes. These minor changes are due to C# being the language of use and due to design choices for the *Job*, *JobCollection* and *Scheduler* classes.

Each sorting algorithm uses a *JobCollection* called *Jobs* which is a property of a *schedule*. *Jobs* is an array of *job* objects and has some *capacity* and *count*. If we consider that the collection is expressed as $Jobs[0, \ldots, capacity - 1]$, then the number of jobs in the collection is expressed as $Jobs[0, \ldots, count - 1]$. For the purposes of pseudocode and time efficiency, *count* will be considered as n and the worst case time efficiency function will be relative to it. This is because when sorting, null values can be ignored as they are 'empty'. Therefore, the sorting algorithms need only iterate over the jobs in a collection — hence the time efficiency is relative to *count* rather than *capacity*.

Additionally, each algorithm used a method from the *JobCollection* class called `ToArray()` which creates an independent *IJob* array so as to prevent editing the original, unsorted array of jobs. This method requires first instantiating a new *IJob* array with the same capacity as the original, next, each job is added to the copy. Copying each job requires looping through the original array from the first job, to the last job, in other words, *count* number of times. Finally, returning the array is one basic operation. Thus, `ToArray()` is considered as **n+2** basic operations.

2.1 First-Come, First-Served

First-come, first-serve sorts a *JobCollection* in non-descending order of the arrival time; *timeReceived* for the jobs in it.

It can access a property of *Scheduler* called *Jobs* which is a *JobCollection* with some capacity and count. Additionally, a *job* in *Jobs* has a parameter *timeReceived* which can be accessed using property accessors in C#; i.e using *job.timeReceived*.

2.1.1 Pseudocode Notation

ALGORITHM FirstComeFirstServed()

Input: No input, however, uses a property of a Schedule that is a *JobCollection* called *Jobs* with $Count = n$ and some capacity. Uses a *JobCollection* method called `ToArray()`.
Output: An array $A[0 \ldots n-1]$ with $n-1$ jobs sorted in non-descending order of $TimeReceived$ of a job.

```
1: A ← copies Jobs property to independent IJob array with length the same as capacity of Jobs
2: for i ← 0 to n − 2 do
3:     min ← i
4:     for j ← i + 1 to n − 1 do
5:         if A[j].TimeReceived < A[min].TimeReceived then
6:             min ← j
7:     Swap A[i] and A[min]
8: return A
```

2.1.2 Theoretical time efficiency

Observing the pseudocode, **line 1** uses the `ToArray` method from the *JobCollection* class in the implementation and as mentioned earlier in the report, counts as $n + 2$ basic operations.

The nested `for-loop` decribes a selection sort algorithm. In this case it works by looping through an array, with each iteration finding the smallest element and swapping it with the left-most, unsorted element in the array. In the worst case, the first iteration would make *n-1* comparisons to find the smallest element and swap it to the first position in the array. The second iteration would make *n-2* comparisons; ignoring the first element, and finding the next smallest element and placing it in the second position in the array. The last iteration would simply compare the two last items and swap them. The sum of the number of comparions can be mathematically expressed by the following summation equation.

$$(n-1) + (n-2) + \cdots + 1 = \sum_{i=1}^{n-1} i$$

The pseudocode reflects this with the outer loop in **line 2** iterating a maximum of *n-1* times. The inner loop is executed *n-1-0* times in the first iteration when i is 0, *n-1-1* times in the second iteration when i is 1, until the last iteration when it executes once. Or simply put, for each *n-1* iteration of the outer loop, the inner loop performs the one basic operation in **line 6**, and as such, executes *n-1-i* times since after each i iteration, the first i elements in array A will be sorted. The number of basic operations for the nest `for-loop` can be expressed by the following summation.

$$\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} (1) = \sum_{i=0}^{n-2} n - i - 1 = (n-1) + (n-2) + \cdots + 1 = \sum_{i=1}^{n-1} i$$

Additionally, in the worst case scenario, the *swap* function runs for each iteration of the outer loop — $n - 1$ times. *Swap* is considered as 3 basic operations based on how C# would swap two items in an array. Example code for swapping items in positions i and $i + 1$ of an array of *IJob* objects is shown below.

```
IJob temporaryJob = IJobArray[i+1];
IJobArray[i+1] = IJobArray[i];
IJobArray[i] = temporaryJob;
```

Thus, it counts as $3(n - 1)$ basic operations for the whole algorithm. Thus, the worst case efficiency function *C(n)*, for an array with n jobs and some *capacity*, (where *capacity* $\geq n$) can be described mathematically as the below equation.

$$\begin{aligned} C_{worst\_fcfs}(n) &= n + 2 + \sum_{i=1}^{n-1} i + 3(n-1) \\ &= n + 2 + 3n - 3 + \frac{n(n-1)}{2} \\ &= 4n - 1 + \frac{1}{2}n^2 - \frac{1}{2}n \\ &= \frac{1}{2}n^2 + 3.5n - 1 \end{aligned}$$

$$\texttt{Thus } C_{worst\_fcfs}(n) = \frac{1}{2}n^2 + \frac{7}{2}n - 1, \quad C_{worst\_fcfs}(n) \in O(n^2)$$

It can be observed from above that *FirstComeFirstServed* makes $4n - 1$ more operations than the worst case efficiency of a normal *selection* sort algorithm. However, it has the same worst case class complexity of $O(n^2)$.

2.2 Shortest Job First

Shortest Job First sorts a *JobCollection* in non-descending order of *executionTime* for the jobs in it.

2.2.1 Pseudocode Notation

Similar to the pseudocode for 'First-come, first served', 'Shortest Job First' considers the *count* of `schedule.Jobs` as n.

ALGORITHM FirstComeFirstServed()

Input: No input, however, uses a property of a Schedule that is a *JobCollection* called *Jobs* with $Count = n$ and some capacity. Uses a *JobCollection* method called `ToArray()`.
Output: An array $A[0 \dots n-1]$ with $n-1$ jobs sorted in non-descending order of *TimeReceived* of a job.

```
A ← copies Jobs property to independent IJob array with length the same as capacity of Jobs
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if A[j].ExecutionTime < A[min].ExecutionTime then
            min ← j
    Swap A[i] and A[min]
return A
```

2.2.2 Theoretical time efficiency

By observation, the pseudocode for *Shortest Job First* is nearly identical to *First-Come, First-Served*, bar the different parameters tested in the **if** statements. This means that the theoretical worst case time efficiency function and the big-O class will be identical.

$$\texttt{Thus } C_{worst\_SJF}(n) = \frac{1}{2}n^2 + \frac{7}{2}n - 1, \quad C_{worst\_SJF}(n) \in O(n^2)$$

2.3 Priority

Priority sorts a *JobCollection* in non-ascending order of *priority* for the jobs in it. Non-ascending as in 9 (the highest) is first, and 1 (the lowest) would be last.

2.3.1 Pseudocode Notation

ALGORITHM FirstComeFirstServed()

Input: No input, however, uses a property of a Schedule that is a *JobCollection* called *Jobs* with $Count = n$ and some capacity. Uses a *JobCollection* method called `ToArray()`.
Output: An array $A[0 \dots n-1]$ with $n-1$ jobs sorted in non-descending order of *TimeReceived* of a job.

```
A ← copies Jobs property to independent IJob array with length the same as capacity of Jobs
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if A[j].Priority > A[min].Priority then
            min ← j
    Swap A[i] and A[min]
return A
```

2.3.2 Theoretical time efficiency

Similarly to the previous two methods, the only difference in the pseudocode is the different **if** statements. As such, the time efficiency and big-O class will be the same.

$$\texttt{Thus } C_{worst\_priority}(n) = \frac{1}{2}n^2 + \frac{7}{2}n - 1, \quad C_{worst\_priority}(n) \in O(n^2)$$

2.4 Real world implications

It can be better understood how the time efficiency impacts computational time by displaying a table that compares how many operations will be run depending on n; the number of jobs in a JobCollection for a schedule, alongside how long it will take.

n	C_worst	seconds	minutes
1	4	0.004	6.6667E-05
10	67	0.067	0.00111667
100	5152	5.152	0.08586667
1000	501502	501.502	8.35836667
10000	50015002	50015.002	833.583367
100000	5000150002	5000150	83335.8334
1000000	5E+11	500001500	8333358.33
10000000	5E+13	5E+10	833333583

Figure 1: One operation takes 1 ms

From the above table, even if we consider that each operation takes 1 millisecond, with an array of length 1000, a computer will theoretically run 501,502 basic operations which take around 8 and a half minutes. A tenfold increase raises the execution time to an extreme 34 hours.

2.4.1 Limitations

Thus, a major limitation for the methods in scheduler is that it is inefficient for large arrays.

However, as this assignment does not involve testing with large arrays, this limitation is negligible. However for future extensions to this project, it is worth looking into applying more efficient sorting methods than selection. Some of these include insertion, quick-sort and merge sort.

3 Test Plan

Testing was completed using an MSTest project provided by Microsoft in the Visual Studio IDE in the .Net 7.0 framework. The operating system was MacOS Ventura 13.2.1 (22D68) for the M1 Mac-book Pro.

To ensure that all aspects of the software system cover functional, non-functional and boundary cases of methods, a comprehensive test plan was designed. The plan has been split into three sections that will delve deeper into what was tested each of the three ADTs and why.

3.1 Testing Job.cs

The Job class contained the following methods:

1. `IsValidId(uint id)`
2. `IsValidExecutionTime(uint executiontime)`
3. `IsValidPriority(uint priority)`
4. `IsTimeReceived(uint time)`

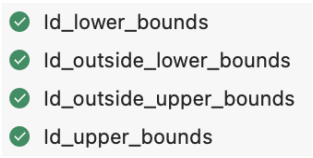
The goal of each of each of these methods was to return a Boolean value based on whether the parameters they take are valid. It was specified that Id needs to be between 1 and 9 inclusive, and priority needs to be between 1 and 999 inclusive. Additionally, execution time and time received need to simply be greater than 0.

For this reason, testing these methods involved providing boundary cases as input and numbers outside the boundary as input. Testing invalid parameter data types was not required as errors are automatically thrown by Visual Studio.

3.1.1 IsValidId(uint id)

It was important to test if IsValidId returned true for all values between 1 and 999, and for 1 and 999 themselves. Additionally, it should return false for any values outside the bounds. A summarisation of the results of testing is below and all cases pass, and the method performs as specified. The test data, results and testing code in the appendix for IsValidId.

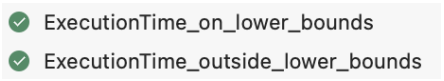
Unit testing summarisation



3.1.2 IsValidExecutionTime(uint executiontime)

Similarly, it was important that IsValidExecutionTime returned true when executionTime was greater than 0, and false for values that were 0 or below. Hence, the values 0 and 1 were used for testing. As summarised below, both tests passed and in the method performs as specified. The test data, results and testing code in the appendix for IsValidExecutionTime.

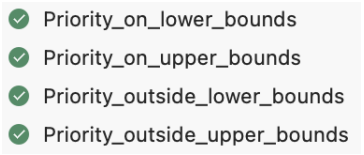
Unit testing summarisation



3.1.3 IsValidPriority(uint priority)

In a similar fashion, IsValidPriority was tested for its boundary and functional requirements (requirement is to be between 1 and 9 inclusive). It also passed each test summarised below. The test data, results and testing code in the appendix for IsValidPriority.

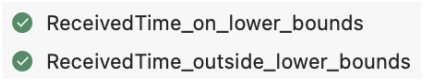
Unit testing summarisation



3.1.4 IsTimeRereceived(uint time)

Similarly, IsTimeReceived needed to return true when receivedTime was greater than 0 and false for values that were 0 or below. Hence, the values 0 and 1 were used for testing. As summarised below, both tests passed and in the method performs as specified. The test data, results and testing code in the appendix for IsTimeReceived.

Unit testing summarisation



3.2 Testing JobCollection.cs

The goal of JobCollection is to store and manipulate a collection of jobs. This includes methods for adding and removing a job from a collection, checking if a job is in a collection and finding a job in a collection. It includes an additional method that returns an independent IJob[] array with the same jobs as the collection.

Since these methods all involve collections of jobs, they all use the same job objects, they are common for each test. The jobs used for testing are listed below.

1. `Add(IJob job)`
2. `Contains(uint id)`
3. `Find(uint id)`
4. `Remove(uint id)`

Since these methods all involve collections of jobs, they all use the same *job* objects, they are common for each test. Additionally, each test used a collection with capactity 5, hence an empty *JobCollection* was also used as a public variable. The data used for testing are listed below.

Common Test Data

```
IJob job1 = new Job(1, 400, 10, 2);
IJob job2 = new Job(2, 20, 14, 5);
IJob job3 = new Job(3, 101, 1, 2);
IJob job4 = new Job(4, 40, 78, 4);
IJob job5 = new Job(5, 18, 54, 3);
IJob job6 = new Job(6, 2, 213, 8);
const uint capacity = 5;
IJobCollection coll = new JobCollection(capacity);
```

Figure 2: IJob objects used throughout testing of JobCollection

3.2.1 Add(IJob job)

Before testing Contains, Find and Remove. It was important to test if adding jobs to a collection performed as specified.

The first test involved adding a job to an empty array, expecting it to append and return true. Once this was confirmed, the second test was performed and involved adding a job to full array, expecting it to not add the job and return false. The third test involved adding a job that already exists in the collection, expecting it not to add the job and return false. The last test involved adding a null-type job to an empty array, expecting it to not add the job and return false. The data and results are summarised below. The test data, results and testing code for each test can be found in the appendix for Add.

Unit testing summarisation

- Add_duplicate_job
- Add_null_job
- Add_to_empty_arrray
- Add_to_full_arrray

3.2.2 Contains(uint id)

Contains was specified to return true if the supplied *id* matched to a job in a collection and false otherwise. Additionally, if the number of jobs in a collection; *count*, was 0, the method would return false without looping through the collection. Thus, testing involved checking whether the correct boolean values were returned if a job wasn't in a collection, was in a collection, and if a collection had no jobs to begin with. The summary below shows that each of these tests passed and test specific data, results and code can be found in the appendix for Contains

It is worth noting that testing the scenario where the capacity is one could have been useful to safeguard against incorrect ranges in a `for` loop, however, a `while` loop was used thus making this test unnecessary.

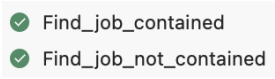
Unit testing summarisation

- Contains_count_0
- Contains_job_in_collection
- Contains_job_not_in_collection

3.2.3 Find(uint id)

Find was specified to return null if the supplied *id* didn't match with any jobs in a collection and returning the job object if it did match. Additionally, it used *Contains* to first check if the collection contained a job with the supplied *id*. Testing however didn't check if this line ran as the *Contains* method was already tested to perform as specified before *Find*. Thus testing only checked whether *null* or a correct *job* was returned depending on the supplied id. The below summary shows that both tests passed. Test specific data, results and code can be found in the appendix for Find.

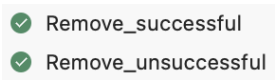
Unit testing summarisation



3.2.4 Remove(uint id)

Remove was specified to remove a job in a collection that matched the specified *id*, decrease the *count* by 1 and then return true. If the job doesn't exist, then simply return false. Thus it was tested whether the method returns false when not removed and *count* remains unchanged. It was also tested that when *Remove* removes a job successfully, the method returns true, the count decrements by one, and the removed job can no longer be found in the array. The summarisation of results below shows that both tests passed and *Remove* operates as specified. Test data, results and code can be found in the appendix for Remove.

Unit testing summarisation

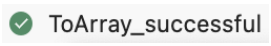


3.2.5 ToArray

ToArray was specified to simply return an independent array with the same jobs in a collection. It was important to check that ToArray returned the same jobs in the same order, thus this was the only test completed.

The summarisation of results below shows that both tests passed and *Remove* operates as specified. Test data, results and code can be found in the appendix for ToArray.

Unit testing summarisation



3.3 Testing Scheduler.cs

3.3.1 Unit testing methodology

As mentioned in the beginning of the report, the three alogorithms designed were `FirstComeFirstServed`, `ShortestJobFirst` and `Priority`. Each method was tested data under the same set of conditions, however used different job objects as the test data.

It is important to note that when the phrase **tested parameter** is mentioned, this refers to either, *receivedTime*, *executionTime* or *priority*, depending on whether `FirstComeFirstServed`, `ShortestJobFirst` and `Priority` are being tested respectively.

Testing worked by creating an array of *uint* values called *expected* which has the expected order of the *tested parameter* for each job. An *actual* array of the same type was created which contains the *tested parameter* for each job sorted using the relevant method.

These were then compared using `bool assertion = Enumerable.SequenceEqual(actual, expected)` which would return true if each value in the two arrays was the same, i.e., the *FirstComeFirstServed* method correctly sorted the *JobCollection*, and false otherwise. Since unit testing was used,

`Assert.IsTrue(assertion)`, was added to the testing method.

3.3.2 Testing methods

One condition was testing if a collection was sorted correctly when all jobs had unique parameters for the *tested parameter*. For example, *Priority* would be tested to sort a collection with jobs that all had unique *priority* parameters. The goal of this test was to ensure that the order of a sorted collection was correct.

The next condition also used unique parameters for the *tested parameter*, but used a partially full collection. This was important to ensure that the sorting methods would return arrays with null values at the end and wouldn't throw exceptions when encountering null objects.

Thus, these conditions were tested using the methods;
`unique_jobs_in_a_full_collection` and `unique_jobs_in_a_partial_collection`.

Additionally, in order to test whether multiple computing jobs that have the same *tested parameter*, are executed in any order, two other methods used collections that had some jobs with the same *tested parameter*.

Thus two other methods were used for testing. These were `common_jobs_in_a_full_collection` and `common_jobs_in_a_partial_collection`.

The last test involved sorting a collection with a single job. This was important to ensure that the methods would not use indexes out of the range of the collection. This test was named `collection_capacity_one`.

It was deemed unnecessary to supply an empty collection to a sorting method as the methods are based on the *Count* variable; the number of jobs, rather than *Capacity*. Hence this was not tested.

Thus the list of test methods can be summarised below.

1. `unique_jobs_in_a_full_collection`
2. `unique_jobs_in_a_partial_collection`
3. `common_jobs_in_a_full_collection`
4. `common_jobs_in_a_partial_collection`
5. `collection_capacity_one`

3.3.3 FirstComeFirstServed

Each test used the same *job* objects, and a *JobCollection* with capacity 7. Note that *priority* and *executionTime* are held constant as they are not relevant to the *FirstComeFirstServed* sorting algorithm.

Common Test Data

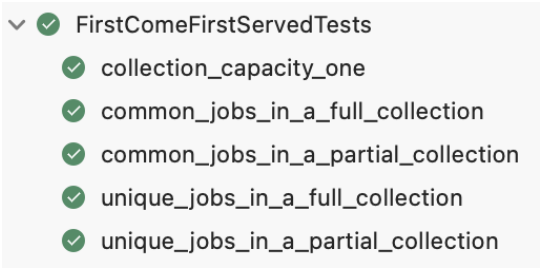
```
private IJob job1 = new Job(1, 525, 1, 1);
private IJob job2 = new Job(2, 412, 1, 1);
private IJob job3 = new Job(3, 823, 1, 1);
private IJob job4 = new Job(4, 789, 1, 1);
private IJob job5 = new Job(5, 431, 1, 1);
private IJob job6 = new Job(6, 392, 1, 1);
private IJob job7 = new Job(7, 948, 1, 1);
private IJob job8 = new Job(8, 431, 1, 5); // same timeReceived as job5
private IJob job9 = new Job(9, 525, 1, 1);  // same timeReceived as job1
private uint capacity = 7;
```

Figure 3: IJob objects used throughout testing of FirstComeFirstServed

While actual unit testing was done differently, the test data, expected results, and test results are simply displayed as lists of jobs. The exact *Jobs* used did differ for each test and this input data along with expected and resulting data can be found in the appendix for FirstComeFirstServed.

From the below summary of tests it can be seen that each test passed and thus *FirstComeFirstServed* was verified to perform as specified.

Unit testing summarisation



3.3.4 ShortestJobFirst

Similarly to *FirstComeFirstServed*, methods for testing *ShortestJobFirst* used the same *job* objects, and a *JobCollection* with capacity 7. Note that *priority* and *receivedTime* are held constant as they are not relevant to the *ShortestJobFirst* sorting.

Common Test Data

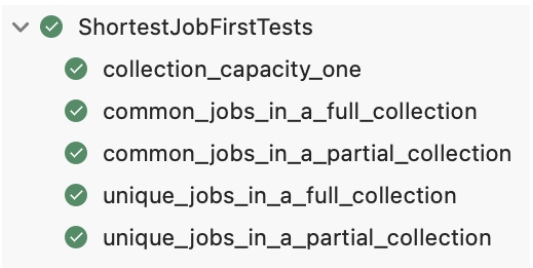
```
private IJob job1 = new Job(1, 1, 52, 1);
private IJob job2 = new Job(2, 1, 41, 1);
private IJob job3 = new Job(3, 1, 82, 1);
private IJob job4 = new Job(4, 1, 78, 1);
private IJob job5 = new Job(5, 1, 43, 1);
private IJob job6 = new Job(6, 1, 39, 1);
private IJob job7 = new Job(7, 1, 94, 1);
private IJob job8 = new Job(8, 1, 43, 5); // same executionTime as job5
private IJob job9 = new Job(9, 1, 52, 1);  // same executionTime as job1
private uint capacity = 7;
```

Figure 4: IJob objects used throughout testing of ShortestJobFirst

While actual unit testing was done differently, the test data, expected results, and test results are simply displayed as lists of jobs. The exact *Jobs* used did differ for each test and this input data along with expected and resulting data can be found in the appendix for ShortestJobFirst.

From the below summary of tests it can be seen that each test passed and thus *ShortestJobFirst* was verified to perform as specified.

Unit testing summarisation



3.3.5 Priority

Similarly to the other two sorting algorithms, testing methods for *Priority* used the same *job* objects, and a *JobCollection* with capacity 7. Note that *executionTime* and *receivedTime* are held constant as they are not relevant to the *ShortestJobFirst* sorting. **Common Test Data**

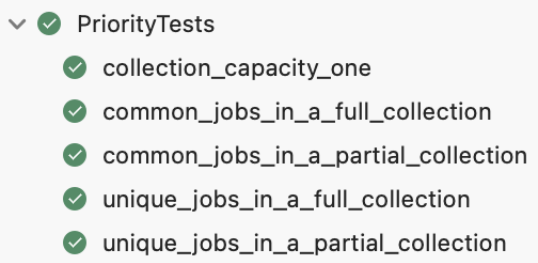
```
private IJob job1 = new Job(1, 1, 52, 5);
private IJob job2 = new Job(2, 1, 41, 7);
private IJob job3 = new Job(3, 1, 82, 2);
private IJob job4 = new Job(4, 1, 78, 3);
private IJob job5 = new Job(5, 1, 43, 5);
private IJob job6 = new Job(6, 1, 39, 9);
private IJob job7 = new Job(7, 1, 94, 1);
private IJob job8 = new Job(8, 1, 43, 5); // same priority as job5
private IJob job9 = new Job(9, 1, 52, 4);  // same priority as job1
private uint capacity = 7;
```

Figure 5: IJob objects used throughout testing of Priority

While actual unit testing was done differently, the test data, expected results, and test results are simply displayed as lists of jobs. The exact *Jobs* used did differ for each test and this input data along with expected and resulting data can be found in the appendix for Priority.

From the below summary of tests it can be seen that each test passed and thus *Priority* was verified to perform as specified.

Unit testing summarisation



4 References

Levitin, A. (2012). Introduction to The Design & Analysis of Algorithms. In A. Levitin, Introduction to The Design &Analysis of Algorithms (Vol. 3). New Jersey, United States of America: Pearson Education.

5 Appendix

5.1 Job ADT test data, results and code

5.1.1 IsValidId()

```
[TestMethod]
public void Id_outside_lower_bounds()
{
    uint id = 0;
    bool actual = Job.IsValidId(id);
    Assert.IsFalse(actual);
}

[TestMethod]
public void Id_on_lower_bounds()
{
    uint id = 1;
    bool actual = Job.IsValidId(id);
    Assert.IsTrue(actual);
}

[TestMethod]
public void Id_on_upper_bounds()
{
    uint id = 999;
    bool actual = Job.IsValidId(id);
    Assert.IsTrue(actual);
}

[TestMethod]
public void Id_outside_upper_bounds()
{
    uint id = 1000;
    bool actual = Job.IsValidId(id);
    Assert.IsFalse(actual);
}
```

Figure 6: Unit testing for IsValidId

5.1.2 IsValidExecutionTime()

```
[TestMethod]
public void ExecutionTime_outside_lower_bounds()
{
    uint executiontime = 1;
    bool actual = Job.IsValidExecutionTime(executiontime);
    Assert.IsTrue(actual, "Valid_ExecutionTime_min failed");
}

[TestMethod]
public void ExecutionTime_on_lower_bounds()
{
    uint executiontime = 0;
    bool actual = Job.IsValidExecutionTime(executiontime);
    Assert.IsFalse(actual);
}
```

Figure 7: Unit testing for IsValidExecutionTime

5.1.3 IsValidPriority()

```
[TestMethod]
public void Priority_on_lower_bounds()
{
    uint priority = 0;
    bool actual = Job.IsValidPriority(priority);
    Assert.IsFalse(actual);
}

[TestMethod]
public void Priority_outside_lower_bounds()
{
    uint priority = 1;
    bool actual = Job.IsValidPriority(priority);
    Assert.IsTrue(actual);
}

[TestMethod]
public void Priority_on_upper_bounds()
{
    uint priority = 9;
    bool actual = Job.IsValidPriority(priority);
    Assert.IsTrue(actual);
}

[TestMethod]
public void Priority_outside_upper_bounds()
{
    uint priority = 10;
    bool actual = Job.IsValidPriority(priority);
    Assert.IsFalse(actual);
}
```

Figure 8: Unit testing for IsValidPriority

5.1.4 IsTimeReceived()

```
[TestMethod]
public void ReceivedTime_outside_lower_bounds()
{
    uint timeReceived = 0;
    bool actual = Job.IsTimeReceived(timeReceived);
    Assert.IsFalse(actual);
}

[TestMethod]
public void ReceivedTime_on_lower_bounds()
{
    uint timeReceived = 1;
    bool actual = Job.IsTimeReceived(timeReceived);
    Assert.IsTrue(actual);
}
```

Figure 9: Unit testing for IsTimeReceived

5.2 JobCollection ADT test data, results and code

5.2.1 Add()

```
[TestMethod]
public void Add_to_empty_arrray()
{
    uint capacity = 5;
    IJobCollection coll = new JobCollection(capacity);
    Assert.IsTrue(coll.Add(job1) == true);
}

[TestMethod]
public void Add_to_full_arrray()
{
    uint capacity = 5;
    IJobCollection coll = new JobCollection(capacity);
    coll.Add(job1); coll.Add(job2); coll.Add(job3); coll.Add(job4); coll.Add(job5);
    Assert.IsTrue(coll.Add(job6) == false);
}

[TestMethod]
public void Add_duplicate_job()
{
    uint capacity = 5;
    IJobCollection coll = new JobCollection(capacity);
    coll.Add(job1); coll.Add(job2); coll.Add(job3); coll.Add(job4);
    Assert.IsTrue(coll.Add(job1) == false);
}

[TestMethod]
public void Add_null_job()
{
    uint capacity = 5;
    IJobCollection coll = new JobCollection(capacity);
    IJob? null_job = null;
    Assert.IsTrue(coll.Add(null_job!) == false, $"expected false");
}
```

Figure 10: Unit testing for Add

5.2.2 Contains()

```
[TestMethod]
public void Contains_count_0()
{
    uint capacity = 5;
    IJobCollection coll = new JobCollection(capacity);
    bool actual = coll.Contains(job1.Id);
    Assert.IsFalse(actual);
}

[TestMethod]
public void Contains_job_in_collection()
{
    uint capacity = 5;
    IJobCollection coll = new JobCollection(capacity);
    coll.Add(job1); coll.Add(job2); coll.Add(job3); coll.Add(job4); coll.Add(job5);
    bool actual = coll.Contains(job1.Id);
    Assert.IsTrue(actual);
}

[TestMethod]
public void Contains_job_not_in_collection()
{
    uint capacity = 5;
    IJobCollection coll = new JobCollection(capacity);
    coll.Add(job1); coll.Add(job2); coll.Add(job3); coll.Add(job4); coll.Add(job5);
    bool actual = coll.Contains(job6.Id);
    Assert.IsFalse(actual);
}
```

Figure 11: Unit testing for Contains

5.2.3 Find()

```
[TestMethod]
public void Find_job_not_contained()
{
    coll.Add(job1); coll.Add(job2); coll.Add(job3); coll.Add(job4); coll.Add(job5);
    Assert.IsNull(coll.Find(job6.Id));
}

[TestMethod]
public void Find_job_contained()
{
    coll.Add(job1); coll.Add(job2); coll.Add(job3); coll.Add(job4); coll.Add(job5);
    Assert.IsTrue((coll.Find(job5.Id) == job5));
}
```

Figure 12: Unit testing for Find

5.2.4 Remove()

```
[TestMethod]
public void Remove_unsuccessful()
{
    coll.Add(job1); coll.Add(job2); coll.Add(job3); coll.Add(job4); coll.Add(job5);
    uint previousCount = coll.Count;
    Assert.IsTrue((coll.Remove(job6.Id) == false) && (coll.Count== previousCount) );
}

[TestMethod]
public void Remove_successful()
{
    coll.Add(job1); coll.Add(job2); coll.Add(job3); coll.Add(job4); coll.Add(job5);
    uint previousCount = coll.Count;
    // Remove returns true, count decrements by one, and the removed job
    // can no longer be found
    Assert.IsTrue((coll.Remove(job3.Id) == true)
        && (coll.Count == previousCount-1)
        && (coll.Find(job3.Id) == null));
}
```

Figure 13: Unit testing for Remove

5.2.5 ToArray()

```
[TestMethod]
public void ToArray_successful()
{
    coll.Add(job1); coll.Add(job2); coll.Add(job3); coll.Add(job4); coll.Add(job5);
    // We know that collection contains jobs in the order [job1, job2, job3, job4, job5]
    // As we cannot look at the jobs parameter, we need to forcefully create
    // an IJob[] array with these jobs
    IJob[] actual_Copy = new IJob[5]{job1,job2,job3,job4,job5};

    // IJob[] array from ToArray() method
    IJob[] coll_ToArray = coll.ToArray();

    bool ToArrayCorrect = Enumerable.SequenceEqual(actual_Copy, coll_ToArray);
    Assert.IsTrue(ToArrayCorrect);
}
```

Figure 14: Unit testing for ToArray

5.3 Scheduler ADT test data, results and code

5.3.1 FirstComeFirstServed

```
Input Data
Job(jobId: 1, timeReceived: 525, executionTime: 1, priority: 1)
Job(jobId: 2, timeReceived: 412, executionTime: 1, priority: 1)
Job(jobId: 3, timeReceived: 823, executionTime: 1, priority: 1)
Job(jobId: 4, timeReceived: 789, executionTime: 1, priority: 1)
Job(jobId: 5, timeReceived: 431, executionTime: 1, priority: 1)
Job(jobId: 6, timeReceived: 392, executionTime: 1, priority: 1)
Job(jobId: 7, timeReceived: 948, executionTime: 1, priority: 1)

Expected results
Job(jobId: 6, timeReceived: 392, executionTime: 1, priority: 1)
Job(jobId: 2, timeReceived: 412, executionTime: 1, priority: 1)
Job(jobId: 5, timeReceived: 431, executionTime: 1, priority: 1)
Job(jobId: 1, timeReceived: 525, executionTime: 1, priority: 1)
Job(jobId: 4, timeReceived: 789, executionTime: 1, priority: 1)
Job(jobId: 3, timeReceived: 823, executionTime: 1, priority: 1)
Job(jobId: 7, timeReceived: 948, executionTime: 1, priority: 1)

Actual Results
Job(jobId: 6, timeReceived: 392, executionTime: 1, priority: 1)
Job(jobId: 2, timeReceived: 412, executionTime: 1, priority: 1)
Job(jobId: 5, timeReceived: 431, executionTime: 1, priority: 1)
Job(jobId: 1, timeReceived: 525, executionTime: 1, priority: 1)
Job(jobId: 4, timeReceived: 789, executionTime: 1, priority: 1)
Job(jobId: 3, timeReceived: 823, executionTime: 1, priority: 1)
Job(jobId: 7, timeReceived: 948, executionTime: 1, priority: 1)
```

Figure 15: Unit testing for unique_jobs_in_a_full_collection

```
Input Data
Job(jobId: 1, timeReceived: 525, executionTime: 1, priority: 1)
Job(jobId: 2, timeReceived: 412, executionTime: 1, priority: 1)
Job(jobId: 3, timeReceived: 823, executionTime: 1, priority: 1)
Job(jobId: 4, timeReceived: 789, executionTime: 1, priority: 1)
Job(jobId: 5, timeReceived: 431, executionTime: 1, priority: 1)
Job(jobId: 6, timeReceived: 392, executionTime: 1, priority: 1)
Job(jobId: 7, timeReceived: 948, executionTime: 1, priority: 1)

Expected results
Job(jobId: 6, timeReceived: 392, executionTime: 1, priority: 1)
Job(jobId: 2, timeReceived: 412, executionTime: 1, priority: 1)
Job(jobId: 5, timeReceived: 431, executionTime: 1, priority: 1)
Job(jobId: 1, timeReceived: 525, executionTime: 1, priority: 1)
Job(jobId: 4, timeReceived: 789, executionTime: 1, priority: 1)
Job(jobId: 3, timeReceived: 823, executionTime: 1, priority: 1)
Job(jobId: 7, timeReceived: 948, executionTime: 1, priority: 1)

Actual Results
Job(jobId: 6, timeReceived: 392, executionTime: 1, priority: 1)
Job(jobId: 2, timeReceived: 412, executionTime: 1, priority: 1)
Job(jobId: 5, timeReceived: 431, executionTime: 1, priority: 1)
Job(jobId: 1, timeReceived: 525, executionTime: 1, priority: 1)
Job(jobId: 4, timeReceived: 789, executionTime: 1, priority: 1)
Job(jobId: 3, timeReceived: 823, executionTime: 1, priority: 1)
Job(jobId: 7, timeReceived: 948, executionTime: 1, priority: 1)
```

Figure 16: Unit testing for unique_jobs_in_a_partial_collection

```
Input Data
Job(jobId: 1, timeReceived: 525, executionTime: 1, priority: 1)
Job(jobId: 8, timeReceived: 431, executionTime: 1, priority: 5)
Job(jobId: 3, timeReceived: 823, executionTime: 1, priority: 1)
Job(jobId: 4, timeReceived: 789, executionTime: 1, priority: 1)
Job(jobId: 5, timeReceived: 431, executionTime: 1, priority: 1)
Job(jobId: 6, timeReceived: 392, executionTime: 1, priority: 1)
Job(jobId: 9, timeReceived: 525, executionTime: 1, priority: 1)

Expected results
Job(jobId: 6, timeReceived: 392, executionTime: 1, priority: 1)
Job(jobId: 8, timeReceived: 431, executionTime: 1, priority: 5)
Job(jobId: 5, timeReceived: 431, executionTime: 1, priority: 1)
Job(jobId: 1, timeReceived: 525, executionTime: 1, priority: 1)
Job(jobId: 9, timeReceived: 525, executionTime: 1, priority: 1)
Job(jobId: 4, timeReceived: 789, executionTime: 1, priority: 1)
Job(jobId: 3, timeReceived: 823, executionTime: 1, priority: 1)

Actual Results
Job(jobId: 6, timeReceived: 392, executionTime: 1, priority: 1)
Job(jobId: 8, timeReceived: 431, executionTime: 1, priority: 5)
Job(jobId: 5, timeReceived: 431, executionTime: 1, priority: 1)
Job(jobId: 1, timeReceived: 525, executionTime: 1, priority: 1)
Job(jobId: 9, timeReceived: 525, executionTime: 1, priority: 1)
Job(jobId: 4, timeReceived: 789, executionTime: 1, priority: 1)
Job(jobId: 3, timeReceived: 823, executionTime: 1, priority: 1)
```

Figure 17: Unit testing for common_jobs_in_a_full_collection

```
Input Data
Job(jobId: 1, timeReceived: 525, executionTime: 1, priority: 1)
Job(jobId: 8, timeReceived: 431, executionTime: 1, priority: 5)
Job(jobId: 4, timeReceived: 789, executionTime: 1, priority: 1)
Job(jobId: 5, timeReceived: 431, executionTime: 1, priority: 1)
Job(jobId: 9, timeReceived: 525, executionTime: 1, priority: 1)


Expected results
Job(jobId: 8, timeReceived: 431, executionTime: 1, priority: 5)
Job(jobId: 5, timeReceived: 431, executionTime: 1, priority: 1)
Job(jobId: 1, timeReceived: 525, executionTime: 1, priority: 1)
Job(jobId: 9, timeReceived: 525, executionTime: 1, priority: 1)
Job(jobId: 4, timeReceived: 789, executionTime: 1, priority: 1)


Actual Results
Job(jobId: 8, timeReceived: 431, executionTime: 1, priority: 5)
Job(jobId: 5, timeReceived: 431, executionTime: 1, priority: 1)
Job(jobId: 1, timeReceived: 525, executionTime: 1, priority: 1)
Job(jobId: 9, timeReceived: 525, executionTime: 1, priority: 1)
Job(jobId: 4, timeReceived: 789, executionTime: 1, priority: 1)
```

Figure 18: Unit testing for common_jobs_in_a_partial_collection

```
Input Data
Job(jobId: 1, timeReceived: 525, executionTime: 1, priority: 1)

Expected results
Job(jobId: 1, timeReceived: 525, executionTime: 1, priority: 1)

Actual Results
Job(jobId: 1, timeReceived: 525, executionTime: 1, priority: 1)
```

Figure 19: Unit testing for collection_capacity_one

5.3.2 Shortest Job First

```
Input Data
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 1)
Job(jobId: 2, timeReceived: 1, executionTime: 41, priority: 1)
Job(jobId: 3, timeReceived: 1, executionTime: 82, priority: 1)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 1)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 1)
Job(jobId: 6, timeReceived: 1, executionTime: 39, priority: 1)
Job(jobId: 7, timeReceived: 1, executionTime: 94, priority: 1)

Expected results
Job(jobId: 6, timeReceived: 1, executionTime: 39, priority: 1)
Job(jobId: 2, timeReceived: 1, executionTime: 41, priority: 1)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 1)
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 1)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 1)
Job(jobId: 3, timeReceived: 1, executionTime: 82, priority: 1)
Job(jobId: 7, timeReceived: 1, executionTime: 94, priority: 1)

Actual Results
Job(jobId: 6, timeReceived: 1, executionTime: 39, priority: 1)
Job(jobId: 2, timeReceived: 1, executionTime: 41, priority: 1)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 1)
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 1)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 1)
Job(jobId: 3, timeReceived: 1, executionTime: 82, priority: 1)
Job(jobId: 7, timeReceived: 1, executionTime: 94, priority: 1)
```

Figure 20: Unit testing for unique_jobs_in_a_full_collection

```
Input Data
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 1)
Job(jobId: 2, timeReceived: 1, executionTime: 41, priority: 1)
Job(jobId: 3, timeReceived: 1, executionTime: 82, priority: 1)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 1)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 1)



Expected results
Job(jobId: 2, timeReceived: 1, executionTime: 41, priority: 1)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 1)
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 1)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 1)
Job(jobId: 3, timeReceived: 1, executionTime: 82, priority: 1)



Actual Results
Job(jobId: 2, timeReceived: 1, executionTime: 41, priority: 1)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 1)
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 1)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 1)
Job(jobId: 3, timeReceived: 1, executionTime: 82, priority: 1)
```

Figure 21: Unit testing for unique_jobs_in_a_partial_collection

```
Input Data
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 1)
Job(jobId: 8, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 3, timeReceived: 1, executionTime: 82, priority: 1)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 1)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 1)
Job(jobId: 6, timeReceived: 1, executionTime: 39, priority: 1)
Job(jobId: 9, timeReceived: 1, executionTime: 52, priority: 1)

Expected results
Job(jobId: 6, timeReceived: 1, executionTime: 39, priority: 1)
Job(jobId: 8, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 1)
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 1)
Job(jobId: 9, timeReceived: 1, executionTime: 52, priority: 1)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 1)
Job(jobId: 3, timeReceived: 1, executionTime: 82, priority: 1)

Actual Results
Job(jobId: 6, timeReceived: 1, executionTime: 39, priority: 1)
Job(jobId: 8, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 1)
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 1)
Job(jobId: 9, timeReceived: 1, executionTime: 52, priority: 1)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 1)
Job(jobId: 3, timeReceived: 1, executionTime: 82, priority: 1)
```

Figure 22: Unit testing for common_jobs_in_a_full_collection

```
Input Data
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 1)
Job(jobId: 8, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 1)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 1)
Job(jobId: 9, timeReceived: 1, executionTime: 52, priority: 1)



Expected results
Job(jobId: 8, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 1)
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 1)
Job(jobId: 9, timeReceived: 1, executionTime: 52, priority: 1)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 1)



Actual Results
Job(jobId: 8, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 1)
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 1)
Job(jobId: 9, timeReceived: 1, executionTime: 52, priority: 1)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 1)
```

Figure 23: Unit testing for common_jobs_in_a_partial_collection

```
Input Data
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 1)



Expected results
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 1)



Actual Results
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 1)
```

Figure 24: Unit testing for collection_capacity_one

5.3.3 Priority

```
Input Data
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 5)
Job(jobId: 2, timeReceived: 1, executionTime: 41, priority: 7)
Job(jobId: 3, timeReceived: 1, executionTime: 82, priority: 2)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 3)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 6, timeReceived: 1, executionTime: 39, priority: 9)
Job(jobId: 7, timeReceived: 1, executionTime: 94, priority: 1)

Expected results
Job(jobId: 6, timeReceived: 1, executionTime: 39, priority: 9)
Job(jobId: 2, timeReceived: 1, executionTime: 41, priority: 7)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 5)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 3)
Job(jobId: 3, timeReceived: 1, executionTime: 82, priority: 2)
Job(jobId: 7, timeReceived: 1, executionTime: 94, priority: 1)

Actual Results
Job(jobId: 6, timeReceived: 1, executionTime: 39, priority: 9)
Job(jobId: 2, timeReceived: 1, executionTime: 41, priority: 7)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 5)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 3)
Job(jobId: 3, timeReceived: 1, executionTime: 82, priority: 2)
Job(jobId: 7, timeReceived: 1, executionTime: 94, priority: 1)
```

Figure 25: Unit testing for unique_jobs_in_a_full_collection

```
Input Data
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 5)
Job(jobId: 2, timeReceived: 1, executionTime: 41, priority: 7)
Job(jobId: 3, timeReceived: 1, executionTime: 82, priority: 2)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 3)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 5)


Expected results
Job(jobId: 2, timeReceived: 1, executionTime: 41, priority: 7)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 5)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 3)
Job(jobId: 3, timeReceived: 1, executionTime: 82, priority: 2)


Actual Results
Job(jobId: 2, timeReceived: 1, executionTime: 41, priority: 7)
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 5)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 3)
Job(jobId: 3, timeReceived: 1, executionTime: 82, priority: 2)
```

Figure 26: Unit testing for unique_jobs_in_a_partial_collection

```
Input Data
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 5)
Job(jobId: 8, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 3, timeReceived: 1, executionTime: 82, priority: 2)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 3)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 6, timeReceived: 1, executionTime: 39, priority: 9)
Job(jobId: 9, timeReceived: 1, executionTime: 52, priority: 4)

Expected results
Job(jobId: 6, timeReceived: 1, executionTime: 39, priority: 9)
Job(jobId: 8, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 5)
Job(jobId: 9, timeReceived: 1, executionTime: 52, priority: 4)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 3)
Job(jobId: 3, timeReceived: 1, executionTime: 82, priority: 2)

Actual Results
Job(jobId: 6, timeReceived: 1, executionTime: 39, priority: 9)
Job(jobId: 8, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 5)
Job(jobId: 9, timeReceived: 1, executionTime: 52, priority: 4)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 3)
Job(jobId: 3, timeReceived: 1, executionTime: 82, priority: 2)
```

Figure 27: Unit testing for common_jobs_in_a_full_collection

```
Input Data
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 5)
Job(jobId: 8, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 3)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 9, timeReceived: 1, executionTime: 52, priority: 4)


Expected results
Job(jobId: 8, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 5)
Job(jobId: 9, timeReceived: 1, executionTime: 52, priority: 4)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 3)


Actual Results
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 5)
Job(jobId: 8, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 5, timeReceived: 1, executionTime: 43, priority: 5)
Job(jobId: 9, timeReceived: 1, executionTime: 52, priority: 4)
Job(jobId: 4, timeReceived: 1, executionTime: 78, priority: 3)
```

Figure 28: Unit testing for common_jobs_in_a_partial_collection

```
Input Data
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 5)




Expected results
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 5)




Actual Results
Job(jobId: 1, timeReceived: 1, executionTime: 52, priority: 5)
```

Figure 29: Unit testing for collection_capacity_one