

CAB301: Algorithms and Complexity – Assignment 1

Ryan Indrananda - n10852565

Queensland University of Technology

Table of Contents

1. Algorithm Design	3
2. Algorithm Analysis.....	5
3. Algorithm Testing.....	8
a. Test Plan	8
b. Test Data and Results	11
4. References.....	23

1. Algorithm Design

The provided Abstract Data Types (ADT) outline multiple methods in order to correctly implement three commonly used scheduling strategies. All of these strategies were implemented using selection sort algorithms. This algorithm essentially works in the following way (Tang, 2023):

1. Search for the minimum data item in a collection of n elements.
2. Swap the minimum data item with the first data item ($n = 1$) in the collection.
3. Repeat steps 1 and 2 for the last $n-1$ elements.

We can find the implementations of these scheduling strategies for the collection of computing jobs under the Scheduler ADT:

The First Come First Served (FCFS) method sorts a job collection in non-descending order of when the job was received (timeReceived field). It returns an array which contains all of the jobs within the scheduler and its elements, with the time received sorted in non-descending order. The pseudocode of this algorithm is:

ALGORITHM *FirstComeFirstServed()*

```
// Sorts the jobs in non-descending order of time received using selection sort
// Output: Array A[0..n - 1] sorted in non-descending order of time received
A ← this.Jobs Array
arrayLength ← A Length
for i ← 0 to arrayLength - 2 do
    min ← i
    for j ← i + 1 to arrayLength - 1 do
        if A[j].TimeReceived < A[min].TimeReceived    min ← j
    temp ← A[min]
    A[min] ← A[i]
    A[i] ← temp
return A
```

$A \leftarrow \text{this.Jobs Array}$ is done by using the `ToArray` method implemented in the `JobCollection` ADT. $\text{arrayLength} \leftarrow A \text{ Length}$ is done by doing `A.Length` in C#. $\text{temp} \leftarrow A[\text{min}]$ until $A[i] \leftarrow \text{temp}$ can be shortened to swap $A[j]$ and $A[\text{min}]$, but for the purpose of further analysis we will keep it unshortened. These apply for all the following scheduling strategies as well.

The Shortest Job First (SJF) method sorts a job collection in non-descending order of how long the job will be executed for (`executionTime` field). It returns an array which contains all of the jobs within the scheduler and it's elements, with the execution time sorted in non-descending order. The pseudocode of this algorithm is:

ALGORITHM *ShortestJobFirst()*

// Sorts the jobs in non-descending order of execution time using selection sort

// Output: Array $A[0..n - 1]$ sorted in non-descending order of execution time

$A \leftarrow \text{this.Jobs Array}$

$\text{arrayLength} \leftarrow A \text{ Length}$

for $i \leftarrow 0$ **to** $\text{arrayLength} - 2$ **do**

$\text{min} \leftarrow i$

for $j \leftarrow i + 1$ **to** $\text{arrayLength} - 1$ **do**

if $A[j].\text{ExecutionTime} < A[\text{min}].\text{ExecutionTime}$ $\text{min} \leftarrow j$

$\text{temp} \leftarrow A[\text{min}]$

$A[\text{min}] \leftarrow A[i]$

$A[i] \leftarrow \text{temp}$

return A

The Priority method sorts a job collection in non-ascending order of priority, or how important the job is (Priority field). It returns an array which contains all of the jobs within the scheduler and its elements, with the time received sorted in non-descending order. The pseudocode of this algorithm is:

ALGORITHM *Priority()*

```
// Sorts the jobs in non-ascending order of priority using selection sort
// Output: Array A[0..n - 1] sorted in non-ascending order of priority
A ← this.Jobs Array
arrayLength ← A Length
for i ← 0 to arrayLength - 2 do
    min ← i
    for j ← i + 1 to arrayLength - 1 do
        if A[j].Priority > A[min].Priority    min ← j
    temp ← A[min]
    A[min] ← A[i]
    A[i] ← temp
return A
```

2. Algorithm Analysis

We may then analyse these algorithms to determine the required resources of said algorithms in relation to the amount of data the algorithm processes. To theoretically analyse the time efficiency of these algorithms, we must consider size of the input and basic operation of the algorithms, then create a summation formula for the number of times the aforementioned basic operation is performed in the worst scenario, and finally solve the summation formula using arithmetic rules, resulting in an exact efficiency equation or identification of the algorithm's efficiency class utilizing Big-O notation (Tang, 2023).

The size of the input within all the algorithms is the number of items within the jobs array, or the length *arrayLength* of A.

The basic operation of the algorithms is number of key comparisons ‘ $A[j].TimeReceived < A[min].TimeReceived$ ’, ‘ $A[j].ExecutionTime < A[min].ExecutionTime$ ’, and ‘ $A[j].Priority > A[min].Priority$ ’ for FCFS, SJF, and Priority respectively. As the algorithms all make use of selection sort, an analysis of one algorithm theoretically applies for all the algorithms. For the purposes of the efficiency analysis, we will use the FCFS algorithm.

ALGORITHM *FirstComeFirstServed()*

```
// Sorts the jobs in non-descending order of time received using selection sort
// Output: Array A[0..n – 1] sorted in non-descending order of time received

A ← this.Jobs Array          c1
arrayLength ← A Length       c2
for i ← 0 to arrayLength – 2 do
    min ← i                   c3
    for j ← i + 1 to arrayLength – 1 do
        if A[j].TimeReceived < A[min].TimeReceived c4    min ← j    c5
    temp ← A[min]             c6
    A[min] ← A[i]             c7
    A[i] ← temp               c8
return A                     c9
```

As selection sort algorithms always perform the same number of key comparisons in all cases, the computation time of this algorithm in all cases may be approximated by the following function:

$$T(n) = c1 + c2 + (c3 + (c4 + c5) + c6 + c7 + c8) + c9$$

where $c1$, $c2$, $c3$, $c4$, $c5$, $c6$, $c7$, $c8$, and $c9$ are machine dependent, and $n = arrayLength$. If we assume that they all take 2 microseconds to process for a particular machine, then:

$$T(n) = 2 + 2 + (2 + (2 + 2) * (arrayLength - 1) + 2 + 2 + 2) * (arrayLength - 2) + 2$$

Thus, when $arrayLength = 10$,

$$T(10) = 2 + 2 + (2 + (2 + 2) * (10 - 1) + 2 + 2 + 2) * (10 - 2) + 2$$

$$T(10) = 4 + (2 + 4 * 9 + 2 + 2 + 2) * 8 + 2$$

$$T(10) = 4 + (2 + 36 + 2 + 2 + 2) * 8 + 2$$

$$T(10) = 4 + 44 * 8 + 2$$

$$T(10) = 360 \text{ microseconds}$$

When $arrayLength = 100$,

$$T(100) = 2 + 2 + (2 + (2 + 2) * (100 - 1) + 2 + 2 + 2) * (100 - 2) + 2$$

$$T(100) = 4 + (2 + 4 * 99 + 2 + 2 + 2) * 98 + 2$$

$$T(100) = 4 + (2 + 396 + 2 + 2 + 2) * 98 + 2$$

$$T(100) = 4 + 404 * 98 + 2$$

$$T(100) = 39,598 \text{ microseconds}$$

When $arrayLength = 1000000$,

$$T(1000000) = 2 + 2 + (2 + (2 + 2) * (1000000 - 1) + 2 + 2 + 2) * (1000000 - 2) + 2$$

$$T(1000000) = 4 + (2 + 4 * 999999 + 2 + 2 + 2) * 999998 + 2$$

$$T(1000000) = 4 + (2 + 3,999,996 + 2 + 2 + 2) * 999998 + 2$$

$$T(1000000) = 4 + 4,000,004 * 999998 + 2$$

$$T(1000000) = 3,999,995,999,998 \text{ microseconds}$$

The basic operation only occurs once in the innermost loop body, which occurs for j equal to $i + 1$ to $arrayLength - 1$, inclusive (Tang, 2023):

$$\sum_{j=i+1}^{arrayLength-1} 1 = (arrayLength - 1) - (i + 1) + 1 = arrayLength - 1 - i$$

The outermost loop occurs for i equal to 0 to $arrayLength - 2$, inclusive:

$$\begin{aligned}
 & \sum_{i=0}^{arrayLength-2} (arrayLength - 1 - i) \\
 &= (arrayLength - 1) + (arrayLength - 2) + \dots + 1 = \sum_{i=1}^{arrayLength-1} i \\
 &= \frac{arrayLength(arrayLength - 1)}{2}
 \end{aligned}$$

Therefore, the algorithm's worst-case efficiency is:

$$\begin{aligned}
 C_{\text{worst}}(arrayLength) &= \frac{(arrayLength - 1)arrayLength}{2} \\
 &= \frac{1}{2}arrayLength^2 - \frac{1}{2}arrayLength \in O(arrayLength^2) \\
 &= O(n^2)
 \end{aligned}$$

3. Algorithm Testing

a. Test Plan

All testing will be done manually through the creation of a Program.cs file and Main method. The file begins by instantiating JobCollection and Scheduler.

```
// New instances of JobCollection and Scheduler
JobCollection aCollection = new JobCollection(10);
Scheduler scheduler = new(aCollection);
```

The input data we are using is as follows:

```
// Create new jobs
IJob job = new Job(523, 90, 23, 8);
IJob job2 = new Job(966, 46, 15, 2);
IJob job3 = new Job(26, 11, 50, 5);
IJob job4 = new Job(553, 35, 12, 1);
IJob job5 = new Job(346, 79, 6, 5);
IJob job6 = new Job(560, 95, 47, 2);
IJob job7 = new Job(132, 13, 18, 8);
IJob job8 = new Job(741, 92, 37, 9);
IJob job9 = new Job(267, 11, 50, 5);
IJob job10 = new Job(583, 97, 22, 2);
```



```
Input data:
Job(jobId: 523, timeReceived: 90, executionTime: 23, priority: 8)
Job(jobId: 966, timeReceived: 46, executionTime: 15, priority: 2)
Job(jobId: 26, timeReceived: 11, executionTime: 50, priority: 5)
Job(jobId: 553, timeReceived: 35, executionTime: 12, priority: 1)
Job(jobId: 346, timeReceived: 79, executionTime: 6, priority: 5)
Job(jobId: 560, timeReceived: 95, executionTime: 47, priority: 2)
Job(jobId: 132, timeReceived: 13, executionTime: 18, priority: 8)
Job(jobId: 741, timeReceived: 92, executionTime: 37, priority: 9)
Job(jobId: 267, timeReceived: 11, executionTime: 50, priority: 5)
Job(jobId: 583, timeReceived: 97, executionTime: 22, priority: 2)

Number of jobs in the collection: 10
Capacity of collection: 10
```

We will test to see if each scheduling strategy behaves as intended. FCFS should sort the collection in non-descending order of time received, SJF should sort the collection in non-descending order of execution time, and Priority should sort the collection in non-ascending order of priority. As each scheduler returns an array, we will make use of a helper method called 'printArray' which will print all the elements of the array.

```
static void printArray(IJob[] A)
{
    for (int i = 0; i < A.Length; i++)
        Console.WriteLine(A[i]);
}
```

Under the Job ADT, which is used to define the fields that make up a job (their ID, time received, execution time and priority), we find methods ensuring the suitability of the values of these fields. For example, the IsValidId method ensures that a user-supplied job ID is valid, returning True if the supplied ID is between 1 and 999 inclusive, and returning False otherwise. The IsValidExecutionTime and IsTimeReceived methods return True if the supplied execution time and time received is greater than 0 non-inclusive, and returning False otherwise. Finally, the IsValidPriority method returns True if the supplied priority is between 1 and 9 inclusive, and False otherwise. To test that all these methods behave as intended, we will test for edge cases, specifically trying to add jobs with an ID of 1, 999, 1000, 0, and/or a time received or execution time of 0 and 1. Non-valid values will throw the user an ArgumentOutOfRangeException Exception.

Under the JobCollection ADT, representing a collection of computing jobs, we find various methods by which we may interact with said collection. It's invariants state that the capacity of a job collection should always be equal or greater than one, count should never be greater than capacity, and there are no duplicate jobs in the collection.

The Add method is used to populate the JobCollection with new jobs called within Program.cs. The condition for this method to run states that as long as the number of jobs does not exceed the capacity of the collection, the new job is not null, and the job does not already exist in the collection, then the job is added to the collection, the count of number of jobs increases by one, all other jobs in the collection remain unchanged, and the method returns True. Otherwise, the job is not added to the collection, the count and capacity of the collection remain unchanged, and the method returns False. To test for this, we will begin by populating the collection with jobs that have valid values of ID, time received, execution time, and priority defined under the Job ADT. If the method behaves as intended, it will populate the collection normally. Further on, we will test if the method will add a job that already exists in the population. We must also test that the method does not add a null job to the collection, and does not add any job to the collection if it has reached the specified capacity of the JobCollection instance. We want the method to return an error message to the user that they cannot add a pre-existing job, a null job, or any job if the collection is full, and continue to further tests gracefully as well as fulfilling the aforementioned post-conditions.

Next, the Contains method tells the user if a certain job exists within the collection by returning True only if the job does exist, otherwise it will return False if it does not. Either way, the jobs, count, and capacity of the collection all remain unchanged. To test for this, we will check the collection using the method for both jobs we know do and do not exist and see if it satisfies the post-conditions.

The Find method shows the user a copy of the job that they are looking for by calling a specified ID. If the job does not exist, then the method will return null, otherwise all aspects of the collection remain unchanged. To test for this, we will try to find jobs in the collection that both we know exist and do not exist and see if it satisfies the post-conditions.

The Remove method provides the user the functionality of removing a job of their choosing from the collection. It returns True if the job exists in the collection, removes the job from the collection and deducts the count by one. Otherwise, it returns False if the job does not exist and all aspects of the collection remain unchanged. To test this, we will try to remove jobs in the collection that both we know exist and do not exist and check to see if it satisfies the post-conditions.

Lastly, we need to implement a ToArray method which aids in displaying the entire collection as an array to the user. It promises to return an array of the collection that contains

the same jobs as the collection. This method will continuously be used throughout Program.cs, specifically whenever the collection is edited using the Add or Remove methods. Paired with the helper printArray method in Program.cs, we can check if the returned array has the correct jobs, job count, and capacity as expected.

The test will be done on a 64-bit Windows 11 Home version 22H2 build 22621.1413 operating system laptop, with an AMD Ryzen 7 4800HS 2.9 GHz CPU, NVIDIA GeForce GTX 1660 Ti Max-Q dGPU, AMD Radeon(TM) iGPU, and 16GB of RAM. It is connected via Ethernet. All C# code implementations were done and will be ran in Microsoft Visual Studio Community 2022 (64-bit) version 17.4.4 with .NET Framework version 4.8.09032.

b. Test Data and Results

Test 1: Add new jobs to the JobCollection instance with the 'Add' and 'ToArray' methods. See section 3a. Test Plan for the corresponding jobs we are adding.

```
// Display current count and capacity of jobs in the collection.
Console.WriteLine("Number of jobs in the collection: " + aCollection.Count);
Console.WriteLine("Capacity of collection: " + aCollection.Capacity + "\n");

// Populate JobCollection with new jobs
aCollection.Add(job);
aCollection.Add(job2);
aCollection.Add(job3);
aCollection.Add(job4);
aCollection.Add(job5);
aCollection.Add(job6);
aCollection.Add(job7);
aCollection.Add(job8);
aCollection.Add(job9);
aCollection.Add(job10);

// Convert JobCollection into an IJob array called 'jobs' using the ToArray method
IJob[] jobs = aCollection.ToArray();

// Print jobs array to console and check if all jobs are added correctly.
Console.WriteLine("Input data:");
printArray(jobs);
Console.WriteLine("\nNumber of jobs in the collection: " + aCollection.Count);
Console.WriteLine("Capacity of collection: " + aCollection.Capacity + "\n");
```

Result:

```
Test 1: Add new jobs to the JobCollection instance with the 'Add' and 'ToArray' methods.

Number of jobs in the collection: 0
Capacity of collection: 10

Job 523 does not currently exist in the collection.
Job 523 has been added! There are 1 jobs in the collection.

Job 966 does not currently exist in the collection.
Job 966 has been added! There are 2 jobs in the collection.

Job 26 does not currently exist in the collection.
Job 26 has been added! There are 3 jobs in the collection.

Job 553 does not currently exist in the collection.
Job 553 has been added! There are 4 jobs in the collection.

Job 346 does not currently exist in the collection.
Job 346 has been added! There are 5 jobs in the collection.

Job 560 does not currently exist in the collection.
Job 560 has been added! There are 6 jobs in the collection.

Job 132 does not currently exist in the collection.
Job 132 has been added! There are 7 jobs in the collection.

Job 741 does not currently exist in the collection.
Job 741 has been added! There are 8 jobs in the collection.

Job 267 does not currently exist in the collection.
Job 267 has been added! There are 9 jobs in the collection.

Job 583 does not currently exist in the collection.
Job 583 has been added! There are 10 jobs in the collection.

Input data:
Job(jobId: 523, timeReceived: 90, executionTime: 23, priority: 8)
Job(jobId: 966, timeReceived: 46, executionTime: 15, priority: 2)
Job(jobId: 26, timeReceived: 11, executionTime: 50, priority: 5)
Job(jobId: 553, timeReceived: 35, executionTime: 12, priority: 1)
Job(jobId: 346, timeReceived: 79, executionTime: 6, priority: 5)
Job(jobId: 560, timeReceived: 95, executionTime: 47, priority: 2)
Job(jobId: 132, timeReceived: 13, executionTime: 18, priority: 8)
Job(jobId: 741, timeReceived: 92, executionTime: 37, priority: 9)
Job(jobId: 267, timeReceived: 11, executionTime: 50, priority: 5)
Job(jobId: 583, timeReceived: 97, executionTime: 22, priority: 2)

Number of jobs in the collection: 10
Capacity of collection: 10

-----
```

The Add and ToArray methods work as intended. The jobs were successfully added to the collection, count updates as intended, and capacity stays the same.

Test 2: First Come First Served Sort

```
// Apply algorithm to collection and assign to new IJob array.
IJob[] firstComeFirstServedSort = scheduler.FirstComeFirstServed();

// Print the sorted array to the console and check if it functions as intended.
Console.WriteLine("Schedule by First Come First Served:");
printArray(firstComeFirstServedSort);
Console.WriteLine("-----");
```

Result:

```
Test 2: First Come First Served Sort

Schedule by First Come First Served:
Job(jobId: 26, timeReceived: 11, executionTime: 50, priority: 5)
Job(jobId: 267, timeReceived: 11, executionTime: 50, priority: 5)
Job(jobId: 132, timeReceived: 13, executionTime: 18, priority: 8)
Job(jobId: 553, timeReceived: 35, executionTime: 12, priority: 1)
Job(jobId: 966, timeReceived: 46, executionTime: 15, priority: 2)
Job(jobId: 346, timeReceived: 79, executionTime: 6, priority: 5)
Job(jobId: 523, timeReceived: 90, executionTime: 23, priority: 8)
Job(jobId: 741, timeReceived: 92, executionTime: 37, priority: 9)
Job(jobId: 560, timeReceived: 95, executionTime: 47, priority: 2)
Job(jobId: 583, timeReceived: 97, executionTime: 22, priority: 2)
-----
```

FCFS behaves as intended. The timeReceived field is sorted in non-descending order.

Test 3: Shortest Job First Sort

```
// Apply algorithm to collection and assign to new IJob array.
IJob[] shortestSort = scheduler.ShortestJobFirst();

// Print sorted array to the console.
Console.WriteLine("Schedule by Shortest Job First:");
printArray(shortestSort);
Console.WriteLine("-----");
```

Result:

```
Test 3: Shortest Job First Sort

Schedule by Shortest Job First:
Job(jobId: 346, timeReceived: 79, executionTime: 6, priority: 5)
Job(jobId: 553, timeReceived: 35, executionTime: 12, priority: 1)
Job(jobId: 966, timeReceived: 46, executionTime: 15, priority: 2)
Job(jobId: 132, timeReceived: 13, executionTime: 18, priority: 8)
Job(jobId: 583, timeReceived: 97, executionTime: 22, priority: 2)
Job(jobId: 523, timeReceived: 90, executionTime: 23, priority: 8)
Job(jobId: 741, timeReceived: 92, executionTime: 37, priority: 9)
Job(jobId: 560, timeReceived: 95, executionTime: 47, priority: 2)
Job(jobId: 267, timeReceived: 11, executionTime: 50, priority: 5)
Job(jobId: 26, timeReceived: 11, executionTime: 50, priority: 5)
-----
```

SJF behaves as intended. The executionTime field is sorted in non-descending order.

Test 4: Priority Sort

```
// Apply algorithm to collection and assign to new IJob array.
IJob[] prioritySort = scheduler.Priority();

// Print sorted array to the console.
Console.WriteLine("Schedule by Priority:");
printArray(prioritySort);
Console.WriteLine("-----");
```

Result:

```
Test 4: Priority Sort

Schedule by Priority:
Job(jobId: 741, timeReceived: 92, executionTime: 37, priority: 9)
Job(jobId: 132, timeReceived: 13, executionTime: 18, priority: 8)
Job(jobId: 523, timeReceived: 90, executionTime: 23, priority: 8)
Job(jobId: 346, timeReceived: 79, executionTime: 6, priority: 5)
Job(jobId: 26, timeReceived: 11, executionTime: 50, priority: 5)
Job(jobId: 267, timeReceived: 11, executionTime: 50, priority: 5)
Job(jobId: 966, timeReceived: 46, executionTime: 15, priority: 2)
Job(jobId: 560, timeReceived: 95, executionTime: 47, priority: 2)
Job(jobId: 583, timeReceived: 97, executionTime: 22, priority: 2)
Job(jobId: 553, timeReceived: 35, executionTime: 12, priority: 1)
-----
```

Priority behaves as intended. The priority field is sorted in non-ascending order.

Test 5: Remove jobs from the collection using the Remove method.

```
// Display current collection.
Console.WriteLine("Current collection:");
printArray(jobs);
Console.WriteLine();

/* Remove jobs from the collection based on job ID and assign new collection to jobs.
   In this test case, five of the existing ten jobs will be removed. */
aCollection.Remove(267);
aCollection.Remove(966);
aCollection.Remove(560);
aCollection.Remove(741);
aCollection.Remove(583);
jobs = aCollection.ToArray();

// Print new jobs array to console. Check if corresponding jobs are removed.
Console.WriteLine("\nNew collection:");
printArray(jobs);
Console.WriteLine("\nNumber of jobs in the collection: " + aCollection.Count);
Console.WriteLine("Capacity of collection: " + aCollection.Capacity + "\n");
```

Result:

```
Test 5: Remove jobs from the collection using 'Remove' method.

Current collection:
Job(jobId: 523, timeReceived: 90, executionTime: 23, priority: 8)
Job(jobId: 966, timeReceived: 46, executionTime: 15, priority: 2)
Job(jobId: 26, timeReceived: 11, executionTime: 50, priority: 5)
Job(jobId: 553, timeReceived: 35, executionTime: 12, priority: 1)
Job(jobId: 346, timeReceived: 79, executionTime: 6, priority: 5)
Job(jobId: 560, timeReceived: 95, executionTime: 47, priority: 2)
Job(jobId: 132, timeReceived: 13, executionTime: 18, priority: 8)
Job(jobId: 741, timeReceived: 92, executionTime: 37, priority: 9)
Job(jobId: 267, timeReceived: 11, executionTime: 50, priority: 5)
Job(jobId: 583, timeReceived: 97, executionTime: 22, priority: 2)

Job 267 has been removed!
Job 966 has been removed!
Job 560 has been removed!
Job 741 has been removed!
Job 583 has been removed!

New collection:
Job(jobId: 523, timeReceived: 90, executionTime: 23, priority: 8)
Job(jobId: 26, timeReceived: 11, executionTime: 50, priority: 5)
Job(jobId: 553, timeReceived: 35, executionTime: 12, priority: 1)
Job(jobId: 346, timeReceived: 79, executionTime: 6, priority: 5)
Job(jobId: 132, timeReceived: 13, executionTime: 18, priority: 8)

Number of jobs in the collection: 5
Capacity of collection: 10
-----
```

Remove method behaves as intended. The corresponding jobs were removed. Count updates correctly, and the capacity and other existing jobs remain unchanged.

Test 6: Removing a job that does not exist in the collection.

```
aCollection.Remove(999);
aCollection.Remove(519);
aCollection.Remove(44);

// Print jobs array to console. Collection, count, and capacity should all remain the same.
Console.WriteLine("\nCollection, count, and capacity stay the same:");
printArray(jobs);
Console.WriteLine("\nNumber of jobs in the collection: " + aCollection.Count);
Console.WriteLine("Capacity of collection: " + aCollection.Capacity + "\n");
```

Result:

```
Test 6: Removing a job that does not exist in the collection.

Job 999 does not exist in the collection and cannot be removed.
Job 519 does not exist in the collection and cannot be removed.
Job 44 does not exist in the collection and cannot be removed.

Collection, count, and capacity stay the same:
Job(jobId: 523, timeReceived: 90, executionTime: 23, priority: 8)
Job(jobId: 26, timeReceived: 11, executionTime: 50, priority: 5)
Job(jobId: 553, timeReceived: 35, executionTime: 12, priority: 1)
Job(jobId: 346, timeReceived: 79, executionTime: 6, priority: 5)
Job(jobId: 132, timeReceived: 13, executionTime: 18, priority: 8)

Number of jobs in the collection: 5
Capacity of collection: 10
```

Remove method when a job does not exist works as intended. An error message is displayed, and the collection, count, and capacity all remain unchanged.

Test 7: Functionality of 'Contains' method.

```
// Test if the method returns true if we pass job IDs we know exists in the job collection.
Console.WriteLine("Check if collection contains jobs we know exists (with ID):");
aCollection.Contains(346);
aCollection.Contains(26);
aCollection.Contains(132);

// Test if the method returns false if we pass job IDs we know do not exist in the job collection.
Console.WriteLine("\nCheck if collection contains jobs we know do not exist:");
aCollection.Contains(999);
aCollection.Contains(92);
aCollection.Contains(672);

Console.WriteLine("\nThe data remains unchanged:");
printArray(jobs);
Console.WriteLine("\nNumber of jobs in the collection: " + aCollection.Count);
Console.WriteLine("Capacity of collection: " + aCollection.Capacity + "\n");
```


Result:

```
Test 7: Functionality of 'Contains' method.

Check if collection contains jobs we know exists (with ID):
Job 346 exists in the collection.
Job 26 exists in the collection.
Job 132 exists in the collection.

Check if collection contains jobs we know do not exist:
Job 999 does not currently exist in the collection.
Job 92 does not currently exist in the collection.
Job 672 does not currently exist in the collection.

The data remains unchanged:
Job(jobId: 523, timeReceived: 90, executionTime: 23, priority: 8)
Job(jobId: 26, timeReceived: 11, executionTime: 50, priority: 5)
Job(jobId: 553, timeReceived: 35, executionTime: 12, priority: 1)
Job(jobId: 346, timeReceived: 79, executionTime: 6, priority: 5)
Job(jobId: 132, timeReceived: 13, executionTime: 18, priority: 8)

Number of jobs in the collection: 5
Capacity of collection: 10
```

Contains method behaves as intended. It correctly tells the user if a job does or does not exist in collection. The collection, count, and capacity also all remain unchanged.

Test 8: Find a job in the collection based on their job ID using the Find method.

```
// Find jobs that we know exist.
Console.WriteLine("Find and display jobs with IDs that exist in the collection:");
aCollection.Find(26);
aCollection.Find(346);
aCollection.Find(132);

// Find jobs that we know do not exist (to check if the null post-condition works).
Console.WriteLine("\nFind and display jobs with IDs that do not exist in the collection:");

aCollection.Find(999);
aCollection.Find(888);

Console.WriteLine("\nThe data remains unchanged:");
for (int i = 0; i < jobs.Length; i++)
    Console.WriteLine(jobs[i]);
Console.WriteLine();
Console.WriteLine("Number of jobs in the collection: " + aCollection.Count);
Console.WriteLine("Capacity of collection: " + aCollection.Capacity + "\n");
```

Result:

```
Find and display jobs with IDs that exist in the collection:
Job(jobId: 26, timeReceived: 11, executionTime: 50, priority: 5)
Job(jobId: 346, timeReceived: 79, executionTime: 6, priority: 5)
Job(jobId: 132, timeReceived: 13, executionTime: 18, priority: 8)

Find and display jobs with IDs that do not exist in the collection:
Job 999 does not exist.
Job 888 does not exist.

The data remains unchanged:
Job(jobId: 523, timeReceived: 90, executionTime: 23, priority: 8)
Job(jobId: 26, timeReceived: 11, executionTime: 50, priority: 5)
Job(jobId: 553, timeReceived: 35, executionTime: 12, priority: 1)
Job(jobId: 346, timeReceived: 79, executionTime: 6, priority: 5)
Job(jobId: 132, timeReceived: 13, executionTime: 18, priority: 8)

Number of jobs in the collection: 5
Capacity of collection: 10
```

Find behaves as intended. It correctly displays the job with the correct corresponding job ID. If it does not exist it returns a null and displays a message. The collection, count, and capacity all remain unchanged

Test 9: Adding jobs that already exists

```
IJob job11 = new Job(26, 58, 23, 3);
IJob job12 = new Job(346, 32, 69, 7);
IJob job13 = new Job(132, 40, 14, 9);
aCollection.Add(job11);
aCollection.Add(job12);
aCollection.Add(job13);
jobs = aCollection.ToArray();

Console.WriteLine();
Console.WriteLine("Collection remains unchanged: ");
printArray(jobs);
Console.WriteLine("\nNumber of jobs in the collection: " + aCollection.Count);
Console.WriteLine("Capacity of collection: " + aCollection.Capacity + "\n");
```

Result:

```
Test 9: Adding a job that already exists.

Job 26 exists in the collection.
Job 346 exists in the collection.
Job 132 exists in the collection.

Collection remains unchanged:
Job(jobId: 523, timeReceived: 90, executionTime: 23, priority: 8)
Job(jobId: 26, timeReceived: 11, executionTime: 50, priority: 5)
Job(jobId: 553, timeReceived: 35, executionTime: 12, priority: 1)
Job(jobId: 346, timeReceived: 79, executionTime: 6, priority: 5)
Job(jobId: 132, timeReceived: 13, executionTime: 18, priority: 8)

Number of jobs in the collection: 5
Capacity of collection: 10
```

Add's post-condition for pre-existing jobs behaves as intended. The job is not added, and the collection, count, and capacity remain unchanged.

Test 10: Adding a job that is null

```
IJob job14 = null;
aCollection.Add(job14);
Console.WriteLine("Collection remains unchanged: ");
printArray(jobs);
Console.WriteLine("\nNumber of jobs in the collection: " + aCollection.Count);
Console.WriteLine("Capacity of collection: " + aCollection.Capacity + "\n");
```

Result:

```
Test 10: Adding a job that is not null.

Job has not been added because it is null.

Collection remains unchanged:
Job(jobId: 523, timeReceived: 90, executionTime: 23, priority: 8)
Job(jobId: 26, timeReceived: 11, executionTime: 50, priority: 5)
Job(jobId: 553, timeReceived: 35, executionTime: 12, priority: 1)
Job(jobId: 346, timeReceived: 79, executionTime: 6, priority: 5)
Job(jobId: 132, timeReceived: 13, executionTime: 18, priority: 8)

Number of jobs in the collection: 5
Capacity of collection: 10
```

Add's post-condition for null jobs behaves as intended. The job is not added, and the collection, count, and capacity remain unchanged.

Test 11: Going above capacity of the collection

```
IJob job15 = new Job(235, 46, 19, 7);
IJob job16 = new Job(621, 36, 16, 3);
IJob job17 = new Job(153, 15, 6, 5);
IJob job18 = new Job(649, 34, 23, 8);
IJob job19 = new Job(953, 12, 34, 9);
IJob job20 = new Job(12, 7, 8, 2);
IJob job21 = new Job(219, 17, 6, 4);
IJob job22 = new Job(21, 46, 35, 6);
aCollection.Add(job15);
aCollection.Add(job16);
aCollection.Add(job17);
aCollection.Add(job18);
aCollection.Add(job19);
aCollection.Add(job20);
aCollection.Add(job21);
aCollection.Add(job22);

jobs = aCollection.ToArray();
printArray(jobs);
Console.WriteLine("\nNumber of jobs in the collection: " + aCollection.Count);
Console.WriteLine("Capacity of collection: " + aCollection.Capacity + "\n");
```

Result:

```
Test 11: Going above capacity of the collection.

Job 235 does not currently exist in the collection.
Job 235 has been added! There are 6 jobs in the collection.

Job 621 does not currently exist in the collection.
Job 621 has been added! There are 7 jobs in the collection.

Job 153 does not currently exist in the collection.
Job 153 has been added! There are 8 jobs in the collection.

Job 649 does not currently exist in the collection.
Job 649 has been added! There are 9 jobs in the collection.

Job 953 does not currently exist in the collection.
Job 953 has been added! There are 10 jobs in the collection.

Job 12 does not currently exist in the collection.
The collection is full! Please remove jobs before adding more.

Job 219 does not currently exist in the collection.
The collection is full! Please remove jobs before adding more.

Job 21 does not currently exist in the collection.
The collection is full! Please remove jobs before adding more.

Job(jobId: 523, timeReceived: 90, executionTime: 23, priority: 8)
Job(jobId: 26, timeReceived: 11, executionTime: 50, priority: 5)
Job(jobId: 553, timeReceived: 35, executionTime: 12, priority: 1)
Job(jobId: 346, timeReceived: 79, executionTime: 6, priority: 5)
Job(jobId: 132, timeReceived: 13, executionTime: 18, priority: 8)
Job(jobId: 235, timeReceived: 46, executionTime: 19, priority: 7)
Job(jobId: 621, timeReceived: 36, executionTime: 16, priority: 3)
Job(jobId: 153, timeReceived: 15, executionTime: 6, priority: 5)
Job(jobId: 649, timeReceived: 34, executionTime: 23, priority: 8)
Job(jobId: 953, timeReceived: 12, executionTime: 34, priority: 9)

Number of jobs in the collection: 10
Capacity of collection: 10
```

Add's pre-condition for count always being equal or less than capacity behaves as intended. The jobs are not added, and count and capacity are shown to be equal, thus the collection is full.

Test 12: Job edge case tests

```
aCollection.Remove(553);
aCollection.Remove(132);
aCollection.Remove(621);
aCollection.Remove(26);
aCollection.Remove(235);

Console.WriteLine();

IJob job23 = new Job(1, 1, 1, 1);
IJob job24 = new Job(999, 999, 999, 9);
//IJob job25 = new Job(1000, 46, 21, 6); // ArgumentOutOfRangeException
//IJob job26 = new Job(0, 64, 23, 1); // ArgumentOutOfRangeException
//IJob job27 = new Job(1, 0, 1, 1); // ArgumentOutOfRangeException
//IJob job28 = new Job(1, 1, 0, 1); // ArgumentOutOfRangeException

aCollection.Add(job23);
aCollection.Add(job24);
//aCollection.Add(job25);
//aCollection.Add(job26);
//aCollection.Add(job27);
//aCollection.Add(job28);

jobs = aCollection.ToArray();
printArray(jobs);
Console.WriteLine("\nNumber of jobs in the collection: " + aCollection.Count);
Console.WriteLine("Capacity of collection: " + aCollection.Capacity + "\n");
```

Results:

```
Test 12: Job edge case and exception tests.

Job 553 has been removed!
Job 132 has been removed!
Job 621 has been removed!
Job 26 has been removed!
Job 235 has been removed!

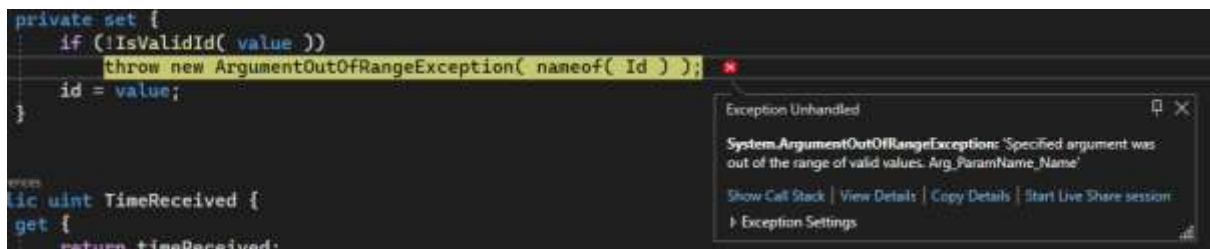
Job 1 does not currently exist in the collection.
Job 1 has been added! There are 6 jobs in the collection.

Job 999 does not currently exist in the collection.
Job 999 has been added! There are 7 jobs in the collection.

Job(jobId: 523, timeReceived: 90, executionTime: 23, priority: 8)
Job(jobId: 346, timeReceived: 79, executionTime: 6, priority: 5)
Job(jobId: 153, timeReceived: 15, executionTime: 6, priority: 5)
Job(jobId: 649, timeReceived: 34, executionTime: 23, priority: 8)
Job(jobId: 953, timeReceived: 12, executionTime: 34, priority: 9)
Job(jobId: 1, timeReceived: 1, executionTime: 1, priority: 1)
Job(jobId: 999, timeReceived: 999, executionTime: 999, priority: 9)

Number of jobs in the collection: 7
Capacity of collection: 10
```

Behaves as intended. Jobs with valid values are added to the collection, and the printed array and count reflects that. When jobs 25-28 are uncommented, each will throw an `ArgumentOutOfRangeException` exception, meaning that the methods implemented in `Job.cs` work as intended.



The screenshot shows a Visual Studio editor with a C# code snippet. The code defines a private set with an `IsValidId` method that throws an `ArgumentOutOfRangeException` if the value is invalid. Below the set, there is a `TimeReceived` property. An exception dialog is open on the right, titled "Exception Unhandled", showing the message: "System.ArgumentOutOfRangeException: 'Specified argument was out of the range of valid values. Arg_ParName: Name'". The dialog also includes links for "Show Call Stack", "View Details", "Copy Details", "Start Live Share session", and "Exception Settings".

```
private set {  
    if (!IsValidId( value ))  
        throw new ArgumentOutOfRangeException( nameof( Id ) );  
    id = value;  
}  
  
public uint TimeReceived {  
    get {  
        return timeReceived;  
    }  
}
```

All tests were successful, and all implemented methods behave as intended.

4. **References**

- Tang, Maolin (2023, February 27). *CAB301 Algorithms and Complexity: Introduction to Algorithms and Complexity* [PowerPoint slides]. Canvas. <https://canvas.qut.edu.au/>
- Tang, Maolin (2023, March 6). *CAB301 Algorithms and Complexity: Analysis of Algorithms* [PowerPoint slides]. Canvas. <https://canvas.qut.edu.au/>
- Tang, Maolin (2023, March 20). *CAB301 Algorithms and Complexity: Basic Sorting Algorithms* [PowerPoint slides]. Canvas. <https://canvas.qut.edu.au/>