# Bachelor of IT (Computer Science)
# Assignment 1
## CAB301 - Algorithms and Complexity

*Dane Madsen*
*n10983864@qut.edu.au*

# Contents

# 1 IsValidId Method

## 1.1 Algorithm Design

This method checks whether a provided job ID is valid. It achieves this by checking that the provided ID is greater than the minimum valid ID (1) and less than the maximum valid ID (999). If the ID is meets these criteria, the method returns true indicating the ID is valid, otherwise it returns false indicating the ID is invalid.

**ALGORITHM** *IsValidId(v)*
// Given an integer ($v$)
// Returns True if $v$ is a valid job ID
// Otherwise returns False
**if** $v \geq 1$ **and** $v \leq 999$
    **return** *True*
**else**
    **return** *False*

## 1.2 Testing

# 2 IsValidExecutionTime Method

## 2.1 Algorithm Design

This method simply checks whether a provided job execution time is valid. It achieves this by simply checking whether the execution time is greater than 0. If the execution time is greater than 0, the method returns true indicating the execution time is valid, otherwise it returns false indicating the execution time is invalid.

**ALGORITHM** *IsValidExecutionTime(v)*
// Given an integer ($v$)
// Returns True if $v$ is a valid job execution time
// Otherwise returns False
**if** $v > 0$
    **return** *True*
**else**
    **return** *False*

## 2.2 Testing

# 3 IsValidPriority Method

## 3.1 Algorithm Design

This method checks whether a provided job priority is valid. It achieves this by checking that the provided priority is greater than or equal to the minimum valid priority (1) and less than or equal to the maximum valid priority (9). If the priority is meets these criteria, the method returns true indicating the priority is valid, otherwise it returns false indicating the priority is invalid.

**ALGORITHM** *IsValidPriority(v)*
// Given an integer ($v$)
// Returns True if $v$ is a valid job priority
// Otherwise returns False
**if** $v \geq 1$ **and** $v \leq 9$
    **return** *True*
**else**
    **return** *False*

## 3.2 Testing

# 4 IsTimeReceived Method

## 4.1 Algorithm Design

This method checks whether a provided job time received is valid. It achieves this by checking that the provided time received is greater than zero. If the time received is greater than zero, the method returns true indicating the time received is valid, otherwise it returns false indicating the time received is invalid.

**ALGORITHM** *IsTimeReceived(v)*
// Given a job time received ($v$)
// Returns True if $v$ is a valid time received
// Otherwise returns False
**if** $v > 0$
    **return** *True*
**else**
    **return** *False*

## 4.2 Testing

# 5 Add Method

## 5.1 Algorithm Design

This method adds a job to the job collection. It achieves this by first checking that the job doesn't already exist in the collection. If the job does already exist in the collection, the method returns false indicating the job was not added to the collection. If the job does not already exist in the collection, the method adds the job to the collection, increments the count variable and returns true.

**ALGORITHM** *Add(v)*
// Let $(n)$ be count
// Given a job $(v)$
// Returns True if $v$ was added to the jobs array $(J)$
// Otherwise returns False
**for** $i \leftarrow 0$ **in** $n$ - *1* **do**
    **if** $v.id = J[i].id$
        **return** *False*
**else**
    $J[n] \leftarrow v$
    $n \leftarrow n + 1$
    **return** *True*

## 5.2 Testing

# 6 Contains Method

## 6.1 Algorithm Design

This method is used to check if a job exists in the job collection. It achieves this by checking if there is a job in the collection with the same ID as the provided job ID. If there is a job with the same ID the method returns true, otherwise it returns false.

**ALGORITHM** *Contains(v)*
// Let $(n)$ be count
// Given an integer $(v)$
// Returns True if a job with the ID $v$ exists in the jobs array $(J)$
// Otherwise returns False

```
for i ← 0 in n - 1 do
        if v = J[i].id
                return True
else
        return False
```

## 6.2 Testing

# 7 Find Method

## 7.1 Algorithm Design

This method is used to find a job in the job collection. It achieves this by using the provided job ID to find the job in the collection. If the job is found, the method returns the job, otherwise it returns null.

**ALGORITHM** *Find(v)*
// Let $(n)$ be count
// Given an integer $(v)$
// Returns the job with the ID $v$ if it exists in the jobs array $(J)$
// Otherwise returns null
**for** $i \leftarrow 0$ **in** $n$ - *1* **do**
    **if** $v = J[i].id$
        **return** *J[i]*
**return** *null*

## 7.2 Testing

# 8 Remove Method

## 8.1 Algorithm Design

This method is used to remove a job from the job collection. It achieves this by using the provided job ID to find the job in the collection. If the job is found, the method removes the job from the collection, decrements the count variable and returns true. If the job is not found, the method returns false.

**ALGORITHM** *Remove(v)*
// Let $(n)$ be count
// Given an integer $(v)$
// Returns True if a job with the ID $v$ was removed
// from the jobs array $(J)$
// Otherwise returns False
**for** $i \leftarrow 0$ **in** $n$ - *1* **do**
    **if** $v = J[i].id$
        **for** $j \leftarrow 0$ **in** $n$ - *2* **do**
            $J[j] \leftarrow J[j+1]$
        $n \leftarrow n - 1$

**return** *True*
**return** *False*

## 8.2 Testing

# 9 ToArray Method

## 9.1 Algorithm Design

This method is used to convert the job collection to an array. It achieves this by creating a new array of the same size as the job collection and then copying the jobs from the job collection to the new array. The method then returns the new array.

**ALGORITHM** *ToArray()*
// Let ($n$) be count
// Returns a new array of copied from the jobs array ($J$)
$A \leftarrow newJob[n]$
**for** $i \leftarrow 0$ **in** $n$ - *1* **do**
    $A[i] \leftarrow J[i]$
**return** $A$

## 9.2 Testing

# 10 FirstComeFirstServed Method

## 10.1 Algorithm Design

This method is used to sort jobs for first come first served scheduling. It achieves this by using the selection sort algorithm to sort the jobs by their arrival time.

**ALGORITHM** *FirstComeFirstServed()*
// Returns a new array of jobs sorted by their arrival time
$A \leftarrow Jobs.ToArray()$
**for** $i \leftarrow 0$ **in** $A.Length$ - *1* **do**
    **for** $j \leftarrow i + 1$ **in** $A.Length$ **do**
        **if** $A[i].TimeRecieved > A[j].TimeRecieved$
            $temp \leftarrow A[i]$
            $A[i] \leftarrow A[j]$
            $A[j] \leftarrow temp$
**return** $A$

## 10.2 Analysis

To implement selection sort, this method uses a nested for loop to iterate through the array of jobs. The outer loop performs n - 1 iterations, and the inner loop performs 1, 2, ..., n - 2, n - 1 iterations for every iteration of the outer loop, where n is the length of the jobs array. Therefore, the total number of iterations can be expressed as 1 + 2 + ... + (n - 2) + (n - 1), and using the rule of sum of an arithmetic sequence this equation can be further simplified to n(n - 1)/2. By dropping the constant factor and the lower order terms, the total number of iterations can be expressed as O($n^2$). Therefore, the worst case time complexity of this method is O($n^2$).

## 10.3 Testing

# 11 Priority Method

## 11.1 Algorithm Design

This method is used to sort jobs for priority scheduling. It achieves this by using the insertion sort algorithm to sort the jobs by their priority.

```
ALGORITHM Priority()
  // Returns a new array of jobs sorted by their priority
  A ← Jobs.ToArray()
  for i ← 1 in A.Length do i++
      for j ← i in 0 do j--
          if A[j].Priority < A[j − 1].Priority
              temp ← A[j]
              A[j] ← A[j − 1]
              A[j − 1] ← temp
          else
              break
  return A
```

## 11.2 Analysis

To implement insertion sort, like the previous method this method uses a nested for loop to iterate through the array of jobs. However, this time the inner loop iterates in reverse. The outer loop performs n iterations, and the inner loop performs n - 1, n - 2, ..., 2, 1 iterations for every iteration of the outer loop, where n is the length of the jobs array. Therefore, the total number of iterations can be expressed as (n - 1) + (n - 2) + ... + 2 + 1, and by again using the rule of sum of an arithmetic sequence this equation can be further simplified to n(n - 1)/2. This is the same as the previous method so therefore, the worst case time complexity of this method is O($n^2$).

### 11.3 Testing

# 12 ShortestJobFirst Method

## 12.1 Algorithm Design

This method is used to sort jobs for shortest job first scheduling. It achieves this by using the bubble sort algorithm to sort the jobs by their execution time.

**ALGORITHM** *ShortestJobFirst()*
  // Returns a new array of jobs sorted by their execution time
  $A \leftarrow Jobs.ToArray()$
  **for** $i \leftarrow 0$ **in** $A.Length - 1$ **do**
    **for** $j \leftarrow 0$ **in** $A.Length - i - 1$ **do**
      **if** $A[j].ExecutionTime > A[j+1].ExecutionTime$
        $temp \leftarrow A[j]$
        $A[j] \leftarrow A[j+1]$
        $A[j+1] \leftarrow temp$
  **return** $A$

## 12.2 Analysis

To implement bubble sort, again this method uses a nested for loop to iterate through the array of jobs. The outer loop performs n - 1 iterations, and the inner loop performs 1, 2, ..., n - 3, n - 2 iterations for every iteration of the outer loop, where n is the length of the jobs array. Therefore, the total number of iterations can be expressed as 1 + 2 + ... + (n - 3) + (n - 2), and by again using the rule of sum of an arithmetic sequence this equation can be further simplified to n(n - 1)/2. This is the same as the previous two methods so therefore, the worst case time complexity of this method is O($n^2$).

## 12.3 Testing