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

This method uses unit testing to test that the method is correctly validating the job ID. Using a for loop, the unit test first tests every valid ID (1-999) to ensure that the method returns true. It then tests two invalid IDs (0 and 1000) to ensure that the method returns false.

Test Result: PASS

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

This method uses unit testing to test that the method is correctly validating the job execution time. Using a for loop, the unit test first tests 100 valid execution times (1-100) to ensure that the method returns true. It then tests one invalid execution time (0) to ensure that the method returns false.

Test Result: PASS

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

This method uses unit testing to test that the method is correctly validating the job priority. Using a for loop, the unit test first tests every valid priority (1-9) to ensure that the method returns true. It then tests two invalid priorities (0 and 10) to ensure that the method returns false.

Test Result: PASS

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

This method uses unit testing to test that the method is correctly validating the job time received. Using a for loop, the unit test first tests 100 valid time received (1-100) to ensure that the method returns true. It then tests one invalid time received (0) to ensure that the method returns false.

Test Result: PASS

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

This method uses unit testing to test that the method is correctly adding jobs to the collection. First, the unit test creates a job with valid properties, and then creates a job collection. The unit test then attempts to add the job to the job collection. If this first attempt is successful, the unit test then attempts to add the job to the job collection again. If the second attempt is unsuccessful, the unit test passes. This ensures that the method can add a new job, but will not overwrite a job if a job in the job collection already has the same ID as the job being added.

Test Result: PASS

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 $i \leftarrow 0$ **in** n - *1* **do**
    if $v = J[i].id$
        return *True*
else
    return *False*

6.2 Testing

This method uses unit testing to test that the method is correctly checking if a job exists in the collection. First, the unit test creates a job with valid properties, and then creates a job collection. Before adding the job to the job collection, the unit test first checks that the job does not exist in the job collection. If the job does not exist in the job collection, the unit test then adds the job to the job collection. Finally, the unit test again checks that the job exists in the job collection. If the job exists in the job collection, the unit test passes. This ensures that jobs are created independantly of the job collection, and that the job collection can correctly check if a job exists in the collection.

Test Result: PASS

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

This method uses unit testing to test that the method is correctly finding a job in the collection. First, the unit test creates a job with valid properties, and then creates a job collection. Before adding the job to the job collection, the unit test first checks that the job cannot be found in the job collection. If the job cannot be found in the job collection, the unit test then adds the job to the job collection. Finally, the unit test again checks that the job can be found in the job collection. If the job can be found in the job collection, the unit test passes. This ensures that the job collection can correctly find a job in the collection.

Test Result: PASS

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

This method uses unit testing to test that the method is correctly removing a job from the collection. First, the unit test creates a job with valid properties, and then creates a job collection. Before adding the job to the job collection, the unit test first attemmps to remove the job from the job collection. If the job cannot be removed from the job collection, the unit test then adds the job to the job collection. Finally, the unit test again attempts to remove the job from the job collection. If the job can be removed from the job collection, the unit test passes. This ensures that the job collection can correctly remove a job from the collection.

Test Result: PASS

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

This method uses unit testing to test that the method is correctly converting the job collection to an array. First, the unit test creates a job with valid properties, and then creates a job collection. Before adding the job to the job collection, the unit test first creates an array from the job collection and checks that is empty. If the array is empty, the unit test then adds the job to the job collection and then again creates an array from the job collection. Finally, the unit test checks that the array contains a single job. If the array contains a single job, the unit test passes. This ensures that the job collection can correctly convert the job collection to an array, and that the array contains the same ammount of jobs as the job collection.

Test Result: PASS

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

This method uses unit testing to test that the method is correctly sorting the jobs by their arrival time. First, the unit test creates an empty job array and then creates a job collection. Next, nine jobs are created with valid properties and added to the empty job array. The unit test then adds the jobs from the job array to the Jobs collection in reverse order, this is to ensure the jobs are not already sorted by their arrival time.

Next, a scheduler is created using the Jobs collection, and a new job array is created using the FirstComeFirstServed method. The unit test then checks that every job in the new job array exists in their repective index in the job array. It also checks that every job is distinct by checking an identicle job does not exist at any other index. If these conditions are met the unit test passes.

Test Result: PASS

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

The unit test for this method is the same as the previous method. The only difference is that Priority() is used instead of FirstComeFirstServed().

First, the unit test creates an empty job array and then creates a job collection. Next, nine jobs are created with valid properties and added to the empty job array. The unit test then adds the jobs from the job array to the Jobs collection in reverse order, this is to ensure the jobs are not already sorted by their priority.

Next, a scheduler is created using the Jobs collection, and a new job array is created using the Priority method. The unit test then checks that every job in the new job array exists in their repective index in the job array. It also checks that every job is distinct by checking an identicle job does not exist at any other index. If these conditions are met the unit test passes.

Test Result: PASS

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

The unit test for this method is the same as the previous two methods. The only difference is that ShortestJobFirst() is used instead of FirstComeFirstServed() or Priority().

First, the unit test creates an empty job array and then creates a job collection. Next, nine jobs are created with valid properties and added to the empty job array. The unit test then adds the jobs from the job array to the Jobs collection in reverse order, this is to ensure the jobs are not already sorted by their execution time.

Next, a scheduler is created using the Jobs collection, and a new job array is created using the ShortestJobFirst method. The unit test then checks that every job in the new job array exists in their respective index in the job array. It also checks that every job is distinct by checking an identicle job does not exist at any other index. If these conditions are met the unit test passes.

Test Result: PASS