COMP90038 Algorithms and Complexity

Assignment 1

Semester 2, 2018

Name Tianyao Cai **StudentID** 940900

September 2, 2018

Problem 1

(a) It is searching for an element k in sorted array A where element k is equal to its index.

(b) This algorithm starts by checking the value range of the array:

If $A[lo] > lo$ or $A[hi] < hi$, then we cannot find the position to meet the condition.

Then it compares the array's middle element $A[mid]$ with its index mid:

If $A[mid] = mid$, then we are done.

If $A[mid] > mid$, then search the sub-array up to $A[mid-1]$ recursively.

If $A[mid] < mid$, then search the sub-array from $A[mid+1]$ recursively.

From my perspective, the input sorted array A should have additional property as follows, which makes the algorithm perform well:
Elements cannot be repeated in the array. If elements can be repeated, the algorithms may miss the element it is looking for because of the range checking.

Problem 2

CountC is the main function which is designed for finding the number of occurrences of c in sorted array A, which calls the functions LocateFirstC and LocateLastC. First of all, the algorithm locates the positions where the first and the last c appear. Then it will calulate the answer using the 2 positions we already get. This 2 sub-function is based on the Binary Search, thus the complexity of this algorithm should be in $O(logn)$. Their pseudo codes are as follows:

```
function CountC($A[\cdot], lo, hi, c$)
    $first \leftarrow$ LocateFirstC($A[\cdot], lo, hi, c$)
    $last \leftarrow$ LocateLastC($A[\cdot], lo, hi, c$)
    if $first == -1$ then
```

```
        return 0
    return $last - first + 1$


function LocateFirstC($A[\cdot], lo, hi, c$)
    if $lo > hi$ then
        return -1
    $mid \leftarrow \lfloor (lo + hi)/2 \rfloor$
    if $A[mid] == c$ then
        if $mid == 0$ or ($mid > 0$ and $A[mid - 1]\ ! = c$) then
            return $mid$
        else
            $hi \leftarrow mid - 1$
    else if $A[mid] > c$ then
        $hi \leftarrow mid - 1$
    else
        $lo \leftarrow mid + 1$
    return LocateFirstC($A[\cdot], lo, hi, c$)

function LocateLastC($A[\cdot], lo, hi, c$)
    if $lo > hi$ then
        return -1
    $mid \leftarrow \lfloor (lo + hi)/2 \rfloor$
    if $A[mid] == c$ then
        if $mid == A.length() - 1$ or ($mid < A.length() - 1$ and $A[mid + 1]\ ! = c$) then
            return $mid$
        else
            $lo \leftarrow mid + 1$
    else if $A[mid] > c$ then
        $hi \leftarrow mid - 1$
    else
        $lo \leftarrow mid + 1$
    return LocateLastC($A[\cdot], lo, hi, c$)
```

To count how many cs in the array A, the initial call is CountC($A[\cdot], 0, n - 1, c$).

Problem 3

(a) The worst case occurs under the circumstance that when all variables' value are false can make the f true, or there is no solution for f. The elements of Array A can be random because the algorithm will replace all elements with certain values.

(b) In the worse case, if $n = 0$ then the function will call Check 1 time. If $n > 0$, the function will call itself 2 times.(the judgement statement and the return statement in "else" section) Thus, we succeed in setting up the recurrence relation and initial condition for the algorithm's number of Check function call $C(n)$:

$$C(n) = 2C(n - 1) \qquad for\ n > 0$$
$$C(0) = 1$$

Now, we can use backward substitution to solve this recurrence:

$$\begin{aligned}
C(n) &= 2C(n-1) \\
&= 2^2C(n-2) \\
&= 2^3C(n-3) \\
&= \cdots \\
&= 2^nC(0) \\
&= 2^n
\end{aligned}$$

Thus the closed form for the recurrence relation is $C(n) = 2^n$, the worst case complexity of this algorithm is $\Theta(2^n)$.

Problem 4

(a) The function ROBOTMOVE(p) is designed as follows, where p is the linked list storing the route for the robot. Stack is used to record the way the robot has traveled. The complexity of this algorithm is in $\Theta(k)$, where k is the length of the route.

```
function RobotMove($p$)
    if $p$ == null then
        return null
    $s \leftarrow init(stack)$        //create an empty stack.
    while $p$ ! = $null$ then
        Move($p.val$)
        $s.push(p.val)$
        $p \leftarrow p.next$
    while $s$ is not empty do
        $value \leftarrow s.pop()$
        if $value$ == 'N' then
            Move('S')
        else if $value$ == 'S' then
            Move('N')
        else if $value$ == 'E' then
            Move('W')
        else
            Move('E')
    return null
```

(b) Using the idea from Breadth First Search, the algorithm I designed to solve this problem is as below. This algorithm, first of all, uses Breadth First Search to find one of the shortest route from Home Base point to Goal point. Then it uses some data structures (array, stack and linked list) to return the entire shortest route from Home Base point to Goal point. It is easy to prove the algorithm below runs in $O(n^2)$.

In this function, array *marked* is used to record whether the points have been visited or not. Array *edgeTo* is used to store the parent of the current point, e.g. $edgeTo[i] \leftarrow j$ means that in the shortest route we found, j is the previous point of i. Given the start point and the end point, stack *path* is a tool used to generate the entire shortest route.

```
function ShortestRoute($A, n, h, g$)
    $marked \leftarrow init(array, n, false)$    //create an boolean array filled with false.
    $queue \leftarrow init(queue)$    //create an empty queue.
    $edgeTo \leftarrow init(array, n)$    //create an empty array.
    $shortestRoute \leftarrow init(linked\ list)$    //create an empty linked list.
    $path \leftarrow init(stack)$    //create an empty stack.
    $marked[h] \leftarrow true$
    $queue.enqueue(h)$
    while $queue$ is not empty do
        $v \leftarrow queue.dequeue()$
        for $w \leftarrow 0$ to $n - 1$ do
            if $A[v, w]$ is not null and ! $marked[w]$ then
                $edgeTo[w] \leftarrow v$
                $marked[w] \leftarrow true$
                if $w == g$ then
                    empty the entire $queue$
                    $break$
                $queue.enqueue[w]$
    $i \leftarrow g$
    while $i\ != h$ do
        $path.push(A[edgeTo[i], i])$
        $i \leftarrow edgeTo[i]$
    while $path$ is not empty do
        $shortestRoute.append(path.pop())$
    return $shortestRoute$
```