THE UNIVERSITY OF MELBOURNE

COMP90038 Algorithms and Complexity SM2, 2018
Assignment 1

Peiyong Wang 955986

September 4, 2018

1 Problem One

A Clearly this algorithm searches in an array for a value which equals to its index, like A[9]=9.

B The algorithm firstly examines whether A[lo] > lo or A[hi] < hi. If so, it means all values are bigger or smaller than their indexes, and in this situation the algorithm will return -1, indicating there is no possibility in the input array that $\exists i \in \mathbb{N}^+, s.t. A[i] = i$. If the input array satisfy this condition, then the algorithm will use hi and lo to compute mid and check whether A[mid] = mid. If so, return mid; if not, the algorithm will split the array into two halves and if A[mid] is bigger than mid the algorithm will continuing its search in the lower half, otherwise it will search in the higher half of the input array and recursively call the function itself.

However, to make this algorithm work fine, the input array must only have on value that equals to its key, and there should be no duplicates. If it has many values that they all equal to their keys repectively, the algorithm will only find one. If the input array have duplicates there is a chance that there will be a A[lo]>lo or A[hi]<hi situation, which won't pass the first if-clause in the algorithm.

2 Problem Two

See Algorithm 1.

3 Problem Three

A From Figure 3.1, which is a "state tree" that in this tree a path starts from node "start" (but not include node start) and ends at a node that in the binary tree's bottom indicates a possible collections of booleans that could satisfy the formula. For example, the array $A = [true, false, false]$ in the question can be converted to a path starts from node 1, then go to node 4 and ends at node 10. So we can tell that for the worst case scenario of the algorithm stated in the question, we need to traverse all the paths in the state tree. The time complexity of binary tree traversal is $O(n + n - 1) = O(n)$, where n is the number of nodes in the binary tree. If we let $n = number\ of\ elements\ in\ array\ A$, then the total number of nodes in the binary tree is $1 + \sum_{i=0}^{n} 2 \cdot 2^i = 2^{n+2} - 1$. So the worst case time complexity is $O(2^{n+2} - 1) = O(2^n)$

Algorithm 1 Count the number of occurrences of a certain integer x in an array $A[]$

Require: $Array\ A[\cdot]$,$Length\ of\ the\ array$ n, $Integer$ x

Ensure: Number of occurrence of integer x

```
function NumberCount(A[·], x, n)
    firstApperence ← FirstOccurrenceSearch(A[·], 0, n − 1, x, n)
    if firstApperence = −1 then
        return firstApperence
    end if
    lastApperence ← LastOccurrenceSearch(A[], firstApperence, n − 1, x, n)
    return lastApperence − firstApperence + 1
end function

function FirstOccurrenceSearch(A[·], low, high, x, n)
    if high ⩾ low then
        mid ← (low + high)//2                          ▷ "//" means integer division
    end if
    if {mid = 0 OR x > A[mid − 1]} AND {A[mid] = x} then
        return mid
    else
        if x > A[mid] then
            return FirstOccurrenceSearch(A[·], mid + 1, high, x, n)
        else
            return FirstOccurrenceSearch(A[·], low, mid − 1, x, n)
        end if
    end if
    return -1
end function

function LastOccurrenceSearch(A[·], low, high, x, n)
    if high ⩾ low then
        mid ← (low + high)//2                          ▷ "//" means integer division
    end if
    if {mid = n − 1 OR x < A[mid − 1]} AND {A[mid] = x} then
        return mid
    else
        if x < A[mid] then
            return LastOccurrenceSearch(A[·], low, mid − 1, x, n)
        else
            return LastOccurrenceSearch(A[·], mid + 1, high, x, n)
        end if
    end if
    return -1
end function
```

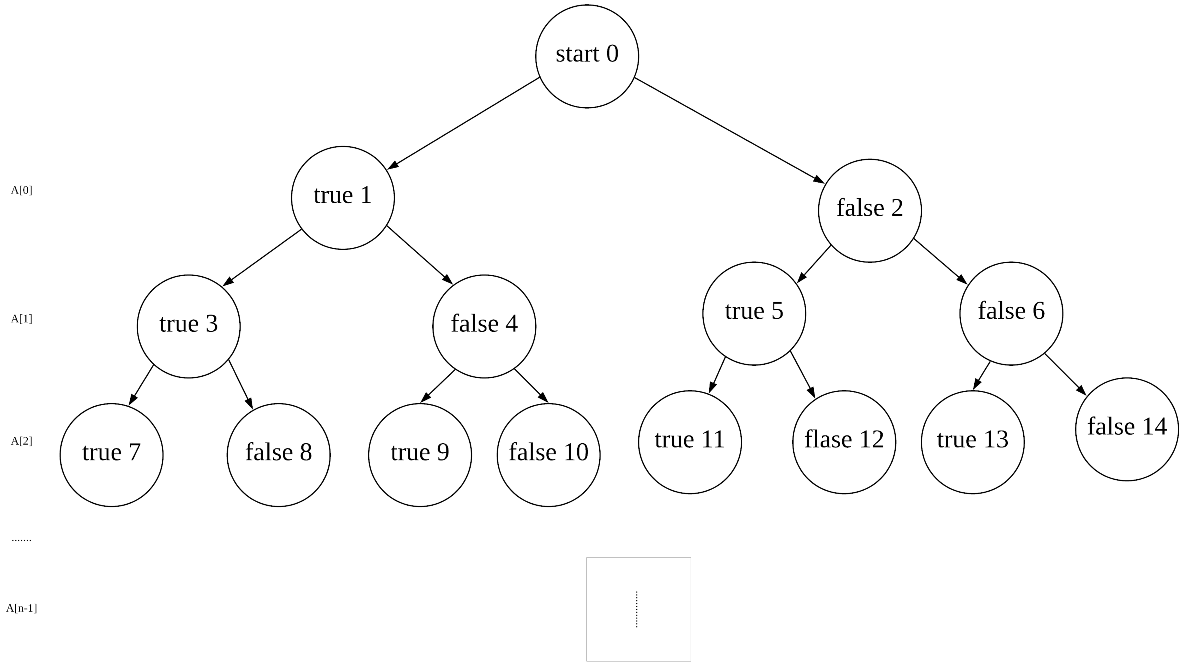

Figure 3.1: State Tree

B From paragraph A, we can easily see that:

$$C(1) = 2 \tag{3.1}$$

$$C(n) = 2C(n-1) \tag{3.2}$$

Then we can have that:

$$\begin{aligned} C(n) &= 2C(n-1) \\ &= 2(2(C(n-2))) \\ &= \cdots \\ &= 2^{n-1}C(1) \\ &= 2^n \end{aligned} \tag{3.3}$$

So the time complexity is $\Theta(2^n)$.

4 Problem Four

A See Algorithm 2.

B See Algorithm 3.

Algorithm 2 Follow a given route linked list to move to the destination and then go back.

Require: head of the linked list of route

```
function GoAndBack(head)
    p ← head                                        ▷ Starting from the home base
    goBack ← empty stack
    while p is not null do
        Move(p.val)
        goBack.push(p.val)                          ▷ Push the current position to the stack
        p ← p.next
    end while
    hashBack ← { "N":"S","E":"W","S":"N","W":"E" }  ▷ This is a hash table that stores the opposite directons, like a dictionary in Python, eg: hashBack("N")="S".
    while goBack is not null do
        Move(hashBack(goBack.pop))  ▷ Pop the top element of the stack,use the hash table to find its opposite direction, and call the Move function
    end while
end function
```

Algorithm 3 Use BFS to search the shortest route from home base to goal

Require: Graph adjacency matrix $A[\cdot,\cdot]$, start, goal
Ensure: Route list

function SHORTESTPATH($A[\cdot,\cdot], start, goal$)
  $pathOfNodes \leftarrow$ BFSPATHS($start$)
  $reverseRoute \leftarrow$ empty list []
  $currentVortex \leftarrow goal$
  while $pathOfNodes[currentVortex] \neq -1$ **do**
    $reverseRoute$.append($currentVortex$)
    $currentVortex \leftarrow pathOfNodes[currentVortex]$
  end while
  $routeList \leftarrow$ empty list []
  $j \leftarrow reverseRoute$.length
  while $j \geq 1$ **do** $routeList$.append($A[reverseRoute[j], reverseRoute[j-1]]$)
  end while
  $routeLinkedList \leftarrow$ empty linked list with head q and length is the same as $routeList$
  for $k = 0; k < routeList.length; k++$ **do**
    $q.value \leftarrow routeList[k]$
    $q \leftarrow q.next$
  end for
  return $routeLinkedList$
end function

function BFSPATHS(start)
  for each vortex v **do**
    $flag[v] \leftarrow false$
    $pathList[v] \leftarrow -1$ ▷ Maintain a list that records what is the previous node of a node in the BFS searching path
  end for
  $Q \leftarrow empty\ queue$
  $flag[start] \leftarrow true$
  ENQUEUE(Q,$start$)
  while Q is not empty **do**
    $v \leftarrow$ DEQUEUE(Q)
    for each w adjacent to v **do**
      if $flag[w] = true$ **then**
        $flag[w] \leftarrow true$
        $pathList[w] \leftarrow v$
        ENQUEUE(Q, w)
      end if
    end for
  end while
  return $pathList$
end function