The University of Melbourne
School of Computing and Information Systems
COMP90038 Algorithms and Complexity

Assignment 1, Semester 2 2018

Released: Friday August 17 2018. Deadline: Sunday September 2 2018 23:59

Objectives

To improve your understanding of the time complexity of algorithms and recurrence relations. To develop problem-solving and design skills. To improve written communication skills; in particular the ability to present algorithms clearly, precisely and unambiguously.

Problems

1. Consider the following mystery algorithm. *Note: the division operation is integer division, i.e. it rounds down its result to the nearest integer.*

```
function Mystery($A[\cdot], lo, hi$)
    //Input is an array of integers of length $n$
    if $A[lo] > lo$ or $A[hi] < hi$ then
        return -1
    $mid \leftarrow lo + (hi - lo)/2$
    if $A[mid] = mid$ then
        return $mid$
    else
        if $A[mid] > mid$ then
            return Mystery($A, lo, mid - 1$)
        else
            return Mystery($A, mid + 1, hi$)
```

(a) This algorithm clearly resembles binary search. What is it searching for? [1 mark]

Solution:
A fixed point (value equal to index), i.e. $A[i] = i$.

(b) Like binary search, it relies on the input array A being sorted. Explain how it works and what additional property of the array A it relies upon. *Hint: think about sorted arrays for which this algorithm might* not *find what it is looking for.* [3 marks]

Solution:
It relies on the elements in A being distinct: no element is repeated in A.
Let $c_{lo} = A[lo]$. Since elements in A are distinct, for all $0 < j \leq hi - lo$, $A[lo + j] \geq c_{lo} + j$. Also, let $c_{hi} = A[hi]$. Then since elements in A are distinct, for all $0 < j \leq hi - lo$, $c_{hi} - j \geq A[hi - j]$. When $A[lo] > lo$, $c_{lo} > lo$. Hence, by transitivity we have for all $0 < j \leq hi - lo$, $A[lo + j] > lo + j$. Likewise when $A[hi] < hi$, $c_{hi} < hi$. Hence by transitivity we have for all $0 < j \leq hi - lo$, $hi - j > A[hi - j]$. Hence in either of these cases, what we are looking for cannot be in the array.
Otherwise, we have $A[lo] \leq lo$ and $A[hi] \geq hi$, so what we are looking for might be in the array between lo and hi. We take the mid-point, as in binary search. When $A[|] > mid$, then clearly what we are looking for can only be in the part of the array before |. So $| -1$ becomes the new hi and we recurse. Likewise, when $A[|] <|$, then clearly what we are looking for can only be in the part of the array after |. So $| +1$ becomes the new lo and we recurse. Otherwise, we've found what we are looking for.

2. Design an algorithm that, given a sorted array of integers A of length n, and an integer c, finds the number of occurrences of c in A. Your algorithm's complexity should be in $O(\log n)$ even when array A contains many cs relative to other elements (i.e. its complexity should be *independent* of the number of cs in the array A). [5 marks]

Solution:
We do a search to find the highest and lowest indices i for which $A[i] = c$. Then we use subtraction to calculate the total, since the array is sorted. One can design a single binary search style algorithm which keeps track of two sub-ranges within the array A, one for the maximum index j and one for the minimum index i. However we can also simply run two logarithmic searches one after the other and the resulting complexity will still be in $\Theta(\log n)$. This is what we do.

```
function FindMin($A[\cdot], c, lo, hi$)
    //Input is an array of integers of length $n$
    if $lo > hi$ then
        return -1
    $mid \leftarrow lo + (hi - lo)/2$
    if $A[mid] = c$ then
        if $mid > lo$ then
            if $A[mid-1] \neq c$ then
                return $mid$
            else
                return FindMin($A[\cdot], c, lo, mid - 1$)
        else
            return $mid$
    if $A[mid] > c$ then
        return FindMin($A[\cdot], c, lo, mid - 1$)
    else
        return FindMin($A[\cdot], c, mid + 1, hi$)
function FindMax($A[\cdot], c, lo, hi$)
    //Input is an array of integers of length $n$
    if $lo > hi$ then
        return -1
    $mid \leftarrow lo + (hi - lo)/2$
    if $A[mid] = c$ then
        if $mid < hi$ then
            if $A[mid+1] \neq c$ then
                return $mid$
            else
                return FindMax($A[\cdot], c, mid + 1, hi$)
        else
            return $mid$
    if $A[mid] > c$ then
        return FindMax($A[\cdot], c, lo, mid - 1$)
    else
        return FindMax($A[\cdot], c, mid + 1, hi$)
function CountOccurrences($A[\cdot], c$)
    $mini \leftarrow$ FindMin($A[\cdot], c, 0, len(A) - 1$)
    if $mini = -1$ then
        return 0
    $maxi \leftarrow$ FindMax($A[\cdot], c, 0, len(A) - 1$)
    return $maxi - mini + 1$
```

3. The *boolean satisfiability problem* is the problem of determining whether a propositional logic formula involving variables (e.g. "$x_0 \wedge (x_1 \vee \neg x_2)$") can be *satisfied*, i.e. whether there exist values for the boolean variables that make the formula true. When this is the case we say that the formula is *satisfiable*; otherwise it is called *unsatisfiable* (e.g. the previous formula is satisfiable: let $x_0 = \mathsf{true}$, $x_1 = \mathsf{false}$ and $x_2 = \mathsf{false}$, then the formula becomes $\mathsf{true} \wedge (\mathsf{false} \vee \neg\mathsf{false})$ which is true).

   Given a collection of n variables $x_0, \ldots, x_{n-1}$, a *valuation* is an array A of length n that holds boolean values such that each $A[i]$ defines the value of variable x_i. (The valuation that demonstrated the satisfiability of the above formula would be the array A for which $A[0] = \mathsf{true}$, $A[1] = \mathsf{false}$, and $A[2] = \mathsf{false}$.)

   Imagine you have a function CHECK(f, A) that given a propositional formula f and valuation A for f's variables, decides whether formula f is true.

   Given such a function, one can write a recursive, brute force algorithm to decide the satisfiability of a boolean formula f. One possible algorithm is as follows. This solution enumerates all possible valuations. For each, it calls CHECK until it finds a solution.

```
function CHECKSAT($f, A[\cdot], i, n$)
    //Input is an array of length $n$
    //i is an integer in the range 0 to $n$
    if $i = n$ then
        return CHECK($f, A$)
    $A[i] \leftarrow$ true
    if CHECKSAT($f, A, i+1, n$) then return true
    else
        $A[i] \leftarrow$ false
        return CHECKSAT($f, A, i+1, n$)
```

   To check satisfiability on formula f, the initial call is for an arbitrary array A of length n, and is: CHECKSAT$(f, A, 0, n)$.

   Your tasks:

   (a) What is the worst case input to the CHECKSAT algorithm? [1 mark]

   Solution:
   The worst case is a formula that is either: (1) unsatisfiable or (2) satisfiable only when all variables are false.

   (b) Assuming the size of the input to the CHECKSAT algorithm is n, the number of variables in f, analyse its worst case complexity, where the basic operation is calling the CHECK function. That is, define a recurrence relation $C(n)$ for the number of calls to CHECK for an input of size n. Use telescoping (aka backward substitution) to find a closed form for the recurrence relation, and from that express the complexity in terms of Θ ("Big-Theta" notation). [2 marks]

Solution:

$$C(0) = 1$$
$$C(n) = 2C(n-1)$$

$$\begin{aligned} C(n) &= 2C(n-1) \\ &= 4C(n-2) \\ &= 8C(n-3) \\ &= 2^i C(n-i) \\ &= 2^n C(n-n) \\ &= 2^n C(0) \\ &= 2^n \end{aligned}$$

Hence, $C(n) = 2^n$, so $C(n) \in \Theta(2^n)$.

4. Suppose you work for a software company designing the software for a robot. The robot moves on tracks and it can move in only four directions: North, South, East, and West. A tiny example scenario is depicted in Figure 1 in which the black lines are the tracks, North corresponds to up, South corresponds to down, West corresponds to left, and East corresponds to right. Each of the blue dots represents an intersection point between tracks. There are n such points and each is numbered from 0 to $n-1$.

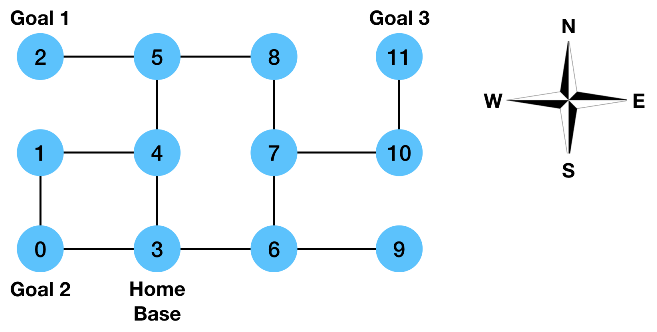

Figure 1: A tiny example track network for the robot.

The track network is stored as an $n \times n$ matrix A. Each matrix cell is either empty, represented by the symbol '-', or it contains one of the four letters 'N', 'S', 'E', 'W', corresponding to the four directions respectively. When cell $A[i][j]$ is not empty and contains one of the four letters, it means that the robot can move from intersection i to intersection j by moving in the direction indicated by that letter. The matrix corresponding to the network in Figure 1 is as follows.

	0	1	2	3	4	5	6	7	8	9	10	11
0	-	N	-	E	-	-	-	-	-	-	-	-
1	S	-	-	-	E	-	-	-	-	-	-	-
2	-	-	-	-	-	E	-	-	-	-	-	-
3	W	-	-	-	N	-	E	-	-	-	-	-
4	-	W	-	S	-	N	-	-	-	-	-	-
5	-	-	W	-	S	-	-	-	E	-	-	-
6	-	-	-	W	-	-	-	N	-	E	-	-
7	-	-	-	-	-	-	S	-	N	-	E	-
8	-	-	-	-	-	W	-	S	-	-	-	-
9	-	-	-	-	-	-	W	-	-	-	-	-
10	-	-	-	-	-	-	-	W	-	-	-	N
11	-	-	-	-	-	-	-	-	-	-	S	-

The robot's job is to travel from its Home Base to various Goal points and back again. However after reaching a Goal point it must return to its Home Base.

(a) Suppose each route for the robot are stored in a linked list. For instance, to move from the Home Base point in Figure 1 to the Goal 3 point, the robot could follow the route ['N','N','E','S','E','N'].

Write a function that takes such a list as its input. The function should instruct the robot to follow the route and then to return to its original starting point (by going back the way it came). Your function instructs the robot to move in a particular direction by calling the function MOVE(d), where d is one of the four letters 'N','S','E','W'. For instance, calling MOVE('W$'$) causes the robot to move West. Your algorithm should run in $\Theta(k)$, where k is the length of the given route. [3 marks]

Solution:
Use a stack to store the return route. Walk the route. For each node encountered, instruct the robot to move accordingly and then push the reverse direction onto the stack. Then return by repeatedly popping the stack and instructing the robot accordingly.

```
function OppositeDirection($direction$)
    if $direction$ = ‘E’ then
        return ‘W’
    else if $direction$ = ‘S’ then
        return ‘N’
    else if $direction$ = ‘W’ then
        return ‘E’
    else if $direction$ = ‘N’ then
        return ‘S’
procedure FollowRoute($l$)
    //Input is a linked list
    initialise $path$ as an empty stack
    while $l \neq NULL$ do
        Move($l.val()$)
        $path.push$(OppositeDirection($l.val()$))
        $l \leftarrow l.next()$
    while $path$ is non-empty do
        $direction \leftarrow path.pop()$
        Move($direction$)
```

(b) Write an algorithm that given a Goal point g (an integer between 0 and $n-1$ inclusive) constructs the shortest route to that Goal from the given Home Base point h (an integer between 0 and $n-1$ inclusive). You may assume that such a route exists, that g and h are distinct, and that if multiple shortest routes exist your algorithm merely needs to return any of them. That is, your function should return a linked list that stores a shortest route from h to g. For instance, if called with $g = 11$ and $h = 3$ for the network in Figure 1, it would produce the route [‘E’,‘N’,‘E’,‘N’]. (Note this is much shorter than the route mentioned in the previous question.)

Your algorithm should run in $O(n^2)$. [5 marks]

Solution:
Use BFS. For each element in the node queue, have it store the route followed so far to reach that node. When we reach the goal we are guaranteed to have gotten there by the shortest route: if a shorter route existed we would have found it on a previous iteration.

```
function BFS($G < V, E >, h, g$)
    //Input is a graph $G$, a home point $h$, and the goal point $g$
    mark each node in $V$ with 0
    $init(queue)$                          ▷ create an empty queue to store nodes
    $init(paths)$                          ▷ create an empty queue to store paths
    $init(path)$                           ▷ create an empty linked list
    $inject(queue, h)$
    $inject(paths, path)$
    mark $h$ with 1
    while $queue$ is non-empty do
        $u \leftarrow eject(queue)$                      ▷ dequeues $u$
        $p \leftarrow eject(paths)$    ▷ dequeues $p$, a linked list containing path to the current node
        for each $v$ marked with 0 do
            if $E[u, v] \neq$ '$-$' then
                mark $v$ with 1
                $p2 \leftarrow$ a copy of the entire linked list $p$
                $p2.enqueue(E[u, v])$            ▷ add the direction to path
                if $v = g$ then
                    return $p2$
                $inject(queue, v)$
                $inject(paths, p2)$
```

Submission and Evaluation

- You must submit a PDF document via the LMS. Note: handwritten, scanned images, and/or Microsoft Word submissions are not acceptable — if you use Word, create a PDF version for submission.
- Marks are primarily allocated for correctness, but elegance of algorithms and how clearly you communicate your thinking will also be taken into account. Where indicated, the complexity of algorithms also matters.
- We expect your work to be neat – parts of your submission that are difficult to read or decipher will be deemed incorrect. Make sure that you have enough time towards the end of the assignment to present your solutions carefully. Time you put in early will usually turn out to be more productive than a last-minute effort.
- You are reminded that your submission for this assignment is to be your own individual work. For many students, discussions with friends will form a natural part of the undertaking of the assignment work. However, it is still an individual task. You should not share your answers (even draft solutions) with other students. Do not post solutions (or even partial solutions) on social media or the discussion board. It is University policy that cheating by students in any form is not permitted, and that work submitted for assessment purposes must be the independent work of the student concerned.

  Please see `https://academicintegrity.unimelb.edu.au`

If you have any questions, you are welcome to post them on the LMS discussion board *so long as you do not reveal details about your own solutions.* You can also email the Head Tutor, Lianglu Pan (lianglu.pan@unimelb.edu.au) or the Lecturer, Toby Murray (toby.murray@unimelb.edu.au). In your message, make sure you include COMP90038 in the subject line. In the body of your message, include a precise description of the problem.

Late Submission and Extension

Late submission will be possible, but a late submission penalty will apply: a flagfall of 2 marks, and then 1 mark per 12 hours late.

Extensions will only be awarded in extreme/emergency cases, assuming appropriate documentation is provided simply submitting a medical certificate on the due date will not result in an extension.