COMP3506 Homework 1

Weighting: 15%

Due date: 21st August 2020, 11:55 pm

Questions

1. Consider the following algorithm, CoolAlgorithm, which takes a **positive** integer n and outputs another integer. Recall that '&' indicates the bitwise AND operation and '$a >> b$' indicates the binary representation of a shifted to the right b times.

```
1:  procedure CoolAlgorithm(int n)
2:      $sum \leftarrow 0$
3:      if $n \% 2 == 0$ then
4:          for $i = 0$ to $n$ do
5:              for $j = i$ to $n^2$ do
6:                  $sum \leftarrow sum + i + j$
7:              end for
8:          end for
9:      else
10:         while $n > 0$ do
11:             $sum \leftarrow sum + (n \& 1)$
12:             $n \leftarrow (n >> 1)$
13:         end while
14:     end if
15:     return $sum$
16: end procedure
```

Note that the runtime of the above algorithm depends not only on the size of the input n, but also on a numerical property of n. For all of the following questions, you must assume that n is a positive integer.

(a) (3 marks) Represent the running time (i.e. the number of primitive operations) of the algorithm when the input n is **odd**, as a mathematical function called $T_{\text{odd}}(n)$. State all assumptions made and explain all your reasoning.

Solution: Assume code is executed in a single processor machine. Takes 1 unit of time for arithmetic, logical operations and bit shifting; 1 unit for assignment and return. Primitive operations:
$T_{odd}(n) = 1 + 2 + (\frac{n+1}{2})(3 + 2) + 1$
$T_{odd}(n) = 5(\frac{n+1}{2}) + 4$

(b) (2 marks) Find a function $g(n)$ such that $T_{\text{odd}}(n) \in O(g(n))$. Your $g(n)$ should be such that the Big-O bound is as tight as possible (e.g. no constants or lower order terms). Using the formal definition of Big-O, prove this bound and explain all your reasoning.

(Hint: you need to find values of c and n_0 to prove the Big-O bound you gave is valid).

Solution: f(n) is O(g(n)) if there are positive constants c and n_0 such that f(n) $\leq$ c * g(n) for n $\geq n_0$
By dropping the lower order terms of f(n), g(n) = n
$5\ (\frac{n+1}{2}) + 4 \leq c * n$
$4 \leq c * n - 5\ (\frac{n+1}{2})$
$4 \leq c * n - \frac{5n}{2} + \frac{5}{2}$
$\frac{3}{2} \leq (c - \frac{5}{2}) * n$
$n \geq \frac{3}{2} \div (c - \frac{5}{2})$
Choose c = $\frac{7}{2}$ and $n_0 = \frac{3}{2}$

(c) (2 marks) Similarly, find the tightest Big-Ω bound of $T_{\text{odd}}(n)$ and use the formal definition of Big-Ω to prove the bound is correct. Does a Big-Θ bound for $T_{\text{odd}}(n)$ exist? If so, give it. If not, explain why it doesn't exist.

Solution: f(n) is Ω(g(n)) if there exist positive constants c and n_0 such that f(n) $\geq$ c * g(n) for all n $\geq$ n_0
Need to prove: 5 $(\frac{n+1}{2})$+ 4 $\geq$ 1 * n ; where c = 1 and g(n) = n
Let $n_0 = 0$, LHS = $\frac{13}{2}$, RHS = 0
LHS $\geq$ RHS
$\therefore 5(\frac{n+1}{2})+ 4 = \Omega(n)$

f(n) is Θ(g(n)) if it is Ω(n) and O(n)
Since we have proven that f(n) is Ω(n) and O(n), Big-Θ bound exists. Let $c_1 = \frac{7}{2}$, $c_2 = 1$, $n_0 = \frac{3}{2}$

(d) (3 marks) Represent the running time (as you did in part (a)) for the algorithm when the input n is **even**, as a function called $T_{\text{even}}(n)$. State all assumptions made and explain all your reasoning. Also give a tight Big-O and Big-Ω bound on $T_{\text{even}}(n)$. You do **not** need to formally prove these bounds.

Solution: Assume code is executed in a single processor machine. Takes 1 unit of time for arithmetic, logical operations and bit shifting; 1 unit for assignment and return. For the inner for loop, n^2 is counted for each iteration therefore the inner for loop runs n (n^2 + 1) times. Primitive operations:
$T_{even}(n)$ = 1 + 2 + (n+1) + n (n^2 + 1) + $3n^2 + 1$
$T_{even}(n)$ = $n^3 + 3n^2 + 2n + 5$
$T_{even}(n)$ = O(n^3),$T_{even}(n)$ = Ω(n^2)

(e) (2 marks) The running time for the algorithm has a best case and worst case, and which case occurs for a given input n to the algorithm depends on the parity of n.

Give a Big-O bound on the **best case** running time of the algorithm, and a Big-Ω bound on the **worst case** running time of the algorithm (and state which parity of the input corresponds with which case).

Solution:
Big-O bound on the best case running time is O(n) when n is odd
BigΩ bound on the worst case running time is $\Omega(n^2)$ when n is even.

(f) (2 marks) We can represent the runtime of the entire algorithm, say $T(n)$, as

$$T(n) = \begin{cases} T_{\text{even}}(n) & \text{if } n \text{ is even} \\ T_{\text{odd}}(n) & \text{if } n \text{ is odd} \end{cases}$$

Give a Big-Ω and Big-O bound on $T(n)$ using your previous results. If a Big-Θ bound for the entire algorithm exists, describe it. If not, explain why it doesn't exist.

Solution:
Big-Ω bound = n Big-O bound = n^3
Big-Θ bound does not exist as there is no 2 c values that satisfies a function that fits the upper and lower bound for both T_{even} and T_{odd}

(g) (2 marks) Your classmate tells you that Big-O represents the worst case runtime of an algorithm, and similarly that Big-Ω represents the best case runtime. Is your classmate correct? Explain why/why not. Your answers for (e) and (f) *may* be useful for answering this.

Solution:
This is correct as the Big-O for represents the worst case runtime while the Big-Ω represents the best case runtime. For this given algorithm, the Big-O for the best case scenario is O(n). Comparing this to the Big-O bound for the entire algorithm, O(n^3). Since O(n^3) > O(n)
$\therefore O(n^3)$ represents the worst case scenario.
For the Big-Ω bound on the worst case runtime, f(n) = $\Omega(n^2)$ > Ω(n)
$\therefore\Omega$(n) is the best case scenario.

(h) (1 mark) Prove that an algorithm runs in $\Theta(g(n))$ time if and only if its worst-case running time is $O(g(n))$ and its best-case running time is $\Omega(g(n))$.

Solution:
Let T(n) = runtime of an algorithm.
By definition, T(n) = Θ(g(n)) if there exists constants c_1 and c_2

such that c_1 * g(n) $\leq$T(n) $\leq$ c_2 * g(n) for all n $\geq$ n_0
Let T_{worst}(n) be the worst-case scenario and T_{best}(n) be the best case scenario,
0 $\leq$ c_1 g(n) $\leq$ T_{best}(n) for all n > n$_{best}$ $= \Omega$(g(n))
0 $\leq$ T$_{worst}$(n) $\leq$ c$_2$ * g(n) for all n > n$_{worst}$ $= O(g(n))$
$\therefore$ 0 $\leq$ c_1 g(n) $\leq$ T_{best} $\leq$ T_{worst} $\leq$ c_2 g(n) for n > $n_{best,worst}$ where O(n) is the worst case running time and Ω(n) is the best case running time.

2. (a) (4 marks) Devise a **recursive** algorithm that takes a sorted array A of length n, containing distinct (not necessarily positive) integers, and determines whether or not there is a position i (where $0 \leq i < n$) such that $A[i] = i$.

- Write your algorithm in pseudocode (as a procedure called FindPosition that takes an input array A and returns a boolean).
- Your algorithm should be as efficient as possible (in terms of time complexity) for full marks.
- You will not receive any marks for an iterative solution for this question.
- You are permitted (and even encouraged) to write helper functions in your solution.

Solution:
Input: a sorted array, A
Output: boolean, whether or not there is a positive integer i such that A[i] = i ; 0 ≤ i < n

```
function FindPosition($A, n$)
    $start \leftarrow 0$
    $end \leftarrow n - 1$
    return SEARCH(A, start, end)
end function
function SEARCH($A, first, last$)
    if $last < first$ then
        return $false$
    end if
    $mid \leftarrow \lfloor (first + last)/2 \rfloor$
    if $A[mid] < mid$ then
        return SEARCH(A, (mid+1), last)
    else if $A[mid] > mid$ then
        return SEARCH(A, first, (mid-1))
    else
        return $true$

```

(b) (1 mark) Show and explain all the steps taken by your algorithm (e.g. show all the recursive calls, if conditions, etc) for the following input array: $[-1, 0, 2, 3, 10, 11, 23, 24, 102]$.

Solution:
FindPosition will call SEARCH(A, 0, 8) with A being the sorted array.
Line 7 of the code will be checked but the if statement's condition will not be executed.
Line 10 will be executed, mid = 4
Line 11 will be checked but 10 is not < 4.
Line 13 will be executed and SEARCH will be called recursively with SEARCH(A, 0, 3)
This is the first recursive call of SEARCH. Line 7 is checked but not executed. Line 10 will be executed, mid = 1
Line 11 will be executed since 0 < 1, SEARCH will be called again recursively with SEARCH(A, 2, 3)
Line 7 will be checked again but not executed. Line 10 will be executed, mid = 2
Line 11 will be checked but 2 is not < 2
Line 13 will be checked but 2 is not > 2
Line 16 will be executed and returns true. The recursion breaks and Line 4 will return true.

(c) (3 marks) Express the worst-case running time of your algorithm as a mathematical recurrence, $T(n)$, and explain your reasoning. Then calculate a Big-O (or Big-Θ) bound for this recurrence and show all working used to find this bound (Note: using the Master Theorem below for this question will not give you any marks for this question).

Solution:

$$T(n) = \begin{cases} O(1) & \text{for } n > 0 \\ O(1) + T(\frac{n}{2}) & \text{for } n = 0 \end{cases}$$

FindPosition runs in O(1) and SEARCH runs in O(1) if n=0
∴ if n=0, the algorithm runs in constant time O(1) since only Line 8 of the SEARCH algorithm is

executed. The check will not depend on the size of A.
If the SEARCH algorithm is executed recursively at least once, then the time it would take would be $T(\frac{n}{2})$ as only half of the function's array is being processed.
To calculate Big-O bound:
For $T(\frac{n}{2})$, every node has 1 child and each child has $\frac{n}{2}$ input size from A. The height is log_2n from $1 = \frac{n}{2^h}$.
Total complexity = number of levels * work done at each level

$$= \sum_{i=0}^{\log_2 n} n * (\frac{1}{2})^i$$

$$= n * \sum_{i=0}^{\log_2 n} (\frac{1}{2})^i$$

Using

$$\sum_{i=0}^{h} ar^i = \frac{a(r^{h+1} - 1)}{r - 1}$$

$$= \frac{n * (\frac{1}{2})^{\log_2 n+1} - 1}{\frac{1}{2} - 1}$$

$$-2n((\frac{1}{2})^{\log_2 n+1} - 1)$$

$$= -2n(\frac{1^{\log_2 n+1}}{2^{\log_2 n+1}} - 1)$$

$$= \frac{-2n}{2n}(1^{\log_2 n+1} - 2n)$$

$$= -1 * 1^1 * 1^{\log_2 n} - 2n$$

$$= -\log_2 n - 2n \in O(\log_2 n)$$

(d) The master theorem is a powerful theorem that can be used to quickly calculate a tight asymptotic bound on a mathematical recurrence. A simplified version is stated as follows: Let $T(n)$ be a non-negative function that satisfies

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + g(n) & \text{for } n > k \\ c & \text{for } n = k \end{cases}$$

where k is a non-negative integer, $a \geq 1$, $b \geq 2$, $c > 0$, and $g(n) \in \Theta(n^d)$ for $d \geq 0$. Then,

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

i. (1 mark) Use the master theorem, as stated above, to find a Big-Θ bound (and confirm your already found Big-O) for the recurrence you gave in (b). Show all your working.
Solution:
Runtime of T(n) = T($\frac{n}{2}$) + 1
a = 1, b=2 , c= 1 ; these satisfy the conditions as 1 = 1, 2 = 2, 1 > 0 and d =0
By using the second case for Θ, Θ(n^{0}log n) since 1 = 2^0
$\therefore$ T(n) $\in$ Θ(log n)

ii. (1 mark) Use the master theorem to find a Big-Θ bound for the recurrence defined by

$$T(n) = 5 \cdot T\left(\frac{n}{3}\right) + n^2 + 2n$$

and $T(1) = 100$. Show all working.
Solution:
a= 5 , b=3 , c=1, d=2 ; these satisfy the conditions as 5 >1, 3 > 2, 1 >0 and 2 > 0 for d
3^2 > 5 ; this matches the first case of the master theorem.
$\therefore$ T(n) $\in$ Θ(n^2)

iii. (1 mark) Use the master theorem to find a Big-Θ bound for the recurrence defined by

$$T(n) = 8 \cdot T\left(\frac{n}{4}\right) + 5n + 2\log n + \frac{1}{n}$$

and $T(1) = 1$. Show all working.
Solution:
a= 8, b= 4, c= 1, d= 1; these satisfy the conditions as 8 > 1, 4 > 2, 1 > 0 and 1 > 0 for d
4 < 8 ; this matches the third case of the master theorem.
$\therefore$ T(n) $\in$ Θ(n$^{\log_4 8}$)

(e) (2 marks) Rewrite (in pseudocode) the algorithm you devised in part (a), but this time **iteratively**. Your algorithm should have the same runtime complexity of your recursive algorithm. Briefly explain how you determined the runtime complexity of your iterative solution.
Solution:

```
function FindPosition($A, n$)
    left ← 0
    right ← n-1
    while left ≤ right do
        $mid \leftarrow \lfloor (left + right)/2 \rfloor$
        if A[mid] = mid then
            return true
        else if mid < A[mid] then
            right ← mid - 1
        else
            left ← mid + 1
        end if
    end while
    return false
```

The height of this search tree would be log$_2$n; since h = $\frac{n}{2^h}$ = 1 , h = log$_2$n.
This is the same as the recursive method.
$\therefore$ Runtime complexity is O($\log_2 n$)

(f) (2 marks) While both your algorithms have the same runtime complexity, one of them will usually be faster in practice (especially with large inputs) when implemented in a procedural programming language (such as Java, Python or C). Explain which version of the algorithm you would implement in Java - and why - if speed was the most important factor to you. You may need to do external research on how Java method calls work in order to answer this question in full detail. Cite any sources you used to come up with your answer.

In addition, explain and compare the space complexity of your both your recursive solution and your iterative solution (also assuming execution in a Java-like language).

Solution:

3. In the support files for this homework on Blackboard, we have provided an interface called `CartesianPlane` which describes a 2D plane which can hold elements at (x, y) coordinator pairs, where x and y could potentially be negative.

   (a) (5 marks) In the file `ArrayCartesianPlane.java`, you should implement the methods in the interface `CartesianPlane` using a multidimensional array as the underlying data structure.

   Before starting, ensure you read and understand the following:

   - Your solution will be marked with an automated test suite.
   - Your code will be compiled using Java 11.
   - Marks may be deducted for poor coding style. You should follow the CSSE2002 style guide, which can be found on Blackboard.
   - A sample test suite has been provided in `CartesianPlaneTest.java`. This test suite is not comprehensive and there is no guarantee that passing these will ensure passing the tests used during marking. It is recommended, but not required, that you write your own tests for your solution.
   - You may not use anything from the Java Collections Framework (e.g. ArrayLists or HashMaps). If unsure about whether you can use a certain import, ask on Piazza.
   - Do not add or use any static member variables. Do not add any **public** variables or methods.
   - Do not modify the interface (or `CartesianPlane.java` at all), or any method signatures in your implementation.

   (b) (1 mark) State (using Big-O notation) the memory complexity of your implementation, ensuring you define all variables you use. Briefly explain how you came up with this bound.

   Solution:
   Given that the array either contains an element or null value, the 2D array size size is N * N storing M elements.
   ∴ the memory complexity is O(M) or O(N^2)

   (c) (1 mark) Using the bound found above, evaluate the overall memory efficiency of your implementation. You should especially consider the case where your plane is very large but has very few elements.

   Solution:
   If the array size is very large but with very few inputs, then the data structure will become very inefficient. There will be coordinates where no element is present. However, this can be useful if a scenario where specific coordinates with large input sizes are required. It will be more efficient if the array is also sorted as retrieving the element could be achieved with a search algorithm. This type of array is also faster when it is accessed by rows (i.e. the element is stored in a lower index row) vs being accessed by columns.

   (d) (3 marks) State (using Big-O notation) the time complexity of the following methods:

   - `add`
   - `get`
   - `remove`
   - `resize`
   - `clear`

   Ensure you define all variables used in your bounds, and briefly explain how you came up with the bounds. State any assumptions you made in determining your answers. You should simplify your bounds as much as possible.

   Solution:
   Assumptions: Takes 1 unit of time for arithmetic, logical operations, assignment and return.
   add: O(1) for the comparative operations and assignment of element to the array index.There are 9 primitive operations.
   ∴ Overall complexity is constant time, O(1)

   get: O(1) for the comparative operations and return of the array. There are 9 primitive operations.
   ∴ Overall complexity is constant time, O(1)

   remove: 9 primitive operations ,O(1) for accessing get method, O(1) for comparative, assignment and

return statement; 4 primitive operations.
∴ Overall complexity is constant time, O(1)

clear: for the nested for loop, best and worst case time complexity is O(n*m) where n would be traversing through with i variable and m would be traversing through with j variable.
∴ Overall complexity is O(n*m) or if n=m, complexity is O(n^2)

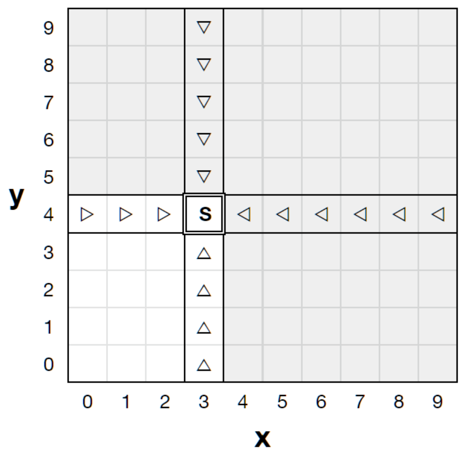
resize: O(1) for the if statements comparing the 4 bound variables of the Cartesian plane. It is accessing a helper method which has a nested for loop. This will run O(n*m) where n is the x bounds and m is the y bounds. If n=m, then this will run O(n^2).

4. The UQ water well company has marked out an $n \times n$ grid on a plot of land, in which their hydrologists know exactly one square has a suitable water source for a water well. They have access to a drill, which uses drill bits and can test one square at a time. Now, all they they need is a strategy to find this water source.

Let the square containing the water source be (s_x, s_y). After drilling in a square (x, y), certain things can happen depending on where you drilled.

- If $x > s_x$ or $y > s_y$, then the drill bit breaks and must be replaced.
- If $x = s_x$ or $y = s_y$, the hydrologists can determine which direction the water source is in.

Note that both the above events can happen at the same time. Below is an example with $n = 10$ and $(s_x, s_y) = (3, 4)$. The water source is marked with **S**. Drilling in a shaded square will break the drill bit, and drilling in a square with a triangle will reveal the direction.



(a) (3 marks) The UQ wat er well company have decided to hire you - an algorithms expert - to devise a algorithm to find the water source as efficiently as possible.

Describe (you may do this in words, but with sufficient detail) an algorithm to solve the problem of finding the water source, assuming you can break as many drill bits as you want. Provide a Big-O bound on the number of holes you need to drill to find it with your algorithm. Your algorithm should be as efficient as possible for full marks.

You may consult the hydrologists after any drill (and with a constant time complexity cost to do so) to see if the source is in the drilled row or column, and if so which direction the water source is in.

(Hint: A linear time algorithm is not efficient enough for full marks.)

Solution: Drill in the centre $[\frac{x}{2}]$, $[\frac{y}{2}]$.Consult the hydrologist after each drill. There will be 4 cases. Case 1, the drill breaks and the direction is unknown. For this case, move diagonally down (i.e. x-1, y-1 from current position).
Case 2, drill doesn't break but direction is known. For this case, move diagonally up (i.e. x+1, y+1 from current position).
For case 3 and 4, you will need to find the direction since both cases have directions that can be determined. Case 3, drill breaks and direction is known. For this case, move -ve towards direction (i.e. if direction is towards x, move x-1 keeping other coordinate constant) Case 4, drill doesn't break and direction is known. For this case, move positive towards direction.
To find the direction, try decreasing x by 1, if direction can still be determined, then keep moving towards x, otherwise move in y direction; +ve or -ve depending on case 3 or case 4.
keep repeating and check for each case after each drill.

Big-O bound for the number of holes drilled will be halved. This is similar to binary search where the runtime is $O(\log n)$

(b) (5 marks) The company, impressed with the drilling efficiency of your algorithm, assigns you to another $n \times n$ grid, which also has a water source you need to help find. However, due to budget cuts, this time you can only break 2 drill bits (at most) before finding the source. (Note that you are able to use a 3rd drill bit, but are not allowed to ever break it).

Write **pseudocode** for an algorithm to find the source while breaking at most 2 drill bits, and give a tight Big-O bound on the number of squares drilled (in the worst case). If you use external function calls (e.g. to consult the hydrologist, or to see if the cell you drilled is the source) you should define these, their parameters, and their return values.

Your algorithm's time complexity should be as efficient as possible in order to receive marks. (Hint: A linear time algorithm is not efficient enough for full marks.)

Solution: