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 ← 0        1
3:     if n % 2 == 0 then      2
4:         for i = 0 to n do        1+(n+1)+n
5:             for j = i to n² do        1 + n² - i + 1 + n² - i
6:                 sum ← sum + i + j        3
7:             end for
8:         end for
9:     else
10:        while n > 0 do        log(n)
11:            sum ← sum + (n & 1)        3
12:            n ← (n >> 1)        2
13:        end while
14:    end if
15:    return sum        1
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:
$T_{\text{odd}}(n) = (2 + 3) * \log(n) + 1 + 2 + 1 = 5\log(n) + 4$
Explanation: There is 1 operation in sum = 0, then 2 operations in **if**, 1 operation in return, then the while loop operates $\log(n)$.Celling times which represent the number of bits in n's binary. Also There are total 5 operation in while loop.

(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:
$T_{\text{odd}}(n) \in O(log(n))$
when $c = 8$, $n_0 = 4$.
$5\log(n) + 4 \le 8log(\text{n}) \quad for \; n \ge 4$

(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:
$T_{\text{odd}}(n) \in \Omega (\log(n))$
when $c = 2$, $n_0 = 4$.
$5\log(n) + 4 \geq 2log(n) \quad for \; n \geq 4$

$T_{\text{odd}}(n) \in \Theta (\log(n))$
Because in $T_{\text{odd}}(n)$, there exists Big-Ω and Big-O bounds that are equal.

(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:
I have written the operations in each step on the right of the code which is provided for us. But in this case we can not calculate the total operations by just easily add and time each steps up because there is an i. So I'm going to treat these two for loop as a whole. Then the operation of $j < n^2$ can be represented by $\sum_{i=0}^{n+1} n^2 - i$. And **j++** *can be represented by* $\sum_{i=0}^{n} n^2 - i$.

$T_{\text{even}}(n) = 1 + 2 + 1 + (n + 1) + n + 1 + (\sum_{i=0}^{n+1} n^2 - i) + (\sum_{i=0}^{n} n^2 - i) + 3 * (\sum_{i=0}^{n} n^2 - i) + 1$
$= 5n^3 + 3.5n^2 + 5.5n + 8$

$T_{\text{even}}(n) \in O(n^3)$
$T_{\text{even}}(n) \in \Omega(n^3)$

(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:
best case: $T_{\text{odd}}(n) \in O(log(n))$
worst case $T_{\text{even}}(n) \in \Omega(n^3)$

(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:

$$T(n) = \begin{cases} T_{\text{even}}(n) \in \Omega(n^3) & \text{if } n \text{ is even} \\ T_{\text{odd}}(n) \in O(log(n)) & \text{if } n \text{ is odd} \end{cases}$$

No, there not exists a Big-Θ bound for the entire algorithm, because the Big-Ω and Big-O bounds that are not equal.

(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:
In my opinion, i think this is half half. The best case and worst case represent specific operation times.

2

For example, when n is odd we can say the best case is when n = 1 and the running time is 4. But this is fairly useless for analysis. So usually we talk about the best/worst case for certain types of input like n is even or odd in this question, then we can use Big-O represents the worst case and Big-Ω represents the best case runtime.

(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:
proof1:
Let f(n) represent the algorithm, where f(n) $\in \Theta(g(n))$:

There exist positive constants $c1$, $c2$, n_0
$c1 * g(n) \leq f(n) \leq c2 * g(n)$ $\qquad for$ n$\geq$n$_0$

Because there exist $c1$, n_0, that
$c1 * g(n) \leq f(n)$ $\qquad for$ n$\geq$n$_0$
Then f(n) $\in \Omega(g(n))$

Because there exist $c2$, n_0, that
f(n) $\leq c2 * g(n)$ $\qquad for$ n$\geq$n$_0$
Then f(n) $\in O(g(n))$

$proof2$:
$Let\ f(n)\ represent\ the\ algorithm,\ where\ f(n) \in O(g(n))\ and\ f(n) \in \Omega(g(n)).$

There exist $c1$, n_0, that
$c1 * g(n) \leq f(n)$ $\qquad for$ n$\geq$n$_0$
There exist $c2$, n_0, that
f(n) $\leq c2 * g(n)$ $\qquad for$ n$\geq$n$_0$

As a result, there exist positive constants $c1$, $c2$, n_0
$c1 * g(n) \leq f(n) \leq c2 * g(n)$ $\qquad for$ n$\geq$n$_0$
then f(n) $\in \Theta(g(n))$

From proof1 and proof2, we can conclude 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 $\Omega(g(n))$.

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 $0 \le i < n$) 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:
   Algorithm FindPosition(A, left, right)
   . **Input**: a sorted array A
   . **Output**: whether or not there is a position i (where $0 \le i < n$) such that $A[i] = i$.
   . if right < left then
   .   return false
   . m ← $\lfloor (left + right) \div 2 \rfloor$
   . if $A[m] < m$ then
   .   return FindPosition(A, m + 1, right)
   . else if $A[m] > m$ then
   .   return FindPosition(A, left, m - 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:
   . A[4] > 4
   .   FindPosition(A, 0, 3)
   . A[1] < 1
   .   FindPosition(A, 2, 3)
   . A[2] == 2
   .   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:
   The worst-case is when there is no **i** that A[i] = i, or there is an **i** that A[i] = i when we search until we divide this array into **n** part.

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

   The total operations is $\log_2 n + \log_2 n$
   Then $T(n) \in O(log(n))$

   (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:
From the recurrence I gave in (b). we can have:
a = 1, b = 2, c = 1, d = 0
Because $b^d = 2^0 = 1 = a$
Then $T(n) \in \Theta(log(n))$

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 = 100, d = 2
Because $b^d = 3^2 = 9 > a$
Then $T(n) \in \Theta(n^2))$

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
Because $b^d = 4^1 = 4 < a$
Then $T(n) \in \Theta(n^{1.5}))$

(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:
Algorithm FindPosition(A, left, right)
.   **Input**: a sorted array A
.   **Output**: whether or not there is a position i (where $0 \leq i < n$) such that $A[i] = i$.
.   while left < right do
.   m $\leftarrow \lfloor (left + right) \div 2 \rfloor$
.   if $A[m] < m$ then
. $left = m + 1$
.   else if $A[m] > m$ then
. $right = m - 1$
.   else
.   return true
.   return false

Explanation: The worst-case is when there is no **i** that A[i] = i, or there is an **i** that A[i] = i when we search until we divide this array into **n** part.   Which means left = right, the times we divided are the total iterations which is $\log_2 n$.

(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:
I will implement this algorithm by using iteration. Firstly, we can find that the code size, runtime complexity, input size and output size in both version are almost same. Then recursion has to store all function calls in a stack to return back to the caller functions but iteration need not, which leads to recursion take up more memory size and executes slower than iteration in this case. So in this case, using iteration is better than recursion. ("What is the difference between iteration and recursion?", 2020)

The space complexity of recursion $\in O(\text{nlog(n)})$
The space complexity of iteration $\in O(\text{n})$

Recursion solution has bigger space complexity than iteration solution.

Reference List :
What is the difference between iteration and?.(2020).Retrieved 16 *August* 2020, *from*
https : *//afteracademy.com/blog/what − is − the − difference − between − iteration − and − recursion*

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:
The memory complexity $\in O(n)$ where n represents total number of cells. Because in constructor I create a grid which contain n cells inside and other variables are constant.

(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:
The memory efficiency of my implementation which memory complexity belong to O(n) is very low when my plane is very large but has very few elements. Lots of cells take up memories but storing a null value.

(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:

$add \in O(1)$
$get \in O(1)$
$remove \in O(1)$
$resize \in O(n^2)$
$clear \in O(n^2)$
If we assume N is the number of cells in the plane, then the Big-O of `resize` and `clear` actually is O(N), although their is a for loop inside a for loop in both method.

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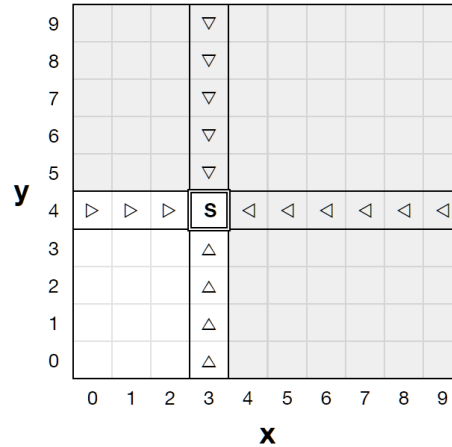
(a) (3 marks) The UQ water well company have decided to hire you - an algorithms expert - to devise a algorithm to find the water source as efficiently as possible.

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:
Firstly, do binary search on the squares which is on the diagonal of the grid which is y = x, each time ask hydrologists until find both squares each of them indicate the horizontal(s_y, s_y) and vertical(s_x, s_x) water source direction.   Then the position (s_x, s_y) is the water source.

The number of holes you need to drill to find $\in O(log(n))$.

(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:

/**

This algorithm is to find the position of the water source, which along the direction of y=x, then search every 3 steps. The input x and y at first is (0,0).

*/

Algorithm FindWaterSource(A, x, y)

. **Input**: an $n \times n$ grid A, and the length of A, n and the coordinate of the last drilled square x and y.

. **Output**: the square position of the water source (x,y)

. if IsWaterSource(x, y) then

.   return (x, y)

. if IsDrillbroke then

.   return FindWaterSource(A, x - 1, y - 1)

. else

.   if IsOnDirection(x, y) then

.   if DirectionIsUp then

.   return FindWaterSource(A, x, y + 1)

.   else if DirectionIsRight then

.   return FindWaterSource(A, x + 1, y)

.   else if DirectionIsDown then

.   return FindWaterSource(A, x, y - 1)

.   else

.   return FindWaterSource(A, x - 1, y)

.   else

.   return FindWaterSource(A, x + 3, y + 3)

FindWaterSource $\in O(n)$ *where* n *is the length of the grid.*

5. **(COMP7505 only)** Binary search is fast because each step halves the search array. Can we do better? Maybe we could *quarter* the input at every step. Would this make the algorithm faster?

This "quaternary search" algorithm takes a sorted array A of length n and an integer x. It determines which quarter of A the value x would occur in, then recurses into that subarray of length $n/4$. When reaching an array of length $n \leq 4$, it returns the index of x or -1 if x cannot be found.

(a) (3 marks) From the description above, write pseudocode for a quaternary search algorithm.
Solution:

(b) (1 mark) Express the worst case running time of quaternary search as a recurrence.
Solution:

(c) (2 marks) Solve the recurrence above to determine a Big-O bound on the worst case running time of quaternary search.
Solution:

(d) (1 mark) Is quaternary search faster than binary search? If so, is it substantially faster? Explain briefly.
Solution:

(e) (3 marks) What if we go *even further*? Suppose we can do k-ary search, which reduces n to n/k at each step (e.g. binary is 2-ary search). With (c) in mind, we hypothesise that k-ary search has complexity $O(\log_k n)$.

Now, we can use n-ary search to search inside an n-length array. Our search finishes in one iteration because $n/n = 1$ and $O(\log_n n) = O(1)$. We've solved algorithms!

Unfortunately, nothing is that easy (or we wouldn't be here). Explain why this is *not* the case and comment on the actual performance of k-ary search.
Solution: