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
3:      if n % 2 == 0 then
4:          for i = 0 to n do
5:              for j = i to n^2 do
6:                  sum ← sum + i + j
7:              end for
8:          end for
9:      else
10:         while n > 0 do
11:             sum ← sum + (n & 1)
12:             n ← (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: Assumptions:

i. Bitwise, assignment and mathematical operations are done in constant time.

Operations:

i. Initialise sum: constant (1)

ii. If comparison: constant (1)

   A. While loop: runs $(\lfloor log_2(n) \rfloor + 1) + 1$ times

   B. Inner sum update: constant (1)

   C. Inner n update: constant (1)

iii. Inner total: 2 constant operations run $\lfloor log_2(n) \rfloor + 1$ times. Complexity: $2 * \lfloor log_2(n) \rfloor + 2$

Thus the total sum of operations is $2 * \lfloor log_2(n) \rfloor + 5$.
$\therefore$ The number of primitive operations performed when n is odd is $T_{odd}(n) = 2 * \lfloor log_2(n) \rfloor + 5$.

(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:

Using the result from 1.a, $T_{odd}(n) = 2 * \lfloor log_2(n) \rfloor + 5$.

Guessing a function g(n) of the form: $g(n) = log_2(n)$.

We need:

$$T_{odd}(n) = 2 * \lfloor log_2(n) \rfloor + 5 \leq c * g(n), \forall n \geq n_0 \tag{1}$$

Thus,

$$T_{odd}(n) = 2 * \lfloor log_2(n) \rfloor + 5 \leq 2 * log_2(n) + 7 \leq c * log_2(n), \forall n \geq n_0 \tag{2}$$

This inequality must be satisifed:

$$2 * log_2(n) + 7 \leq c * log_2(n) \tag{3}$$

Selecting c = 7 and $n_0 = 2$ we get that:

$$T_{odd}(2) = 2 * log_2(2) + 5 \leq 7 * log_2(2)$$
$$T_{odd}(2) = 7 \leq 7$$

With c = 7 and $n = 3$:

$$T_{odd}(2) = 2 * log_2(3) + 5 \leq 7 * log_2(3)$$
$$T_{odd}(2) = 8.16 \leq 11.09$$

Since,

- g(n) is stricly increasing for n $\geq$ 2
- and, $\lfloor log_2(n) \rfloor \leq log_2(n), \forall n \geq 1$

Thus, inequality (1) is satisfied with c = 7 and $n_0 = 2$

$$\therefore T_{odd}(n) \text{ has a runtime complexity of O}(log_2(n))$$

(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:

Using the result from 1.a, $T_{odd}(n) = 2 * \lfloor log_2(n) \rfloor + 5$.

Guessing a function g(n) of the form: $g(n) = log_2(n)$.

We need:

$$T_{odd}(n) = 2 * \lfloor log_2(n) \rfloor + 5 \geq c * g(n), \forall n \geq n_0 \tag{4}$$

Selecting c = 1 and $n_0 = 1$ we get that:

$$T_{odd}(2) = 2 * \lfloor log_2(1) \rfloor + 5 \geq 7 * log_2(1)$$
$$T_{odd}(2) = 5 \geq 0$$

Since,

- g(n) and $\lfloor log_2(n) \rfloor$ is strictly positive for n $\geq$ 2
- and, $\lfloor log_2(n) \rfloor + 1 \geq log_2(n), \forall n \geq 1$ (by the nature of the floor function)
- Consequently, the inequality $2 * \lfloor log_2(n) \rfloor + 5 \geq log_2(n), \forall n \geq 1$ is also true

Thus, inequality (2) is satisfied with c = 1 and $n_0 = 1$

$$\therefore T_{odd}(n) \text{ has a runtime complexity of } \Omega(log_2(n))$$

In order for a Big-Θ bound for $T_{odd}(n)$ to exist there must exist a g(n), c_1, c_2 and n_0 such that:

$$c_1 * g(n) \leq T_{odd}(n) \leq c_2 * g(n), \forall n \geq n_0 \; T_{odd}(n) = c * g(n) \; 2 * \lfloor log_2(1) \rfloor + 5 = c * g(n)$$

Thus by using our results from above, Big-Θ bound for $T_{odd}(n)$ is $\Theta(log_2(n))$ with $c_1 = 1, c_2 = 7$ and $n_0 = max(1, 2)$.

(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:

- sum $\rightarrow$ 0 (1)
- if (1)
- for i (n+1)
  - for j (n^2)
    - sum $\leftarrow$ sum + i + j (1)
- $T_{even}(n) = 1 + 1 + (n+1) * n^2 * 1 = n^3 + n^2 + 2$

Thus ignoring lower order terms:
Big-O bound: O(n^3)
Big-Ω: Ω(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 Big-O Bound (n is odd): O($log_2(n)$)
Worst-case Big-Ω Bound (n is even): Ω(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:
The Big-O and Big-Ω bounds for T(n) are the Big-O bound for the worst case and the Big-Ω bound for the best case.
These have been calculated above and are

- Big-O for worst case: O(n^3) $\implies$ $T(n) \in O(n^3)$
- Big-Ω for best case: Ω($log_2(n)$) $\implies$ $T(n) \in \Omega(log_2(n))$

Since the functions (g(n)s) for the Big-O and Big-Ω case are not equal (since $T(n)$ is a piecewise function composed of functions of different runtimes) a Big-Θ bound DNE as there exists no such $g(n)$, c_1, c_2 and n_0 such that $c_1 * g(n) \leq T(n) \leq c_2 * g(n), \forall n \geq n_0$.

(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: The classmate is correct. By definition, Big-O represents an upper-bound on the runtime of an algorithm and thus represents the worst (highest) possible runtime of an algorithm. Big-Ω represents an lower-bound on the runtime of an algorithm and thus represents the best (lowest) possible runtime of an algorithm. This is exemplified in the solutions to e) and f) which show that the best case bound for T(n) is the Big-Ω bound for the best case sub-function in the piecewise function ($T_{\text{odd}}(n)$). Conversely, the worst case bound for T(n) is the Big-O bound for the worst case sub-function in the piecewise function ($T_{\text{even}}(n)$).

(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:
Proof ($\rightarrow$):

Assume that an algorithm runs in $\Theta(g(n))$ time, where T(n) represents the running time of that algorithm.
Then the following inequality holds true:

$$c_1 * g(n) \leq T(n) \leq c_2 * g(n), \forall n \geq n_0$$

Examining the first half of the inequality we obtain:

$$c_1 * g(n) \leq T(n), \forall n \geq n_0$$

This statement is equivalent to T(n) is lower-bounded by $\Omega(g(n))$ (best case)
Examining the second half of the inequality we obtain:

$$T(n) \leq c_2 * g(n), \forall n \geq n_0$$

This statement is equivalent to T(n) is upper-bounded by $O(g(n))$ (worst case)
$\therefore$ The right implication holds.

Proof ($\leftarrow$):
Now considering the reverse implication:
Assume that the runtime of an algorithm (T(n)) is bounded by $\Omega(g(n))$ and $O(g(n))$.
Thus the following inequalities are true:

- $c_1 * g(n) \leq T(n), \forall n \geq n_1$
- and, $T(n) \leq c_2 * g(n), \forall n \geq n_2$

Consequently, taking the max of n_1 and n_2 we are able to combine the inequalities producing the following inequality:

$$c_1 * g(n) \leq T(n) \leq c_2 * g(n), \forall n \geq n_0 \text{ where } n_0 = max(n_1, n_2)$$

This statement is equivalent to T(n) $\in \Theta(g(n))$.
$\therefore$ The left implication holds. $\square$

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:

```
1: procedure FINDPOSITION(int A[])
2:     Input: a sorted array A.
3:     Output: (bool) whether there exists an i such that i == A[i]
4:     size ← length(A)
5:     return BINSEARCH(A, 0, size − 1)
6: end procedure

7: procedure BINSEARCH(int A[], int min, int max)
8:     Input: a sorted array A, min and max indices of search area.
9:     Output: whether there exists an i such that i == A[i]
10:    if max ≥ min then
11:        mid ← ⌊(min + max)/2⌋
12:        if mid == A[mid] then
13:            return true
14:        end if
15:        if mid > A[mid] then
16:            return BINSEARCH(A, (mid + 1), max)
17:        else
18:            return BINSEARCH(A, min, (mid − 1))
19:        end if
20:    end if
21:    return false
22: end procedure
```

   Assumptions:

   - length(A) returns the length of A and runs in O(1) time.
   - returns run in O(1) time.
   - Floor function runs in O(1) time.

   (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: Step-By-Step Runthrough

   i. $FINDPOSITION(A = [-1, 0, 2, 3, 10, 11, 23, 24, 102])$ (1)

   ii. $BINSEARCH(A, 0, 9 - 1)$ (5): Calls driver function with calculated min and max indices.

   A. (max ≥ min) == true (if triggered) (10): Passes base case test.

   B. mid = 4 (11): Calculates the middle index.

   C. (mid < A[mid]) == True (else triggered) (17): Fails return case then succeeds the else clause (i.e if i does exists such that A[i] == i then it must exist in the bottom half of the current search area (since the array is sorted and contains distinct elements))

   D. return BINSEARCH(A, 0, 3) (18): (if an i exists such that i == A[i], it is in the lower half of the array)

   - mid = 2 (11): Calculate new middle point.
   - (mid == A[mid]) == True (first if triggered) (12): Passes return case.

- return true (13): Element found that satisfies condition so true is returned.

iii. Cascades down call stack to **return** *true* in FINDPOSITION.

(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:

```
1:  procedure FINDPOSITION(int A[])
2:      size ← length(A) (1)
3:      return BINSEARCH(A, 0, size − 1) (1)
4:  end procedure

5:  procedure BINSEARCH(int A[], int min, int max)
6:      if max ≥ min then
7:          mid ← ⌊(min + max)/2⌋ (1)
8:          if mid == A[mid] then
9:              return true (1)
10:         end if
11:         if mid > A[mid] then
12:             return BINSEARCH(A, (mid + 1), max) (1)
13:         else
14:             return BINSEARCH(A, min, (mid − 1)) (1)
15:         end if
16:     end if
17:     return false (1)
18: end procedure
```

Recursive Relation of BINSEARCH:

- In the worst case (i == A[i] DNE):
  - Each time BINSEARCH is called it is called on (in the worst case (even array)) a subarray of length $(n/2)$ and for each call atmost 2 operations must be performed.
- Thus the recurrence relation is T(n/2) + 2.
- In order to fully define this relation a base case must be established.
- Thus using the base case for n = 1 we obtain, T(n) = 3, for $n = 1$
- Additionally, a edge case for n = 0 was created as the algorithm can take empty arrays
- Combining these cases we get a complete function for $T_{BIN}(n)$:
  - $T_{BIN}(n) = \begin{cases} 1 & \text{if n} = 0 \\ 3 & \text{if n} = 1 \\ T(\lfloor n/2 \rfloor) + 2 & \text{if n} > 1 \end{cases}$
- Now including the inital call to FINDPOSITION:
  - $T(n) = \begin{cases} 3 & \text{if n} = 0 \\ 5 & \text{if n} = 1 \\ T(\lfloor n/2 \rfloor) + 4 & \text{if n} > 1 \end{cases}$
- Note: In order to handle the case when n is odd a floor function has been applied to $n/2$ to ensure it is an integer, however this does not affect the case when n is even and has a no effect on the worst case runtime of the algorithm.

Guess that $T(n) \in O(log_2(n))$.

Proof by induction:

Base Case:

- $n_1 = 2$
- $T(2) = T(1) + 4 = 5 + 4 = 9 \leq 10 * log_2(2)$

Assume that $T(n_1) \leq c * log_2(n_1)$ for $n_1 < n$:

- $T(n) = T(\lfloor n/2 \rfloor) + 4 \leq c * log_2(\lfloor n/2 \rfloor) + 4 \leq c * log_2(n/2) + 4$
- $T(n) = T(\lfloor n/2 \rfloor) + 4 \leq c * log_2(n/2) + 4$
- $T(n) = T(\lfloor n/2 \rfloor) + 4 \leq c * log_2(n) - c * log_2(2) + 4$
- $T(n) = T(\lfloor n/2 \rfloor) + 4 \leq c * log_2(n) - c * log_2(2) + 4 \leq c * log_2(n)$, for $c \geq 4$

Thus, in the worst case (n > 0): T(n) $\in O(log_2(n))$ with $c = max(4, 10)$ and $n_0 \geq 2$.

(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:
The floor function is omitted out of T(n) and instead integer division is assumed to truncate down while utilising the master function.
Additionally the following function definition of T(n) was used (in order to apply the master function):

- $T(n) = \begin{cases} 5 & \text{if n = 1} \\ T(n/2) + 4 & \text{if n > 1} \end{cases}$

T(n) is in the form of the master theorem with:

- a = 1
- b = 2
- c = 5
- g(n) = 4 $\in \Theta(n^d)$ for $d = 0$
- d = 0

Thus since a = 1 = $2^0 = b^d$, $T(n) \in \Theta(log(n))$ and the Big-O bound found above ($O(log_2(n))$) is correct.

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:
T(n) is in the form of the master theorem with:

- a = 5
- b = 3
- g(n) = $n^2 + 2n \in \Theta(n^d)$ for $d = 2$

Thus since a = 5 < $3^2 = b^d$, $T(n) \in \Theta(n^2)$

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:
T(n) is in the form of the master theorem with:

- a = 8
- b = 4
- g(n) = $5n + 2log(n) + 1/n \in \Theta(n^d)$ for $d = 1$

Thus since a = $8 > 4^1 = b^d$, $T(n) \in \Theta(n^{log_4(8)})$

(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:

```
1: procedure FINDPOSITION(int A[])
2:     Input: a sorted array A.
3:     Output: whether there exists an i such that i == A[i]
4:     size ← length(A) (1)
5:     return BINSEARCH(A, 0, size − 1) (1)
6: end procedure

7: procedure BINSEARCH(int A[], int min, int max)
8:     Input: a sorted array A, min and max index of search area.
9:     Output: whether there exists an i such that i == A[i]
10:    while max ≥ min do (log(n) + 1)
11:        mid ← ⌊(min + max)/2⌋ 1
12:        if mid == A[mid] then
13:            return true (1)
14:        end if
15:        if mid > A[mid] then
16:            min ← mid + 1 (1)
17:        else
18:            max ← mid − 1 (1)
19:        end if
20:    end while
21:    return false (1)
22: end procedure
```

Note: (g(n)) represents the runtime of that operation.

The runtime complexity of the iterative algorithm is O(log(n)). This is seen by observing that, similar to the recursive case, with each run of the while loop the search space is halved. Additionally, T(n) = $3 + 4 * log(n - 1) \leq 3 + 4 * log(n)$, assuming $n > 1$. This indicates a O(log(n)) runtime of the while loop while all other operations are done in constant time.

(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: Run-time differences:

i. Recursion is slower due to the overhead maintaining the stack i.e
   A. The call stack which contains the "stack frame" (all local variables for the specific recursion level)

ii. Additionally, recursion takes additional time 'climbing' back up the call stack to the original function call.

Due to these differences the iterative implementation should be implemented for best runtimes.

Difference in space complexity:

i. As mentioned above recursion requires maintenance of a call stack and stack frames.

ii. As such each call takes up more memory initalising variables and storing the parameters local to the specific level of the recursion process.

iii. For an especially large number of necessary inputs which must be passed to consecutive recursive calls, this leads to a severe use of storage.

iv. This memory is slowly released as the "pointer" moves up the call stack resolving each recursion call.

v. On the otherhand, the iterative solution only ever uses one set of initalised parameters and consistently updates them each time a loop is performed.

vi. Now assuming space complexity defined by bytes:

   A. Recursive: $(n_{size} + 1) + (n_{size} + 1 + 1 + 1)$*log(n) $\implies$ O(nlog(n))

   B. Iterative: $(n_{size} + 1) + n_{size} + 1 + 1 \implies$ O(n)

Considering these points, it is clear that the iterative solution has a much better space effieciency and complexity.

Accessed Material(s):

i. http://www.fredosaurus.com/JavaBasics/methods/methods-25-calls.html

ii. https://www.tutorialspoint.com/what-are-the-differences-between-recursion-and-iteration-in-java

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: Assumptions:

   - Integers take one byte to store.
   - Memory Complexity is in terms of bytes.
   - Array takes up m x n bytes + 1 to store the reference variable.
   - Function inputs are not considered to be stored in memory used by the implementation.

   In the general case the implementation initialises these variables:

   - minimumX: Integer (1)
   - maximumX: Integer (1)
   - minimumY: Integer (1)
   - maximumY: Integer (1)
   - array: Size of arr + reference to arr ((m*n) + 1)
   - xDim: Integer (1)
   - yDim: Integer (1)
   - xShift: Integer (1)
   - yShift: Integer (1)

   However when resize is run these additional variables are constructed:

   - xDim(2): Integer (1)
   - yDim(2): Integer (1)
   - array(2): Size of arr + reference to arr ((a*b) + 1)
   - newXShift: Integer (1)
   - newYShift: Integer (1)

   Note:

   - The memory used to store these variables is released after the end of the function call.
   - varname(2) represents a local variable with the same name as a class wide variable.
   - $array(2) \geq array$.

   $\therefore$ At any point the implementation uses at most: $M(m, n, a, b) = 14 + (m * n) + (a * b)$. Whilst, generally using $9 + (m * n)$. Thus the memory complexity of my implementation is O((m*n)+(a*b)) since for c sufficiently large (c = 8), $14 + (m * n) + (a * b) \leq c * ((m * n) + (a * b)), \forall m, n, a, b > 0$

(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: As shown by the bound found above, the implementation's memory usage increases drastically with the array(s)' dimensions. As such in the worst case, the computer must store m*n+a*b bytes of data whilst many of the array cells might have uninitalised values (null). This method would be effective at storing a full array of data values however is extrememly inefficient when storing few spaced out elements. This inefficiency is especially true when the array is initalised (as no elements are stored initally) and when performing significant resizing of the array (where $a >> n, b >> m$)
Note: >> represents much greater than.

(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:

- Throwing an exception is done in constant time.
- Accessing and assigning an item between a two dimensional array is done in constant time.
- Stacked addition then assignment is a constant time operation.
- Additionally all bounds were found by reducing to highest order term(s) and finding satisfying values for c and n_0

add:

- int posX = x+this.xShift; (1)
- int posY = y+this.yShift; (1)
- this.array[posX][posY] = element; (1)
- throw new IllegalArgumentException(); (1)
- catch block runs once or not at all; (1)
- Therefore $T_a(n) = 1 + 1 + 1 + 1 * 1$ and this is clearly O(1) with c = 4 and $n_0 = 1$

get:

- int posX = x+this.xShift; (1)
- int posY = y+this.yShift; (1)
- return this.array[posX][posY]; (1)
- Therefore $T_a(n) = 1 + 1 + 1$ and this is clearly O(1) with c = 3 and $n_0 = 1$

remove:

- int posX = x+this.xShift; (1)
- int posY = y+this.yShift; (1)
- T element = this.array[posX][posY]; (1)
- if statement check (1)
  - return false; (1)
- this.array[posX][posY] = null; (1)
- return true; (1)
- Therefore, in the worst case (if false) $T_a(n) = 1 + 1 + 1 + 1 + 1$ and this is clearly O(1) with c = 5 and $n_0 = 1$

clear:

- for i loop (n) where n = xDim.
- for j loop (m) where m = yDim.

- this.array[i][j] = null; (1)
- Therefore $T_a(n) = n * m * 1$ and this is clearly O(n*m) with c = 2, $n_0 = 1$ and $m_0 = 1$

resize:

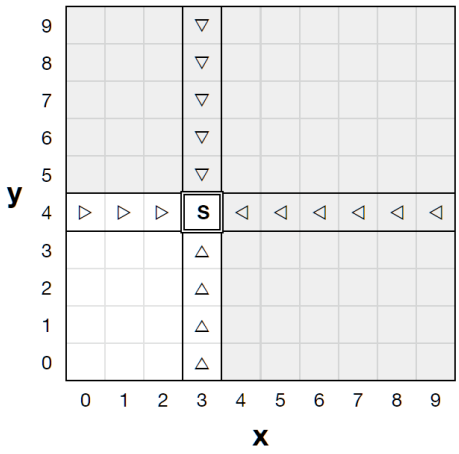
- if statement (best case runtime thus ignoring)
- int xDim = newMaximumX - newMinimumX + 1; (1)
- int yDim = newMaximumY - newMinimumY + 1; (1)
- T[][] array = (T[][])new Object[xDim][yDim]; ($new_m * new_n$) where new_m = newXDim and new_n = newYDim.
- int newXshift = -newMinimumX; (1)
- int newYshift = -newMinimumY; (1)
- for i loop (m)
  - for j loop (n)
    - T temp = this.array[i][j]; (1)
    - int x = i + this.minimumX + newXshift; (1)
    - int y = j + this.minimumY + newYshift; (1)
    - array[x][y] = temp; (1)
- Therefore $T_a(n) = 4 + (new_m * new_n) + (4 * n * m)$ and this is clearly O($n * m + new_m * new_n$) with c = 6, $n_0 = 2$, $m_0 = 2$, $new_m = 2$ $new_n = 2$

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 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:

   Assumptions:

   - After every drill check if it is the source, if its shows the direction or if the drill is broken.
   - External Calls are done in constant time.

   Algorithm:

   - DIAGSEARCH
     - Starting from position (0)(0), assuming a grid running from (0, 0) to (n-1)(n-1).
     - 1: Move diagonally up $\lfloor (max - min/2) \rfloor$ and drill and check a hole. If the source is found move to step 5. Else if the direction of the flow is found move to step 4. Otherwise, move to step 2 or 3 depending on the state of the drill bit.
     - 2: If the drill is not broken set the current position's (a, a) index (a) to min and repeat 1.
     - 3: If the drill breaks at the current hole (a, a), move back to the previous dug hole (located at (b, b)) and move $(\lceil ((a - b)/2) \rceil))$ steps diagonally drilling and checking after each step.
     - 4: If the direction of the flow is found enter LINSEARCH mode.

  - 5: If the source is drilled into return it.
- LINSEARCH
  - 6: Starting at position (a, a) where the flow was found, move ($\lceil((n-1)-a)/2\rceil$) and drill a hole. This step involves storing the min (a), the max (n-1) and mid = $\lceil((n-1)-a)/2\rceil$.
  - 7: If the direction is (left/down) set max = mid - 1, mid = $\lceil((max)-min)/2\rceil$ and move to (a, mid) or (mid, a) for left/down respectively. Drill a hole if the source is found return the current position of the drill. Otherwise, depending on the direction repeat 1 or 2.
  - 8: If the direction is (right, up) set min = mid + 1, mid = $\lceil((max)-min)/2\rceil$ and move to (a, mid) or (mid, a) for right/up respectively. Drill a hole if the source is found return the current position of the drill. Otherwise, depending on the direction repeat 1 or 2.

Since for DIAGSEARCH the number of required diagonal searches is halved each time the a hole is drilled, all other operations are done in constant time and a necessary element is guaranteed to be in the search area the runtime is:

- $T_{DIAG}(n) = T(n/2)$
- As seen in an earlier question: $T_{DIAG}(n) \in O(log(n))$
- Additionally, since LINSEARCH behaves similarly to DIAGSEARCH over a similar search area: $T_{LIN}(n) = T(n/2)$
- Thus, $T(n) = T_{DIAG}(n) + T_{LIN}(n) = T(n/2) + T(n/2) = 2T(n/2) \implies T(n) \in O(log(n))$ (by the master theorem)

(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:

Assumptions:

- Assuming an grid indexed from 1 to n.
- Tuple (position) returns are allowed.
- || is the logical or operation.
- There is guaranteed water source within the array.
- All external functions run in O(n) time.
- null == false in if statements

```
1: procedure FINDSOURCE(int A[][], int n)
2:     Input: An two dimensional array of n*n size
3:     Output: The location of the water source inf the form (x, y)
4:     if n == 1 then
5:         return (n, n)
6:     end if
7:     return DIAGSEARCH(A, 1, n)
8: end procedure

9: procedure DIAGSEARCH(int A[][], int min, int max)
10:    Input: An two dimensional array with a search area of [min, max]
11:    Output: The location of the water source in the form (x, y)
12:    interval ← OPTIMISESTEPS(max)
13:    pos ← min
14:    while !CHECKDRILLBIT() do
15:        prevPos ← pos
16:        pos ← pos + interval
17:        DRILLHOLE(pos, pos)
18:        dir ← CHECKWATERFLOW(pos, pos)
19:        if CHECKSOURCE(pos, pos) then
20:            return (pos, pos)
21:        else if dir then
22:            if dir == "v" then
23:                return (pos, LINSEARCH(A, pos, pos, max, dir))
24:            else
25:                return (LINSEARCH(A, pos, min, pos, dir), pos)
26:            end if
27:        end if
28:        interval ← interval - 1
29:    end while
30:    breakPoint ← pos
31:    pos ← prevPos
32:    while pos != breakPoint do
33:        pos ← pos + 1
34:        DRILLHOLE(pos, pos)
35:        if CHECKDRILLBIT then
36:            return (1, 1)
37:        end if
38:        dir ← CHECKWATERFLOW(pos, pos)
39:        if dir then
40:            if dir == v then
41:                return (pos, LINSEARCH(A, pos, pos, max, dir))
42:            else
43:                return (LINSEARCH(A, pos, pos, max, dir), pos)
44:            end if
45:        end if
46:    end while
47: end procedure
```

```
1: procedure LINSEARCH(int A[][], int index, int min, int max, char dir)
2:     Input: An two dimensional array with a search area of [min, max] along a line prescribed by index and dir.
3:     Output: The second coordinate of the water source (x or y coordinate depending on dir)
4:     pos ← min
5:     interval ← OPTIMISESTEPS(max - min)
6:     while !CHECKDRILLBIT() do
7:         prevPos ← pos
8:         pos ← pos + interval
9:         if dir == v then
10:            DRILLHOLE(index, pos)
11:            if CHECKSOURCE(index, pos) then
12:                return pos
13:            end if
14:        else
15:            DRILLHOLE(pos, index)
16:            if CHECKSOURCE(pos, index) then
17:                return pos
18:            end if
19:        end if
20:        interval ← interval - 1
21:    end while
22:    breakpoint ← pos
23:    pos ← prevPos
24:    while pos != breakpoint do
25:        pos ← pos + 1
26:        if dir == v then
27:            DRILLHOLE(index, pos)
28:            if CHECKSOURCE(index, pos) then
29:                return pos
30:            end if
31:        else
32:            DRILLHOLE(pos, index)
33:            if CHECKSOURCE(pos, index) then
34:                return pos
35:            end if
36:        end if
37:    end while
38: end procedure
```

EXT FUNCTION DEF(s):

---

procedure DRILLHOLE(int A[][], int x, int y)
    Input: The array and the position of the cell which is being drilled.
    Output:
end procedure
procedure CHECKWATERFLOW(int A[][], int x, int y)
    Input: The array and the position of the cell which is being checked for water flow.
    Output: The direction of the water (vertical or horizotnal) flow, expressed as a char, if it can be told by the current drill hole, null otherwise
end procedure
procedure CHECKSOURCE(int A[][], int x, int y)
    Input: The array and the position of the cell which is being checked for water flow.
    Output: True if the source is located at the current position (otherwise false)
end procedure
procedure CHECKDRILLBIT
    Input:
    Output: True if the current drill bit is broken (otherwise false) Note: This method also replaces the drill bit if it is broken.
end procedure
procedure OPTIMIZESTEPS(int n)
    Input: The size of a search area (n).
    Output: Returns the optimial step size.
Comment: This function finds the smallest integer q that satisfies: q + q-1 + ... + 1 $\geq$ n. This q allows us to partition the search space such that at worst q 'operations' must be performed.
    return $\lceil \frac{-1+\sqrt{1-8*n}}{2} \rceil$
end procedure

Thus examining the runtime of the algorithm (in the worst case):

- $T_{FINDSOURCE}(n) = 1$
- $T_{DIAGSEARCH}(n) = 5 + c * \lceil \frac{-1+\sqrt{1+8*n}}{2} \rceil$ (Source in the $(\lceil \frac{-1+\sqrt{1-8*n}}{2} \rceil - 1)$ column or row)
- $T_{LINSEARCH}(n) = 2 + d * \lceil \frac{-1+\sqrt{1+8*(\lceil \frac{-1+\sqrt{1+8*n}}{2} \rceil - 1)}}{2} \rceil$, since in the worst case DIAGSEARCH leaves LINSEARCH with a search space of $(\lceil \frac{-1+\sqrt{1+8*n}}{2} \rceil - 1)$ magnitude.
- All other procedures run in O(1).
- c and d represent the number constant time operations which are run within DIAGSEARCH and LINSEARCH procedures respectively. These constant coefficients have no effect on the resultant Big-O bound and have thus been ignored.
- $T(n) = T_{FINDSOURCE}(n) + T_{DIAGSEARCH}(n) + T_{LINSEARCH}(n) = 8 + \lceil \frac{-1+\sqrt{1+8*n}}{2} \rceil + \lceil \frac{-1+\sqrt{1+8*\lceil \frac{-1+\sqrt{1+8*n}}{2} \rceil}}{2} \rceil$
- Thus, ignoring lower order terms, $T(n) \in O(\sqrt{n})$