COMP3506 Homework 1

Weighting: 15%

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

Overview

This purpose of this assignment is for you to become familiar with understanding the main concepts and notation of asymptotic analysis of algorithms, to gain practice writing simple mathematical proofs, to learn how to read and write pseudocode algorithms, and to practice using binary search and writing and analysing recursive algorithms.

Marks

This assignment is worth 15% of your total grade. COMP3506 students will be marked on questions 1 to 4 out of **50 marks**. COMP7505 are required to additionally do question 5 and will be marked out of **60 marks**.

Partial marks *may* be awarded for partially correct answers and answers lacking justification where it is required.

Submission Instructions

- Your answers to the written questions should be submitted as a file called **A1-[Your Student Number Here].pdf** via the Homework 1 - Written submission. Do not include your full name in your pdf submission.
- **Hand-written answers will not be marked.** If you are comfortable with the LATEX typesetting system, it is strongly recommended that you write your answers using it, however it is not a requirement.
- Your solution to Q3a will be submitted via Gradescope to the Homework 1 - Q3 Programming submission. You should only submit your completed ArrayCartesianPlane.java file. No marks will be awarded for noncompiling submissions, or submissions which import non-supported 3rd party libraries. You should follow all constraints laid out in question 3 or risk losing marks for the question.

Late Submissions and Extensions

Late submissions will not be accepted. It is your responsibility to ensure you have submitted your work well in advance of the deadline (taking into account the possibility of computer, internet, or Gradescope issues). You are allowed to submit multiple times, and only the latest submission before the deadline will be marked. See the ECP for information about extensions.

Academic Misconduct

This assignment is an individual assignment. Posting questions or copying answers from the internet is considered cheating, as is sharing your answers with classmates. All your work (including code) will be analysed by sophisticated plagiarism detection software.

Students are reminded of the University's policy on student misconduct, including plagiarism. See the course profile and the School web page: http://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism.

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.

```
procedure CoolAlgorithm(int n)
    $sum \leftarrow 0$
    if $n \% 2 == 0$ then
        for $i = 0$ to $n$ do
            for $j = i$ to $n^2$ do
                $sum \leftarrow sum + i + j$
            end for
        end for
    else
        while $n > 0$ do
            $sum \leftarrow sum + (n \& 1)$
            $n \leftarrow (n >> 1)$
        end while
    end if
    return $sum$
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 whenthe input n is **odd**, as a mathematical function called $T_{\text{odd}}(n)$. State all assumptions made and explain all your reasoning.

Solution:

sum <- 0	1
if n % 2 == 0 then	1 + 1
else	n is odd
while n > 0 do	log2 (n) + 1 (rounds down to whole number)
sum <- sum + (n & 1)	1 + 1 + 1
n <- (n >> 1)	1 + 1
end while	1 for n < 0
end if	
return sum	1

n <- (n >> 1), n is odd, divided by 2 each loop, therefore there would be $\log_2(n) + 1$ loops in total

$T_{\text{odd}}(n) = 1 + 1 + 1 + (\lfloor \log_2(n) \rfloor + 1) \cdot (3 + 2) + (\lfloor \log_2(n) \rfloor + 1) + 1 + 1$

$\quad = 6 \cdot \lfloor log_2(n) \rfloor + 11$

(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:

assume $T_{odd}(n) \in O(\log(n))$ $\quad g(n) = \log(n)$

there must be some c, n_0 and $c > 0$, such that $T_{odd}(n) \leq c \cdot \log(n)$ for all $n > n_0$.

$6 \cdot \log_2(n) + 8 \leq c \cdot \log(n)$

$6 \cdot \log_2(n) - c \cdot \frac{log_2(n)}{log_2(10)} \leq -8$

$\left(6 - \frac{c}{log_2(10)}\right) \cdot log_2(n) \leq -8$

pick $c = 24$, $n_0 = 93$, $T_{odd}(n) \leq c \cdot \log(n)$ for all $n > n_0$

thus $g(n) = \log(n)$

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

Solution:

assume $T_{odd}(n) \in \Omega(\log(n))$ $\quad g(n) = \log(n)$

there must be some c, n_0 and $c > 0$, such that $T_{odd}(n) \geq c \cdot \log(n)$ for all $n > n_0$.

$6 \cdot \log_2(n) + 8 \geq c \cdot \log(n)$

$6 \cdot \log_2(n) - c \cdot \frac{log_2(n)}{log_2(10)} \geq -8$

$\left(6 - \frac{c}{log_2(10)}\right) \cdot log_2(n) \geq -8$

pick $c = 16$, $n_0 = 1$, $T_{odd}(n) \geq c \cdot \log(n)$ for all $n > n_0$

thus $g(n) = \log(n)$

Big-Θ bound for $T_{odd}(n)$ exists and $T_{odd}(n) \in \Theta(\log(n))$

Since $T_{odd}(n) \in O(\log(n))$ and $T_{odd}(n) \in \Omega(\log(n))$

(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 $T_{even}(n)$. State all assumptions made and explain all your reasoning. Also give a tight Big-O and Big-Ω bound on $T_{even}(n)$. You do **not** need to formally prove these bounds.

Solution:

sum <- 0	1 primitive operation
if n%2 == 0	2 primitive operations, since % and ==
for i = 0 to n	for(i=0; i<n; ++i), there are n loops when i < n and 1 operation when i == n
for j = i to n 2	for(j=i; j<n^2 ; ++j), there are $n^3 - ½n^2 + ½n$ loops when j < n^2 and n times operations when j == n^2
sum < −sum + i + j	3 primitive operations for each loop
return sum	1 primitive operation

when						
i = 0;	j = 0,	1,	2,	3 … $n^2 - 1$		n^2 loops
i = 1;	j =	1,	2,	3 … $n^2 - 1$		$n^2 - 1$ loops
i = 2;	j =		2,	3 … $n^2 - 1$		$n^2 - 2$ loops
…						
i = n -1;	j = n – 1,	n,	n + 1,	n + 2 … $n^2 - 1$		$n^2 - n + 1$ loops

total loops = $(n^2 - n + 1) + (n^2 - n + 2) + ... + (n^2 - 2) + (n^2 - 1) + n^2$

$= (n^2 - (n - 1)) + (n^2 - (n - 2)) + ... + (n^2 - 2) + (n^2 - 1) + n^2$

$= n \cdot n^2 + (-1 - 2 - ... - (n-2) - (n-1))$

$= n^3 - (1 + 2 + ... + (n-2) + (n-3))$

$= n^3 - \frac{(n-1)\cdot n}{2}$

$= n^3 - \frac{1}{2}n^2 + \frac{1}{2}n$

$T_{even}(n) = 1 + 2 + n + 1 + (n^3 - \frac{1}{2}n^2 + \frac{1}{2}n) + 3\cdot(n^3 - \frac{1}{2}n^2 + \frac{1}{2}n) + n + 1$

$= 4n^3 - 2n^2 + 4n + 5$

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

(e) (2 marks) The running time for the algorithm has a best case and worst case, and which case occurs fora 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
- $T_{even}(n) \in O(1)$ when n=0, the algorithm runs 5 primitive operations
- $T_{odd}(n) \in O(1)$ when n=1, the algorithm runs 11 primitive operations

worst – case
- $T_{even}(n) \in \Omega(n^3)$ proved above
- $T_{odd}(n) \in \Omega(\log(n))$ proved above

(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:

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

Big-Θ bound does not exist, since the entire algorithm has two separated functions, if a Big-Θ bound is asymptotically equal to one of the functions, it must keep away from the other one.

(g) (2 marks) Your classmate tells you that Big-O represents the worst case runtime of an algorithm, andsimilarly 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:

Not correct.

Big-O and Big-Ω are bounds or value range under a certain case. There is no kind of relationship of the type Big-O represents worst case, Big-Ω represents the best case. All types of notation can be used when talking about best, average, or worst case of an algorithm.

(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

set best-case running time $f(n) \in O(g_0(n))$

since best–case running time $f(n) \in \Omega(g(n))$

thus $g_0(n) \geq g(n)$ for $n \geq n_0$, n_0 is some positive integer

also worst-case running time is $O(g(n))$

thus $g_0(n) \leq g(n)$ for $n \geq n_1$, n_1 is some positive integer

thus $g(n) \geq g_0(n) \geq g(n)$ for $n \to \infty$

thus $g_0(n) = g(n)$

thus $f(n) \in O(g(n))$

since $f(n) \in \Omega(g(n))$ and $f(n) \in O(g(n))$ for $n>n_2$, n_2 is some integer

therefore best-case running time $f(n) \in \Theta(g(n))$

set worst-case running time $f(n) \in \Omega(g_1(n))$

since worst–case running time $f(n) \in O(g(n))$

thus $g_1(n) \leq g(n)$ for $n \geq n_3$, n_3 is some positive integer

also best-case running time is $\Omega(g(n))$

thus $g_1(n) \geq g(n)$ for $n \geq n_4$, n_4 is some positive integer

thus $g(n) \geq g_1(n) \geq g(n)$ for $n \to \infty$

thus $g_1(n) = g(n)$

thus $f(n) \in \Omega(g(n))$

since $f(n) \in \Omega(g(n))$ and $f(n) \in O(g(n))$ for $n > n_5$, n_5 is some positive integer

therefore worst-case running time $f(n) \in \Theta(g(n))$

since both best-case and worst-case running time is $\Theta(g(n))$

therefore the entire algorithm runs in $\Theta(g(n))$

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 $0 \leq i < n$) 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:

```
Algorithm FindPosition(A, left, right)
  Input: a sorted array A
  Output: whether there is a position i such that A[i] = i between A[left] and A[right]
  if left > right then
          return false
  m <- ⌊ (left + right) ÷ 2 ⌋
  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
end algorithm
```

helper function:

```
public boolean FindPosition(int[] A, int left, int right) {
    if (left > right) {
        return false;
    }
    int m = (left + right) / 2;
    if (A[m] < m) {
        return FindPosition(A, m + 1, right);
    } else if (A[m] > m) {
        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, 0, 8)

-1	0	2	3	10	11	23	24	102

FindPosition(A, 0, 8)

left (0) < right (8) and m = 4, A[4] = 10 > 4, thus return *FindPosition(A, 0, 3)*

-1	0	2	3

left (0) < right (3) and m = 1, A[1] = 0 < 1, thus return *FindPosition(A, 2, 3)*

2	3

left (2) < right (3) and m = 2, A[2] = 2, thus 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:

```
Algorithm FindPosition(A, left, right)
  Input: a sorted array A
  Output: whether there is a position i such that A[i] = i between A[left] and A[right]
  if left > right then                          } O(1)
        return false
  m <- ⌊ (left + right) ÷ 2 ⌋                   → O(1)
  if A[m] < m then                              ┐
        return FindPosition(A, m+1, right)      │
  else if A[m] > m then                         ├ O(n/2)
        return FindPosition(A, left, m-1)       ┘
  else
        return true
end algorithm
```

Each level halve the size of array, thus:

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

$$\begin{aligned} T(n) &= T(n/2) + O(1) \\ &= T(n/2^2) + O(1) + O(1) \\ &= T(n/2^3) + O(1) + O(1) + O(1) \\ &\dots \\ &= T(n/n) + O(1) \cdot \log_2 n \\ &= T(1) + O(1) \cdot \log_2 n \\ &= O(1) + O(1) \cdot \log_2 n \end{aligned}$$

Thus $T(n)$ is $O(\log n)$.

(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 nonnegative 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:

$T(n)$ given in question (c) can be written in:

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

thus $a = 1$, $b = 2$, $d = 0$

substitute the values of a, b, d into the master theorem asymptotic bound,

$a = 1$, $b = 2$, $d = 0$ make the equation $a = b^d$ true

thus $T(n) \in \Theta(n^d \log n) = \Theta(\log n)$

this agrees with the result of question (c) $T(n) \in O(\log n) \in \Theta(\log n)$.

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:

$T(n)$ can be written in:

$$T(n) = \begin{cases} 5 \cdot T\left(\frac{n}{3}\right) + n^2 + 2n & \text{for } n > 1 \\ 100 & \text{for } n = 1 \end{cases}$$

According to the master theorem, coefficient $a = 5$, $b = 3$

since $g(n) = n^2 + 2n \in \Theta(n^2)$

thus coefficient $d = 2$

$a = 5$, $b = 3$, $d = 2$ make the inequation $a < b^d$ true

thus $T(n) \in \Theta(n^2)$

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:

$T(n)$ can be written in:

$$T(n) = \begin{cases} 8 \cdot T\left(\frac{n}{4}\right) + 5n + 2\log n + \frac{1}{n} & \text{for } n > 1 \\ 1 & \text{for } n = 1 \end{cases}$$

According to the master theorem, coefficient $a = 8$, $b = 4$

$g(n) = 5n + 2\log n + \frac{1}{n}$

assume for some c, n_0 and c > 0, such that $5n + 2\log n + \frac{1}{n} \leq c \cdot n$ for *all* $n > n_0$

$(5 - c)n + 2\log n + \frac{1}{n} \leq c \cdot n$

$2\log n + \frac{1}{n} \leq (c - 5) \cdot n$

pick c = 6, $n_0 = 1$

thus $g(n) \in O(n)$

assume for some c, n_1 and c > 0, such that $5n + 2\log n + \frac{1}{n} \geq c \cdot n$ for *all* $n > n_1$

$(5 - c)n + 2\log n + \frac{1}{n} \geq c \cdot n$

$2\log n + \frac{1}{n} \geq (c - 5) \cdot n$

pick c = 4, $n_1 = 1$

thus $g(n) \in \Omega(n)$

thus $g(n) \in \Theta(n) = \Theta(n^d)$

thus coefficient $d = 1$

$a = 8$, $b = 4$, $d = 1$ make the inequation $a > b^d$ true

thus $T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_4 8})$

(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:

```
Algorithm FindPosition(A, left, right)
  **Input**: a sorted array A
  **Output**: whether there is a position i such that A[i] = i between A[left] and A[right]
  while left < right do                          } O(n/2)
          m <- ⌊ (left + right) ÷ 2 ⌋             }
          if A[m] < m then                       ┐
                  left <- m + 1                  │
          else if A[m] > m then                  │ O(1)
                  right <- m – 1                 │
          else                                   │
                  return true                    ┘
          end if
  end while
  return false                                 → O(1)
end algorithm
```

helper function:

```java
public boolean FindPosition(int[] A, int left, int right) {
    while (left < right) {
        int m = (left + right) / 2;
        if (A[m] < m) {
            left = m + 1;
        } else if (A[m] > m) {
            right = m - 1;
        } else {
            return true;
        }
    }
    return false;
}
```

For each loop the algorithm halves the length of sorted array

$T(n) = T(n/2) + O(1)$

$= O(1) + O(1) \cdot \log_2 n$

Thus the iterative algorithm running time is $O(\log n)$.

(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 would implement iterative algorithm in Java, because Java uses method calls by value while passing reference variables as well, it takes time to create a copy of references and passes them as parameters to the function, so the deeper the depth of recursive algorithm is, the more method calls are called, the more time it takes. However iterative algorithm only has one layer of depth, thus it doesn't have that issue.

Both the recursive solution's and iterative solution's space complexity is O(1).

The iterative algorithm access one element of the array for each loop, requires constant space to perform operations, therefore space complexity is O(1).

Although the recursive algorithm has depth, but here the depth doesn't hold any value, thus doesn't occupy much memory space, so it's the same with the iterative algorithm, accesses one element of the array for each level of depth, therefore space complexity is O(1).

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:
   Assume the number of cells on the plane is *N*, the memory complexity of the plane is *O(N).*
   Memory held by the class of ArrayCartesianPlane is constant, thus memory complexity is *O(1).*
   Memory held by the method resize() is constant, thus memory complexity is *O(1).*
   Thus overall memory complexity is *O(N) + O(1) + O(1) = O(N).*

   (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:
   When the plane is small, it has a high memory efficiency, but once the plane is very large but has very few elements, it will waste a lot of memory space because of the null value, meanwhile it will significantly increase the time complexity as well.

   (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:

```
public void add(int x, int y, T element) throws IllegalArgumentException {
    if (x < minimumX || x > maximumX) {                                        3
        throw new IllegalArgumentException("x-coordinate is out of bounds");
    } else if (y < minimumY || y > maximumY) {                                 3   O(1)
        throw new IllegalArgumentException("y-coordinate is out of bounds");
    }
    plane[x - minimumX][y - minimumY] = element;                               4
}
```

add time complexity is *O(1)*

```
public T get(int x, int y) throws IndexOutOfBoundsException {
    return plane[x - minimumX][y - minimumY];                      3
}
```

get time complexity is *O(1)*

```
public boolean remove(int x, int y) throws IndexOutOfBoundsException {
    if (plane[x - minimumX][y - minimumY] == null) {                    4
        return false;
    } else {
        plane[x - minimumX][x - minimumX] = null;                       4
        return true;
    }
}
```

remove time complexity is *O(1)*

```
public void resize(int newMinimumX, int newMaximumX, int newMinimumY,
                   int newMaximumY) throws IllegalArgumentException {
    if (newMinimumX > newMaximumX || newMinimumY > newMaximumY) {                       3
        throw new IllegalArgumentException();
    }
    int newWidth = newMaximumX - newMinimumX + 1;                                       3
    int newHeight = newMaximumY - newMinimumY + 1;                                      3
    T[][] newPlane = (T[][])new Object[newWidth][newHeight];
    for (int i = 0; i < width; ++i) {                                                 O(N)
        for (int j = 0; j < height; ++j) {
            if (plane[i][j] != null) {                                                   2
                if ((i + minimumX) < newMinimumX || (i + minimumX) > newMaximumX
                        || (j + minimumY) < newMinimumY || (j + minimumY) > newMaximumY) {  11
                    throw new IllegalArgumentException();
                } else {
                    newPlane[i + minimumX - newMinimumX][j + minimumY - newMinimumY]     7
                            = plane[i][j];
                }
            }
        }
    }
    this.minimumX = newMinimumX;
    this.maximumX = newMaximumX;
    this.minimumY = newMinimumY;
    this.maximumY = newMaximumY;                                                        7
    this.width = newWidth;
    this.height = newHeight;
    this.plane = newPlane;
```

resize time complexity is 3 + 3 + 3 + (2 + 11 + 7) · *O(N)* + *7* = *20* · *O(N)* + *O(1)*

= O(N) N is number of cells on plane

```
public void clear() {
    for (int i = 0; i < width; ++i) {        } O(N)
        for (int j = 0; j < height; ++j) {
            plane[i][j] = null;                2
        }
    }
}
```

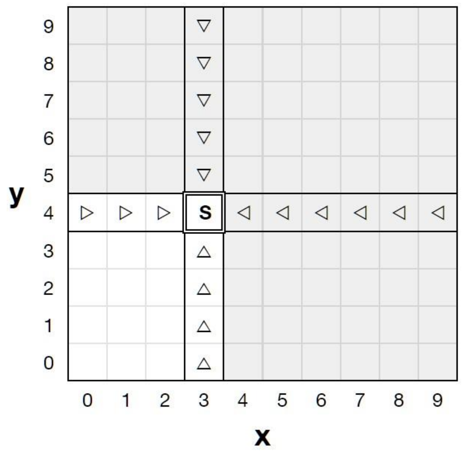
clear time complexity is $2 \cdot O(N) = O(N)$

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 aalgorithm 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:

~~First of all, drill on the four vertices, in order to find out the origin *(0, 0)*. *O(1)*~~

Firstly use binary search to find $(S_x, 0)$: $O(\log n)$

1. along $y = 0$, find the midpoint of x bound as x_m , drill on $(x_m, 0)$

2. if drill bit was broken, pick the midpoint between left end and *(x_m, 0)* as new *(x_m, 0)*, original *(x_m, 0)* as right end, drill on *(x_m, 0)*
3. if drill bit was not broken, pick the midpoint between right end and *(x_m, 0)* as new *(x_m, 0)*, original *(x_m, 0)* as left end, drill on *(x_m, 0)*
4. repeat the steps above until find *(S_x, 0)* (when the direction can be found after drilling)

Secondly keep drilling along column $x = S_x$, use binary search and the direction revealed to find S_y .*O(log n)*

Finally there you have it, water source *S* is (S_x, S_y).

The Big-O bound on the number of holes you need to drill is *2· O(log n) = O(log n)*

(b) (5 marks) The company, impressed with the drilling efficiency of your algorithm, assigns you to another *n* × *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:

In terms of : 1. Two sides of a triangle are greater than the third

2. double step size for each drilling

```
Algorithm FindWaterSource(A, x, y)
    Input: a 2D array A (n × n grid)
    Output: the position of water source ($S_x$ , $S_y$)
    x <- 0
    y <- 0
    drill at (x , y)
    if (x , y) is water source then
        return (0 , 0)
    else if found direction then
        return AlongDirectionFindWaterSource(x , y)
    else
        x <- 1
        y <- 1
    DoubleStepSizeDrilling(x , y)      (may break drill bit 1 time)
    x <- x/2 + 1
    y <- y/2 + 1
    return StepByStepDrilling(x , y)
```

```
HelperFucntion AlongDirectionFindWaterSource(x , y)
    **Input**: the position of last drilling
    **Output**: the position of water source $(S_x , S_y)$
    if x = $S_x$ then (since we've found the direction along the column)
        i <-  1
        while (drill bit is not broken) do
            drill at ($S_x$ , y + i)
            if ($S_x$ , y + i) is water source then
                return ($S_x$ , y + i)
            i <-  2 * i
        end while                              (may break drill bit 1 time)
        i <-  i / 2 + 1
        while (drill bit is not broken) do
            drill at ($S_x$ , y + i)
            if ($S_x$ , y + i) is water source then  (this case must happen)
                return ($S_x$ , y + i)
            i <-  i + 1
        end while

    else if y =  $S_y$ then (found the direction along the row)
        algorithm is the same with above
    end if


HelperFucntion DoubleStepSizeDrilling(x , y)
    **Input**: the position to drill at
    **Output**: the position of water source $(S_x , S_y)$
    drill at (x , y)
    while (drill bit is not broken) do
        if (x , y) is water source then
            return (x , y)
        else if found direction then
            return AlongDirectionFindWaterSource(x , y)
        end if
        x <- x * 2
        y <- y * 2
        drill at (x , y)
    end while
```

```
HelperFucntion StepByStepDrilling(x , y)
    **Input**: the position to drill at
    **Output**: the position of water source $(S_x , S_y)$
    drill at (x , y)
    while (drill bit is not broken) do
        if (x , y) is water source then            (one of two cases must happen)
            return (x , y)
        else if found direction then
            return AlongDirectionFindWaterSource(x , y)
        end if
        x <- x + 1
        y <- y + 1
        drill at (x , y)
    end while
```

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.

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

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

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

(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.