COMP7505 – Homework Task 1

Joel Thomas 44793203

1.

a)

Program counter (line)	Code	Explanation	Number of primitive operations
2	$sum \leftarrow 0$	1 operation for assignment	1
3	if $n\%2 == 0$	1 operation for evaluating $n\%2$, 1 operation for testing equality	2
10	while $n > 0$ do	1 operation per iteration (total $\log_2 n$), 1 final operation for final check. Total of $\log_2 n$ operations because each iteration, n loses a power of 2 (after bitwise right shift)	$\log_2 n + 1$
11	$sum \leftarrow sum + (n\&1)$	1 operation for assignment, 1 operation for addition, 1 operation for evaluating bitwise AND	$3 \cdot \log_2 n = 3\log_2 n$
12	$n \leftarrow (n \gg 1)$	1 operation for assignment, 1 operation for evaluating bitwise right shift operator	$2 \cdot \log_2 n = 2\log_2 n$
15	return sum	1 operation for returning value	1
Total			$T_{odd}(n) = 6\log_2 n + 5$

Table 1: Running time of coolAlgorithm when n is odd

b)

$T_{odd}(n) = 6\log_2 n + 5$

Dropping constants and lower order terms:

$T_{odd}(n) \in O(\log_2 n)$

Proof:

$6\log_2 n + 5 \leq c \cdot \log_2 n$

$5 \leq (c-6)\log_2 n$

$\log_2 n \geq \frac{5}{c-6}$

$n \geq 2^{\frac{5}{c-6}}$

Choose $c = 7, n_0 = 2^{\frac{5}{7-6}} = 2^5 = 32$

$\therefore T_{odd}(n) \in O(\log_2 n)$ for $c = 7, n \geq 32$

c)

$T_{odd}(n) \in \Omega(\log_2 n)$

Proof:

$6\log_2 n + 5 \geq c \cdot \log_2 n$

Since $5 > 0 \rightarrow 6\log_2 n + 5 > 6\log_2 n \; \forall n > 0$

Choose $c = 6, n_0 = 1$

$\therefore T_{odd}(n) \in \Omega(\log_2 n)$ for $c = 6, n_0 = 1$

$T(n) \in \Theta\big(g(n)\big)$ if $T(n) \in O\big(g(n)\big)$ and $T(n) \in \Omega\big(g(n)\big)$. Using the formal definition for Big-Θ bound:

$6\log_2 n \leq 6\log_2 n + 5 \leq 7\log_2 n, \quad c_1 = 6, c_2 = 7, \quad n = \max\{n_1, n_2\} = \max\{1,32\} = 32$

$\therefore T(n) \in \Theta(\log_2 n)$

d)

Program counter (line)	Code	Explanation	Number of primitive operations
2	$sum \leftarrow 0$	1 operation for assignment	1
3	if $n\%2 == 0$	1 operation for evaluating $n\%2$, 1 operation for testing equality	2
4	for $i = 0$ to n do	1 operation for assignment, 1 operation per iteration for evaluating n (total $n + 2$), 1 operation per iteration for evaluating $i < n + 1$ (total $n + 2$)	$1 + n + 2 + n + 2$ $= 2n + 5$
5	for $j = 0$ to n^2 do	1 operation for assignment, 1 operation per iteration for evaluating n^2 (total $n^2 + 2$), 1 operation per iteration for evaluating $j < n^2 + 1$ (total $n^2 + 2$). Multiply everything by $(n + 1)$ due to for loop on line 4	$(n + 1)[1 + (n^2 + 2) + (n^2 + 2)]$ $= (n + 1)[2n^2 + 5]$ $= 2n^3 + 2n^2 + 5n + 5$
6	$sum \leftarrow sum + i + j$	1 operation for assignment, 1 operation each per addition. Multiply everything by $(n + 1)(n^2 + 1)$ due to for loops on line 4 and line 5	$(n + 1)(n^2 + 1)[1 + 1 + 1]$ $= 3(n + 1)(n^2 + 1)$ $= 3(n^3 + n^2 + n + 1)$ $3n^3 + 3n^2 + 3n + 1$
15	return sum	1 operation for returning value	1
Total			$T_{even}(n) = 5n^3 + 5n^2 + 10n + 15$

Table 2: Running time of coolAlgorithm when n is even

Since $T_{even}(n) = 5n^3 + 5n^2 + 10n + 15$ is a simply polynomial, using the property that polynomials have the same $g(n)$ for a Big-O and Big-Ω and dropping constants and lower order terms:

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

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

e)

Since $\log_2 n \ll n^3 \ \forall n > 0$ so $T_{odd}(n) \ll T_{even}(n)$ in runtime complexity and thus the best case occurs when n is odd and the worst case occurs when n is even.

$T_{odd}(n) \in O(\log_2 n)$

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

f)

$$T(n) = \begin{cases} T_{even}(n) \in \Theta(n^3), & n \text{ is even} \\ T_{odd}(n) \in \Theta(\log_2 n), & n \text{ is odd} \end{cases}$$

$\therefore T(n) \in O(n^3), T(n) \in \Omega(\log_2 n)$

$T(n) \in \Theta(g(n))$ if $T(n) \in O(g(n))$ and $T(n) \in \Omega(g(n))$

$Since\ n^3 \neq \log_2 n$ in terms of runtime complexity, $T(n)$ does not have a Big-Θ bound.

g)

The classmate is incorrect. Big-O, Big-Ω and Big-Θ bounds have nothing to do with the runtimes of an algorithm as they are purely used to denote asymptotic bounds on functions. An algorithm can have each of these bounds on the best-case, average-case and worst-case runtimes of an algorithm. Thus, Big-O and Big-Ω bounds do not strictly represent the worst- and best-case runtimes of an algorithm respectively.

h)

Let the algorithm be denoted by $f(n)$. Since $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n)) \rightarrow f(n) \in \Theta(g(n))$ and $\exists\ c_1, c_2 > 0$ and $n \geq \max\{n_1, n_2\}$ such that:

$c_1 g(n) \leq f(n) \leq c_2 g(n)$

$Since\ \Theta(g(n))$ here is a tight bound between the best- and worst-case running times, it implies that the average-case runtime is $\Theta(g(n))$ i.e. the algorithm runs in $\Theta(g(n))$ time.

2.

a)

```
Algorithm findPosition(A)
    Input: a sorted array A
    Output: true or false whether or not there is a position i (0 <= i <= n) such that A[i]=i

    left <- 0
    // length(A) - 1 same as n - 1
    right <- length(A) - 1

    return findPositionHelper(A, left, right)

Algorithm findPositionHelper(A, left, right)
    Input: a sorted array A, left starting postiion and right ending position
    Output: true or false whether or not there is a position i (0 <= i <= n) such that A[i]=i

    if left - right <= 2 then
        for i <- left to right do
            if A[i] = i then
                return true
        return false

    i <- floor((left + right)/2)
    if A[i] = i then
        return true

    return (findPosition(A, left, i) or findPosition(A, i + 1, right))
```

Figure 1: Recursive solution pseudocode

b)

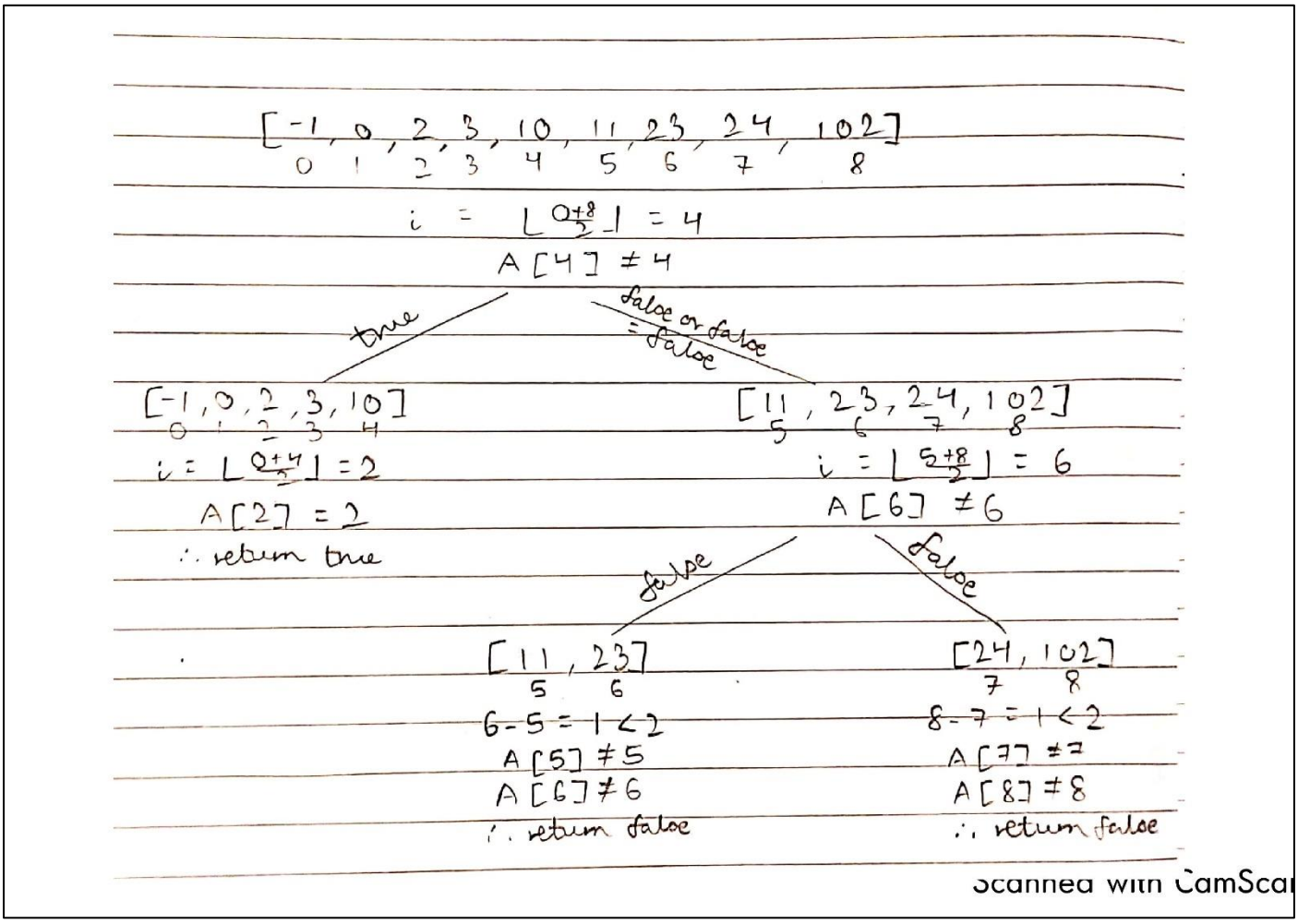

Figure 2 : Steps taken by algorithm for $A = \{-1, 0, 2, 3, 10, 11, 23, 24, 102\}$

Initially, findPosition(A) is called and $left = 0, right = length(A) - 1 = 8$. Since $i = \left\lfloor \frac{0+8}{2} \right\rfloor = 4$ and $A[4] \neq 4$, we call findPositionHelper(A, 0, 4) and findPositionHelper(A, 5, 8). In the first subtree (LHS figure 2), $i = \left\lfloor \frac{0+4}{2} \right\rfloor = 2$ and because $A[2] = 2$, we simply return true from this subtree. In the second subtree (RHS figure 2), $i = \left\lfloor \frac{5+8}{2} \right\rfloor = 6$ and because $A[6] \neq 6$, we again call findPositionHelper(A, 5, 6) and findPositionHelper(A, 7, 8). In both sub-subtrees, since $6 - 5 = 8 - 7 = 1 < 2$, we run a for loop from $left$ to $right$ to check for $A[i] = i$ in both sub-subtree. In the first sub-subtree (LHS of second subtree), $A[5] \neq 5$ and $A[6] \neq 6$ so we return false. Similarly, in the second sub-subtree (RHS of second subtree), $A[7] = 24 \neq 7$ and $A[8] = 102 \neq 8$, so we again return false. Because false or false yields false, the second subtree returns false. Overall, findPositionHelper(A, 0, 8) returns true or false which is true. This makes findPosition(A) return true resulting in the correct solution.

c)

$$T(n) = \begin{cases} O(1), & if\ left - right \leq 2\ or\ A[i] = i \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + O(1), & otherwise \end{cases}$$

Note that $T(n)$ here has the exact same formulation as for a binary sum algorithm. At depth 0, there is 1 sequence of size n. At depth 1, there are 2 sequences, each of size $\frac{n}{2} = \frac{n}{2^1}$. At depth 2, there are 4 sequences, each of size $\frac{n}{4} =$

$\frac{n}{2^2}$. Hence, following this pattern, at arbitrary depth h, there are 2^h sequences, each of size $\frac{n}{2^h}$. If h is the terminal depth of the recursion tree, then:

$$\frac{n}{2^h} = 1$$

$$n = 2^h$$

$$h = \log_2 n$$

We can then write the number of recursive calls as:

$$T(n) = 1 + 2 + 4 + 8 + \cdots + n$$

Using the property that $T(n)$ is a geometric series and the formula $\sum_{i=0}^{h} ar^i = \frac{a(r^{h+1}-1)}{r-1}$

$$T(n) = \sum_{i=0}^{h} 2^i$$

$$T(n) = \sum_{i=0}^{\log_2 n} 2^i$$

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

$$T(n) = 2 \cdot 2^{\log_2 n} - 1$$

$$T(n) = 2n - 1$$

$$\therefore T(n) \in O(n)$$

Another way of explaining why $T(n)$ is $O(n)$ is because even if the only i such that $A[i] = i$ is $i = 0$ and $A[0] = 0$, the algorithm runs on the entire array A, returning true for $i = 0$ and false for all other values. A sequence of {true or false or false or … or false} will return true but only at the very end.

d)

i)

$$T(n) = \begin{cases} O(1), & if\ left - right \leq 2\ or\ A[i] = i \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + O(1), & otherwise \end{cases}$$

$$\rightarrow T(n) = \begin{cases} 1, & n \leq 2 \\ 2T\left(\frac{n}{2}\right) + 1, & n > 2 \end{cases}$$

$$k = 2, \quad a = 2, \quad b = 2, \quad c = 1, \quad g(n) = 1 \in O(1), \quad d = 0$$

$$a = 2 > 2^0 = 1 = b^d$$

$$\therefore T(n) \in \Theta\left(n^{\log_2 2}\right) = \Theta(n^1) = \Theta(n)$$ using the Master theorem.

ii)

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

$$k = 1, \quad a = 5, \quad b = 3, \quad c = 100, \quad g(n) = n^2 + 2n \in O(n^2), \quad d = 2$$

$a = 5 < 3^2 = 9 = b^d$

$\therefore T(n) \in \Theta(n^2)$ using the Master theorem.

iii)

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

$k = 1, \quad a = 8, \quad b = 4, \quad c = 1, \quad g(n) = 5n + 2\log n + \frac{1}{n} \in O(n), \quad d = 1$

$g(n) \in O(n)$ because $\frac{1}{n} \ll \log n \ll n$ asymptotically and hence we drop the $\log n, \frac{1}{n}$ as lower order terms.

$a = 8 > 4^1 = 4 = b^d$

$\therefore T(n) \in \Theta\left(n^{\log_4 8}\right) = \Theta\left(n^{\frac{3}{2}}\right)$ using the Master theorem.

e)

```
1  Algorithm findPosition(A)
2      Input: a sorted array, A
3      Output: whether or not there is a position i (0 <= i <= n) such that A[i]=i
4
5      n <- A.length
6      for i = 0 to n - 1 do
7          if A[i] = i then
8              return true
9
10     return false
11
```

Figure 2: Iterative solution pseudocode

Program counter (line)	Code	Explanation	Number of primitive operations
5	$n \leftarrow A.length$	1 operation for assignment, 1 operation for evaluating length of A	2
6	for $i = 0$ to $n - 1$ do	1 operation for assignment, 1 operation per iteration for evaluating $n - 1$ (total $n + 1$), 1 operation per iteration for evaluating $i < n$ (total $n + 1$)	$1 + n + 1 + n + 1 = 2n + 3$
7	if $A[i] = i$ then	1 operation for evaluating $A[1]$, 1 operation for testing equality. Multiply everything by n due to for loop on line 6	$n \cdot (1 + 1) = 2n$
8	return true	1 operation for returning a value	1
10	return false	1 operation for returning a value	1
Total			$T(n) = 4n + 7$

Table 4: Running time of iterative solution to findPosition

$T(n) = 4n + 7$

Dropping constants and lower order terms:

$T(n) \in O(n)$ which is the same runtime complexity as for the recursive solution.

f)

If speed is an important factor then the iterative solution should be implemented. The recursive solution is inherently slower due to two factors (Ghanem, 2011) – the added task of maintaining the stack (the list structure of function calls and parameters (Bolton, 2019)) as well as uses more memory for maintaining the stack. Thus, the speed of the recursive solution will be much slower relative to the iterative solution using very large inputs. Hence, recursive solutions are generally meant to be used only when iterative solutions are not possible because they are generally cleaner/less complex to implement.

3.

b)

Let:

$m = maxX - minX + 1, \qquad maxX \geq minX$

$n = maxY - minY + 1, \qquad maxY \geq minY$

Then the memory complexity of the algorithm is given by $O(m \cdot n) = O(mn)$ since $m \times n$ cells are always used in the multidimensional array including the null cells and cells containing type T elements.

c)

Implementation is memory inefficient because $m \times n$ cells are always used in this implementation, regardless of how many or rather how few elements there are in the multidimensional array. If $m = n$, then the space complexity is $O(n^2)$ which is worse than a linear-time algorithm. However, an advantage of this implementation is that every single (x, y) coordinate pair is accounted for (either a null cell or cell with type T element in it).

d)

1) add $\in O(1)$ because conditional (if) statements are $O(1)$ and accessing and assigning a particular index of a multidimensional array are $O(1)$. Altogether, this is $O(1)$.

2) get $\in O(1)$ because conditional (if) statements are $O(1)$ and accessing a particular index of a multidimensional array is $O(1)$. Altogether, this is $O(1)$.

3) remove $\in O(1)$ because conditional (if) statements are $O(1)$, accessing a particular index of a multidimensional array is $O(1)$, testing equality is $O(1)$ and returning a boolean is $O(1)$. Altogether, this is $O(1)$.

4) resize $in\ O(mn)$ because conditional (if) statements are $O(1)$, accessing and conditional statements on every element of a multidimensional array is $O(mn)$ (worst-case), and reassigning variables is $O(1)$. Altogether, this is $O(mn)$.

5) clear $\in O(mn)$ because initialising a 2D type T array variable to an empty array of size mn is equivalent to setting every (x, y) coordinate pair in that array to the null type.

4.

a)

For both a) and b), it is assumed that if $x < s_x$ or $y < s_y$ then nothing happens to the drill bit and no information is revealed about the direction of the water source. Hence logically, it is ideal to start in the bottom left corner ($x = y = 0$) of the grid since in the worst-case, the water source is in a different location but the drill bit never breaks.

```
a)
FOUND <- 0
DRILL_BREAKS <- 1
DRILL_DOESNT_BREAK <- 2

Algorithm revealSource(grid, n)
    Input: grid is the plot of land and n is the length of the grid
    Output: array that contains sX and sY
    start <- 0
    end <- n

    sX <- revealSourceX(grid, start, end)
    sY <- revealSourceY(grid, start, end)

    return (sX, sY)

```

Figure 3: Main algorithm to reveal source (s_x, s_y)

```
Algorithm revealSourceX(grid, start, end)
    Input: grid is the plot of land, x is the current x-coordinate and n is the
    length of the grid
    Output: sX - the sources x-coordinate

    if start < end then
        return -1

    x <- floor((start + end) / 2)
    consultResult = drillBreak(grid, x, 0)
    if (consultResult = DRILL_BREAKS) then
        return revealSourceX(grid, start, x - 1)
    else if (consultResult = DRILL_DOESNT_BREAK) then
        return revealSourceX(grid, x + 1, end)
    else
        return x

```

Figure 4: Recursive algorithm to reveal s_x

```
Algorithm revealSourceY(grid, start, end)
    Input: grid is the plot of land, x is the current x-coordinate and n is the
    length of the grid
    Output: sY - the sources Y-coordinate

    if start < end then
        return -1

    y <- floor((start + end) / 2)
    consultResult = drillBreak(grid, 0, y)
    if (consultResult = DRILL_BREAKS) then
        return revealSourceY(grid, start, y - 1)
    else if (consultResult = DRILL_DOESNT_BREAK) then
        return revealSourceY(grid, y + 1, end)
    else
        return y

```

Figure 5: Recursive algorithm to reveal s_y

```
Algorithm drillBreak(int[] grid, int x, int y)
    Input: grid is the plot of land, x is the current x-coordinate and n is the
    length of the grid
    Output: constants denoting whether:
    1) x = sX or y = sY, return FOUND
    2) The drill bit breaks, return DRILL_BREAKS
    3) Nothing happens, return DRILL_DOESNT_BREAK
    // Assumed that has constant time complexity and is provided as a helper
    // function

```

Figure 6: Helper function

Since revealSourceX and revealSourceY are each implemented extremely similar to binary search which has a runtime complexity of $O(\log_2 n)$, these recursive algorithms are $O(\log_2 n)$ each. The helper function drillBreak is assumed to have constant time complexity i.e. $O(1)$ and is pre-provided to us (no need to write pseudocode for it). Hence, the main algorithm revealSource has an overall runtime complexity of:

$T(n) = O(\log_2 n) + O(\log_2 n) = 2 \cdot O(\log_2 n) \in O(\log_2 n) \ll O(n)$ as required.

b)

For b) only, it is assumed that s_x and s_y must exist (unlike a) which allows for no existence of s_x, s_y).

```
b)
FOUND <- 0
DRILL_BREAKS <- 1
DRILL_DOESNT_BREAK <- 2

Algorithm revealSource(grid, n)
    Input: grid is the plot of land and n is the length of the grid
    Output: array that contains sX and sY
    sX = revealSourceX(grid, n)
    sY = revealSourceY(grid, n)

    return (sX, sY)
```

Figure 7: Main algorithm to reveal source (s_x, s_y)

```
Algorithm revealSourceX(grid, n)
    Input: grid is the plot of land and n is the length of the grid
    Output: sX - the sources x-coordinate

    x <- 0
    safeX <- x
    while True
        consultResult <- drillBreak(grid, x, 0)
        if (consultResult = DRILL_BREAKS) then
            break
        else if (consultResult = DRILL_DOESNT_BREAK) then
            safeX <- x
            if (x + n^(1/2) >= n) then
                x <- n - 1
            else
                x <- integer(x + n^(1/2))
        else
            return x

    for i <- safeX to i < x do
        consultResult <- drillBreak(grid, i, 0)
        if (consultResult = FOUND)
            return x
```

Figure 8: Iterative algorithm to reveal s_x

```
Algorithm revealSourceY(grid, n)
    Input: grid is the plot of land and n is the length of the grid
    Output: sY - the sources y-coordinate

    y <- 0
    safeY <- y
    while True
        consultResult <- drillBreak(grid, 0, y)
        if (consultResult = DRILL_BREAKS) then
            break
        else if (consultResult = DRILL_DOESNT_BREAK) then
            safeY <- y
            if (y + n^(1/2) >= n) then
                y <- n - 1
            else
                y <- integer(y + n^(1/2))
        else
            return y

    for j <- safeY to j < y do
        consultResult <- drillBreak(grid, 0, j)
        if (consultResult = FOUND)
            return y
```

Figure 9: Iterative algorithm to reveal s_y

Since revealSourceX and revealSourceY are once again very similar, the have the same runtime complexity. Hence, the runtime complexity will be explained only for revealSourceX. Essentially, we continually increment x with a constant step size of $\sqrt{n}$ (from the start $x = 0$) till either $x = s_x$ or a drill bit breaks. On each iteration, we also increase the $safeX$ variable (x such that the drill bit hasn't broken yet). Upon the first drill break, it is known that s_x must lie somewhere between $safeX$ aand x. So we break the infinite while loop and run a new for loop to iterate over all the elements to $x - 1$ starting from $safeX$ until we successfully find s_x. We repeat the same process to find $y = s_y$ and break another drill bit in doing so. Note that to find s_x, s_y, we have only broken two drill bits thus far as required.

In revealSourceX, in the worst-case scenario, $s_x = n - 1$ so we need to iterate $\frac{n}{\sqrt{n}} = \sqrt{n}$ times in the while loop to get there i.e. $O(\sqrt{n})$ runtime complexity. Similarly, since the step size is constant $\sqrt{n}$, the for loop's time complexity is also $O(\sqrt{n})$. So in total, revealSourceX is $O(\sqrt{n}) + O(\sqrt{n}) = 2 \cdot O(\sqrt{n}) \in O(\sqrt{n})$. This is also the runtime complexity for revealSourceY. The helper function drillBreak is the same as in figure 6 – it is assumed to have constant time complexity i.e. $O(1)$ and is pre-provided to us. Hence, the main algorithm revealSource has an overall runtime complexity of:

$T(n) = O(\sqrt{n}) + O(\sqrt{n}) = 2 \cdot O(\sqrt{n}) \in O(\sqrt{n}) \ll O(n)$ as required.

5.

a)

```
Algorithm quaternarySearch(A, left, right, x)
    Input: a sorted array A, left and right start positions, and integer x to search for in A
    Output: whether x is in the subarray between A[left] and A[right]
    if (right < left) then
        return -1

    if (right - left <= 4) then
        for i = left to right do
            if A[i] = x then
                return i
        return -1

    q1 = floor((right - left)/4)
    q2 = floor((right - left)/2)
    q3 = floor((right - left)*(3/4))

    if x < A[q1] then
        return quaternarySearch(A, left, q1 - 1, x)
    else if A[q1] < x and x < A[q2] then
        return quaternarySearch(A, q1 + 1, q2 - 1, x)
    else if A[q2] < x and x < A[q3] then
        return quaternarySearch(A, q2 + 1, q3 - 1, x)
    else if x > A[q3] then
        return quaternarySearch(A, q3 + 1, right, x)
    else
        if x == A[q1] then
            return q1
        else if x == A[q2] then
            return q2
        else if x == A[q3] then
            return q3

    return -1
```

Figure 3: Quaternary search algorithm pseudocode

b)

$$T(n) = \begin{cases} O(1), & if\ right < left\ or\ left - right \leq 4 < n \\ T\left(\frac{n}{4}\right) + O(1), & otherwise \end{cases}$$

$O(1)$ above in both the base and recursive cases is for evaluating conditionals and assignments.

c)

At depth 0, there is 1 sequence of size n. At depth 1, there are 4 sequences, each of size $\frac{n}{4} = \frac{n}{4^1}$. At depth 2, there are 16 sequences, each of size $\frac{n}{16} = \frac{n}{4^2}$. Hence, following this pattern, at arbitrary depth h, there are 4^h sequences, each of size $\frac{n}{4^h}$. If h is the terminal depth of the recursion tree, then:

$$\frac{n}{4^h} = 1$$

$$n = 4^h$$

$$h = \log_4 n$$

which is the required runtime.

$\therefore T(n) \in O(\log_4 n)$

d)

Using the change of base property of logarithms:

$T(n) \in O(\log_4 n)$

$T(n) \in O\left(\frac{\log_2 n}{\log_2 4}\right)$

$T(n) \in O\left(\frac{1}{2}\log_2 n\right)$

$T(n) \in O(\log_2 n)$

Quaternary search is no faster than binary search. They essentially have the same runtime complexity but noting that quaternary search needs more operations in each function call, quaternary search is more expensive runtime-wise in this manner. In addition, quaternary search adds more complexity to the implementation.

e)

If $k < n$:

$$T(n) = \begin{cases} O(1), & if\ left < right\ or\ left - right \leq k < n \\ kT\left(\frac{n}{k}\right) + O(1), & otherwise \end{cases}$$

Since there are k sequences every recursive call and using the change of log base rule again, $T(n) \in O(\log_k n) \rightarrow$ $T(n) \in O(\log_2 n)$

However, if $k = n$:

$$T(n) = \begin{cases} O(1), & if\ right < left \\ O(n), & if\ left - right \leq n \\ T\left(\frac{n}{n}\right) + O(1), & otherwise \end{cases}$$

$$T(n) = \begin{cases} O(1), & if\ right < left \\ O(n), & if\ left - right \leq n \\ T(1) + O(1), & otherwise \end{cases}$$

We know that it is impossible to get to the third case because from the second case, we either find the element in A, don't find the element in A or throw an index out of bounds error if the index is outside the length of A. So we can disregard the third case:

$$T(n) = \begin{cases} O(1), & if\ right < left \\ O(n), & if\ left - right \leq n \end{cases}$$

Note that an $O(n)$ is the same runtime complexity for the worst-case scenario of searching through an entire array to find whether an element is in the array – this is actually what happens since the third case never gets executed for n-ary search. In general, a k-ary search (with $2 < k < n$) will take many more operations on each recursive call than simple binary search, thus requiring more memory to maintain the call stack. Evidently, it performs worse in runtime than binary search.

References

- Eslam Ghanem, 20/01/11, Eslam Ghanem 's blog, *Recursion VS Iteration (Looping) : Speed & Memory Comparison*, viewed 20/08/20, http://eslamghanem.blogspot.com/2011/01/recursion-vs-iteration-looping-speed.html
- David Bolton, 23/06/19, ThoughtCo., *Definition of Stack in Programming*, viewed 20/08/20, https://www.thoughtco.com/definition-of-stack-in-programming-958162