COMP3506 Homework 1 Weighting: 15%

Mohammad Faiz ATHER (s46484813)

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

Questions

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

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

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

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

Solution:

For n being odd the the execution will jump to the following block of code.

```
$sum \leftarrow 0$                ▷ 1 primitive operation.
while $n > 0$ do                 ▷ executed $\lceil \log_2(n) \rceil + 1$ times.
    $sum \leftarrow sum + (n \& 1)$   ▷ bit-wise, addition, set − $(1+1+1) * \lceil \log_2(n) \rceil$.
    $n \leftarrow (n >> 1)$           ▷ effectively division by 2 (right shifting by 1) and set − $(1+1) * \lceil \log_2(n) \rceil$.
end while                        ▷ Overall primitive operations are still $6 * \lceil \log_2(n) \rceil + 2$.
```

Assumption: Considering the code is being run on a system with the binary number representation is such that the MSB(most significant bit) is the left most bit. In that case the operation on line 12 is division by 2.

It will take $\lceil \log_2(n) \rceil$ divisions for n to become equal to zero as 0's will be introduced to the binary number from the MSB making LSB's fall off.

The approximate running time, $T_{\text{odd}}(n)$ is $f(n) = 6 * \lceil \log_2(n) \rceil + 2$

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

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

Solution:

$f(n) \leq c * g(n) \rightarrow 6 * \lceil \log_2(n) \rceil + 2 \leq c * g(n) \rightarrow c * g(n) - (6 * \lceil \log_2(n) \rceil + 2) \geq 0.$

Let $g(n)$ be equal to $\lceil \log_2(n) \rceil$.

$(c-6) * \lceil \log_2(n) \rceil - 2 \geq 0 \rightarrow (c-6) * \lceil \log_2(n) \rceil \geq 2.$

Now for $c > 0$ and $n \geq n_0 \rightarrow n_0 = 3$ since this is the odd case and to avoid logarithm of 1 that is 0 and $c = 8$ to keep the function positive and greater than two and ceiling functioncan be ignored.

Therefore, $T_{\text{odd}}(n), f(n) = 6 * \lceil \log_2(n) \rceil + 2$ is $O(\log_2(n)))$

(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:

$f(n) \geq c * g(n) \rightarrow 6 * \lceil \log_2(n) \rceil + 2 \geq c * g(n) \rightarrow 6 * \lceil \log_2(n) \rceil + 2 - c * g(n) \leq 0.$

Let $g(n)$ be equal to $\lceil \log_2(n) \rceil$.

$(6-c) * \lceil \log_2(n) \rceil + 2 \leq 0 \rightarrow (6-c) * \lceil \log_2(n) \rceil \leq -2 \rightarrow (c-6) * \lceil \log_2(n) \rceil \geq 2.$

Now for $c > 0$ and $n \geq n_0 \rightarrow n_0 = 3$ since this is the odd case and to avoid logarithm of 1 that is 0 and $c = 8$ to keep the function positive and greater than two and ceiling function can be ignored.

Therefore, $T_{\text{odd}}(n), f(n) = 6 * \lceil \log_2(n) \rceil + 2$ is $\Omega(\log_2(n))$

Since $f(n)$ is $O(\log_2(n))$ and $\Omega(\log_2(n))$ fulfilling the condition $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all $n \geq n_0$ and $c_1, c_2 > 0$ and hence, $T_{\text{odd}}(n), f(n) = 6 * \lceil \log_2(n) \rceil + 2$ is $\Theta(\log_2(n))$.

(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 $T_{\text{even}}(n)$. 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:

```
$sum \leftarrow 0$                                   ▷ set, primitive operation count is – 1
**if** $n \% 2 == 0$ **then**                        ▷ mod and check – 2
  **for** $i = 0$ to $n$ **do**                      ▷ $n$
    **for** $j = i$ to $n^2$ **do**                  ▷ $n^2$
      $sum \leftarrow sum + i + j$    ▷ $n * n^2 - \sum_{i=0}^{n-1} i \rightarrow n^3 - (n-1)((n-1)+1)/2 \rightarrow (2*n^3 - n^2 + n)/2$
    **end for**                                      ▷ $(2*n^3 - n^2 + n)/2 + n^2$
  **end for**   ▷ adding everything up $(2*n^3 - n^2 + n)/2 + n^2 + n + 2 + 1 \rightarrow 2*n^3 + (n^2)/2 + (3/2)*n + 3$
**end if**
```

$T_{\text{even}}(n),\ f(n) = 2 * n^3 + (n^2)/2 + (3/2) * n + 3$

Dropping the lower order terms and constants gives us $T_{\text{even}}(n),\ f(n)$ is $O(n^3)$ and $\Omega(n^3)$

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

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

Solution:

$$T(n) = \begin{cases} T_{\text{even}}(n) & \text{if } n \text{ is even, worst case with } O(n^3). \\ T_{\text{odd}}(n) & \text{if } n \text{ is odd best case with } \Omega(\log_2(n)). \end{cases}$$

(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: No, Big-Θ does not exist as the entire function for $T(n)$ is actually piece-wise for the odd and even case with different upper and lower bounds.

$T(n)$ is $\Omega(\log_2(n))$ and $T(n)$ is $O(n^3)$, hence Big-Θ does not exit.

(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: Yes and no. Although the answer will not be very accurate as we drop the lower order terms and the constants that may affect the actual $T(n)$ but the Big-Ω will provide a $g(n)$ that are a class of functions that will give us a estimate of the best case as it is a bound on the entire $T(n)$ from below multiplied by a $c > 0$. And similarly a Big-O will give us a $h(n)$ multiplied with a constant $d > 0$ that will be an estimate for the upper bound of the entire $T(n)$ giving us a estimate of the worst case.

So, final answer is yes as for big values of n we can get a fair insight on how the algorithm behaves in best and worst cases. Big values of n are the ones that are actually useful.

(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:

Worst case(high) bounded from above by $c_1 * g(n)$ by $O(g(n))$;

Best case(low) bounded from below by $c_2 * g(n)$ by $\Omega(g(n))$;

i.e. $c_2 * g(n) \leq T(n) = f(n) \leq c_1 * g(n)$ — c_1,$c_2 > 0$, $n \geq n_0$;

Hence, $T(n) = f(n)$ is $\Theta(g(n))$.

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:

```
int
FINDPOSITION_R (int *A, int left , int right)
{
        int res = -1;

        if (right < left) return res;

        int m = (left+right)/2;
        if ( A[m] > m )
                res = FINDPOSITION_R (A, left , m-1);
        else if ( A[m] < m )
                res = FINDPOSITION_R (A, m+1, right);
        else if ( A[m] == m )
                return m;
        return res;
}
```

(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:

$$\begin{tikzcd}
FindPosition(A,1,3) \arrow[d] \arrow[dr] & \\
m=(1+3)/2 & if(A[2]==2)\ return\ 2; \\
FindPosition(A,0,3) \arrow[d] \arrow[dr] & \\
m=(0+3)/2 & if(A[1]<1)\ return\ FindPosition(A,1,3); \\
FindPosition(A,0,8) \arrow[d] \arrow[dr] & \\
m=(0+8)/2 & if(A[4]>4)\ return\ FindPosition(A,0,3);
\end{tikzcd}$$

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

Solution:

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

$T(n) = O(1) + T(n/2)$
$= O(1) + (O(1) + T(n/4))$
$= O(1) + (O(1) + (O(1) + T(n/8)))$
...
$= O(1) + O(1) + O(1) + ... + T(n/n)$
$= O(1) + O(1) + O(1) + ... + O(1)$; $\log_2(n)$ times as the divisor increases by a factor of 1/2 each time. Halve of the array is being eliminated from search area each recursive function call.
Therefore, the answer for the worst case is $\log_2(n)$.

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

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

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

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

i. (1 mark) Use the master theorem, as stated above, to find a Big-Θ bound (and confirm your already found Big-O) for the recurrence you gave in (b). Show all your working.

Solution:

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

$a = 1, b = 2, g(n) = 1, c = 1;$
$g(n) \in \Theta(n^d)$ where $d \geq 0$, $d = 0$
$g(n) \in \Theta(n^0) \in \Theta(1)$
$a = b^d$ is condition 2,
So $T(n) \in \Theta(n^d \log(n)) \in \Theta(\log(n))$ for k non-negative being 1 by information about $T(1) = O(1)$.

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:

$$T(n) = \begin{cases} 5 * T(n/3) + n^2 + 2n, & \text{if } n > 1; \\ 100, & \text{if } n = 1. \end{cases}$$

$a = 5, b = 3, g(n) = n^2 + 2n, c = 100;$
$g(n) \in \Theta(n^d)$ where $d \geq 0$, $d = 2$
$g(n) \in \Theta(n^2)$
$a < b^d$ is condition 1,
So $T(n) \in \Theta(n^d) \in \Theta(n^2)$ for k non-negative being 100 by information about $T(1) = 100$.

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:

$$T(n) = \begin{cases} 8 * T(n/4) + 5n + 2\log(n) + 1/n, & \text{if } n > 1; \\ 1, & \text{if } n = 1. \end{cases}$$

$a = 8, b = 4, g(n) = 5n + 2\log(n) + 1/n, c = 1;$
$g(n) \in \Theta(n^d)$ where $d \geq 0$, $d = 1$
$g(n) \in \Theta(n^1)$
$a > b^d$ is condition 3,
So $T(n) \in \Theta(n^{log_b(a)}) \in \Theta(n^{log_4(8)}) \in \Theta(n^{2log_4(4)}) \in \Theta(n^2)$ for k non-negative being 1 by information about $T(1) = 1$.

(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:

```
int
FINDPOSITION_I ( int *A, int left , int right )
{
        int m = -1;
        while (left < right)
        {
                m=(left+right)/2;

                if ( A[m] == m ) return m;
                else if ( A[m] > m ) right = m-1;
                else left = m+1;
        }
        return -1;
}
```

(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: The itertive solution would be more efficient for the JVM as it would have to do extra work for to create the multiple function stacks increasing the load on the CPU and the operating system by increasing the instructions to setup stacks and destroy stacks for the $\log_2(n)s$ times the recursive function is called. Same goes for C or python a recursive solution creates overhead on the CPU, while the iterative solution will run in the same time complexity with reduced overhead of function stacks as well as reduced memory usage compared to recursive where memory is being held by the previous not yet destroyed function stacks to keep track of variables.

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:

   T[m][n] grid; $m * n * int$

   int minimumX;

   int maximumX;

   int minimumY;

   int maximumY;

   $4 * int$

   Total is $m * n + 4 \in O(m * n)$

   (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: Not good, better to store as a 2D linked list sorted by x and y with coordinates inside the struct, this way the number of elements of type T added will be stored only compared to $m * n$ for space complexity this method is very efficient.

   (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:

   (e) `add` $O(1)$

   (f) `get` $O(1)$

   (g) `remove` $O(1)$

   (h) `resize` $O(m * n)$

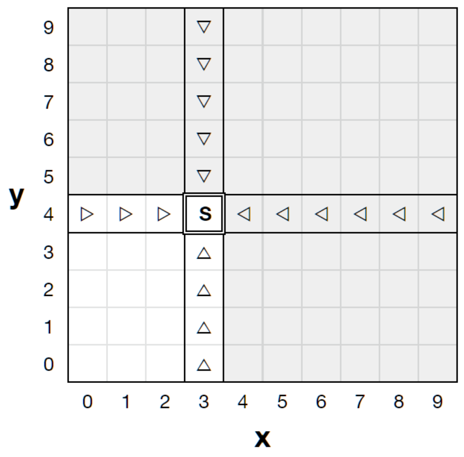
   (i) `clear` $O(m * n)$

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 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:
   Approach: Two dimensional binary search moving in the diagonal direction.
   At most $\log(n^2) = 2\log(n) \in O(log(n))$
   If in one of the x,y direction we land on the arrows we will no longer move in that direction as we know we are on the right row or column.
   Clarification: in the code below for part a) and b) the axis is considered to be $(0, 0)$ is the top-left most. The point $(1, 0)$ is the point to the right of $(0, 0)$ in the top column, i.e. moving in right direction for a increase in x. Also, the array is inverted to axis, i.e. $arr[y_value][x_value]$

```
int
calculateJump (int min, int max)
{
    if (min==max) return min;
    return (max+min)/2;
}

int *
search (int (*arr)[ARR_SIZE_X], int left, int right, int down, int up)
{
    int *res=NULL;
    int j_x = calculateJump (left, right);
    int j_y = calculateJump (down, up);
    switch ( arr[j_y][j_x] )
    {
        case NBREAK:
            res = search (arr, j_x+1, right, j_y+1, up);
            break;
        case BREAK:
            res = search (arr, left, j_x-1, down, j_y-1);
            break;
        case RIGHT:
            res = search (arr, j_x+1, right, j_y, j_y);
            break;
        case LEFT:
            res = search (arr, left, j_x-1, j_y, j_y);
            break;
        case UP:
            res = search (arr, j_x, j_x, j_y+1, up);
            break;
        case DOWN:
            res = search (arr, j_x, j_x, down, j_y-1);
            break;
        case FOUND:
            res = (int *)malloc (sizeof(int)*2);
            res[0] = j_x;
            res[1] = j_y;
            return res;
    }
    return res;
}
```

(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:

$O(n^{1/2})$ by the int $j = calculateJump(n)$ function dividing the array up going up the diagonal i.e. n by n means $n * \sqrt{2}$ top right most point.

from $n = j(j+1)/2$ formula getting $j = (2 * n + 0.25)^{1/2} - 0.5$ taking at most 2 bits drilled and that is $\in O(n^{1/2})$ being better than $O(n)$ because if 1 bit breaks we go switch to iterative search.

So our first jump in the diagonal direction by splitting the jumps in horizontal and vertical is j, $j-1$, $j-2$, ..., 0, giving us a total of n and n in the x and y directions and $n * \sqrt{2}$ in the diagonal if we never break a drill bit meaning the (n, n) is where the S was and this value is bounded by $j = (2 * n + 0.25)^{1/2} - 0.5$ so if we give n as $n * \sqrt{2}$ we still get $O(\sqrt{n})$ even if we break a drill bit we would only have to check that j many spots iterative.

```
int
calculateJump (int min, int max)
{
    if ( min == max ) return min;
    return (int)ceil ((sqrt ( ((2.0 * (max - min)) + 0.25) ) - 0.5 ));
}

int *
search (int (*arr)[ARR_SIZE_X], int left, int right, int down, int up)
{
    return searchR (arr, left, right, down, up,\
                    calculateJump(left, right), calculateJump(down, up));
}

int *
searchR (int (*arr)[ARR_SIZE_X], int left, int right, int down, int up,\
                                    int j_x, int j_y)
{
    int *res=NULL;
    switch ( arr[down+j_y][left+j_x] )
    {
        case NBREAK:
            res = searchR (arr, left+j_x, right, down+j_y, up, j_x-1, j_y-1); break;
        case BREAK:
            return searchI (arr, left, left+j_x-1, down, down+j_y-1, true, true);
        case RIGHT:
            res = searchR (arr, left+j_x, right, down+j_y, down+j_y, j_x-1, 0); break
        case LEFT:
            return searchI (arr, left, left+j_x-1, down+j_y, down+j_y, true, false);
        case UP:
            res = searchR (arr, left+j_x, left+j_x, down+j_y, up, 0, j_y-1); break;
        case DOWN:
            return searchI (arr, left+j_x, left+j_x, down, down+j_y-1, false, true);
        case FOUND:
            res = (int *)malloc (sizeof(int)*2);
            res[0] = left + j_x;    res[1] = down + j_y;
            return res;
    }
    return res;
}

int *
searchI (int (*arr)[ARR_SIZE_X], int left, int right, int down, int up,\
                                bool r_on, bool u_on)
{
    int *res = (int *)malloc (sizeof(int)*2);
    while ( arr[down][left] != FOUND )
    {
        if ( arr[down][left] == RIGHT ) u_on=false;
        else if (arr[down][left] == UP) r_on=false;
        if (r_on) left++;
        if (u_on) down++;
    }
    res[0] = left; res[1] = down;
    return res;
}
```

—-This is the end of question 4—-

All of the code with minimal tests are given below. To aid in explanation of functions of questions and are not part of the answers, this just provides reference.

compilation.txt

```
gcc -o findPosition findPosition.c -Wall -Werror -pedantic -lm -g -pg; ./findPosition
gcc -o search search.c -Wall -Werror -pedantic -lm -g -pg; ./search
gcc -o fastSearch fastSearch -Wall -Werror -pedantic -lm -g -pg; ./fastSearch
```

findPosition.c Question 2

```c
#include <stdio.h>

int
FINDPOSITION_R (int *, int , int);

int
FINDPOSITION_I (int *, int , int);

//Q2)b)
int
main (void)
{
  int A[9] = {-1,0,2,3,10,11,23,24,102};

  int res = FINDPOSITION_R ( A, 0, 8 );

  printf ("%d\n", res);

  res = FINDPOSITION_I ( A, 0, 8 );

  printf ("%d\n", res);

  return 0;
}

//Q2)a)
int
FINDPOSITION_R (int *A, int left, int right)
{
  int res = -1;

  if (right < left) return res;

  int m = (left+right)/2;
  if ( A[m] > m )
    res = FINDPOSITION_R (A, left, m-1);
  else if ( A[m] < m )
    res = FINDPOSITION_R (A, m+1, right);
  else if ( A[m] == m )
    return m;
  return res;
}

//Q2)e)
int
FINDPOSITION_I ( int *A, int left, int right )
{
  int m = -1;
  while (left < right)
  {
    m=(left+right)/2;

    if ( A[m] == m ) return m;
    else if ( A[m] > m ) right = m-1;
    else left = m+1;
  }
  return -1;
}
```

search.c Question 4

```c

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define NFOUND  -1
#define FOUND   0
#define BREAK   1
#define NBREAK  2
#define RIGHT   3
#define LEFT    4
#define UP      5
#define DOWN    6

#define ARR_SIZE_X  20
#define ARR_SIZE_Y  20

void
show (int (*)[ARR_SIZE_X]);

int *
search (int (*)[ARR_SIZE_X], int, int, int, int);

int
calculateJump (int, int);

/*
int arr[ARR_SIZE_Y][ARR_SIZE_X] = {
    {NBREAK,NBREAK,NBREAK,UP,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,UP,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,UP,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,UP,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {RIGHT,RIGHT,RIGHT,FOUND,LEFT,LEFT,LEFT,LEFT,LEFT,LEFT},\
    {BREAK,BREAK,BREAK,DOWN,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {BREAK,BREAK,BREAK,DOWN,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {BREAK,BREAK,BREAK,DOWN,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {BREAK,BREAK,BREAK,DOWN,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {BREAK,BREAK,BREAK,DOWN,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK}
};
*/

int arr[ARR_SIZE_Y][ARR_SIZE_X] = {
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {RIGHT,RIGHT,RIGHT,RIGHT,RIGHT,RIGHT,RIGHT,RIGHT,RIGHT,FOUND,\
    LEFT,LEFT,LEFT,LEFT,LEFT,LEFT,LEFT,LEFT,LEFT,LEFT},\
    {BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,DOWN,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,DOWN,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,DOWN,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,DOWN,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,DOWN,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK}
};

int
main (void)
{
    show (arr);
    int *res = search (arr, 0, ARR_SIZE_X-1, 0, ARR_SIZE_Y-1);
    printf ("x=%d y=%d\n", res[0], res[1]);
    free (res);
    return 0;
}

void
show (int (*arr)[ARR_SIZE_X])
{
    for (int j = 0; j < ARR_SIZE_Y; j++)
    {
        for (int i = 0; i < ARR_SIZE_X; i++)
        {
            printf ("%d ", arr[j][i]);
        }
        printf ("\n");
    }
}

int
calculateJump (int min, int max)
{
    if (min==max) return min;
    return (max+min)/2;
}

int *
search (int (*arr)[ARR_SIZE_X], int left, int right, int down, int up)
{
    int *res=NULL;
    int j_x = calculateJump (left, right);
    int j_y = calculateJump (down, up);
    switch ( arr[j_y][j_x] )
    {
        case NBREAK:
            res = search (arr, j_x+1, right, j_y+1, up);
            break;
        case BREAK:
            res = search (arr, left, j_x-1, down, j_y-1);
            break;
        case RIGHT:
            res = search (arr, j_x+1, right, j_y, j_y);
            break;
        case LEFT:
            res = search (arr, left, j_x-1, j_y, j_y);
            break;
        case UP:
            res = search (arr, j_x, j_x, j_y+1, up);
            break;
        case DOWN:
            res = search (arr, j_x, j_x, down, j_y-1);
            break;
        case FOUND:
            res = (int *)malloc (sizeof(int)*2);
            res[0] = j_x;
            res[1] = j_y;
            return res;
    }
    return res;
}
```

fastSearch.c Question 4

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdbool.h>

#define NFOUND  -1
#define FOUND    0
#define BREAK    1
#define NBREAK   2
#define RIGHT    3
#define LEFT     4
#define UP       5
#define DOWN     6

#define ARR_SIZE_X   20
#define ARR_SIZE_Y   20

void
show (int (*)[ARR_SIZE_X]);

int *
search (int (*)[ARR_SIZE_X], int, int, int, int);

int *
searchR (int (*)[ARR_SIZE_X], int, int, int, int, int, int);

int *
searchI (int (*)[ARR_SIZE_X], int, int, int, int, bool, bool);

int
calculateJump (int, int);

int arr[ARR_SIZE_Y][ARR_SIZE_X] = {
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,NBREAK,UP,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {RIGHT,RIGHT,RIGHT,RIGHT,RIGHT,RIGHT,RIGHT,RIGHT,RIGHT,FOUND,\
    LEFT,LEFT,LEFT,LEFT,LEFT,LEFT,LEFT,LEFT,LEFT,LEFT},\
    {BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,DOWN,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,DOWN,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,DOWN,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,DOWN,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK},\
    {BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,DOWN,\
    BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK,BREAK}
};


int
main (void)
{
    show (arr);
    int *res = search (arr, 0, ARR_SIZE_X-1, 0, ARR_SIZE_Y-1);
    printf ("x=%d y=%d\n", res[0], res[1]);
    free (res);
    return 0;
}

void
show (int (*arr)[ARR_SIZE_X])
{
    for (int j = 0; j < ARR_SIZE_Y; j++)
    {
        for (int i = 0; i < ARR_SIZE_X; i++)
        {
            printf ("%d ", arr[j][i]);
        }
        printf ("\n");
    }
}

int
calculateJump (int min, int max)
{
    if ( min == max ) return min;
    return (int)ceil ((sqrt ( ((2.0 * (max - min)) + 0.25) ) - 0.5 ));
}

int *
search (int (*arr)[ARR_SIZE_X], int left, int right, int down, int up)
{
    return searchR (arr, left, right, down, up,\
                calculateJump(left, right), calculateJump(down, up));
}

int *
searchR (int (*arr)[ARR_SIZE_X], int left, int right, int down, int up,\
                                    int j_x, int j_y)
{
    int *res=NULL;
    switch ( arr[down+j_y][left+j_x] )
    {
        case NBREAK:
            res = searchR (arr, left+j_x, right, down+j_y, up, j_x-1, j_y-1); break;
        case BREAK:
            return searchI (arr, left, left+j_x-1, down, down+j_y-1, true, true);
        case RIGHT:
            res = searchR (arr, left+j_x, right, down+j_y, down+j_y, j_x-1, 0); break;
        case LEFT:
            return searchI (arr, left, left+j_x-1, down+j_y, down+j_y, true, false);
        case UP:
            res = searchR (arr, left+j_x, left+j_x, down+j_y, up, 0, j_y-1); break;
        case DOWN:
            return searchI (arr, left+j_x, left+j_x, down, down+j_y-1, false, true);
        case FOUND:
```

```c
            res = (int *)malloc (sizeof(int)*2);
            res[0] = left + j_x;     res[1] = down + j_y;
            return res;
    }
    return res;
}

int *
searchI (int (*arr)[ARR_SIZE_X], int left, int right, int down, int up,\
                                bool r_on, bool u_on)
{
    int *res = (int *)malloc (sizeof(int)*2);
    while ( arr[down][left] != FOUND )
    {
        if ( arr[down][left] == RIGHT ) u_on=false;
        else if (arr[down][left] == UP) r_on=false;
        if (r_on) left++;
        if (u_on) down++;
    }
    res[0] = left; res[1] = down;
    return res;
}
```