Assignment 1

Name: Hongzhi Fu
Student Number: 1058170

1.

```
// Abs(n) means absolute value of a number n
function IsAffected((px, py), (x, y), b)
    if Abs(px - x) <= b and Abs(py - y) <= b then
        return True
    else
        return False
```

2. On the one hand, the purpose of **if** statement is to determine whether low index and high index have the same value after recursively calling the function **MarkAffectedDC.** On the other hand, the **if** statement makes the algorithm stop calling itself repetitively, which means to return value if the initial condition met, because divide and conquer algorithm tells us we divide the large problem into half(one third or others) recursively until the smallest problem can be solved directly. If we eliminate the **if/else** statement, because we don't have an instruction to our program when it is time to stop recursion, the recursive statement will be infinitely executed. Concretely, our program always inspects whether A[lo] is affected by human player for each recursion.

3. As we can see from the question, the basic operations are calling IsAffected and Mark, which is 2, so we can conclude that T(1) = 2. In addition, the pseudocode shows for each recursion, the function **MarkAffectedDC** calls itself twice and divide the size of problem **n** into half, so the recurrence relation should be T(n) = 2T(n/2) so the fourth choice is correct.

4. According to recurrence relation T(n) = 2T(n/2) for n > 1 and T(1) = 2, we can assume that $n = 2^k$ if n is the power of 2, so back substitution can be denoted as follows:
$T(2^k) = 2T(2^{k-1})$ substitute $T(2^{k-1}) = 2T(2^{k-2})$
$T(2^k) = 2[2T(2^{k-2})] = 2^2T(2^{k-2})$ substitute $T(2^{k-2}) = 2T(2^{k-3})$
...
$T(2^k) = 2^iT(2^{k-i})$
...
$T(2^k) = 2^kT(2^{k-k}) = 2^kT(1) = 2*2^k$
After substituting 2^k to the original variable n, $T(n) = 2n \in \Theta(n)$. Hence, we can prove the worst case complexity of the MarkAffectedDC algorithm is in $\Theta(n)$

5.

```
// Max(n1, n2,...) returns the maximum value of the multiple values n1, n2, ...
// Abs(n) means absolute value of a number
function LessOrEqualTo((x1, x2), (x2, y2))
    If Max(Abs(px - x1), Abs(py - y1)) <= Max(Abs(px - x2), Abs(py - y2)) then
        return True
    else
        return False
```

6.

```
function FindAffected((px, py), A, b)
    lo ← 0
    hi ← length(A) - 1
    While lo <= hi do
        mid ← lo + ⌊(hi - lo) / 2⌋
        if IsAffected((px, py), A[mid], b) then
            lo ← mid + 1
        else
            hi ← mid - 1
    return A[0..lo-1]
```

The returned value is a sub-array of A whose elements will be affected by human player.

7. In the worst case, all elements need to be marked, which is N. The function **FindAffected** will return exactly entire array of A. By doing this, it needs log(N) times comparisons. After that, implementing the behaviour of **MarkAffected**, we need to mark N enemy players, so the program needs to execute a total of ($\log_2(N) + N$) times operation, so the worst case complexity in this case should be $\Theta(N)$.

8. Because of uniform distribution of enemy AIs on the game board, the probability of enemy AIs in or on the boarders of a box of bound b should be the ratio of the area of boarder of the box, whose bound is b, and the area of the game board, which is $(2M)^2$. Thus, the probability $P = (2b)^2 / (2M)^2 = b^2/M^2$. If the number of enemy AIs is N, we can deduce that there are b^2N/M^2 expected enemy players, which is also the expected complexity. Hence, d will be replaced by b^2N/M^2.

9. For average case, the number of expected enemy players that need to be marked is b^2N/M^2, but no matter how many enemy players are, we always pass the entire array A to our divide-and-conquer algorithm as a parameter, so the inspection of each element in the array A is needed to determine whether an enemy AI is affected by human player. Hence, the complexity of average case should be $\Theta(N)$.

10. For the best case of finding affected enemy AIs and marking them, the number of enemy players that needs to be marked is always a constant, even the growth of size of N. The primitive divide-and-conquer algorithm will always compare N times to determine whether a particular position should be marked or not, so the time complexity for best case is $\Theta(N)$. Using pre-sorted array, however, we need to spend $\Theta(\log N)$ complexity to select sub-array of the sorted array that need to be marked, and another constant complexity to mark them. Hence, the total complexity of the algorithm should be $\Theta(\log N) + \Theta(1) = \Theta(\log N)$.

11.

function CanDirectlyCommunicate((x_1, y_1), (x_2, y_2), (p_x, p_y), b)
    $a \leftarrow y_2 - y_1$
    $b \leftarrow x_1 - x_2$
    $c \leftarrow x_1y_2 - x_2y_1$
    if $(a(p_x + b) + b(p_y + b) - c)(a(p_x - b) + b(p_y - b) - c) <= 0$ **or** $(a(p_x + b) + b(p_y - b) - c)(a(p_x - b) + b(p_y + b) - c) <= 0$ **then**
        return True
    else
        return False

Description: straight line through two points (x_1, y_1), (x_2, y_2) can be defined as $ax + by = c$, where $a = y_2 - y_1$, $b = x_1 - x_2$, $c = x_1y_2 - x_2y_1$. If we plug in the coordinates of each pair of diagonal points($[(p_x + b, p_y + b), (p_x - b, p_y - b)]$ and $[(p_x + b, p_y - b), (p_x - b, p_y + b)]$), we will find that if two enemy players cannot be directly communicated, the coordinates of each pair must be sperated by the straight line or one(two) of the points is(are) on the straight line. In other words, at least one of the products of $a(p_x + b) + b(p_y + b) - c$ and $a(p_x - b) + b(p_y - b) - c$ or $a(p_x + b) + b(p_y - b) - c$ and $a(p_x - b) + b(p_y + b) - c$ are less or equal than zero. Hence, we can use if-else statement to determine whether two enemy AIs can be directly communicated with each other based on the boolean expression $(a(p_x + b) + b(p_y + b) - c)(a(p_x - b) + b(p_y - b) - c) <= 0$ **or** $(a(p_x + b) + b(p_y - b) - c)(a(p_x - b) + b(p_y + b) - c) <= 0$.

12.

function ShortestPath(M[ ][ ], N, A, n1, n2)
    mark each vertex to 0
    count ← 0
    D ← ∅ // A dictionary to store vertices' predecessor
    TempList ← [ ]
    LinkedList ← NULL // LinkedList is a pointer points to the first node or NULL
    Inject (queue, n1)
    count ← count + 1
    mark A[n1] with count
    while queue is not empty **do**
        v ← eject(queue)
        neighbours ← **GetNeighboursIndices**(M[ ][ ], N, v)
        for i ← 0 to len(neighbours) - 1 **do**
            if A[neighbours[i]] is marked with 0 **then**

```
                    count ← count + 1
                    mark A[neighbours[i]] with count
                    Inject (queue, neighbours[i])
                    RecordPredecessor(v, neighbours[i], D)
                if neighbours[i] = n2
                    return PresentLinkedList(D, n1, n2, TempList, LinkedList)
    return LinkedList
function GetNeighboursIndices(M[ ][ ], N, i)
    // to obtain a vertex's all neighbours with indices
    neighbours ← [ ]
    for j ← 0 to N - 1 do
        if M[i][j] = 1 then
            Append j into List neighbours
    return neighbours
function RecordPredecessor(parent, child, D)    // put key-value pair into the dictionary D
    D[child] ← parent
function PresentShortestPath(D, n1, v, TempList, LinkedList)
    while v ≠ n1 do
        Append v into TempList
        v ← D(v)    // Given the key v and assign its value to v
    LinkedList.next ← node(TempList[len(TempList) - 1])    // create a node holding an index
and a pointer points to the next node
    p ← LinkedList // p is a pointer which points to the node(here points to the first node)
    for i ← len(TempList) - 2 down to 0
        p.next ← node(TempList[i])
        p ← p.next
    return LinkedList
```