Problem 1:

(a) Since the function provided has two if statement and following by three recursive function call, each with a third slice of the original array, the number of basic operations is:

$$W(n) = \begin{cases} 1 & when\ n = 0 \\ 2 & when\ n = 1 \\ \left(3 * W\left(\frac{n}{3}\right)\right) + 2\ for\ n > 1 \end{cases}$$

*Note: W(n) = number of basic operations, n = size of Array
*Note: assume $n = 3^m\ (m = positive\ integers)$

Provence:

W(n) = 3 x W($\frac{n}{3}$) + 2
= 3^1 x W($\frac{n}{3^1}$) + 2 x 3^0 where W($\frac{n}{3^1}$) = 3^1 x W($\frac{n}{3^2}$) + 2
substitute W($\frac{n}{3^1}$) with above
= 3^1 x (3^1 x W($\frac{n}{3^2}$) + 2) + 2 x 3^0
= 3^2 x W($\frac{n}{3^2}$) + 2 x 3^0 + 2 x 3^1
= … continue above until

W(n) = 3^m x W($\frac{n}{3^m}$) + 2 x 3^0 + 2 x 3^1 + … + 3^{m-1}
= 3^m x W($\frac{n}{3^m}$) + 2 x ($\frac{3^m-1}{2}$) we know n = 3^m from assumption
= 3^m x W($\frac{3^m}{3^m}$) + $(3^m - 1)$
= 3^m x W(1) + $(3^m - 1)$ since W(1) = 2
= 3^m x 2 + $(3^m - 1)$
= $3^{m+1} - 1$ n = 3^m, then m = $\log_3 n$
= $3^{\log_3 n+1} - 1$
= $3n - 1$

(b) We know the basic operation is the line if n == … and the number of basic operation is W(n) = 3n – 1, then the T(n) = c x W(n)

To find the upper bound, we can use O-notation:
Based on Levintin's notation, if we can prove t(n) is bounded above by some constant multiple of g(n), we can say T(n) is in O(g(n).

To find T(n) <= cg(n) for all n >= n0
We can assume that some n0 and c to satisfy the above condition:
3n-1 <= 3n for n >= 1
T(n) is in O(g(n)), equivalently T(n) is in O(n)

Similar principle from above apply to Omega-notation:
Based on Levintin's notation, if we can prove t(n) is bounded below by some constant multiple of g(n), we can say T(n) is in O(g(n).

To find T(n) >= cg(n) for all n >= n0
Assume that some n0 and c to satisfy the above condition
3n-1 >= 2n for n >= 1
W(n) is in Ω(g(n)), equivalently T(n) is in Ω(n)

Since W(n) is both upper bounded and lower bounded linear time complexity, then T(n) = $\theta(n)$

Use Master Theorem, we have same answer:
W(n) = 3^1 x W($\frac{n}{3^1}$) + 2 x 3^0
Where a = 3, b = 3, d = 0
Since $a > b^d$, T(n) = $\theta(n^{\log_b a})$
= $\theta(n^{\log_3 3})$
= $\theta(n^1)$
= $\theta(n)$

(c) Based on the provided modification to the function above, the best case scenario is that the k value is always in the first index position. During the recursive function call on array A of n length, **a** is always true for every following recursive function call.
Then essentially, we don't need to call **b** & **c** at all, so the complexity in **best case** becomes:

W(n) = $W\left(\frac{n}{3}\right) + 2$ where $W\left(\frac{n}{3}\right) = W\left(\frac{n}{3*3}\right) + 2$, substitute original we have:

W(n) = $W\left(\frac{n}{3^2}\right) + 2 + 2$

W(n) = $W\left(\frac{n}{3^m}\right) + (2 + 2 + \cdots m\ times + \cdots\ + 2)$

W(n) = $W\left(\frac{3^m}{3^m}\right) + 2m$

W(n) = $W(1) + 2m$ where W(1) = 2

W(n) = $2m + 2$ where m = $\log_3 n$

W(n) = $2(\log_3 n + 1)$

The **worst case** scenario is opposite to above, k value is at the last index of A. Then we will always have to call a, then b, then c for all recursive calls. In this case, the time complexity is same as question (a):

W(n) = 3 x W($\frac{n}{3}$) + 2
= $3n - 1$

Problem 2:

Pseudocode:

```
function AreCloseFriends(num1, num2)          # num1, num2 = non-negative decimal integers

    create array A[0..7] with all 0s          # for any octal number, it only contains 0 to 7

    while num1 != 0 do
        remainder <-- num1 % 8
        A[remainder] <-- A[remainder] + 1     # once we get a part of octal number, add one to counter
        num1 <-- floor(num1 / 8)

    while num2 != 0 do
        remainder <-- num2 % 8
        A[remainder] <-- A[remainder] – 1     # once we get a part of octal number, minus one to counter
        num2 <-- floor(num2 / 8)

    for i <-- 0 to 7 do                       # if any number of the counter not equal 0, meaning num1 and
                                              # num2 octal number have different components
        if A[i] != 0 then
            return false

    return true
```

Problem 3:

(a) Assume to initialize a stack ADT, we use assume stack = stack_ADT():
Accordingly:
stack.PUSH(x) : push item x into stack
stack.POP() : pop one item from stack
stack.SIZE() : show the current stack size

Assume we are not build a new class for ENQUEUE(x) and DEQUEUE(). So we have to initialize helper stack objects as the queue objects, note DEQUEUE() does not need to take arguments.

ENQUEUE(x) : enqueue item x into the queue
DEQUEUE() : dequeue one item from the queue

Pseudocode:

```
create stack_ADT() queue_main                 # initialize a stack/queue main object
```

```
create stack_ADT() queue_help                    # initialize a stack/queue helper object


function ENQUEUE(x)
    queue_main.PUSH(x)

function DEQUEUE()
    if queue_help.SIZE() = 0 then
        if queue_help.SIZE() = 0 then            # if both queue is empty, nothing to dequeue
            return

        while queue_main.SIZE() != 0 do
            queue_helper.PUSH(queue_main.POP())          # put all item to helper, in reversed order

    itemToDequeue <-- queue_helper.POP()         # pop last one from main, record its value

    return itemToDequeue                         # return item value dequeued from queue
```

(b) ENQUEUE() one items is:
Its time complexity is $\theta(1)$. As it needs to perform PUSH() 1 times and PUSH() is a constant-time operation. Hence, the time complexity is $\theta(1)$.

DEQUEUE() one items depends on several factors:
Let x <-- queue_main.SIZE()
Let y <-- queue_help.SIZE()

First, we need to call SIZE() twice to find out whether they are empty. If both empty, nothing to dequeue, return null. If y is not 0 (check SIZE()), POP() from queue_help. This is the best case, and the time complexity is constant time, hence $\theta(1)$.

However, if there are x items in queue_main, and n items to dequeue, it needs to put all item to queue_help, which takes 3 operation each transfer (check SIZE(), POP() from queue_main and PUSH() to queue_helper). Then, POP() from queue_help. Time complexity is related to the size of queue_main, in linear time. Hence, the worst case complexity is $\theta(n)$.

(c) For above discussion, the worst case of DEQUEUE() is not been considered with Amortised analysis. If the worst case only appears rarely during many runs, its time complexity can be considered as the most cases' complexity. In this case, $\theta(1)$.

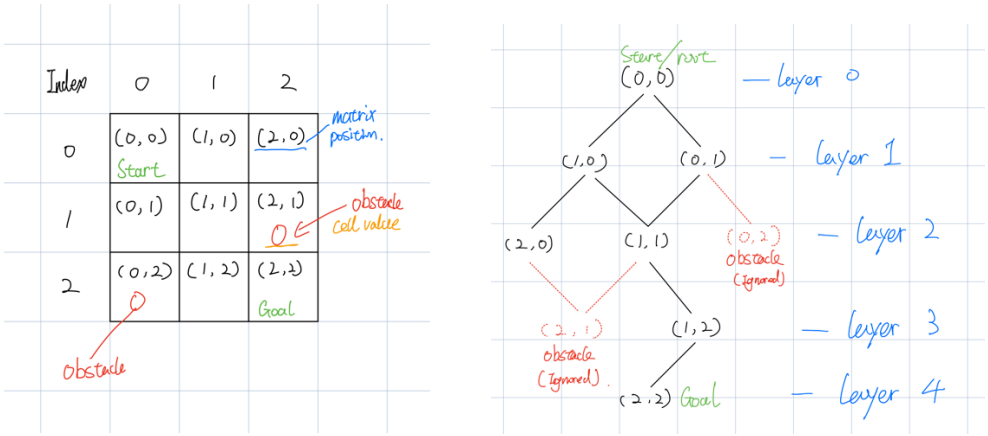
To call ENQUEUE(x) and DEQUEUE() n times in a random order, the worst case would only appear few times, while most of the cases, the DEQUEUE() is running at a constant time, $\theta(1)$.

One may argue under the extreme worst case, we may call DEQUEUE() after every time we call ENQUEUE(). That is, for every DEQUEUE() call, we need to transfer items in queue_main to queue_help. So, a method of $\theta(n) * n$ times should be $\theta(n^2)$ in terms of time complexity.

However, since we always DEQUEUE() before each enqueue, the size of queue_main, x, is always 1. The total operations for DEQUEUE() is $\theta(1) * n$. Hence, the time complexity for the entire process is still in linear time, $\theta(n)$. Same principle applies when DEQUEUE() every two call of ENQUEUE, although the size x becomes two, we also reduced half the times to transfer all items to queue_main to queue_help. In summary, the worst case time complexity is $\theta(n)$.

Problem 4:

(a) In this case, I use BFS method as the main method and use a tree as the graph to map the maze. For simplicity, let's call a (i, j) position as matrix position. And the cell value is the value referenced at this cell (for example M[i][j] = 0, where 0 is the cell value).



Sample 3 x 3 maze and tree diagram

First, I set the starting point as the root. On the tree model, it is at the top. In the maze, it is at (0,0). Based on its matrix position value, we can find its neighbour matrix.

To ensure we only travel by accessible position

If the curr_matrix (a, b) is in the middle of a maze, with no obstacle around, it should have 4 neighbours:

(a + 1, b)
(a – 1, b)
(a, b + 1)
(a, b – 1)

If any of these neighbours happen to be an obstacle (i.e. M(a + 1, b) = 0 as provided in the problem), we ignore it from the possible neighbour set.

To ensure we stay in boundaries

If any of these neighbours are out of the boundaries (i.e. curr_matrix (a, b) is on the edge of the n x n rectangles), then we ignore it from the possible neighbour set. We may use guards like if a + 1 < 0 or a + 1 > n -1 to distinguish the cells out of boundaries.

To ensure we don't iterate over visited cell

We record every cell visited and cell to visit by using list and queue. If a cell is visited, we add it to visited list. When we find its child cells (neighbour cells), we put() them to to_visit queue. Then, we get() one cell from to_visit stack, check if it is in the visited list. If yes, ignore. If not, repeat above process.

When we get the neighbour cells. We add them to the next layer of the tree map, and link each of them to the root (edge). Meaning from the first vertex (root) we can have access to each of the second layer vertex, as there is an edge linked between them.

For each vertex in second layer, we repeat the above process. 1) find the neighbours of this vertex (ignore obstacle, out-of-boundaries, and vertex already as mentioned above). 2) add each of them to the next layer, and link the parent vertex and child vertex with an edge. 3) if two parent vertex happen to have a common child vertex, draw edge from the child vertex to both parents.

Repeat the above process until we find the vertex, of which the matrix position is (n-1, n-1), or to_visited queue is empty.

To store the graph, we can use adjacency lists. The list record every vertex's parent to its child vertexs.
Use the sample 3 x 3 maze above, the list:

(0, 0) --> (1, 0), (0, 1)
(0, 1) --> (1, 1)
(1, 0) --> (2, 0), (1, 1)
(2, 0) -->
(1, 1) --> (1, 2)
(1, 2) --> (2, 2)

To improve the efficiency when trace back from goal to root, we can record every child vertex's parents vertex:

(0, 0) -->
(0, 1) --> (0, 0)
(1, 0) --> (0, 0)
(2, 0) --> (1, 0)
(1, 1) --> (1, 0), (0, 1)
(1, 2) --> (1, 1)
(2, 2) --> (1, 2)

(b) To determine if there is a path from start point to goal point, we can check the Child to Parents adjacency list. If the goal point's matrix position is in the Child vertex list, it means we have found the goal point from the start point.

If the goal point's matrix position is **not** in the Child vertex list, it means we cannot find a viable path from the start point to the goal point.

The worst case scenario happens when the goal point is the furthest point from the start. Meaning in our tree map, the goal point is always at the very bottom.

Assume setting/getting a vertex/edge label takes O(1) time
Every vertex is accessed twice, 1) DEQUEUE from to_visit 2) Add to visited
Every edge is checked twice, 1) finding neighbours 2) check if visited

Hence, the time complexity is O(V + E) while V is the number of vertex and E is the number of edges.

(c) To find the route from starting point to goal point, we start from the goal point vertex. 1) choose any parent vertex of the goal vertex, record its matrix position. 2) set the parent vertex as the child vertex, and repeat the previous step, until we reach the root vertex. As we are travelling from the bottom of the tree to top. As long as we are moving upwards, we are on the right track.

In a BFS tree map, the shortest path or the minimal number of movements is always the layer differences from the root to the goal.

Comparing to DFS, BFS may not be as fast to find the viable path between start point and goal point, when the goal point is always set to the furthest distance. However, it is more reliable in searching for the shortest path as it iterates all possible move before moving to the next layer, and do not need to deal with trace back routes.