Name: Le Thanh Cong

Student ID: 20245998

# Programming – TP 2

## Exercise 1:

Suppose we are observing a frog which would like to cross a river. The river is split into x units and at each unit there may or may not exist a stone. The frog can jump on a stone, but it must not jump into the water. Given a list of stone positions (in units), determine if the frog is able to cross the river by landing on the last stone. Initially, the frog is on the first stone and assumes the first jump must be 1 unit. If the frog's last jump was k units, then its next jump must be either k - 1, k, or k + 1 units. Note that the frog can only jump in the forward direction. You may assume that the list of positions is sorted in ascendant order.

Idea: We will use the DFS approach with stack implementation

```python
def frog_cross_river(stones: list[int]):
    # Create a stack
    # Value in the stack will be in format (stone, step)
    # Stone: the stone value itself
    # Step: the number of step that the frog use to jump to the stone
    stone_stack = []

    # Append the first stone to the stack
    stone_stack.append([0, 1])

    # Loop until stack have no value
    while(len(stone_stack) > 0):
        stone, step = stone_stack.pop()

        # Consider 3 possible case step-1, step, step+1
        for i in [-1, 0, 1]:
            # Check if there is a stone on each case
            if step + i > 0 and stone + step + i in stones:
                # If we reach the last stone, return True
                if stone + step + i == stones[-1]:
                    return True

                # Add new (stone, step) to the stack
                stone_stack.append([stone+step+i, step+i])

    # After the while loop and we still not reach the end stone,
    # It means that the frog can't cross the river, so return False.
    return False

#stones = [0,1,3,5,6,8,12,17]
stones = [0,1,2,3,4,8,9,11]
print(frog_cross_river(stones))
```

## <u>Exercise 2:</u>

We have an array of integers T, there is a sliding window of size k which is moving from the very left of the array to the very right. We can only see the k numbers in the window. Each time the sliding window moves right by one position. Return the max of each sliding window.

Idea:

- If we use the most straight forward algorithm, which is check the largest value in the k-length window, the complexity of this algorithm is n*k
- An upgrade method is to use the k-length sorted list and update it using binary search algorithm. The purpose of it is to reduce the time to get the largest number in the sliding window. The time complexity of Binary Search is O(log(k)). However, the time complexity to insert a value in a list is still O(k). So the advantage of Binary Search will be mitigated, but it will be faster compared to the first approach.
- Python has a library that supports binary search, which is **bisect.**

```
-   import bisect
-
-   def find_max_of_sliding_window(nums: list[int], k):
-       result = []
-       sorted_list = []
-       # initialize the k-length sorted_list with k first elements
-       for i in range(k):
-           bisect.insort(sorted_list, nums[i])
-
-       result.append(sorted_list[-1])
-       for i in range(k, len(nums)):
-           # remove the i-k th element from the list
-           idx = bisect.bisect_left(sorted_list, nums[i-k])
-           sorted_list.pop(idx)
-           # add the ith element to the list
-           bisect.insort(sorted_list, nums[i])
-           # add the last value of sorted_list to the result list
-           result.append(sorted_list[-1])
-
-       return result
-
-   #T = [1,3,-1,-3,5,3,6,7]
-   #k = 3
```

```
-
-   T = [1, -1]
-   k = 1
-
-   print(find_max_of_sliding_window(T, k))
```

## Problem 1:

The n-queens puzzle is the problem of placing n queens on an n x n chessboard such that no two queens attack each other.

Given an integer n, return all distinct solutions to the n-queens puzzle.

Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space, respectively.

Idea: Use backtracking algorithms with stopping conditions

```
# Check if placing a queen on this location is valid or not
def isValid(row, col, board, n):
    # check column
    for i in range(n):
        if board[row][i] == 'Q':
            return False
    # check row
    for i in range(n):
        if board[i][col] == 'Q':
            return False
    # check forward diagonal
    i = row
    j = col
    while (i >= 0 and j >= 0):
        if board[i][j] == 'Q':
            return False
        i -= 1
        j -= 1

    i = row
    j = col
    while (i < n and j < n):
        if board[i][j] == 'Q':
            return False
        i += 1
        j += 1
    # check backward diagonal
```

```python
    i = row
    j = col
    while (i >= 0 and j < n):
        if board[i][j] == 'Q':
            return False
        i -= 1
        j += 1

    i = row
    j = col
    while (i < n and j >= 0):
        if board[i][j] == 'Q':
            return False
        i += 1
        j -= 1
    return True

def backtracking(col, n, board, result):
    # if col >=n then if have one result, save it and return
    if (col >= n):
        tmp = []
        for i in range(n):
            tmp_row = ""
            for j in range(n):
                tmp_row += board[i][j]
            tmp.append(tmp_row)
        result.append(tmp)
        return

    # Loop through the row
    for i in range(n):
        # Check valid
        if isValid(i, col, board, n):
            # set the value to Q
            board[i][col] = 'Q'
            # go to next column
            backtracking(col+1, n, board, result)
            # backtracking by setting the value to '.'
            board[i][col] = '.'

def NQueen(n):
    result = []

    # create a board
    board = [["." for i in range(n)] for i in range(n)]
    # call the backtrack function starting from column 0
    backtracking(0, n, board, result)

    return result

print(NQueen(4))
```

## Problem 2:

An undirected, connected graph of N nodes (labeled 0, 1, 2, ..., N-1) is given as graph.

graph.length = N, and j != i is in the list graph[i] exactly once, if and only if nodes i and j are connected.

Return the length of the shortest path that visits every node. You may start and stop at any node, you may revisit nodes multiple times, and you may reuse edges.

Idea: Use BFS algorithm with queue implementation and bimask to store the state of visited nodes

```python
from collections import deque
def find_shortest_path(graph):
    # initialize to result (in this case the step)
    result = 0

    n = len(graph)

    # value inside the set and queue is in format (node, state)
    # node is in node itself
    # state is the bimask that store the visited_state like 1001, it means we
have visited the 4th node and 1st node
    visited_set = set()
    queue_node = deque()

    # Add all the (node, state) in the set and queue
    # At this time the state of each node is all 0 except for the node
    # For example 1st node: (1, 1), 2nd node: (2, 2), 3rd node: (3, 4)
    for i in range(n):
        state = 1 << i
        visited_set.add((i, state))
        queue_node.append((i, state))

    while True:
        # loop through all value in queue_node, this is the step "result"
        for i in range(len(queue_node)):

            node, state = queue_node.popleft()
            # if all nodes have been visited, if n = 4 then 1111 mean all nodes
are visited
            # if yes return result
            if state == (1<<n) - 1:
                return result

            # For each neighbor of the node
            for neighbor in graph[node]:
                # we add the neighbor to the state
                new_state = state | (1 << neighbor)
                # If the (neighbor, new_state) has not been visited, add it to
the queue and set
                if (neighbor, new_state) not in visited_set:
                    visited_set.add((neighbor, new_state))
                    queue_node.append((neighbor, new_state))

        # increase the step
        result += 1

graph = [[1,2,3],[0],[0],[0]]
print(find_shortest_path(graph))
```