**A close-up of a logo

Description automatically generated**

Name: Le Thanh Cong

Student ID: 20245998

Programming – TP4

**Exercise 1**: Given an input string (s) and a pattern (p), implement regular expression matching with support for '.' and '\*' where:

• '.' Matches any single character.

• '\*' Matches zero or more of the preceding elements.

The matching should cover the entire input string (not partial).

Idea: use a while loop with index of p and s to check the similarity

def valid\_pattern(s, p):

    # Initialize the index for pattern p

    i = 0

    # Simplify the pattern string by removing redundant characters after '\*'

    while i < len(p):

        # If the current character is preceded by a '\*' and matches the character before '\*',

        # it is redundant, so remove it from the pattern

        if p[i-1] == '\*':

            if p[i] == p[i-2]:

                # Remove the redundant character from pattern p

                p = p[:i] + p[i+1:]

                continue

        i += 1

    # Initialize indices for the input string (s) and the pattern (p)

    i = 0

    j = 0

    # Match the input string (s) against the simplified pattern (p)

    while i < len(s) and j < len(p):

        # If match, move on

        if s[i] == p[j]:

            i += 1

            j += 1

            continue

        # If p[j] = '.' move on

        if p[j] == '.':

            i += 1

            j += 1

            continue

        #'\*' in the pattern

        if p[j] == '\*':

            # If the current character in the string matches the character before '\*',

            # move to next index in s

            if s[i] == s[i-1]:

                i += 1

                continue

            # If the characters don't match, move to the next character in the pattern

            if s[i] != s[i-1]:

                j += 1

                continue

        # If no conditions match, return False (pattern doesn't match)

        return False

    # If both the string and pattern are not fully processed

    if i != len(s) and j != len(p):

        return False

    # If both string and pattern are fully processed, the pattern matches the string

    return True

p = 'a\*cc.b'

s = 'aaaaacceb'

print(valid\_pattern(s, p))

Exercise 2:

You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

A paper with writing on it

Description automatically generated

The result will be the total of Xb, b from 0 to int(n/2) + 1

#Idea: a is the number of 1 step taken

#      b is the number of 2 step taken

# First we find all combination of a and b such that a + 2\*b = n

# Then we calculate the number of permutation for that by this formula: (a + b)!/a!/b!    (repeated permutation)

def climb\_stair(n):

    total = 0

    tmp = 1

    total += tmp

    for b in range(1, int(n/2)+1):

        a = n - 2\*b

        tmp = tmp \* (a+1)\*(a+2)/(a+b+1)/b

        total += tmp

    return int(total)

print(climb\_stair(4))

Exercise 3:

Given a rows x cols binary matrix filled with 0's and 1's, find the largest rectangle containing only 1's and return its area.

Idea: I transform this problem from 2 dimension matrix into a problem with one dimension vector list this

A close-up of a paper

Description automatically generated

Now for each row in the right, we need to find a subarray such that the min(subarray) \* length(subarray) is maximum in that. I implemented this function in get\_max\_area(arr)

def get\_max\_area(arr):

    n = len(arr)

    s = []

    result = 0

    for i in range(n):

        while s and arr[s[-1]] >= arr[i]:

            # The popped item is to be considered as the

            # smallest element of the histogram

            tp = s.pop()

            # For the popped item previous smaller element is

            # just below it in the stack (or current stack top)

            # and next smaller element is i

            width = i if not s else i - s[-1] - 1

            res = max(result, arr[tp] \* width)

        s.append(i)

    # For the remaining items in the stack, next smaller does

    # not exist. Previous smaller is the item just below in

    # stack.

    while s:

        tp = s.pop()

        curr = arr[tp] \* (n if not s else n - s[-1] - 1)

        res = max(result, curr)

    return res

def find\_max\_rectangle\_area(board: list[list[int]]):

    row = len(board)

    col = len(board[0])

    max\_area = 0

    temp = [0] \* col

    for i in range(row):

        for j in range(col):

            # if board[i][j] = 1 then we add 1 to temp[j]

            if board[i][j] == 1:

                temp[j] += 1

            # if board[i][j] = 0 then we reset the temp[j]

            else:

                temp[j] = 0

        max\_temp = get\_max\_area(temp)

        max\_area = max(max\_area, max\_temp)

    return max\_area

board = [[1, 0, 1, 0, 0],

         [1, 0, 1, 1, 1],

         [1, 1, 1, 1, 1],

         [1, 0, 0, 1, 0]]

print(find\_max\_rectangle\_area(board))

Exercise 4:

You are given an integer array prices where prices[i] is the price of a given stock on the ith day. Design an algorithm to find the maximum profit. You may complete at most k transactions. Notice that you may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

Idea: I use a recursion approach with dynamic programming

class Solution:

    def \_\_init\_\_(self):

        # Initialize a dynamic programming table

        self.dp = []

    def h(self, prices, k, i, t, can\_buy):

        # Base case: If we reach the end of the prices array or the transaction limit

        if i >= len(prices) or t == k:

            return 0

        # If the result for the current state is already computed, return it

        if self.dp[i][t][can\_buy] != -1:

            return self.dp[i][t][can\_buy]

        # Skip the current day (do nothing)

        profit = self.h(prices, k, i + 1, t, can\_buy)

        if can\_buy:

            # Option to buy the stock on the current day

            profit = max(profit, -prices[i] + self.h(prices, k, i + 1, t, 0))

        else:

            # Option to sell the stock on the current day

            profit = max(profit, prices[i] + self.h(prices, k, i + 1, t + 1, 1))

        # Store the computed result in the DP table

        self.dp[i][t][can\_buy] = profit

        return profit

    def maxProfit(self, k, prices):

        n = len(prices)

        # Initialize the DP table: dp[n][k][2] with -1

        self.dp = [[[-1] \* 2 for \_ in range(k)] for \_ in range(n)]

        # Start from day 0, with 0 transactions, and the ability to buy

        return self.h(prices, k, 0, 0, 1)

# Test the solution

sol = Solution()

k = 2

prices = [3, 2, 6, 5, 0, 3]

print(sol.maxProfit(k, prices))  # Output the maximum profit