Assignment 1:

袋子里最少数目的球

```
class Solution:
    def minimumSize(self, nums, maxOperations):
        left, right = 1, max(nums)
        ans = right

        while left <= right:
            mid = (left + right) // 2
            operation = 0
            for num in nums:
                # integer-safe and correct count of splits needed
                operation += (num - 1) // mid

            if operation <= maxOperations:
                ans = mid
                right = mid - 1
            else:
                left = mid + 1

        return ans
```

note：二分查找

String Task

```
s = input().strip()
vowels = set("aoyeui")

result = []

for ch in s:
    if ch.lower() not in vowels:
        result.append("." + ch.lower())

print("".join(result))
```

.lower(), .upper()

add to a string: use append

Goldbach Conjecture

```
def is_prime(x):
    if x < 2:
        return False
    for i in range(2, int(x ** 0.5) + 1):
        if x % i == 0:
            return False
    return True
S = int(input())
for A in range(2, S):
    B = S - A
    if is_prime(A) and is_prime(B):
        print(A, B)
        break
```

** is sqrt

% is remainder

多项式时间复杂度

```
s = input().strip()
terms = s.split('+') #split at specific terms
max_power = 0
for term in terms: #every character
    # 处理系数和指数
    if 'n' not in term:
        # 常数项，相当于 n^0
        coef = int(term)
        power = 0
    else:
        # 拆分系数
        if term.startswith('n'): #startswith()
            coef = 1
            rest = term
        else:
            coef_part, rest = term.split('n', 1)
            coef = int(coef_part)
        # 拆分指数
        if '^' in rest:
            power = int(rest.split('^')[1])
        else:
            power = 1  # 兜底处理 n 的情况（虽然题目一般不会给）
    if coef != 0:
        max_power = max(max_power, power)

print(f"n^{max_power}")
```

Assignment 3

移动零

```
class Solution:
    def moveZeroes(self, nums):
        non_zero_pos = 0
        for i in range(len(nums)):
            if nums[i] != 0:
                nums[non_zero_pos] = nums[i]
                nums[i] = nums[non_zero_pos]
                non_zero_pos += 1
        for i in range((non_zero_pos, len(nums))):
            nums[i] = 0

        return nums
```

use a pointer to indicate which position you want to keep track

有效的括号

```
class Solution(object):
    def isValid(self, s):
        stack = []
        pairs = {')': '(', '}': '{', ']': '['} #mapping whihc ones pair together
        for char in s:
            if char in '({[':
                stack.append(char)
            else:
                if not stack or stack[-1] != pairs[char]:
                    return False
                stack.pop()
        return not stack
```

mapping pairs is cruical for this question

stack operations

- stack[]
- stack append()
- stack.pop()

not stack = if it is empty

杨辉三角

```
class Solution:
    def generate(self, numRows: int):
        triangle = []

        for i in range(numRows):
            row = [1] * (i + 1)  # 初始化每行
            for j in range(1, i):
                row[j] = triangle[i-1][j-1] + triangle[i-1][j]
            triangle.append(row)

        return triangle
```

Assignment 4

矩阵运算

```
def read_matrix():
    r, c = map(int, input().split())
    mat = []
    for _ in range(r):
        mat.append(list(map(int, input().split())))
    return r, c, mat
# 读取三个矩阵
ar, ac, A = read_matrix()
br, bc, B = read_matrix()
cr, cc, C = read_matrix()
# 检查乘法是否合法
if ac != br:
    print("Error!")
    exit()
```

w

```
# 乘法结果维度
mr mc = ar, bc
# 检查加法是否合法
if mr != cr or mc != cc:
    print("Error!")
    exit()
# 计算 A · B
D = [[0] * mc for _ in range(mr)]
for i in range(mr):
    for j in range(mc):
        for k in range(ac):
            D[i][j] += A[i][k] * B[k][j]
# 计算 (A · B) + C
E = [[D[i][j] + C[i][j] for j in range(mc)] for i in range(mr)]
# 输出结果
for row in E:
    print(" ".join(map(str, row)))
```

the importnat thing is t**o compare column and rows, assign correctly.**

draw out the matrix, understand who is mulitplying who

二维矩阵上的卷积运算

```python
# 读取输入
m, n, p, q = map(int, input().split())
matrix = [list(map(int, input().split())) for _ in range(m)]
kernel = [list(map(int, input().split())) for _ in range(p)]
# 输出矩阵大小
out_rows = m - p + 1
out_cols = n - q + 1
# 卷积计算
result = [[0] * out_cols for _ in range(out_rows)]
for i in range(out_rows):
    for j in range(out_cols):
        s = 0
        for x in range(p):
            for y in range(q):
                s += matrix[i + x][j + y] * kernel[x][y]
        result[i][j] = s
# 输出结果
for row in result:
    print(" ".join(map(str, row)))
```

倒排索引

```python
# Read number of documents
N = int(input())
index = {}  # inverted index: word -> list of doc IDs
for doc_id in range(1, N + 1):
    parts = input().split()
    c = int(parts[0])       # number of words
    words = parts[1:]
    seen = set()  # avoid duplicate words in the same
document
    for word in words:
        if word in seen:
            continue
```

```
            seen.add(word)
            if word not in index:
                index[word] = []
            index[word].append(doc_id)
    # Read number of queries
    M = int(input())
    for _ in range(M):
        query = input().strip()
        if query in index:
            print(" ".join(map(str, index[query])))
        else:
            print("NOT FOUND")
```

use a dictionary to match values to items

use continue to skiip the loop once for this iteration

相 交 链 表

```
class Solution:
    def getIntersectionNode(self, headA, headB):
        # Step 1: Get lengths of both lists
        def get_length(head):
            length = 0
            while head:
                length += 1
                head = head.next
            return length
        lenA = get_length(headA)
        lenwB = get_length(headB)
```

```
        # Step 2: Align both lists by skipping the difference in lengths
        while lenA > lenB:
            headA = headA.next
            lenA -= 1
        while lenB > lenA:
            headB = headB.next
            lenB -= 1
        # Step 3: Traverse both lists and find the intersection
        while headA and headB:
            if headA == headB:
                return headA  # Intersection found
            headA = headA.next
            headB = headB.next
        return None  # No intersection
```

反转链表

```
class Solution:
    def reverseList(self, head):
        prev = None
        curr = head
        while curr:
            nxt = curr.next
            curr.next = prev
            prev = curr
            curr = nxt
        return prev
```

Mock Exam Assignment 5

咒语序列

```
s = input().strip()
stack = [-1]    # 哨兵
max_len = 0
for i, ch in enumerate(s):
    if ch == '(':
        stack.append(i)
    else:   # ch == ')'
        stack.pop()
        if not stack:
            # 重新设定起点
            stack.append(i)
        else:
            max_len = max(max_len, i - stack[-1])
print(max_len)
```

We iterate through the string using enumerate.

If the character is '(', we push its **index** onto the stack.

If the character is ')', we pop from the stack to try to match it.

- If the stack becomes empty after popping, we push the current index as a new base (this) is invalid).
- Otherwise, we update the maximum length using

i - stack[-1].

八皇后

```
# 预计算 8 皇后所有解
solutions = []
def dfs(row, cols, diag1, diag2, path):
    if row == 8:
        # 转成字符串
        solutions.append(''.join(str(c + 1) for c in path))
        return
    for col in range(8):
        if col in cols or (row - col) in diag1 or (row + col) in diag2: #check if there are any in the same column, diagonal or other diagonal direction
            continue
        dfs(
            row + 1,
            cols | {col}, #update column
            diag1 | {row - col}, #update diag1
            diag2 | {row + col}, #update diag2
            path + [col] #
        )
dfs(0, set(), set(), set(), [])
# 排序（按整数大小）
solutions.sort()
# 处理输入
n = int(input())
for _ in range(n):
    b = int(input())
    print(solutions[b - 1])
```

USE DFS

- check for conlfict cases, like same column, diagonal

- backtracking
    - trying a choice, continuing if it works, undoinging it if it leads to a dead end

洋葱

#NAVIGATE TOP DOWN LEFT RIGHT BY HOLDING ONE OF THEM CONSTANT

find patterns in finding layers

find patterns in finding where each layer is

you just have to write out the indexes and compare it to find a function

检测括号嵌套

字符串中可能有 3 种成对的括号，"( )"、"[ ]"、"{}"。请判断字符串的括号是否都正确配对以及有　括号嵌套

```
def checkbrackets(s):
stack = []
nested = False
pairs = {')': '(', ']': '[', '}': '{'}
for ch in s:
if ch in pairs.values():
stack.append(ch)
elif ch in pairs.keys():
if not stack or stack.pop() != pairs[ch]:
return "ERROR"
if stack:
nested = True
if stack:
return "ERROR"
return "YES" if nested else "NO"
s = input()
print(checkbrackets(s)
```

Assignment 6：链表，栈，排序

后续表达式求值

```python
n = int(input())
for _ in range(n):
    expr = input().strip().split()
    stack = []
    for token in expr:
        if token in {"+", "-", "*", "/"}:
            b = stack.pop()
            a = stack.pop()
            if token == "+":
                stack.append(a + b)
            elif token == "-":
                stack.append(a - b)
            elif token == "*":
                stack.append(a * b)
            elif token == "/":
                stack.append(a / b)
        else:
            # 操作数
            stack.append(float(token))
    result = stack.pop()
    print(f"{result:.2f}")
```

回文链表

```python
class Solution:
    def isPalindrome(self, head) -> bool:
        if not head or not head.next:
            return True
        slow = fast = head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
        prev = None
        curr = slow
        while curr:
            next_temp = curr.next
            curr.next = prev
            prev = curr
            curr = next_temp
        left, right = head, prev
        while right:
            if left.val != right.val:
                return False
            left = left.next
            right = right.next
        return True
```

有多少种合法的出栈顺序

```python
n = int(input())
catalan = 1
for i in range(2, n + 1):
    catalan = catalan * (n + i) // i
catalan //= (n + 1)
print(catalan)
```

中序表达式转后续表达式

```
        else:
            while stack and stack[-1] != '(': #is a operator
                top = stack[-1] #check what the top stack is
                prec_top = 2 if top in '*/' else 1 #if top of stack is */, assign 2
                prec_curr = 2 if expr[i] in '*/' else 1 #if current index is */, assign 2
                if prec_top >= prec_curr: #compare top stack with current index
                    output.append(stack.pop()) #output top stack
                else:
                    break
            stack.append(expr[i]) #add operator to stack if not bigger
        i += 1
    while stack:
        output.append(stack.pop()) #when no more digits to check, output all of stack's remaining operators
    return ' '.join(output)
n = int(input())
for _ in range(n):
```

```
def infix_to_postfix(expr):
    output = [] #initialize answer
    stack = []# initialize stack for comparison
    i = 0
    n = len(expr)#length of input
    while i < n:
        if expr[i].isdigit() or expr[i] == '.':  #if it is a number
            num = []
            while i < n and (expr[i].isdigit() or expr[i] == '.'):
                num.append(expr[i])#add to number list
                i += 1
            output.append(''.join(num))#output numbers till next character isn't digit
            continue
        elif expr[i] == '(': #prioritize (
            stack.append('(')
        elif expr[i] == ')':
            while stack and stack[-1] != '(':
                output.append(stack.pop()) #output all the things inside the bracket
            stack.pop()  #pop out right bracket
```

Assignment 7: BFS

二叉树的层序遍历

```python
from collections import deque

class Solution:
    def levelOrder(self, root):
        if not root:
            return []
        res, q = [], deque([root])#create a queue, start from root
        while q:
            level = [] #nodes on this level
            for _ in range(len(q)): #for all the nodes on this level
                node = q.popleft() #pop it
                level.append(node.val) #add it to the result
                if node.left: #add the left children to the queue
                    q.append(node.left)
                if node.right: #add the right children
                    q.append(node.right)
            res.append(level)
        return res
```

For BFS on a tree, you need:

1. **A queue** (usually collections.deque)
2. **An initial element** (the root)
3. **A loop while the queue is not empty**
4. **Level control** (know how many nodes are in the current level)
5. **Add children to the queue**

分割回文串

https://leetcode.cn/problems/palindrome-partitioning/

```
class Solution:
    def partition(self, s):
        res = []
        path = []

        def backtrack(start):
            if start == len(s):
                res.append(path[:])
                return
            for end in range(start + 1, len(s) + 1):
                sub = s[start:end]
                if sub == sub[::-1]:
                    path.append(sub)
                    backtrack(end)
                    path.pop()

        backtrack(0)
        return res
```

岛屿数量

```
class Solution:
    def numIslands(self, grid):
        if not grid:
            return 0
        m, n = len(grid), len(grid[0])
        def dfs(i, j):
            if i < 0 or i >= m or j < 0 or j >= n or
grid[i][j] == '0':
                return
            grid[i][j] = '0'
            dfs(i+1, j)
            dfs(i-1, j)
            dfs(i, j+1)
            dfs(i, j-1)
        count = 0
        for i in range(m):
            for j in range(n):
                if grid[i][j] == '1':
                    count += 1
                    dfs(i, j)
        return count
```

https://leetcode....

DFS is good when: You want to fully explore one connected region , Mark everything visited , Then move on to find the next region

High-level DFS strategy

1. Traverse every

cell in the grid

2. When you see a '1':
    a. You've found a new island
    b. Increment island count
    c. Use DFS to flood-fill the entire island (mark it visited)
3. Continue scanning the grid

DFS essentials (mental checklist)

1. Boundary check
    a. dont go out of bounds
    b. follow the rules/dont revisit nodes
2. Base case (water or already visited)
3. Mark visited
4. Explore neighbors recursively

how to mark "visited"

Two common methods:

Option A (most common): modify the grid

Option B: separate visited set

最深叶节点的最近公共祖先

https://leetcode.cn/problems/lowest-common-ancestor-of-deepest-leaves/

```python
class Solution:
    def lcaDeepestLeaves(self, root: TreeNode) -> TreeNode:
        def dfs(node):
            if not node:
                return None,
            left_lca, left_depth = dfs(node.left)
            right_lca, right_depth = dfs(node.right)
            if left_depth > right_depth:
                return left_lca, left_depth + 1
            elif right_depth > left_depth:
                return right_lca, right_depth + 1
            else:
                return node, left_depth + 1
        lca, _ = dfs(root)
        return lca
```

单词搜索

https://leetcode.cn/problems/word-search/

```python
class Solution:
    def exist(self, board, word):
        m, n = len(board), len(board[0])
        visited = [[False]*n for _ in range(m)]
        def dfs(i, j, k):
            if k == len(word):
                return True
            if i < 0 or i >= m or j < 0 or j >= n:
                return False
            if visited[i][j] or board[i][j] != word[k]:
                return False
            visited[i][j] = True
            res = (
                dfs(i+1, j, k+1) or
                dfs(i-1, j, k+1) or
                dfs(i, j+1, k+1) or
                dfs(i, j-1, k+1)
            )
            visited[i][j] = False
            return res
        for i in range(m):
            for j in range(n):
                if dfs(i, j, 0):
                    return True
        return False
```

十进制转换为十六以下任意进制

```python
from pythonds.basic.stack import Stack

def baseConverter(decNumber,base):
    digits = "0123456789ABCDEF"

    remstack = Stack()

    while decNumber > 0:
        rem = decNumber % base
        remstack.push(rem)
        decNumber = decNumber // base

    newString = ""
    while not remstack.isEmpty():
        newString = newString + digits[remstack.pop()]

    return newString

print(baseConverter(25,2))
print(baseConverter(25,16))
```

```python
from collections import deque
q = deque()
q.append(10)      # 在右端入队
q.append(20)
q.appendleft(5)   # 在左端入队
print(q)          # deque([5, 10, 20])
q.pop()           # 右端出队 → 20
q.popleft()       # 左端出队 → 
extend(iterable) #在队列的右侧扩展一个可迭代对象的元素。
extendleft(iterable) #在队列的左侧扩展一个可迭代对象的元素，注意元素的顺序会反转。
rotate(n) #将队列向右旋转 n 步，如果 n 为负数，则向左旋转。
#两边时间复杂度都是 O (1)
```

顺序查索

```python
def sequentialSearch(alist, item):
    pos = 0
    found = False

    while pos < len(alist) and not found:
        if alist[pos] == item:
            found = True
        else:
            pos = pos + 1

    return found
```

直接插入排序

```python
def insertion_sort(alist):  # 按不减序进行直接插入排序
    for index in range(1, len(alist)):
        current_value = alist[index]
        position = index
        while position > 0 and alist[position - 1] > current_value: #compare from the end of the list
            alist[position] = alist[position - 1]
            position = position - 1
        if position != index:
            alist[position] = current_value
```

二分插入排序

```python
def binary_insertion_sort(alist):
    for i in range(1, len(alist)):
        key = alist[i]  # 当前需要插入的元素
        low = 0  # 二分查找的左边界
        high = i - 1  # 二分查找的右边界
        while low <= high:
            middle = (low + high) // 2
            if alist[middle] < key:
                low = middle + 1  # 在右半部分继续查找
            else:
                high = middle - 1  # 在左半部分继续查找
        j = i - 1
        while j >= low:
            alist[j + 1] = alist[j]  # 向右移动元素，为插入腾出空间
            j -= 1
        alist[low] = key  # 插入当前元素到正确位置
    return alist
```

冒泡排序

```python
def bubble_sort(alist):
    n = len(alist)
    for i in range(n - 1, 0, -1):
        for j in range(i):
            if alist[j] > alist[j + 1]:
                alist[j], alist[j + 1] = alist[j + 1], alist[j]
    return alist
```