CHEATSHEET

算法

二分查找

`small`, `large`为可能取到的边界值。`valid(x)`大于等于/小于等于某一值时为`True`，反之为`False`的情况，注意一下valid里面是大于号还是小于号：

```
def binary_search_greatest_lower_bound(small, large):#找到第一个valid为True的idx
    left, right = small, large
    while left < right:
        mid = (left + right) // 2
        if valid(mid):
            right = mid
        else:
            left = mid + 1
    return left

def binary_search_least_upper_bound(small, large):#在 [small, large] 中找到“最后一个使 valid(x) 为 True 的 x”
    left, right = small, large + 1
    while left < right:
        mid = (left + right) // 2
        if valid(mid):
            left = mid + 1
        else:
            right = mid
    return right - 1
```

滑动窗口

```
n = len(nums)
right = 0
for left in range(n):
    # 做与left有关的操作
    while right < n and (与right有关的某一条件):
        # 做与right有关的操作
        right += 1
    # 做一些操作，如增加计数等，比如n - right + 1
```

求排列的逆序数

```
def merge_two(left, right):
    res = []
```

```
    cnt = 0
    p1 = p2 = 0
    l1, l2 = len(left), len(right)
    while p1 < l1 and p2 < l2:
        if left[p1] <= right[p2]:
            res.append(left[p1])
            p1 += 1
        else:
            res.append(right[p2])
            p2 += 1
            cnt += l1 - p1
    res.extend(left[p1:])
    res.extend(right[p2:])
    return res, cnt

def merge_self(nums):#实际使用时直接调用_, inv_cnt = merge_self(nums)即可。_是排序后
的数组。
    l = len(nums)
    if l == 1:
        return nums, 0
    mid = l >> 1
    merged_left, cnt_left = merge_self(nums[:mid])
    merged_right, cnt_right = merge_self(nums[mid:])
    merged_two, cnt = merge_two(merged_left, merged_right)
    return merged_two, cnt + cnt_right + cnt_left

seq = [2, 6, 3, 4, 5, 1]
_, ans = merge_self(seq)
print(ans)
# 8
```

归并排序（Merge Sort）

基础知识

时间复杂度：

- **最坏情况**: *O*(*n*log*n*)
- **平均情况**: *O*(*n*log*n*)
- **最优情况**: O*(*n*log*n)
- **空间复杂度**: O(n) — 需要额外的内存空间来存储临时数组。
- **稳定性**: 稳定 — 相同元素的相对顺序在排序后不会改变。

应用

- **计算逆序对数**：在一个数组中，如果前面的元素大于后面的元素，则这两个元素构成一个逆序对。归并排序可以在排序过程中修改并计算逆序对的总数。这通过在归并过程中，每当右侧的元素先于左侧的元素被放置到结果数组时，记录左侧数组中剩余元素的数量来实现。

- **排序链表**：归并排序在链表排序中特别有用，因为它可以实现在链表中的有效排序而不需要额外的空间，这是由于链表的节点可以通过改变指针而不是实际移动节点来重新排序。

代码示例

对链表进行排序

```
class ListNode:
    def __init__(self, value=0, next=None):
        self.value = value
        self.next = next


def split_list(head):
    if not head or not head.next:
        return head

    # 使用快慢指针找到中点
    slow = head
    fast = head.next  # fast 从 head.next 开始确保分割平均
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    # 分割链表为两部分
    mid = slow.next
    slow.next = None

    return head, mid


def merge_sort(head):
    if not head or not head.next:
        return head

    left, right = split_list(head)
    left = merge_sort(left)
    right = merge_sort(right)
    return merge_lists(left, right)


# 创建链表：4 -> 2 -> 1 -> 3
head = ListNode(4, ListNode(2, ListNode(1, ListNode(3))))

# 排序链表
sorted_head = merge_sort(head)

# 打印排序后的链表
current = sorted_head
while current:
    print(current.value, end=" -> ")
```

```
    current = current.next
print("None")
```

OJ02299:Ultra-QuickSort

http://cs101.openjudge.cn/2024sp_routine/02299/

与**20018:蚂蚁王国的越野跑**（http://cs101.openjudge.cn/2024sp_routine/20018/）类似。

算需要交换多少次来得到一个排好序的数组，其实就是算逆序对。

```
d = 0


def merge(arr, l, m, r):
    """对l到m和m到r两段进行合并"""
    global d
    n1, n2 = m - l + 1, r - m  # L1和L2的长
    L1, L2 = arr[l:m + 1], arr[m + 1:r + 1]
    # L1和L2均为有序序列
    i, j, k = 0, 0, l  # i为L1指针，j为L2指针，k为arr指针
    '''双指针法合并序列'''
    while i < n1 and j < n2:
        if L1[i] <= L2[j]:
            arr[k] = L1[i]
            i += 1
        else:
            arr[k] = L2[j]
            d += n1 - i  # 精髓所在
            j += 1
        k += 1
    while i < n1:
        arr[k] = L1[i]
        i += 1
        k += 1
    while j < n2:
        arr[k] = L2[j]
        j += 1
        k += 1


def mergesort(arr, l, r):
    """对arr的l到r一段进行排序"""
    if l < r:  # 递归结束条件，很重要
        m = (l + r) // 2
        mergesort(arr, l, m)
        mergesort(arr, m + 1, r)
        merge(arr, l, m, r)


while True:
```

```
    n = int(input())
    if n == 0:
        break
    array = []
    for b in range(n):
        array.append(int(input()))
    d = 0
    mergesort(array, 0, n - 1)
    print(d)
```

快速排序（Quick Sort）

时间复杂度

- **最坏情况**: O*(*n2) — 通常发生在已经排序的数组或基准选择不佳的情况下。
- **平均情况**: O*(*nlog*n)
- **最优情况**: *O*(*n*log*n*) — 适当的基准可以保证分割平衡。
- **空间复杂度**: *O*(log*n*) — 主要是递归的栈空间。
- **稳定性**: 不稳定 — 基准点的选择和划分过程可能会改变相同元素的相对顺序。

应用：k-th元素

代码示例

普通快排

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[-1]
        less = [x for x in arr[:-1] if x <= pivot]
        greater = [x for x in arr[:-1] if x > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)

# 示例数组
array = [10, 7, 8, 9, 1, 5]
sorted_array = quicksort(array)
print(sorted_array)
```

在无序列表中选择第k大

```
def partition(nums, left, right):
    pivot = nums[right]
    i = left
    for j in range(left, right):
```

```
            if nums[j] > pivot:  # 注意这里是寻找第k大，所以使用大于号
                nums[i], nums[j] = nums[j], nums[i]
                i += 1
        nums[i], nums[right] = nums[right], nums[i]
        return i

    def quickselect(nums, left, right, k):
        if left == right:
            return nums[left]
        pivot_index = partition(nums, left, right)
        if k == pivot_index:
            return nums[k]
        elif k < pivot_index:
            return quickselect(nums, left, pivot_index - 1, k)
        else:
            return quickselect(nums, pivot_index + 1, right, k)

    def find_kth_largest(nums, k):
        return quickselect(nums, 0, len(nums) - 1, k - 1)
```

素数筛法——埃氏筛&欧拉筛

```
prime = [True] * (n + 1)
primes = []

# 埃氏筛
p = 2
while p * p <= n:
    if prime[p]:
        primes.append(p)
        for i in range(p * p, n + 1, p):
            prime[i] = False
    p += 1

# 欧拉筛
for i in range(2, n + 1):
    if prime[i]:
        primes.append(i)
    for j in primes:
        if i * j > n:
            break
        prime[i * j] = False
        if i % j == 0:
            break
```

Dilworth定理

最少严格/不严格上升子序列的分割数=最长不严格/严格下降子序列长度。

```
import bisect
arr = [9, 4, 10, 5, 1]
arr.reverse()
a = []
for x in arr:
    idx = bisect.bisect(a, x)
    if idx == len(a):
        a.append(x)
    else:
        a[idx] = x
print(len(a))
# 3
```

KMP算法

用于寻找字符串`text`中出现字符串模式`pattern`的位置。

```
def compute_lps(pattern):
    m = len(pattern)
    lps = [0] * m
    length = 0
    for i in range(1, m):
        while length > 0 and pattern[i] != pattern[length]:
            length = lps[length - 1]
        if pattern[i] == pattern[length]:
            length += 1
        lps[i] = length
    return lps

def kmp_search(text, pattern):
    n = len(text)
    m = len(pattern)
    if m == 0:
        return []
    lps = compute_lps(pattern)
    matches = []

    j = 0
    for i in range(n):
        while j > 0 and text[i] != pattern[j]:
            j = lps[j - 1]
        if text[i] == pattern[j]:
            j += 1
        if j == m:
            matches.append(i - j + 1)
            j = lps[j - 1]
    return matches

print(kmp_search('ABABABABCABABABABCABABABABC', 'ABABCABAB'))
# [4, 13]
```

数据结构

栈

中缀表达式转后缀表达式——调度场算法

初始化运算符栈operator和输出栈result为空。对于表达式tokens的每个token： 如果token为数字：将其压入输出栈。 如果token为左括号：将其压入运算符栈。 如果token为右括号：将运算符栈顶元素弹出并压入输出栈，直到遇到左括号为止。 如果token为运算符：将运算符栈顶元素弹出并压入输出栈，直到运算符栈顶元素优先级小于token或遇到左括号为止。将token压入运算符栈。 将运算符栈顶元素弹出并压入输出栈，直到运算符栈为空为止。

```
priority = {'(': 0,
            '+': 1,
            '-': 1,
            '*': 2,
            '/': 2,
            }
# (3)*((3+4)*(2+3.5)/(4+5))
tokens = ['(', '3', ')', '*', '(', '(', '3', '+', '4', ')', '*', '(', '2', '+',
'3.5', ')', '/', '(', '4', '+', '5', ')', ')']
operator = []
result = []
for token in tokens:
    if token in '+-*/':
        while operator and priority[operator[-1]] >= priority[token]:
            result.append(operator.pop())
        operator.append(token)
    elif token == '(':
        operator.append(token)
    elif token == ')':
        while operator[-1] != '(':
            result.append(operator.pop())
        operator.pop()
    else:
        result.append(token)
while operator:
    result.append(operator.pop())
print(*result)
# 3 3 4 + 2 3.5 + * 4 5 + / *
```

后缀表达式求值

```
def eval_rpn(tokens):
    """
    tokens: List[str], 如 ["2","1","+","3","*"] -> 9
    支持 + - * / , 除法向 0 取整 (LeetCode 150)
    """
```

```
    st = []
    for t in tokens:
        if t in {"+", "-", "*", "/"}:
            b = st.pop()
            a = st.pop()
            if t == "+":
                st.append(a + b)
            elif t == "-":
                st.append(a - b)
            elif t == "*":
                st.append(a * b)
            else:  # "/" ,向 0 取整：Python 的 // 对负数是向下取整，所以用 int(a / b)
                st.append(int(a / b))
        else:
            st.append(int(t))
    return st[-1]
```

表达式建树

```
from dataclasses import dataclass
from typing import List, Tuple, Dict, Optional

# =========================
# 0) Node：统一二叉/多叉
# =========================
@dataclass
class Node:
    val: str
    children: List["Node"]

# =========================
# 1) 字符串 -> tokens
#    支持：整数/变量/运算符/括号/逗号/函数名
# =========================
OPS = {"+", "-", "*", "/", "^"}
def tokenize(expr: str) -> List[str]:
    tokens = []
    i, n = 0, len(expr)
    while i < n:
        ch = expr[i]
        if ch.isspace():
            i += 1
        elif ch.isdigit():
            j = i
            while j < n and expr[j].isdigit():
                j += 1
            tokens.append(expr[i:j])
            i = j
        elif ch.isalpha() or ch == "_":
            j = i
```

```
            while j < n and (expr[j].isalnum() or expr[j] == "_"):
                j += 1
            tokens.append(expr[i:j])  # 变量名或函数名
            i = j
        else:
            tokens.append(ch)         # + - * / ^ ( ) ,
            i += 1
    return tokens

# 一元负号识别：把中缀里的 '-' 识别成 'u-'
def mark_unary_minus(tokens: List[str]) -> List[str]:
    res = []
    prev: Optional[str] = None
    for t in tokens:
        if t == "-":
            if prev is None or prev in OPS or prev in {"(", ","}:
                res.append("u-")
            else:
                res.append("-")
        else:
            res.append(t)
        prev = res[-1]
    return res

# =========================
# 2) 中缀 tokens -> 后缀 tokens (Shunting-yard)
#    支持一元负号 u-，支持函数调用 func(...)
# =========================
PREC = {"+": 1, "-": 1, "*": 2, "/": 2, "^": 3, "u-": 4}
RIGHT_ASSOC = {"^", "u-"}  # 右结合：幂、一元负号

def is_operand(tok: str) -> bool:
    return tok not in OPS and tok not in {"(", ")", ",", "u-"} and not
tok.isidentifier() is False

def infix_to_postfix(tokens: List[str]) -> List[str]:
    out: List[str] = []
    op: List[str] = []

    # 用于识别“函数名”：形如 name '('
    def is_func_name(t: str) -> bool:
        return t.isidentifier() and t not in OPS and t != "u-"

    i = 0
    while i < len(tokens):
        t = tokens[i]

        # 操作数：数字或变量
        if t.isdigit() or (t.isidentifier() and (i + 1 >= len(tokens) or tokens[i
+ 1] != "(")):
            out.append(t)

        # 函数名：先压栈，遇到右括号再输出它
        elif is_func_name(t) and i + 1 < len(tokens) and tokens[i + 1] == "(":
```

```
            op.append(t)

        elif t == "(":
            op.append(t)

        elif t == ",":
            # 逗号：弹出直到遇到 '('
            while op and op[-1] != "(":
                out.append(op.pop())

        elif t in OPS or t == "u-":
            while op and (op[-1] in OPS or op[-1] == "u-"):
                top = op[-1]
                if (top in RIGHT_ASSOC and PREC[top] > PREC[t]) or \
                   (top not in RIGHT_ASSOC and PREC[top] >= PREC[t]):
                    out.append(op.pop())
                else:
                    break
            op.append(t)

        elif t == ")":
            while op and op[-1] != "(":
                out.append(op.pop())
            op.pop()  # pop '('

            # 如果 '(' 前面是函数名，它在栈顶：弹出函数名到输出
            if op and op[-1].isidentifier() and op[-1] not in OPS and op[-1] !=
"u-":
                out.append(op.pop())

        else:
            raise ValueError(f"Unknown token: {t}")

        i += 1

    while op:
        out.append(op.pop())
    return out

# =========================
# 3) 后缀 tokens -> 建树
#    同时支持二叉/一元/多叉（通过 arity_map 控制）
#
#    - 二元运算符：+ - * / ^  -> arity=2
#    - 一元负号：u-           -> arity=1
#    - 多叉函数：max/min/...  -> arity由你提供或从 postfix 中的参数计数获得
#
#    这里提供两种方式：
#    A) 固定元数（arity_map 里直接写死）
#    B) 可变元数函数：在 postfix 里把函数写成 "max@3" 表示 3 个参数
#       （推荐：简单、不会歧义）
# =========================
def build_tree_from_postfix(post: List[str], arity_map: Dict[str, int]) -> Node:
    st: List[Node] = []
```

```
    for t in post:
        # 方式B：解析 "name@k"
        if "@" in t:
            name, k_str = t.split("@", 1)
            k = int(k_str)
            kids = [st.pop() for _ in range(k)][::-1]
            st.append(Node(name, kids))
            continue

        if t in arity_map:
            k = arity_map[t]
            kids = [st.pop() for _ in range(k)][::-1]
            st.append(Node(t, kids))
        else:
            st.append(Node(t, []))
    return st[-1]

# 把 infix 的函数调用变成 postfix 时输出了 "max"，
# 但没告诉参数个数。为了让“多叉”也能建树，
# 我们提供一个小工具：给函数名补上 "@k"。
# 需要你提供 func_arity（固定元数函数）或可变元数都能处理。
def annotate_func_arity(post: List[str], func_arity: Dict[str, int]) -> List[str]:
    # 若函数是固定元数（如 pow(a,b)），直接改成 pow@2
    return [f"{t}@{func_arity[t]}" if t in func_arity else t for t in post]

# ========================
# 4) 中缀建树：直接 1->2->3（不单独再写）
# ========================
def build_tree_from_infix(expr: str,
                          func_arity: Optional[Dict[str, int]] = None) -> Node:
    tokens = mark_unary_minus(tokenize(expr))
    post = infix_to_postfix(tokens)

    # 二元/一元元数表
    arity = {"+": 2, "-": 2, "*": 2, "/": 2, "^": 2, "u-": 1}

    # 如果你想让函数是“多叉/固定叉”，用 func_arity 把它们标注成 name@k
    if func_arity:
        post = annotate_func_arity(post, func_arity)

    return build_tree_from_postfix(post, arity_map=arity)

# ========================
# 5) 树 -> 中缀表达式（带括号，保证不歧义）
#    - 二元：(a+b)
#    - 一元：(-x)    (u- 打印成 -)
#    - 多叉函数：max(a,b,c)
# ========================
def tree_to_infix(root: Node) -> str:
    if not root.children:
        return root.val

    if root.val == "u-":
        return f"(-{tree_to_infix(root.children[0])})"
```

```
    if root.val in OPS and len(root.children) == 2:
        a = tree_to_infix(root.children[0])
        b = tree_to_infix(root.children[1])
        return f"({a}{root.val}{b})"

    # 多叉：函数调用风格
    args = ",".join(tree_to_infix(ch) for ch in root.children)
    return f"{root.val}({args})"

# =========================
# 6) 树 -> 后缀表达式 tokens
#    - 二元/一元：后序输出
#    - 多叉函数：先输出所有孩子，再输出 "name@k"（推荐）
# =========================
def tree_to_postfix(root: Node) -> List[str]:
    out: List[str] = []
    def dfs(node: Node):
        for ch in node.children:
            dfs(ch)
        if node.children:
            # 多叉函数输出 name@k；二元/一元也可输出原符号
            if node.val not in OPS and node.val != "u-" and len(node.children) !=
2:
                out.append(f"{node.val}@{len(node.children)}")
            else:
                out.append(node.val)
        else:
            out.append(node.val)
    dfs(root)
    return out

# =========================
# 调用例子
# =========================
if __name__ == "__main__":
    # ---- 例1：纯二叉 + 一元负号（中缀 -> 后缀 -> 树 -> 中缀/后缀）
    expr1 = "-3 + 4*(2-1)"
    tokens1 = mark_unary_minus(tokenize(expr1))
    post1 = infix_to_postfix(tokens1)
    root1 = build_tree_from_postfix(post1, {"+":2,"-":2,"*":2,"/":2,"^":2,"u-":1})
    print(tokens1)
    # ['u-', '3', '+', '4', '*', '(', '2', '-', '1', ')']  （实际 token 会略有差异
取决于表达式）
    print(post1)
    # ['3','u-','4','2','1','-','*','+']
    print(tree_to_infix(root1))
    # ((-3)+(4*(2-1)))
    print(tree_to_postfix(root1))
    # ['3','u-','4','2','1','-','*','+']

    # ---- 例2：多叉函数 max(1,2,3) + 一元负号
    # 这里用固定元数 func_arity 来标注参数个数（例如 max 有3个参数）
    expr2 = "max(1,2,3) + -x"
```

```
    tokens2 = mark_unary_minus(tokenize(expr2))
    post2 = infix_to_postfix(tokens2)
    post2 = annotate_func_arity(post2, {"max": 3})  # 关键：标注为 max@3
    root2 = build_tree_from_postfix(post2, {"+":2,"-":2,"*":2,"/":2,"^":2,"u-":1})
    print(tokens2)
    # ['max','(','1',',','2',',','3',')','+','u-','x']
    print(post2)
    # ['1','2','3','max@3','x','u-','+']
    print(tree_to_infix(root2))
    # (max(1,2,3)+(-x))
    print(tree_to_postfix(root2))
    # ['1','2','3','max@3','x','u-','+']
```

二叉树版本

```
from dataclasses import dataclass

# ========================
# 0) 二叉表达式树结点
# ========================
@dataclass
class Node:
    val: str
    left: "Node" = None
    right: "Node" = None

OPS = {"+", "-", "*", "/", "^"}
PREC = {"+": 1, "-": 1, "*": 2, "/": 2, "^": 3, "u-": 4}
RIGHT_ASSOC = {"^", "u-"}  # 幂、一元负号右结合

# ========================
# 1) 字符串 -> tokens
# ========================
def tokenize(expr: str) -> list:
    tokens = []
    i, n = 0, len(expr)
    while i < n:
        ch = expr[i]
        if ch.isspace():
            i += 1
        elif ch.isdigit():
            j = i
            while j < n and expr[j].isdigit():
                j += 1
            tokens.append(expr[i:j])
            i = j
        elif ch.isalpha() or ch == "_":
            j = i
            while j < n and (expr[j].isalnum() or expr[j] == "_"):
                j += 1
```

```
            tokens.append(expr[i:j])
            i = j
        else:
            tokens.append(ch)  # + - * / ^ ( )
            i += 1
    return tokens

# 一元负号识别：把中缀里的 '-' -> 'u-'
def mark_unary_minus(tokens: list) -> list:
    res = []
    prev = None
    for t in tokens:
        if t == "-":
            if prev is None or prev in OPS or prev == "(":
                res.append("u-")
            else:
                res.append("-")
        else:
            res.append(t)
        prev = res[-1]
    return res

# =========================
# 2) 中缀 tokens -> 后缀 tokens (Shunting-yard)
# =========================
def infix_to_postfix(tokens: list) -> list:
    out, op = [], []
    for t in tokens:
        if t.isdigit() or (t.isidentifier() and t not in OPS):
            out.append(t)
        elif t in OPS or t == "u-":
            while op and (op[-1] in OPS or op[-1] == "u-"):
                top = op[-1]
                if (top in RIGHT_ASSOC and PREC[top] > PREC[t]) or \
                   (top not in RIGHT_ASSOC and PREC[top] >= PREC[t]):
                    out.append(op.pop())
                else:
                    break
            op.append(t)
        elif t == "(":
            op.append(t)
        elif t == ")":
            while op and op[-1] != "(":
                out.append(op.pop())
            op.pop()
        else:
            raise ValueError(f"Unknown token: {t}")

    while op:
        out.append(op.pop())
    return out

# =========================
# 3) 后缀 tokens -> 建树（二叉 + 一元u-）
```

```python
# ========================
def build_tree_from_postfix(post: list) -> Node:
    st = []
    for t in post:
        if t in OPS:
            r = st.pop()
            l = st.pop()
            st.append(Node(t, l, r))
        elif t == "u-":
            x = st.pop()
            st.append(Node("u-", x, None))
        else:
            st.append(Node(t))
    return st[-1]

# ========================
# 4) 前缀 tokens -> 建树（二叉 + 一元u-）
#    规则：从右往左扫
# ========================
def build_tree_from_prefix(pre: list) -> Node:
    st = []
    for t in reversed(pre):
        if t in OPS:
            l = st.pop()
            r = st.pop()
            st.append(Node(t, l, r))
        elif t == "u-":
            x = st.pop()
            st.append(Node("u-", x, None))
        else:
            st.append(Node(t))
    return st[-1]

# 中缀直接建树：tokenize -> unary -> postfix -> build
def build_tree_from_infix(expr: str) -> Node:
    tokens = mark_unary_minus(tokenize(expr))
    post = infix_to_postfix(tokens)
    return build_tree_from_postfix(post)

# ========================
# 5) 树 -> 中缀表达式（括号保证不歧义）
# ========================
def tree_to_infix(root: Node) -> str:
    if root.left is None and root.right is None:
        return root.val
    if root.val == "u-":
        return f"(-{tree_to_infix(root.left)})"
    return f"({tree_to_infix(root.left)}{root.val}{tree_to_infix(root.right)})"

# ========================
# 6) 树 -> 后缀 tokens
# ========================
def tree_to_postfix(root: Node) -> list:
    if root.left is None and root.right is None:
```

```
        return [root.val]
    if root.val == "u-":
        return tree_to_postfix(root.left) + ["u-"]
    return tree_to_postfix(root.left) + tree_to_postfix(root.right) + [root.val]

# ========================
# 7) 树 -> 前缀 tokens
# ========================
def tree_to_prefix(root: Node) -> list:
    if root.left is None and root.right is None:
        return [root.val]
    if root.val == "u-":
        return ["u-"] + tree_to_prefix(root.left)
    return [root.val] + tree_to_prefix(root.left) + tree_to_prefix(root.right)

# ========================
# 调用例子
# ========================
if __name__ == "__main__":
    expr = "-3 + 4*(2-1) - -x^2"

    tokens = mark_unary_minus(tokenize(expr))
    post = infix_to_postfix(tokens)
    root = build_tree_from_postfix(post)

    print("TOKENS:", tokens)
    print("POST  :", post)
    print("INFIX :", tree_to_infix(root))
    print("POST2 :", tree_to_postfix(root))
    print("PRE   :", tree_to_prefix(root))

    # 用前缀再建回去验证
    root2 = build_tree_from_prefix(tree_to_prefix(root))
    print("INFIX2:", tree_to_infix(root2))
```

单调栈

找到某个数组中，每个元素右边第一个比其更大的数的索引。这时使用单调递减栈。

```
def find_next_greater(nums):
    n = len(nums)
    res = [0] * n
    stack = []
    for i in range(n):
        while stack and nums[i] > nums[stack[-1]]:#变成<即为右边第一个更小数的索引
            res[stack.pop()] = i
        stack.append(i)
    while stack:
        res[stack.pop()] = n
    return res
```

```
print(find_next_greater([4, 5, 2, 25]))
# [1, 3, 3, 4]
```

找到某个数组中，每个元素右边第一个比其更大的数的索引。

```
def find_last_greater_on_left(nums):
    n = len(nums)
    res = [-1] * n
    stack = []  # 单调递减栈
    for i in range(n):
        while stack and nums[stack[-1]] <= nums[i]:# 弹出所有小于等于当前元素的栈顶，
改成>=就是左边最后一个小于其的
            stack.pop()
        if stack:
            res[i] = stack[-1]# 如果栈不为空，栈顶就是左边最近大于当前元素的
        stack.append(i)# 将当前索引入栈
    return res
print(find_last_greater_on_left([4, 5, 2, 25]))  # 输出: [-1, -1, 1, -1]
```

另外，对于单调递增栈，每当处理完一个索引，单调栈内的某一个索引所对应的元素，就是该索引到栈中下一个索引在数组中对应所有元素的最小值。对单调递减栈类似。

链表

反转链表

反转链表并返回新的头节点。

```
def reverse_linked_list(head):
    pre, cur = None, head
    while cur is not None:
        cur.next, pre, cur = pre, cur, cur.next
    return pre
```

合并两个升序链表

```
def merge_two_lists(head1, head2):
    dummy = ListNode(0)
    cur = dummy
    while head1 is not None and head2 is not None:
        if head1.val <= head2.val:
            cur.next, head1 = head1, head1.next
        else:
            cur.next, head2 = head2, head2.next
        cur = cur.next
```

```
    cur.next = head2 if head1 is None else head1
    return dummy.next
```

查找链表中间节点

返回链表的中间节点或中间偏左节点。

```
def find_middle_node(head):
    slow = fast = head
    while fast.next and fast.next.next:
        slow = slow.next
        fast = fast.next.next
    return slow
```

树

二叉搜索树的建立

```
def insert(value, node):
    if value == node.val:
        return
    if value < node.val:
        if node.left is None:
            node.left = TreeNode(value)
        else:
            insert(value, node.left)
    else:
        if node.right is None:
            node.right = TreeNode(value)
        else:
            insert(value, node.right)
```

二叉搜索树的验证

第一种方式是验证中序遍历序列是否是严格递增序列。

```
def isBST(root):
    stack, cur = [], float('-inf')
    while stack or root:
        while root:
            stack.append(root)
            root = root.left
        root = stack.pop()
        if root.val <= cur:
            return False
        cur = root.val
```

```
            root = root.right
        return True
```

第二种方式是验证节点值是否在合法范围内。

```
def helper(node, lower=float('-inf'), upper=float('inf')):
    if node is None:
        return True
    value = node.val
    if value <= lower or value >= upper:
        return False
    return helper(node.left, lower, value) and helper(node.right, value, upper)
```

dp生成卡特兰数

```
def catalan_dp(k: int, one_indexed: bool = False) -> int:
    """
    DP 计算第 k 个卡特兰数
    - 默认 0-index：C0=1, C1=1, C2=2, ...
    - one_indexed=True：第1个=1(=C0)...
    时间 O(k^2)，空间 O(k)
    """
    if one_indexed:
        k -= 1
    if k < 0:
        raise ValueError("k must be >= 0 (or >=1 if one_indexed=True).")

    dp = [0] * (k + 1)
    dp[0] = 1
    for n in range(1, k + 1):
        s = 0
        for i in range(n):
            s += dp[i] * dp[n - 1 - i]#如果题目会要求取模（比如 1e9+7），把 s += ...
改成 s = (s + dp[i]*dp[n-1-i]) % MOD 就行。
        dp[n] = s
    return dp[k]

# 示例
for n in range(6):
    print(n, catalan_dp(n))  # 0..5: 1,1,2,5,14,42
```

Huffman编码

目的：用二叉树的叶节点存储字符，并最小化叶节点深度与叶节点权值之积的总和。

思路：弹出堆中最小的两个节点，合并并入堆。循环往复，直至堆中只剩下根节点。

```
class TreeNode:
    # def __init__(self, val, left=None, right=None): ...
    def __lt__(self, other):
        return self.val < other.val
n = 4
heap = [TreeNode(i) for i in [1, 1, 3, 5]]
heapify(heap)
for _ in range(n - 1):
    left_node, right_node = heappop(heap), heappop(heap)
    merged = TreeNode(left_node.val + right_node.val)
    merged.left, merged.right = left_node, right_node
    heappush(heap, merged)
stack = [(heap[0], 0)]
ans = 0
while stack:
    node, depth = stack.pop()
    if node.left is None and node.right is None:
        ans += node.val * depth
        continue
    if node.right is not None:
        stack.append((node.right, depth + 1))
    if node.left is not None:
        stack.append((node.left, depth + 1))
print(ans)
# 17
```

多叉树的表示——长子-兄弟表示法

一个节点的左指针为其第一个子节点，右指针为其下一个兄弟节点。因此，前序遍历序列不变。

二叉树的前、中、后序遍历(永远先左再右，前中后取决于根的位置)

```
def preorder(root):
    res = []
    def dfs(node):
        if not node:
            return
        res.append(node.val)     # 根
        dfs(node.left)           # 左
        dfs(node.right)          # 右
    dfs(root)
    return res
```

根据遍历序列建立二叉树

以根据前中序遍历序列`preorder`和`inorder`建树为例，时间复杂度`O(n)`。

```
n = len(preorder)
preorder_dict = {preorder[i]: i for i in range(n)}
inorder_dict = {inorder[i]: i for i in range(n)}
def build(pre_left, pre_right, in_left, in_right):
    if pre_left > pre_right:
        return None
    value = preorder[pre_left]
    index = inorder_dict[value]
    root = TreeNode(value)
    root.left = build(pre_left + 1, pre_left + index - in_left, in_left, index -
1)
    root.right = build(pre_left + index - in_left + 1, pre_right, index + 1,
in_right)
    return root
```

根据完全二叉树的层序遍历建树

```
from collections import deque
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
def build_tree_levelorder(level):#level: 层序遍历数组，例如 [1,2,3,4,5] 或
[1,2,3,None,4]。返回根节点 TreeNode
    if not level or level[0] is None:
        return None
    root = TreeNode(level[0])
    q = deque([root])
    i = 1
    while q and i < len(level):
        node = q.popleft()
        if i < len(level) and level[i] is not None:
            node.left = TreeNode(level[i])
            q.append(node.left)
        i += 1
        if i < len(level) and level[i] is not None:
            node.right = TreeNode(level[i])
            q.append(node.right)
        i += 1
    return root
```

并查集

注意可能根据题目需求不同，`union()`的实现方式需要调整。如果是编号从1到n，那要记得union的时候都是i-1；取出来的时候i也要变化。

```
class DisjointSet:
    def __init__(self, k):
        self.parents = list(range(k))
        self.rank = [1] * k
    def find(self, x):
        if self.parents[x] != x:
            self.parents[x] = self.find(self.parents[x])
        return self.parents[x]
    def union(self, x, y):
        x_rep, y_rep = self.find(x), self.find(y)
        if x_rep == y_rep:
            return
        if self.rank[x_rep] < self.rank[y_rep]:
            self.parents[x_rep] = y_rep
        elif self.rank[x_rep] > self.rank[y_rep]:
            self.parents[y_rep] = x_rep
        else:
            self.parents[y_rep] = x_rep
            self.rank[x_rep] += 1
d = DisjointSet(n)
ultparents = []
for i in range(n):
    ultparents.append(d.find(i))#获取每个儿子对应的代表元素，存储在list里面
parent_to_son = {}#获取每个代表元素对应的所有儿子，存储在dict里面
for i,ch in enumerate(ultparents):
    if ch not in parent_to_son:
        parent_to_son[ch] = []
    parent_to_son[ch].append(i)
```

堆实现

```
class BinHeap:
    def __init__(self):
        self.heap_list = [-float('inf')]
        self.size = 0
    def perc_up(self, i):
        while i >> 1 > 0:
            if self.heap_list[i] <self.heap_list[i >> 1]:
                self.heap_list[i], self.heap_list[i >> 1] = self.heap_list[i >>
1], self.heap_list[i]
            i >>= 1
    def insert(self, value):
        self.heap_list.append(value)
        self.size += 1
        self.perc_up(self.size)
    def min_child(self, i):
        if i << 1 | 1 > self.size:
            return i << 1
        if self.heap_list[i << 1] < self.heap_list[i << 1 | 1]:
            return i << 1
```

```
        return i << 1 | 1
    def perc_down(self, i):
        while i << 1 <= self.size:
            c = self.min_child(i)
            if self.heap_list[i] > self.heap_list[c]:
                self.heap_list[i], self.heap_list[c] = self.heap_list[c],
self.heap_list[i]
            i = c
    def pop(self):
        ans = self.heap_list[1]
        self.heap_list[1] = self.heap_list[self.size]
        self.heap_list.pop()
        self.size -= 1
        self.perc_down(1)
        return ans
```

中位数查询

```
class MedianQueryQueue:
    def __init__(self):
        self.small = []
        self.large = []
        self.small_size = 0
        self.large_size = 0
        self.small_search = defaultdict(int)
        self.large_search = defaultdict(int)
    def delete(self):
        while self.small and self.small_search[-self.small[0]] == 0:
            heappop(self.small)
        while self.large and self.large_search[self.large[0]] == 0:
            heappop(self.large)
    def balance(self):
        self.delete()
        if self.small_size > self.large_size + 1:
            num = -heappop(self.small)
            heappush(self.large, num)
            self.small_size -= 1
            self.large_size += 1
            self.small_search[num] -= 1
            self.large_search[num] += 1
        if self.small_size < self.large_size:
            num = heappop(self.large)
            heappush(self.small, -num)
            self.small_size += 1
            self.large_size -= 1
            self.small_search[num] += 1
            self.large_search[num] -= 1
        self.delete()
        if self.small and self.large and -self.small[0] > self.large[0]:
            num1, num2 = -heappop(self.small), heappop(self.large)
            heappush(self.large, num1)
```

```
            heappush(self.small, -num2)
            self.small_search[num1] -= 1
            self.small_search[num2] += 1
            self.large_search[num1] += 1
            self.large_search[num2] -= 1
        self.delete()
    def add(self, x):
        heappush(self.small, -x)
        self.small_size += 1
        self.small_search[x] += 1
        self.balance()
    def remove(self, x):
        if x <= -self.small[0]:
            self.small_size -= 1
            self.small_search[x] -= 1
        else:
            self.large_size -= 1
            self.large_search[x] -= 1
        self.balance()
    def query(self):
        if self.small_size == self.large_size:
            tot = self.large[0] - self.small[0]
            if tot % 2 == 1:
                return round(tot / 2, 1)
            else:
                return tot // 2
        else:
            return -self.small[0]
```

字典树

A写法 查询在不在、前缀有没有

```
class Trie:
    __slots__ = ("ch", "end")
    def __init__(self):
        self.ch = [[-1]*26]   # children indices,数字变成*10
        self.end = [False]    # is_end

    def insert(self, word: str) -> None:
        node = 0
        for c in word:
            i = ord(c) - 97#数字变成int(c)
            nxt = self.ch[node][i]
            if nxt == -1:
                nxt = len(self.ch)
                self.ch[node][i] = nxt
                self.ch.append([-1]*26)#数字变成*10
                self.end.append(False)
            node = nxt
        self.end[node] = True
```

```
    def search(self, word: str) -> bool:
        node = 0
        for c in word:
            i = ord(c) - 97#数字变成int(c)
            node = self.ch[node][i]
            if node == -1:
                return False
        return self.end[node]

    def startsWith(self, prefix: str) -> bool:
        node = 0
        for c in prefix:
            i = ord(c) - 97#数字变成int(c)
            node = self.ch[node][i]
            if node == -1:
                return False
        return True

    # 可选：返回prefix对应的节点下标（很多题要用）
    def walk(self, s: str) -> int:
        node = 0
        for c in s:
            i = ord(c) - 97#数字变成int(c)
            node = self.ch[node][i]
            if node == -1:
                return -1
        return node
trie = Trie()
words = ["apple", "app", "bat"]

for w in words:
    trie.insert(w)

print(trie.search("app"))       # True  ("app" 是一个完整单词)
print(trie.search("ap"))        # False ("ap" 不是完整单词)
print(trie.startsWith("ap"))    # True  ("apple"/"app" 都以 ap 开头)
print(trie.startsWith("ba"))    # True
print(trie.startsWith("cat"))   # False
```

B写法 带计数（支持 erase / 统计单词次数 / 统计前缀次数）,某个单词出现了几次、以某个前缀开头的单词一共有多少个（注意是计数不是布尔）

```
class Trie:
    def __init__(self):
        self.ch = [[-1]*26]#数字变成*10
        self.pass_cnt = [0]   # 经过该节点的单词数
        self.end_cnt  = [0]   # 以该节点结尾的单词数

    def insert(self, word: str) -> None:
```

```
        node = 0
        self.pass_cnt[node] += 1
        for c in word:
            i = ord(c) - 97#数字变成int(c)
            if self.ch[node][i] == -1:
                self.ch[node][i] = len(self.ch)
                self.ch.append([-1]*26)#数字变成*10
                self.pass_cnt.append(0)
                self.end_cnt.append(0)
            node = self.ch[node][i]
            self.pass_cnt[node] += 1
        self.end_cnt[node] += 1

    def countWordsEqualTo(self, word: str) -> int:
        node = 0
        for c in word:
            i = ord(c) - 97#数字变成int(c)
            node = self.ch[node][i]
            if node == -1:
                return 0
        return self.end_cnt[node]

    def countWordsStartingWith(self, prefix: str) -> int:
        node = 0
        for c in prefix:
            i = ord(c) - 97#数字变成int(c)
            node = self.ch[node][i]
            if node == -1:
                return 0
        return self.pass_cnt[node]

    def erase(self, word: str) -> None:
        # 假设 word 一定存在（题目通常保证）；不保证就先查 end_cnt
        node = 0
        self.pass_cnt[node] -= 1
        path = [0]  # 记录走过的节点，用于减计数
        for c in word:
            i = ord(c) - 97#数字变成int(c)
            node = self.ch[node][i]
            path.append(node)
            self.pass_cnt[node] -= 1
        self.end_cnt[node] -= 1
        # 机考一般不需要真正删除节点，计数减掉即可

trie = Trie()
trie.insert("apple")
trie.insert("app")
trie.insert("app")   # app 插两次

print(trie.countWordsEqualTo("app"))        # 2
print(trie.countWordsStartingWith("ap"))    # 3  (apple 1 + app 2)
print(trie.countWordsStartingWith("app"))   # 3

trie.erase("app")    # 删除一次 app
```

```
print(trie.countWordsEqualTo("app"))        # 1
print(trie.countWordsStartingWith("ap"))    # 2  (apple 1 + app 1)
```

树形dp

A类：选不选（一个节点可能选可能不选），选择某个节点说明其父节点已经没选，所以无需再考虑其父节点的选择问题。

例：没有上司的宴会

```
import sys
sys.setrecursionlimit(10**7)

def tree_select_or_not(n, edges, val, root=0, one_indexed=False):
    """
    n: 点数
    edges: [(u,v), ...]  无向边
    val[u]: 选中节点u的收益/权值
    root: 任意选个根
    one_indexed: 若输入点编号1..n则True，从0开始则false

    return: 最优值
    """
    g = [[] for _ in range(n)]
    if one_indexed:
        for a, b in edges:
            a -= 1; b -= 1
            g[a].append(b); g[b].append(a)
    else:
        for a, b in edges:
            g[a].append(b); g[b].append(a)

    dp0 = [0] * n
    dp1 = [0] * n

    def dfs(u, p):#p为u的父节点
        dp1[u] = val[u]     # 选u，先拿到val[u]
        dp0[u] = 0          # 不选u，先从0开始加
        for v in g[u]:
            if v == p:
                continue
            dfs(v, u)
            dp1[u] += dp0[v]
            dp0[u] += max(dp0[v], dp1[v])

    dfs(root, -1)
    return max(dp0[root], dp1[root])
n = int(input())
val = [0]*(n+1)
for i in range(n):
```

```
    val[i+1] = int(input())
edges = []#形式为[(1,2),(2,3)], x[0]是父, x[1]是子
has_parents = [False]*(n+1)
for _ in range(n-1):
    s,t = map(int,input().split())
    has_parents[t] = True
    edges.append((s,t))
root = 1
for i in range(1,n+1):
    if not has_parents[i]:
        root = i
        break
val0 = [val[i] for i in range(1,n+1)]#val应该是下标从0开始
ans = tree_select_or_not(n,edges, val0, root,one_indexed=True)
print(ans)
```

图

连通性判断；如果不联通，返回块数

```
from collections import deque

def connected_components(n, edges, one_indexed=False):
    """
    n: 点数
    edges: 边列表 [(u,v), ...]
    one_indexed: 若输入点编号是 1..n, 设为 True

    return:
      comp_id: 长度 n 的数组，第 i 个点属于哪个连通块(0..k-1)
      comps:   连通块点集列表，comps[c] 是第 c 块包含的点（按输入编号风格返回）
    """
    # --- build graph ---
    g = [[] for _ in range(n)]
    if one_indexed:
        for u, v in edges:
            u -= 1; v -= 1
            g[u].append(v); g[v].append(u)
    else:
        for u, v in edges:
            g[u].append(v); g[v].append(u)

    comp_id = [-1] * n
    comps = []

    for s in range(n):
        if comp_id[s] != -1:
            continue
        cid = len(comps)
        q = deque([s])
        comp_id[s] = cid
```

```
            nodes = [s]

            while q:
                u = q.popleft()
                for v in g[u]:
                    if comp_id[v] == -1:
                        comp_id[v] = cid
                        q.append(v)
                        nodes.append(v)

            comps.append(nodes)

        # 若需要按 1..n 返回点编号
        if one_indexed:
            comps = [[x + 1 for x in comp] for comp in comps]

        return comp_id, comps
    def is_connected(n, edges, one_indexed=False):
        """True 表示整图连通（无向图意义下）"""
        _, comps = connected_components(n, edges, one_indexed=one_indexed)
        return len(comps) == 1
    n = 6
    edges = [(0,1), (1,2), (3,4)]  # 0-1-2 一块，3-4 一块，5 单独一块

    comp_id, comps = connected_components(n, edges)
    print(len(comps))   # 3
    print(comps)        # [[0,1,2], [3,4], [5]]
    print(is_connected(n, edges))  # False
    x = 4# 查询某个点属于哪块
    print(comp_id[x])   # 1  (表示属于 comps[1])
```

最短路径

Dijkstra算法：从堆中heappop出来的步长会越来越大。

```
    import sys
    import heapq
    from collections import defaultdict, deque

    input = sys.stdin.readline
    INF = 10**18

    def build_graph(n, edges, directed=True):
        """
        edges: [(u, v, w), ...]
        directed=True 表示有向图；False 表示无向图（自动加反向边）
        """
        g = [[] for _ in range(n + 1)]  # 1..n，若你是 0..n-1 就改成 n
        for u, v, w in edges:
            g[u].append((v, w))
```

```
        if not directed:
            g[v].append((u, w))
    return g

def dijkstra(n, g, s):
    """
    返回:
      dist[i]  : s->i 最短距离（不可达 INF）
      parent[i]: 最短路树的父节点，用于还原路径；s 的 parent[s]=-1
    """
    dist = [INF] * (n + 1)
    parent = [-1] * (n + 1)
    dist[s] = 0
    pq = [(0, s)]  # (dist, node)

    while pq:
        d, u = heapq.heappop(pq)
        if d != dist[u]:
            continue
        for v, w in g[u]:
            nd = d + w
            if nd < dist[v]:
                dist[v] = nd
                parent[v] = u
                heapq.heappush(pq, (nd, v))
    return dist, parent

def restore_path(parent, s, t):#还原 s->t 的路径（节点序列）。不可达则返回 []
    path = []
    cur = t
    while cur != -1:
        path.append(cur)
        if cur == s:
            break
        cur = parent[cur]
    if path[-1] != s:
        return []
    path.reverse()
    return path

n = 5
edges = [
    (1,2,2),(1,3,5),
    (2,3,1),(2,4,2),
    (3,5,3),(4,5,1)
]
g = build_graph(n, edges, directed=True)#False: 无向图

s, t = 1, 5#s是起点，t是终点
dist, parent = dijkstra(n, g, s)#dist是距离，parent是一个父节点列表，parent[i]=j意思是此最短路径中i节点的上一个节点是j

if dist[t] >= INF//2:
    print("IMPOSSIBLE")
```

```
else:
    print(dist[t])   # 输出 5
    print(restore_path(parent,s,t))#输出[1,2,4,5]
```

Bellman-Ford算法：经过v-1次松弛后，若还能松弛则存在负权回路。时间复杂度O(VE)。

```
def bellman_ford(graph, V, source):
    dist = [float('inf')] * V
    dist[source] = 0
    for _ in range(V - 1):
        for u, v, w in graph:
            if dist[u] != float('inf') and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
    for u, v, w in graph:
        if dist[u] != float('inf') and dist[u] + w < dist[v]: # 存在负权回路
            return None
    return dist
```

Floyd-Warshall算法：可以计算得到存储任意两点间的最小距离的dist，时间复杂度O(V^3)，V为顶点数。对于每个节点分别作为中间节点的情况，去看能否减少某两个节点间的距离。初始两节点i, j间若无边则dist[i][j] == inf。

```
def floyd_warshall(graph):
    V = len(graph)
    dist = [row[:] for row in graph]
    for k in range(V):
        for i in range(V):
            for j in range(V):
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
    return dist
INF = 10**9
graph = [
    [0,   5,   INF, 10],
    [INF, 0,   3,   INF],
    [INF, INF, 0,   1],
    [INF, INF, INF, 0],
]
dist = floyd_warshall(graph)
```

拓扑排序——Kahn算法（同等条件下节点从小到大排列）

```
def topological_sort_small_first(graph):
    degree = defaultdict(int)
    nodes = set(graph.keys())
```

```
    for u in graph:
        for v in graph[u]:
            degree[v] += 1
            nodes.add(v)

    heap = []
    for u in nodes:
        if degree[u] == 0:
            heapq.heappush(heap, u)

    result = []
    while heap:
        u = heapq.heappop(heap)   # 编号最小的先出
        result.append(u)
        for v in graph.get(u, []):
            degree[v] -= 1
            if degree[v] == 0:
                heapq.heappush(heap, v)

    return result if len(result) == len(nodes) else None
graph = {
    "A": ["C", "D"],
    "B": ["D"],
    "C": ["E"],
    "D": ["E"],
    "E": []          # 没有后续课也要写出来，保证节点在 graph 里，非常非常重要
}
order = topological_sort_small_first(graph)
print(order)  # 可能输出: ['A', 'B', 'C', 'D', 'E'] （也可能有别的合法顺序）
graph0 = [defaultdict(int) for _ in range(n+1)]
for _ in range(p):
    i,j,w = map(int,input().split())
    graph0[i][j]+=w
graph = {i:list(graph0[i].keys()) for i in range(1,n+1)}
```

最小生成树——Prim算法&Kruskal算法

找到一棵连接所有n个节点的包含n - 1条边的树，它在所有这样的树中权值之和最小。（无向图，邻接表加双向边）

Prim算法：对由0~n - 1标记节点的图graph，选定某一起始节点，不断选择已生成的树通往外部的边中权值最小的一条，将其加入result中。适用于稠密图。

```
from heapq import heappush, heappop
def prim(graph, n, start=0):
    visited = [False] * n
    visited[start] = True
    heap = []
    result = []
    for (to, w) in graph[start]:
        heappush(heap, (w, start, to))
```

```
    while heap and len(result) < n - 1:
        w, u, v = heappop(heap)
        if visited[v]:
            continue
        visited[v] = True
        result.append((w, u, v))
        for (to, w2) in graph[v]:
            if not visited[to]:
                heappush(heap, (w2, v, to))
    if len(result) != n - 1:
        return None  # 表示没有生成树（图不连通）
    return result
n = 5
graph = [[] for _ in range(n)]
def add_undirected(u, v, w):
    graph[u].append((v, w))
    graph[v].append((u, w))
add_undirected(0, 1, 2)
add_undirected(0, 3, 6)
add_undirected(1, 2, 3)
add_undirected(1, 3, 8)
add_undirected(1, 4, 5)
add_undirected(2, 4, 7)
add_undirected(3, 4, 9)
mst = prim(graph, n, start=0)
if mst is None:
    print("图不连通，没有生成树")
else:
    total = sum(w for w, u, v in mst)
    print("MST edges (w, u, v):")
    for e in mst:
        print(e)
    print("Total weight =", total)
```

还可以基于邻接矩阵实现，时间复杂度没有对数因子，适用于稠密图。

```
def prim_matrix(graph, n, start=0):
    inf = float('inf')
    key = [inf] * n
    key[start] = 0
    visited = [False] * n
    parent = [-1] * n
    for _ in range(n):  # 这里用 n，不是 V
        u = -1# 1) 找到未访问点里 key 最小的 u
        min_key = inf
        for v in range(n):
            if not visited[v] and key[v] < min_key:
                min_key = key[v]
                u = v
        if u == -1:  # 剩余点都不可达 => 图不连通
            return None, None
        visited[u] = True
```

```
        for v in range(n):# 2) 用 u 更新其他点的 key
            w = graph[u][v]
            if not visited[v] and w < key[v]:
                key[v] = w
                parent[v] = u
    return sum(key), parent# -------- 构造邻接矩阵 --------
n = 5
inf = float('inf')
graph = [[inf]*n for _ in range(n)]
for i in range(n):
    graph[i][i] = 0
def add_undirected(u, v, w):
    graph[u][v] = min(graph[u][v], w)
    graph[v][u] = min(graph[v][u], w)
add_undirected(0, 1, 2)
add_undirected(0, 3, 6)
add_undirected(1, 2, 3)
add_undirected(1, 3, 8)
add_undirected(1, 4, 5)
add_undirected(2, 4, 7)
add_undirected(3, 4, 9)
total, parent = prim_matrix(graph, n, start=0)
if total is None:
    print("图不连通，没有 MST")
else:
    print("MST total weight =", total)  # 16
    for v in range(n): # 如果想打印最小生成树的各边（除了start之外，每个点v都由
parent[v]连进来）
        if parent[v] != -1:
            print(parent[v], "-", v, "weight", graph[parent[v]][v])
```

Kruskal算法：对所有边`edges`按权值进行排序，遍历每一条边，利用并查集，如果一条边的两个节点尚未在同一个连通分量中，则将该边加入`result`中。适用于稀疏图。要先写并查集。

```
djs = DisjointSet(n)
result = []
for w, u, v in sorted(edges):
    if djs.find(u) != djs.find(v):
        djs.union(u, v)
        result.append((w, u, v))
        n = 5
edges = [(2, 0, 1),(6, 0, 3),(3, 1, 2),(8, 1, 3),(5, 1, 4),(7, 2, 4),(9, 3, 4),]
mst, total = kruskal(n, edges)
if mst is None:
    print("图不连通，没有 MST")
else:
    print("MST total weight =", total)  # 16
    print("MST edges (w, u, v):")
    for e in mst:
        print(e)
```

强连通单元——Kosaraju算法

根据邻接表`graph`得到`sccs: List[List[int]]`，`sccs`的每一个元素都是强连通单元。 先进行一次DFS并记录完成时间，将图转置后按完成时间的逆序再进行DFS。

```
def dfs1(graph, node, visited, stack):
    visited[node] = True
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs1(graph, neighbor, visited, stack)
    stack.append(node)

def dfs2(graph, node, visited, component):
    visited[node] = True
    component.append(node)
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs2(graph, neighbor, visited, component)

def kosaraju(graph):
    stack = []
    visited = [False] * len(graph)
    for node in range(len(graph)):
        if not visited[node]:
            dfs1(graph, node, visited, stack)

    transposed_graph = [[] for _ in range(len(graph))]
    for node in range(len(graph)):
        for neighbor in graph[node]:
            transposed_graph[neighbor].append(node)

    visited = [False] * len(graph)
    sccs = []
    while stack:
        node = stack.pop()
        if not visited[node]:
            scc = []
            dfs2(transposed_graph, node, visited, scc)
            sccs.append(scc)
    return sccs
```

语法&小技巧

字符串&数字&全排列

```
print(ord('A'), ord('a'), chr(65))
# 65 97 A
print('sdfsa'.replace('s', 'e'))
```

```python
# edfea
print('010980511'.isdigit())
# True
print('114514'.lstrip('1'))
# 4514
print([2,3,1].sort(),[2,3,1].sort(reverse = True))
#[1,2,3] [3,2,1]
print(math.log(1000, 10))
# 2.9999999999999996
print(math.isclose(0.1 + 0.2, 0.3))
# True
print('%.5f' % 2 ** 0.5)
# 1.41421
num = 1.145141919810
print(f'{num:.7f}') # 不能随便加空格!
# 1.1451419

from itertools import permutations
a = 'abc'
for i in permutations(a):
    x = ''.join(i)
    print(x, end=' ')
# abc acb bac bca cab cba

c = ('e', 'f', 'g')
for j in permutations(c, 2):
    print(j)
# ('e', 'f')
# ('e', 'g')
# ('f', 'e')
# ('f', 'g')
# ('g', 'e')
# ('g', 'f')
a = [1,2,3]
a.reverse()#只能自己反转
b = a[::-1]#[3,2,1]
print(a.index(3,0))#寻找3的下标，start=0，输出2

print(abs(-7), divmod(10, 3))
# 7 (3, 1)
print(pow(2, 10), pow(2, 10, 1000))
# 1024 24
print(max([3,1,5]), min("cab"))
# 5 a
print(round(2.675, 2), int(-3.9))
# 2.67 -3
print(sum([1,2,3], 10))
# 16
print(7//3, -7//3)
# 2 -3   # // 是向下取整

a = [2,3,1]; a.sort(); print(a)
# [1, 2, 3]
a = [2,3,1]; print(sorted(a), a)
```

```
# [1, 2, 3] [2, 3, 1]
a = [1,2,3]; a.append(4); a.extend([5,6]); print(a)
# [1, 2, 3, 4, 5, 6]
a = [1,2,3]; a.insert(1, 99); print(a)
# [1, 99, 2, 3]
a = [1,2,3,2]; a.remove(2); print(a)
# [1, 3, 2]   # remove删第一个匹配
a = [1,2,3]; print(a.pop(), a)
# 3 [1, 2]
a = [10,20,30,40]; print(a[1:3], a[::-1])
# [20, 30] [40, 30, 20, 10]
a = [0,1,2,3,4]; a[1:4] = [9,9]; print(a)
# [0, 9, 9, 4]
print([i*i for i in range(5)])
# [0, 1, 4, 9, 16]
print(list(zip([1,2],[3,4])), list(enumerate(['a','b'], start=1)))
# [(1, 3), (2, 4)] [(1, 'a'), (2, 'b')]

print(ord('A'), ord('a'), chr(65))
# 65 97 A
s = "  a,b,  c  "; print(s.strip(), s.split(','))
# a,b,  c ['  a', 'b', '  c  ']
print("abcabc".find("bc"), "abcabc".rfind("bc"))
# 1 4
print("abcabc".count("ab"), "abcabc".replace("ab", "X", 1))
# 2 Xcabc
print("-".join(["a","b","c"]))
# a-b-c
print("010980511".isdigit(), "12.3".isdigit())
# True False
print("114514".lstrip("1"), "114514".rstrip("4"))
# 4514 11451
print("Hello".lower(), "hello".upper(), "hello".capitalize())
# hello HELLO Hello
print("abC".swapcase(), "abc".startswith("a"), "abc".endswith("bc"))
# ABc True True

t = (1,2,3); print(t[0], t[-1], len(t))
# 1 3 3
a,b = (10,20); print(a, b)
# 10 20
print((1,2) + (3,), (1,2)*3)
# (1, 2, 3) (1, 2, 1, 2, 1, 2)
print((1,2,3) > (1,2,0), sorted([(2,1),(1,9)]))
# True [(1, 9), (2, 1)]

import heapq
h = [3,1,4]; heapq.heapify(h); print(h)
# [1, 3, 4]
h = []; heapq.heappush(h, 5); heapq.heappush(h, 2); print(h, heapq.heappop(h), h)
# [2, 5] 2 [5]
h = [5,1,7,3]; heapq.heapify(h); print(heapq.nsmallest(2, h), heapq.nlargest(2,
h))
# [1, 3] [7, 5]
```

```
h = [1,3,2]; heapq.heapify(h); print(heapq.heappushpop(h, 0), h)
# 1 [0, 3, 2]   # 先push再pop（更快）
h = [1,3,2]; heapq.heapify(h); print(heapq.heapreplace(h, 0), h)
# 1 [0, 3, 2]   # 先pop再push（要求非空）
h = []; heapq.heappush(h, -5); heapq.heappush(h, -2); print(-heapq.heappop(h))#最
大堆
# 5

from collections import deque

q = deque([1,2,3]); q.append(4); q.appendleft(0); print(q)
# deque([0, 1, 2, 3, 4])
q = deque([1,2,3]); print(q.popleft(), q.pop(), q)
# 1 3 deque([2])
q = deque([1,2,3]); q.rotate(1); print(q)
# deque([3, 1, 2])
q = deque([1,2,3]); q.rotate(-1); print(q)
# deque([2, 3, 1])
q = deque([1,2,3]); q.extend([4,5]); q.extendleft([9,8]); print(q)
# deque([8, 9, 1, 2, 3, 4, 5])  # extendleft 会“逐个左插”，顺序反过来

import bisect
a = [1,2,2,4]; print(bisect.bisect_left(a, 2), bisect.bisect_right(a, 2))
# 1 3   # 左边界/右边界(插在最后一个2后面)
a = [1,3,5]; bisect.insort(a, 4); print(a)
# [1, 3, 4, 5]
a = [1,3,5]; i = bisect.bisect_left(a, 4); print(i, a[:i], a[i:])
# 2 [1, 3] [5]

s = set([1,2,2,3]); print(s)
# {1, 2, 3}
s = {1,2,3}; print(2 in s, 5 in s)
# True False
s = {1,2,3}; s.add(4); s.discard(2); print(s)
# {1, 3, 4}
A, B = {1,2,3}, {3,4}; print(A|B, A&B, A-B, A^B)
# {1, 2, 3, 4} {3} {1, 2} {1, 2, 4}
A = {1,2}; B = {1,2,3}; print(A.issubset(B), B.issuperset(A))
# True True
s = {1,2,3}; x = s.pop(); print(x, s)
# （随机弹出一个元素）剩余两个元素    # pop弹哪个不保证

d = {"a": 1}; d["b"] = 2; print(d["a"], d.get("c", 0))
# 1 0
d = {}; d["x"] = d.get("x", 0) + 1; print(d)
# {'x': 1}
from collections import defaultdict, Counter
dd = defaultdict(int); dd["k"] += 1; print(dd["k"], dd["missing"])
# 1 0   # 不存在的键自动是0
cnt = Counter("abca"); print(cnt["a"], cnt.most_common(2))
# 2 [('a', 2), ('b', 1)]   # 次数相同的次序不重要
d = {"b":2, "a":1}; print(list(d.keys()), list(d.values()), list(d.items()))
# ['b', 'a'] [2, 1] [('b', 2), ('a', 1)]   # 保持插入顺序(3.7+)
d = {"b":2, "a":1}; print(sorted(d.items()), sorted(d.items(), key=lambda x:
```

```
x[1]))
# [('a', 1), ('b', 2)] [('a', 1), ('b', 2)]

d = {"a":1, "b":2}; print(d.pop("a"), d)
# 1 {'b': 2}
d = {"a":1}; print(d.setdefault("a", 9), d.setdefault("b", 9), d)
# 1 9 {'a': 1, 'b': 9}

from collections import defaultdict
dd = defaultdict(int)
for x in [1,2,1,3,2]: dd[x] += 1
print(dict(dd))#计数
# {1: 2, 2: 2, 3: 1}
dd = defaultdict(list)
for k, v in [("a",1), ("b",2), ("a",3)]: dd[k].append(v)
print(dict(dd))#分组收集：key -> list（Group Anagrams / 按类别归类）
# {'a': [1, 3], 'b': [2]}
dd = defaultdict(set)
for k, v in [("a",1), ("a",1), ("a",2)]: dd[k].add(v)
print({k: sorted(v) for k, v in dd.items()})#分组去重：key -> set
# {'a': [1, 2]}
dd = defaultdict(lambda: defaultdict(int))
dd["u"]["v"] += 1; dd["u"]["v"] += 2
print(dd["u"]["v"], dict(dd["u"]))#嵌套字典：二维计数 / 关系表
# 3 {'v': 3}
from collections import deque
dd = defaultdict(deque)
dd[2].append("x"); dd[2].appendleft("y")
print(list(dd[2]))#桶/队列：key -> deque（按层/按频次分桶）
# ['y', 'x']
g = defaultdict(list)
edges = [(1,2), (1,3), (2,4)]
for u, v in edges: g[u].append(v); g[v].append(u)
print(g[1], g[4])#图邻接表：node -> list（树/图建边最顺手）
# [2, 3] [2]
dd = defaultdict(int)
print("x" in dd, dd["x"], "x" in dd)#defaultdict 的“坑点”提醒（机考常见）。如果只是想
“安全读取不创建键”，用 dd.get("x", 0)。
# False 0 True   # 访问 dd["x"] 会把键创建出来
```

正则表达式

^：匹配开始位置。

$：匹配结束位置。

\：匹配特殊字符，有^$()*+?.\[{|。

*：匹配前面的零次或多次。

+：匹配前面的一次或多次。

?：匹配前面的零次或一次。

|：或。

\d/\D：匹配阿拉伯数字/非阿拉伯数字。

\w/\W：匹配字母、数字、下划线/非字母、数字、下划线。

\s/\S：匹配空白/非空白。

[]：匹配范围内的字符之一，如[ace]可匹配a，[m-p]可匹配o。

[^]：匹配非范围内的字符。

()：将其中的内容作为整体。

```
import re
reg = r'^(0|[1-9][0-9]*)$'
s = '26'
print('yes' if re.match(reg, s) else 'no')
# yes
```

题库

有界图的深度优先搜索

```
n,m,l = map(int,input().split())
edges = []
for _ in range(m):
    a,b = map(int,input().split())
    edges.append((a,b))
start = int(input())
def build_graph(n, edges, directed=True):
    g = [[] for _ in range(n)]  # 1..n, 若你是 0..n-1 就改成 n
    for u, v in edges:
        g[u].append(v)
        if not directed:
            g[v].append(u)
    for ch in g:
        ch.sort()
    return g
g = build_graph(n,edges,directed=False)
#print(g)
visited = [False]*n
step = 0
ans = []
def dfs(u):
    global step
    if step>l:
        return
    ans.append(u)
```

```
    #print(ans)
    for v in g[u]:
        if not visited[v] and v not in ans:#把v not in ans放在append那块判断会WA
            visited[v] = True
            step+=1
            dfs(v)
            step-=1
            visited[v] = False
visited[start] = True
dfs(start)
#ans.sort()
print(' '.join([str(x) for x in ans]))
```

括号生成

```
class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
        ans = []
        tmp = []
        def dfs(left,right):
            if left==n and right==n:
                ans.append(''.join(tmp))
                return
            if left == right:
                tmp.append('(')
                dfs(left+1,right)
                tmp.pop()
            elif left>right:
                tmp.append(')')
                dfs(left,right+1)
                tmp.pop()
                if left<n:
                    tmp.append('(')
                    dfs(left+1,right)
                    tmp.pop()
        dfs(0,0)
        return ans
```

带花费的dijkstra算法

```
import sys
import heapq
from collections import defaultdict, deque

input = sys.stdin.readline
INF = 10**18

def build_graph(n, edges, directed=True):
    g = [[] for _ in range(n + 1)]  # 1..n, 若你是 0..n-1 就改成 n
```

```
    for u, v, w, t in edges:
        g[u].append((v, w,t))
        if not directed:
            g[v].append((u, w,t))
    return g

def dijkstra(n, g, s,k):
    dist = [[INF] * (k + 1) for _ in range(n+1)]
    cost = [INF] * (n+1)
    dist[s][0] = 0
    pq = [(0,0, s)]  # (dist,cost, node)
    while pq:
        d,c, u = heapq.heappop(pq)
        if d != dist[u][c]:
            continue
        for v, w,t in g[u]:
            nd = d + w
            nc = c+t

            if nc<=k and nd < dist[v][nc]:
                dist[v][nc] = nd
                #print((nd,nc,v))
                heapq.heappush(pq, (nd,nc, v))
                #print("heapq:",pq)
    return dist
k = int(input())
n = int(input())
r = int(input())
edges = []
cost = []
for _ in range(r):
    s,d,l,t = map(int,input().split())
    edges.append((s,d,l,t))
g = build_graph(n,edges,directed=True)
s,t = 1,n
dist= dijkstra(n,g,s,k)
if min(dist[n]) < INF//2:
    print(min(dist[n]))
else:
    print(-1)
```

哈密顿激活层（dfs+剪枝）

```
from collections import deque

n, m, k, b = map(int, input().split())

must_pos = {}      # (r,c) -> t
must_time = {}     # t -> (r,c)
for _ in range(k):
    r, c, t = map(int, input().split())
    must_pos[(r, c)] = t
```

```
    must_time[t] = (r, c)

blocked = set()
for _ in range(b):
    x, y = map(int, input().split())
    blocked.add((x, y))

def isin(x, y):
    return 1 <= x <= n and 1 <= y <= m

move = [(-1,0),(1,0),(0,1),(0,-1)]
total = n * m - b

# 起点必须是 t=1 的那个
if 1 not in must_time:
    print(-1)
    exit()

sx, sy = must_time[1]
if (sx, sy) in blocked:
    print(-1)
    exit()

visited = [[False]*(m+1) for _ in range(n+1)]
path = [(sx, sy)]
visited[sx][sy] = True
flag = False

def time_feasible(x, y, step):
    """未来强制点的距离/奇偶性剪枝 + 过去强制点是否错过"""
    for (r, c), t in must_pos.items():
        if t < step:
            if not visited[r][c]:
                return False
        elif t == step:
            if (r, c) != (x, y):
                return False
        else:
            if visited[r][c]:
                continue
            d = abs(r - x) + abs(c - y)
            rem = t - step
            if d > rem:
                return False
            if (rem - d) & 1:   # 奇偶性不匹配
                return False
    return True

def connected_prune(x, y, step):
    """连通性剪枝：从当前位置出发能否到达所有剩余未访问点"""
    need = total - step + 1  # 剩余点数(包含当前)
    q = deque([(x, y)])
    seen = set([(x, y)])
    cnt = 1
```

```
    while q:
        ux, uy = q.popleft()
        for dx, dy in move:
            vx, vy = ux+dx, uy+dy
            if not isin(vx, vy):
                continue
            if (vx, vy) in blocked:
                continue
            if visited[vx][vy] and (vx, vy) != (x, y):
                continue
            if (vx, vy) in seen:
                continue
            seen.add((vx, vy))
            cnt += 1
            q.append((vx, vy))
    return cnt == need

def onward_degree(px, py, curx, cury):
    """Warnsdorff: 优先走“后续选择更少”的格子，减少分支"""
    deg = 0
    for dx, dy in move:
        nx, ny = px+dx, py+dy
        if isin(nx, ny) and (nx, ny) not in blocked and (not visited[nx][ny] or
(nx, ny) == (curx, cury)):
            deg += 1
    return deg

def dfs(x, y, step):
    global flag
    if flag:
        return True

    # 当前步必须满足时序约束
    if step in must_time and must_time[step] != (x, y):
        return False
    if not time_feasible(x, y, step):
        return False
    if not connected_prune(x, y, step):
        return False

    if step == total:
        for r, c in path:
            print(r, c)
        flag = True
        return True

    # 如果下一步有强制点，那就只能走向它
    if (step + 1) in must_time:
        nx, ny = must_time[step + 1]
        if abs(nx - x) + abs(ny - y) != 1:
            return False
        if visited[nx][ny] or (nx, ny) in blocked:
            return False
        visited[nx][ny] = True
```

```
            path.append((nx, ny))
            if dfs(nx, ny, step + 1):
                return True
            path.pop()
            visited[nx][ny] = False
            return False

        # 否则正常尝试四方向（按后续度排序）
        cand = []
        for dx, dy in move:
            nx, ny = x+dx, y+dy
            if not isin(nx, ny):
                continue
            if (nx, ny) in blocked:
                continue
            if visited[nx][ny]:
                continue
            # 若该格是强制点，但时间不对，则不能走
            if (nx, ny) in must_pos and must_pos[(nx, ny)] != step + 1:
                continue
            cand.append((nx, ny))

        cand.sort(key=lambda p: onward_degree(p[0], p[1], x, y))

        for nx, ny in cand:
            visited[nx][ny] = True
            path.append((nx, ny))
            if dfs(nx, ny, step + 1):
                return True
            path.pop()
            visited[nx][ny] = False

        return False

    dfs(sx, sy, 1)
    if not flag:
        print(-1)
```

括号嵌套树 题干： 可以用括号嵌套的方式来表示一棵树。表示方法如下：

1. 如果一棵树只有一个结点，则该树就用一个大写字母表示，代表其根结点。
2. 如果一棵树有子树，则用"树根(子树1,子树2,...,子树n)"的形式表示。树根是一个大写字母,子树之间用逗号隔开，没有空格。子树都是用括号嵌套法表示的树。

给出一棵不超过26个结点的树的括号嵌套表示形式，请输出其前序遍历序列和后序遍历序列。

输入样例代表的树如下图：

输入 一行，一棵树的括号嵌套表示形式 输出 两行。第一行是树的前序遍历序列，第二行是树的后序遍历序列 样例输入 A(B(E),C(F,G),D(H(I))) 样例输出 ABECFGDHI EBFGCIHDA

```
import sys
sys.setrecursionlimit(10**7)

from dataclasses import dataclass
from typing import List, Tuple

@dataclass
class Node:
    val: str
    children: List["Node"]
def parse_tree(s: str, i: int = 0) -> Tuple[Node, int]:
    # 读根（一个大写字母）
    node = Node(s[i], [])
    i += 1
    # 如果后面有孩子列表 "(...)"，解析之
    if i < len(s) and s[i] == '(':
        i += 1  # skip '('
        while True:
            child, i = parse_tree(s, i)
            node.children.append(child)

            if s[i] == ',':
                i += 1  # skip ','
                continue
            elif s[i] == ')':
                i += 1  # skip ')'
                break
    return node, i
def preorder(root: Node, out: List[str]):
    out.append(root.val)
    for ch in root.children:
        preorder(ch, out)
def postorder(root: Node, out: List[str]):
    for ch in root.children:
        postorder(ch, out)
    out.append(root.val)
def main():
    s = sys.stdin.readline().strip()
    root, idx = parse_tree(s, 0)
    pre, post = [], []
    preorder(root, pre)
    postorder(root, post)
    print("".join(pre))
    print("".join(post))
if __name__ == "__main__":
    main()
```