2025Spring CheatSheet

常用库

```
#堆
import heapq
#队列(default字典)
from collections import deque(defaultdict)
#递归上限
from sys import setrecursionlimit
#缓存
from functools import lru_cache
#数学***math库***：最常用的sqrt,对数log(x[,base])、三角sin()、反三角asin()也都有；还有e,pi等常数，inf表示无穷大；返回小于等于x的最大整数floor（）,大于等于ceil（）,判断两个浮点数是否接近isclose（a，b，*，rel_tol=1e-09，abs_tol=0.0）；一般的取幂pow（x，y）,阶乘factorial（x）如果不符合会ValueError,组合数comb（n，k）`math.radians()`将度数转换为弧度，或者使用`math.degrees()`将弧度转换为度数。
import math
math.gcd(a,b)
def catalan_number(n):
return math.comb(2 * n, n) // (n + 1)#卡特兰数，即合法出栈序列总数
#二分库
import bisect
bisect.bisect_right(a,6)#返回在a列表中若要插入6的index（有重复数字会插在右边）
bisect.insort(a,6)#返回插入6后的列表a
#conuter
from collections import Counter
```

`map(function, *iterables)`:将函数应用于传入的每个可迭代对象的各个元素.e.g.

```
squared = list(map(lambda x: x**2, [1, 2, 3, 4]))  # [1, 4, 9, 16]
```

debug

OOP：

`__eq__(self, other)`	`==`	判断相等
`__ne__(self, other)`	`!=`	判断不相等
`__lt__(self, other)`	`<`	判断是否小于
`__le__(self, other)`	`<=`	判断是否小于等于
`__gt__(self, other)`	`>`	判断是否大于
`__ge__(self, other)`	`>=`	判断是否大于等于

方法名	用途说明
`__init__`	构造函数，创建对象时自动调用
`__del__`	析构函数，对象删除前调用
`__str__`	控制 `print(obj)` 时的输出

方法名	用途说明
`__repr__`	控制对象在解释器中的表现
`__len__`	支持 `len(obj)`
`__getitem__`	支持 `obj[key]`
`__setitem__`	支持 `obj[key] = value`
`__iter__`	使对象可迭代（如用于 for 循环）
`__next__`	支持迭代器的下一个元素
`__call__`	使对象可以像函数一样调用
`__enter__` / `__exit__`	用于上下文管理器（with 语句）
`__eq__`, `__lt__`, `__gt__`, `__ne__`, `__le__`, `__ge__`	比较运算
`__add__`, `__sub__`, `__mul__`, 等等	支持算术运算符重载

欧拉筛（每添加一个数，都要借助质数筛去合数）

```
# 胡睿诚 23数院
N=20
primes = []
is_prime = [True]*N
is_prime[0] = False;is_prime[1] = False
for i in range(2,N):
    if is_prime[i]:
        primes.append(i)
    for p in primes: #筛掉每个数的素数倍
        if p*i >= N:
            break
        is_prime[p*i] = False
        if i % p == 0: #这样能保证每个数都被它的最小素因数筛掉！
            break
print(primes)
# [2, 3, 5, 7, 11, 13, 17, 19]
```

Kadane**算法**

最大连续子序列之和

```
def max_subarray_sum(nums):

max_sum = current_sum = nums[0]

for num in nums[1:]:

    current_sum = max(num, current_sum + num)#是否舍弃前缀和

    max_sum = max(max_sum, current_sum)
```

```
return max_sum

nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]

print(max_subarray_sum(nums)) # 输出: 6
```

Manacher algrithom

```
    ns='#'.join(s)  #防止偶数
    n = len(ns)
    dp = [0] * n
    mid = 0
    right = 0
    center = 0
    max_r = 0
    for i in range(n):
        if i < right:
            mirror = 2 * mid - i
            dp[i] = min(dp[mirror], right - i)#镜像对称
        lt = i - dp[i]
        rt = i + dp[i]
        while 0 <= lt - 1 <= rt + 1 < n and ns[lt - 1] == ns[rt + 1]:
            lt -= 1
            rt += 1
            dp[i] += 1
        if dp[i] + i > right:
            right = rt
            mid = i
        if dp[i] > max_r:
            max_r = dp[i]
            center = i
    return ns[center - max_r : center + max_r + 1].replace("#", "")
```

分治算法（求排列的逆序数）

```
def merge_sort(num):#num表示一个数组，我们把它一分为二，再分别排序
    if len(num)==1:
        return num
    mid=len(num)//2
    left_num=num[:mid]
    right_num=num[mid:]
    left_num=merge_sort(left_num)
    right_num=merge_sort(right_num)
    return merged(left_num,right_num,res)
def merged(lst1,lst2,ans):#我们把先前分开的再合拢，同时计算逆序数(子数组内部变换顺序不会影响它们与外界的逆序数)
    l,r=0,0
    merged_lst=[]
    while l<len(lst1) and r<len(lst2):
        if lst1[l]<lst2[r]:
            merged_lst.append(lst1[l])
            l+=1
        else:
```

```
            merged_lst.append(lst2[r])
            r+=1
            ans[0]+=len(lst1)-l
    merged_lst.extend(lst1[l:])
    merged_lst.extend(lst2[r:])
    return merged_lst
```

集合运算

```
union_set = set1 | set2 #并集
intersection_set = set1 & set2 #交集
difference_set = set1 - set2 #差集
symmetric_difference_set = set1 ^ set2 #对称差（各自独有元素的集合）
is_subset = set1.issubset(set2) #bool，判断1是否为2的子集
are_disjoint = set1.isdisjoint(set2) #bool，判断交集是否为空
```

itertools包（排列组合等）

```
import itertools
my_list = ['a', 'b', 'c']
permutation_list1 = list(itertools.permutations(my_list))
permutation_list2 = list(itertools.permutations(my_list, 2))
combination_list = list(itertools.combinations(my_list, 2))
bit_combinations = list(itertools.product([0, 1], repeat=4))

print(permutation_list1)

# [('a', 'b', 'c'), ('a', 'c', 'b'), ('b', 'a', 'c'), ('b', 'c', 'a'), ('c', 'a',
'b'), ('c', 'b', 'a')]

print(permutation_list2)

# [('a', 'b'), ('a', 'c'), ('b', 'a'), ('b', 'c'), ('c', 'a'), ('c', 'b')]

print(combination_list)

# [('a', 'b'), ('a', 'c'), ('b', 'c')]

print(bit_combinations)

# [(0, 0, 0, 0), (0, 0, 0, 1), (0, 0, 1, 0), (0, 0, 1, 1), (0, 1, 0, 0), (0, 1,
0, 1), (0, 1, 1, 0), (0, 1, 1, 1), (1, 0, 0, 0), (1, 0, 0, 1), (1, 0, 1, 0), (1,
0, 1, 1), (1, 1, 0, 0), (1, 1, 0, 1), (1, 1, 1, 0), (1, 1, 1, 1)]
    import itertools
    ans=[[]]
    n=len(nums)
    for i in range(1,n+1):
        ans+=list(itertools.combinations(nums,i))
    return ans
```

括号匹配问题

Shunting Yard算法（逆波兰表达式求值，中序表达式转后序表达式）

由于后序表达式中数字间的相对位置并没有改变，因此唯一需要处理的就是不同运算符之间的相对顺序和插入位置。先创建字典定义运算符之间的优先级，在处理当前运算符时与栈顶的运算符优先级进行比较，由于栈顶元素意味着先出栈，代表更高优先级，从而可以据此判断当前运算符入栈前是否需要将栈顶运算符出栈。括号内部是一个独立的式子，因此将左括号优先级定为0，而遇到右括号则全部出栈。

```
precedence={'(':0,'+':1,'-':1,'*':2,'/':2}
def infix_to_postfix(x):
    stack=list();output=list()
    for k in range(len(x)):
        if x[k]== '(':
            stack.append(x[k])
        elif x[k] in '+-*/':
            while stack and precedence[x[k]]<=precedence[stack[-1]]:
                output.append(stack.pop())
            stack.append(x[k])
        elif x[k]== ')':
            while stack[-1]!='(':
                output.append(stack.pop())
            stack.pop()
        else:
            output.append(x[k])
    while stack:
        output.append(stack.pop())
    return output
```

number buffer技巧

核心思想

1. **累积数字字符**：遇到数字字符时，将其添加到缓冲区（字符串变量）
2. **非数字触发输出**：遇到运算符或括号时，输出缓冲区内容并清空
3. **边界处理**：表达式结束时检查并输出缓冲区剩余内容

链表

反转链表

```
class Solution:
    def reverseList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        p=head;pre=None
        while p:
            nt=p.next
            p.next,pre,p=pre,p,nt
        return pre
```

有序链表合并

```
class Solution:
    def mergeTwoLists(self, list1: Optional[ListNode], list2: Optional[ListNode])
-> Optional[ListNode]:
        dummy=ListNode()
        p=dummy
        p1=list1;p2=list2
        while p1 and p2:
            if p1.val>p2.val:
                p1,p2=p2,p1
            p.next,p,p1=p1,p1,p1.next
        p.next=p1 if p1 else p2
        return dummy.next
```

二叉树

翻转二叉树

```
class Solution:
    def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        if not root:
            return root
        root.left=self.invertTree(root.left)
        root.right=self.invertTree(root.right)
        root.left,root.right=root.right,root.left
        return root
```

对称二叉树

```
class Solution:
    def isSymmetric(self, root: Optional[TreeNode]) -> bool:
        if not root:
            return True
        def isMirror(left: TreeNode, right: TreeNode) -> bool:
            if not left and not right:
                return True
            if not left or not right:
                return False
            return (left.val == right.val) and isMirror(left.left, right.right)
and isMirror(left.right, right.left)

        return isMirror(root.left, root.right)
```

建树

二叉

```
for _ in range(t):
    n,m=map(int,input().split())
    tran=dict()
    tran[-1]=None
    nodes=set();leaves=set()
    for _ in range(n):
        lst=list(map(int,input().split()))
        for i in lst:
            if i not in tran:
                tran[i]=Tree(i)
        cur,lt,rt=lst
        nodes=nodes|{cur,lt,rt};leaves=leaves|{lt,rt}
        tran[cur].left=tran[lt]
        tran[cur].right=tran[rt]
        if lt!=-1:
            tran[lt].parent=tran[cur]
        if rt!=-1:
            tran[rt].parent=tran[cur]
    root=tran[list(nodes-leaves)[0]]
```

列表（这里类似于用图来表示树）

```
s=input()
adjacency_list=defaultdict(list)
stack=[]
dummy='0'
cur=dummy;stack.append(dummy)
for i in s:
    if i=='(':
        cur=stack[-1]
    elif i==')' or i==',':
        stack.pop()
        cur=stack[-1]
    else:
        adjacency_list[cur].append(i)
        stack.append(i)

root=adjacency_list[dummy][0]
```

前中序

```
def build_tree(lst1,lst2):
    if not lst1:
        return None
    root=Tree(lst1[0])
    idx=lst2.index(lst1[0])
    root.left=build_tree(lst1[1:idx+1],lst2[:idx])
    root.right=build_tree(lst1[idx+1:],lst2[idx+1:])
    return root
```

列表式

给定一棵大小**为 n**的树，以数组 parent[0..n-1] 的形式表示，其中 parent[] 中的每个索引 i 代表一个节点，

而 i 处的值表示该节点的**直接**父**节点。**对于根节点，其值为 **-1。**

括号式(栈)

```
    cnt=[];trees=[TreeNode(x[0])]
    for i in range(1,n):
        if x[i].isupper() or x[i]=='*':
            trees.append(TreeNode(x[i]))
        elif x[i]=='(':
            cnt.append(trees[-1])
        elif x[i]==')':
            cnt.pop()
        if not cnt:
            break
        cur=cnt[-1]
        if x[i].isupper() or x[i]=='*':
           if not cur.left:
               cur.left=trees[-1]
           else:
               cur.right=trees[-1]
    root=trees[0]
```

Huffman编码树

根据字符使用频率(权值)生成一棵唯一的哈夫曼编码树。生成树时需要遵循以下规则以确保唯一性：

选取最小的两个节点合并时，节点比大小的规则是:

1. 权值小的节点算小。权值相同的两个节点，字符集里最小字符小的，算小。

例如 ({'c','k'},12) 和 ({'b','z'},12)，后者小。

2.合并两个节点时，小的节点必须作为左子节点

每次合并权值最小的两个节点

```
class Node:
    def __init__(self, weight, char=None):
        self.weight = weight
        self.char = char
        self.left = None
        self.right = None

    def __lt__(self, other):
        if self.weight == other.weight:
            return self.char < other.char
        return self.weight < other.weight
```

```
def build_huffman_tree(characters):
    heap = []
    for char, weight in characters.items():
        heapq.heappush(heap, Node(weight, char))

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        #merged = Node(left.weight + right.weight) #note: 合并后，char 字段默认值是空
        merged = Node(left.weight + right.weight, min(left.char, right.char))
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)

    return heap[0]
```

字典树（Tire）

```
class Node:
    def __init__(self):
        self.son={}
        self.end=False

class Trie:
def __init__(self):
    self.root=Node()

def insert(self, word: str) -> None:
    cur=self.root
    for c in word:
        if c not in cur.son:
            cur.son[c]=Node()
        cur=cur.son[c]
    cur.end=True

def search(self, word: str) -> bool:
    cur=self.root
    for c in word:
        if c not in cur.son:
            return False
        cur=cur.son[c]
    return True if cur.end else False

def startsWith(self, prefix: str) -> bool:
    cur=self.root
    for c in prefix:
        if c not in cur.son:
            return False
        cur=cur.son[c]
    return True
```

树状dp

```
from collections import defaultdict

n=int(input())
tree=list(map(int,input().split()))
tree=[0]+tree
dp=[[0,tree[i]] for i in range(1,n+1)]
children=defaultdict(list)
for i in range(2,n+1):
    children[i//2].append(i)
def dfs(num):
    for child in children[num]:
        dfs(child)
        dp[num-1][0]+=max(dp[child-1][0],dp[child-1][1])
        dp[num-1][1]+=dp[child-1][0]
dfs(1)
print(max(dp[0][0],dp[0][1]))
```

并查

```
def find_parents(x):
    if parents[x]!=x:
        parents[x]=find_parents(parents[x])
    return parents[x]
def union_parents(x,y):
    parents_x=find_parents(x)
    parents_y=find_parents(y)
    parents[parents_y]=parents_x
```

图

桶+BFS

一个桶中所有元素互为邻居

```
buckets=defaultdict(list)
for _ in range(n):
    word=input()
    for i in range(4):
        bucket=f'{word[:i]}_{word[i+1:]}'
        buckets[bucket].append(word)

src,dst=input().split()
stack=deque()
stack.append([src])
visited=set();visited.add(src)
while stack:
    paths=stack.popleft()
    word=paths[-1]
    if word==dst:
```

```
            print(*paths)
            exit()
        for i in range(4):
            bucket=f'{word[:i]}_{word[i+1:]}'
            for neighbor in buckets[bucket]:
                if neighbor in visited:
                    continue
                visited.add(neighbor)
                stack.append(paths+[neighbor])
```

最短路径算法

Dijkstra算法（带有限制）

```
class Solution:
    def findCheapestPrice(self, n: int, flights: List[List[int]], src: int, dst:
int, k: int) -> int:
        graph=defaultdict(list)
        for fro,to,price in flights:
            graph[fro].append((to,price))
        heap=[(0,0,src)]
        INF=float('inf')
        visited=[INF]*n
        while heap:
            cost,time,start=heappop(heap)
            if start==dst:
                return cost
            if time>k or time>=visited[start]:
                continue
            visited[start]=time
            for end,price in graph[start]:
                heappush(heap,(cost+price,time+1,end))
        return -1
```

Bellman—Ford算法（检测负权环）

```
def bellman_ford(graph, V, source):
    # 初始化距离
    dist = [float('inf')] * V
    dist[source] = 0

    # 松弛 V-1 次（代表路径长度不超过V-1的情况下到达节点的最短路径）
    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

    # 检测负权环(若有负权环，路径权重可以被无限缩小)
    for u, v, w in graph:
        if dist[u] != float('inf') and dist[u] + w < dist[v]:
            print("图中存在负权环")
            return None

    return dist
```

```
--------------------------------------------------------------------------------
-------
        # 初始化：到各城最便宜费用
        INF = float('inf')
        dist = [INF] * n
        dist[src] = 0

        # 最多允许 K 次中转 -> 最多使用 K+1 条边
        for _ in range(K + 1):
            # 基于上一轮的结果创建新一轮的 dist
            prev = dist[:]

            # 对每条航班边做松弛
            for u, v, w in flights:
                # 若 u 可达，则尝试用 u -> v 这条边更新 v
                if prev[u] + w < dist[v]:
                    dist[v] = prev[u] + w

            # 下一轮松弛时，依然要基于本轮更新后的 dist，
            # 因此不需要再额外复制

        return dist[dst] if dist[dst] != INF else -1
```

SPFA算法

SPFA算法的基本思想如下：

1. 初始化源节点的最短距离为0，其他节点的最短距离为正无穷大。
2. 将源节点加入队列中，并标记为已访问。
3. 循环执行以下步骤直到队列为空：
    - 从队列中取出一个节点作为当前节点。
    - 遍历当前节点的所有邻接节点：
        - 如果经过当前节点到达该邻接节点的路径比当前记录的最短路径更短，则更新最短路径，并将该邻接节点加入队列中。
4. 当队列为空时，算法结束，所有节点的最短路径已计算出来。

SPFA算法在实际应用中通常表现出良好的性能，尤其适用于稀疏图（边数相对较少）和存在负权边的情况。然而，需要注意的是，如果图中存在负权环路，SPFA算法将无法给出正确的结果。

Floyd-Warshall算法（多源节点）

算法的基本思想是通过一个二维数组来存储任意两个顶点之间的最短距离。初始时，这个数组包含图中各个顶点之间的直接边的权重，对于不直接相连的顶点，权重为无穷大。然后，通过迭代更新这个数组，逐步求得所有顶点之间的最短路径。

🔹 **Floyd-Warshall 算法原理（多源）**

- **思想**：动态规划 + 三重循环
- 状态定义：`dist[i][j]` 表示 i 到 j 的最短路径长度
- 转移方程：

  `dist[i][j]=min(dist[i][j], dist[i][k]+dist[k][j])`

表示是否通过中间点 k 能让路径更短

- 最终得出任意两点之间的最短路径

```
def floyd_warshall(graph):
    V = len(graph)
    dist = [row[:] for row in graph]  # 深拷贝初始图矩阵

    for k in range(V):        # 中间点
        for i in range(V):    # 起点
            for j in range(V):  # 终点
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist
```

最小生成树算法（MST）

Prim算法

- **初始化**：从第一个点开始，将其所有邻接边加入优先队列（最小堆）。
- **贪心选择**：每次从堆中取出距离最小的边，如果该边连接的节点未被访问过，则将其加入最小生成树，并累加距离。
- **更新堆**：将新加入节点的所有未访问邻接边加入堆中，直到所有节点都被访问或堆为空。

```
        INF = float('inf')
        heap = [(0, 0)]
        visited = [INF] * n
        visited[0] = 0
        flag = [False] * n
        while heap:
            distance, pre = heappop(heap)
            flag[pre] = True
            for neighbor in neighbors[pre]:
                if flag[neighbor[-1]]:
                    continue
                new_distance = neighbor[0]
                if new_distance < visited[neighbor[-1]]:
                    visited[neighbor[-1]] = new_distance
                    heappush(heap, (new_distance, neighbor[-1]))
```

Kruskal算法

1. 将图中的所有边按照权重从小到大进行排序。
2. 初始化一个空的边集，用于存储最小生成树的边。
3. 重复以下步骤，直到边集中的边数等于顶点数减一或者所有边都已经考虑完毕：
    - 选择排序后的边集中权重最小的边。
    - 如果选择的边不会导致形成环路（即加入该边后，两个顶点不在同一个连通分量中），则将该边加入最小生成树的边集中。
4. 返回最小生成树的边集作为结果。

```
class UnionFind:
```

```
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if self.rank[px] > self.rank[py]:
            self.parent[py] = px
        else:
            self.parent[px] = py
            if self.rank[px] == self.rank[py]:
                self.rank[py] += 1

def kruskal(n, edges):
    uf = UnionFind(n)
    edges.sort(key=lambda x: x[2])
    res = 0
    for u, v, w in edges:
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            res += w
    if len(set(uf.find(i) for i in range(n))) > 1:
        return -1
    return res

n, m = map(int, input().split())
edges = []
for _ in range(m):
    u, v, w = map(int, input().split())
    edges.append((u, v, w))
print(kruskal(n, edges))
```

两者均使用贪心算法，kruskal 算法的时间复杂度主要来源于对边进行排序，因此其时间复杂度是O(ElogE)，其中E为图的边数。显然 kruskal 适合顶点数较多、边数较少的情况，这和 prim算法恰好相反。于是可以根据题目所给的数据范围来选择合适的算法，即**如果是稠密图(边多)，则用 prim 算法;如果是稀疏图(边少)，则用 kruskal 算法。**

拓扑排序

```
def has_cycle_topo_sort(graph):
    indegree = defaultdict(int)
    for u in graph:
        for v in graph[u]:
            indegree[v] += 1
queue = deque([node for node in graph if indegree[node] == 0])
visited_count = 0
```

```
while queue:
    node = queue.popleft()
    visited_count += 1
    for neighbor in graph[node]:
        indegree[neighbor] -= 1
        if indegree[neighbor] == 0:
            queue.append(neighbor)

return visited_count != len(graph)
```

强连通分量算法（强连通分量是指在有向图中，存在一条路径可以从任意一个顶点到达另一个顶点的一组顶点。）

Kosaraju算法

```
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):
    # Step 1: Perform first DFS to get finishing times
    stack = []
    visited = [False] * len(graph)
    for node in range(len(graph)):
        if not visited[node]:
            dfs1(graph, node, visited, stack)

    # Step 2: Transpose the graph
    transposed_graph = [[] for _ in range(len(graph))]
    for node in range(len(graph)):
        for neighbor in graph[node]:
            transposed_graph[neighbor].append(node)

    # Step 3: Perform second DFS on the transposed graph to find SCCs
    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
```

Tarjan算法

```
# 遍历当前节点的所有邻居
        for neighbor in graph[node]:
            if indices[neighbor] == 0:  # 如果邻居未被访问过
                dfs(neighbor)  # 递归进行DFS
                # 回溯时更新当前节点的low_link值（从子节点继承）
                low_link[node] = min(low_link[node], low_link[neighbor])
            elif on_stack[neighbor]:  # 如果邻居已经被访问且还在栈中（即属于当前SCC路径）
                # 更新当前节点的low_link为邻居的index（回边或横叉边）
                low_link[node] = min(low_link[node], indices[neighbor])

        # 如果当前节点的index等于low_link，说明发现了一个SCC
        if indices[node] == low_link[node]:
            scc = []
            while True:
                top = stack.pop()        # 弹出栈顶元素
                on_stack[top] = False    # 标记不在栈中
                scc.append(top)          # 加入当前SCC集合
                if top == node:          # 直到弹出当前节点为止
                    break
            sccs.append(scc)             # 将找到的SCC加入结果列表
```

KMP算法

Knuth-Morris-Pratt（KMP）算法是**一种用于在文本字符串中查找单词的计算机科学算法。**该算法从左到右依次比较字符。

当出现字符不匹配时，算法会使用一个预处理表（称为“前缀表”）来跳过不必要的字符比较。

```
"""
compute_lps 函数用于计算模式字符串的LPS表。LPS表是一个数组，
其中的每个元素表示模式字符串中当前位置之前的子串的最长前缀后缀的长度。
该函数使用了两个指针 length 和 i，从模式字符串的第二个字符开始遍历。
"""
def compute_lps(pattern):
    """
    计算pattern字符串的最长前缀后缀（Longest Proper Prefix which is also Suffix）表
    :param pattern: 模式字符串
    :return: lps表
    """
    m = len(pattern)
    lps = [0] * m  # 初始化lps数组
    length = 0  # 当前最长前后缀长度
    for i in range(1, m):  # 注意i从1开始，lps[0]永远是0
        while length > 0 and pattern[i] != pattern[length]:
            length = lps[length - 1]  # 回退到上一个有效前后缀长度
        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 0
    lps = compute_lps(pattern)
    matches = []

    # 在 text 中查找 pattern
    j = 0  # 模式串指针
    for i in range(n):  # 主串指针
        while j > 0 and text[i] != pattern[j]:
            j = lps[j - 1]  # 模式串回退
        if text[i] == pattern[j]:
            j += 1
        if j == m:
            matches.append(i - j + 1)  # 匹配成功
            j = lps[j - 1]  # 查找下一个匹配，如果这一行改为j=0，则不会有重叠部分

    return matches


text = "ABABABABCABABABABCABABABABC"
pattern = "ABABCABAB"
index = kmp_search(text, pattern)
print("pos matched: ", index)
# pos matched:  [4, 13]
```

逃生指南

1. 除法是否使用地板除得到整数？（否则 4/2=2.0）

2. 是否有缩进错误?

3. 用于调试的print是否删去?

4. 非一般情况的边界情况是否考虑？（参考取模,序列中连续相等,0）

5. 递归中return的位置是否准确？（缩进问题,逻辑问题）

6. 贪心是否最优？有无更优解?

7. 正难则反（参考 #蒋子轩 23工院# 乌鸦坐飞机）

8. 审题是否准确？ 是否漏掉了输出？（参考简单的整数划分）

9.注意字符串输入的整体性，可以选择用逗号，空格分割单位（参考文字排版）

10.PE：空格；RE：break 打乱输入;TLE:while陷入无限循环。while过程中，一定要注意参数变化

```
from sys import setrecurisonlimit
setrecursionlimit(10000)#python 默认 200
```

11.dp是否注意了循环内外层，有没有预留0条件，数据会不会需要预处理

12.搜索，有没有注意bfs中q-1，有没有加入visited辅助剪枝

13.矩阵行列不要搞错了

14.浅拷贝与深拷贝

15.常量不变