正则-regular expression

.:任意字符
^:开头位置
$:结尾位置
*:0/多个
+:1/多次
{n}:恰好出现n次
\d:匹配一个数字
\w:匹配字母、数字、下划线
?:匹配0/1次
|:前后二选一
邮箱格式验证

```
import re
while True:
    try:
        s=input()
        sag=r'^[\w-]+(\.[\w-]+)*@[\w-]+(\.[\w-]+)+$'
        print('YES' if re.match(sag,s) else 'NO')
    except EOFError:
        break
```

有效数字

```
import re
s=input()
pattern=r'^[+-]?(\d+(\.\d*)?|\.\d+)([eE][+-]?\d+)?$'
ans=re.match(pattern,s)
if ans:
    print('YES')
else:
    print('NO')
```

二分查找-binary search

总结：
第一个>=x : bisect.bisect_left(a,x)
第一个>x:bisect.bisect_right(a,x)
最后一个=x:bisect.bisect_right(a,x)-1

插入x并保持有序:bisect.insort(a,x)
判断元素是否存在

```
i=bisect.bisect_left(a,x)
exists=(i<len(a) and a[i]==x)
```

区间统计公式
a中属于[L,R]的个数

```
cnt=bisect.bisect_right(a,R)-bisect.bisect_left(a,L)
```

python源码

```
left=0
right=len(a)
while left<right:
    mid=(left+right)//2
    if ...:
        right=mid
    else:
        left=mid+1
```

经典二分查找
在排序数组中查找元素的第一个和最后一个位置

```
import bisect
class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        if not nums:
            return [-1,-1]
        i1=bisect.bisect_left(nums,target)
        i2=bisect.bisect_right(nums,target)-1
        if i1<len(nums) and i2<len(nums) and nums[i1]==target and
nums[i2]==target:
            return [i1,i2]
        else:
            return [-1,-1]
```

答案空间二分
二分True和False的状态

```python
class Solution:
    def shipWithinDays(self, weights: List[int], days: int) -> int:
        def check(n):
            cur=0
            day_used=1
            for w in weights:
                if cur+w<=n:
                    cur+=w
                else:
                    day_used+=1
                    cur=w
            return day_used<=days
        l,r=max(weights),sum(weights)
        while l<r:
            mid=(l+r)//2
            if check(mid):
                r=mid
            else:
                l=mid+1
        return l
```

并查集-dsu/union-find

1.普通并查集
省份数量（最经典的展现形式）

```python
class Solution:
    def findCircleNum(self, isConnected: List[List[int]]) -> int:
        n=len(isConnected)
        parent=list(range(n))
        def find(x):
            if parent[x]!=x:
                parent[x]=find(parent[x])
            return parent[x]
        def union(x,y):
            rx=find(x)
            ry=find(y)
            if rx!=ry:
                parent[ry]=rx
        for i in range(n):
            for j in range(n):
                if isConnected[i][j]==1:
                    union(i,j)
        root=set(find(i) for i in range(n))
        return len(root)
```

2.分团合并
行列总结到一个字典里，一团石头里只留一个

```
class Solution:
    def removeStones(self, stones: List[List[int]]) -> int:
        parent={}
        def find(x):
            if parent[x]!=x:
                parent[x]=find(parent[x])
            return parent[x]
        def union(x,y):
            rx=find(x)
            ry=find(y)
            if ry!=rx:
                parent[ry]=rx
        offset=10000
        for r,c in stones:
            if r not in parent:
                parent[r]=r
            if c+offset not in parent:
                parent[c+offset]=c+offset
            union(r,c+offset)
        root=set(find(x) for x in parent)
        return len(stones)-len(root)
```

3.带权重的并查集
除法求值
把权重和父节点都进行存储

```
class Solution:
    def calcEquation(self, equations: List[List[str]], values: List[float],
queries: List[List[str]]) -> List[float]:
        parent={}
        weight={}
        def find(x):
            if parent[x]!=x:
                root,w=find(parent[x])
                parent[x]=root
                weight[x]*=w
            return parent[x],weight[x]
        def union(x,y,value):
            if x not in parent:
                parent[x]=x
                weight[x]=1.0
            if y not in parent:
                parent[y]=y
                weight[y]=1.0
            rx,wx=find(x)
            ry,wy=find(y)
            parent[ry]=rx
            weight[ry]=(value*wx)/wy
        for i in range(len(equations)):
            a,b=equations[i]
            v=values[i]
            union(a,b,v)
        res=[]
        for j in range(len(queries)):
            aa,bb=queries[j]
            if aa not in parent or bb not in parent:
                res.append(-1)
                continue
            rrx,wwx=find(aa)
            rry,wwy=find(bb)
            if rrx!=rry:
                res.append(-1)
            else:
                num=weight[bb]/weight[aa]
                res.append(num)
        return res
```

4.最小生成树-minimum spanning tree

连接所有点的最小费用

```
class Solution:
    def minCostConnectPoints(self, points: List[List[int]]) -> int:
        n=len(points)
        is_mst=[False]*n
        dist=[float('inf')]*n
        res=0
        dist[0]=0
        for _ in range(n):
            u=-1
            min_d=float('inf')
            for i in range(n):#寻找列表中的最小值，并以此为新的起点
                if not is_mst[i] and dist[i]<min_d:
                    min_d=dist[i]
                    u=i
            is_mst[u]=True
            res+=min_d
            for v in range(n):#计算剩余点到新起点的距离，并更新相应距离位置的最小值
                if not is_mst[v]:
                    d=abs(points[v][0]-points[u][0])+abs(points[v][1]-
points[u][1])
                    if d<dist[v]:
                        dist[v]=d
        return res
```

并查集优化

1.按大小合并

```
def find(x):
    if parent[x]!=x:
        parent[x]=find(parent[x])
    return parent[x]
def union(x,y):
    rx,ry=find(x),find(y)
    if rx!=ry:
        parent[ry]=rx
        size[rx]+=size[ry]
n,m=map(int,input().split())
parent=list(range(n+1))
size=[1]*(n+1)
for _ in range(m):
    a,b=map(int,input().split())
    union(a,b)
classes=[size[x] for x in range(1,n+1) if parent[x]==x]
print(len(classes))
print(' '.join(map(str,sorted(classes,reverse=True))))
```

2.按秩合并-关键是比较rank

```
class DSU:
    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])  # 路径压缩
        return self.parent[x]
    def union(self, x, y):
        rx = self.find(x)
        ry = self.find(y)
        if rx == ry:
            return
        if self.rank[rx] < self.rank[ry]:
            self.parent[rx] = ry
        elif self.rank[rx] > self.rank[ry]:
            self.parent[ry] = rx
        else:
            self.parent[ry] = rx
            self.rank[rx] += 1
```

堆-heap（优先队列-priority queue）#

常见用法：
建堆：heapq.heapify(nums) nums为列表
入堆/出堆：heapq.heappush(heap,x)
x=heap.heappop(heap)
组合：
heapq.heappushpop(heap,x)
heapq.heapreplace(heap,x)
1.Top-K最值

```
import heapq
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        heap=[]
        for x in nums:
            if len(heap)<k:
                heapq.heappush(heap,x)
            elif x>heap[0]:
                heapq.heapreplace(heap,x)
        return heap[0]
```

2.存数组
记录频率和数值，按频率入堆排序

```
import heapq
class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        freq={}
        for n in nums:
            freq[n]=freq.get(n,0)+1
        heap=[]
        for n,f in freq.items():
            if len(heap)<k:
                heapq.heappush(heap,(f,n))
            elif f>heap[0][0]:
                heapq.heapreplace(heap,(f,n))
        return [n for f,n in heap]
```

3.连木棍

```
import heapq
class Solution:
    def connectSticks(self, sticks: List[int]) -> int:
        sticks=[]
        if len(sticks)<=1:
            return 0
        total_cost=0
        while len(sticks)>1:
            first=heapq.heappop(sticks)
            second=heapq.heappop(sticks)
            cost=first+second
            total_cost+=cost
            heapq.heappush(heap,cost)
        return total_cost
```

字典树-Trie

经典实现

```
class Solution:
    def replaceWords(self, dictionary: List[str], sentence: str) -> str:
        trie=[]
        trie.append([{},False])
        def insert(word):
            node=0
            for ch in word:
                if ch not in trie[node][0]:
                    trie[node][0][ch]=len(trie)
                    trie.append([{},False])
                node=trie[node][0][ch]
            trie[node][1]=True
        def change_words(word):
            node=0
            res=[]
            for ch in word:
                if trie[node][1]:
                    return ''.join(res)
                if ch not in trie[node][0]:
                    return word
                res.append(ch)
                node=trie[node][0][ch]
            return ''.join(res) if trie[node][1] else word
        for word_ in dictionary:
            insert(word_)
        sen=[]
        for s in sentence.split():
            result=''.join(change_words(s))
            sen.append(result)
        return ' '.join(sen)
```

Trie+dp

连接词

```
class Solution:
    def findAllConcatenatedWordsInADict(self, words: List[str]) -> List[str]:
        trie=[[{},False]]
        def insert(word):
            node=0
            for ch in word:
                if ch not in trie[node][0]:
                    trie[node][0][ch]=len(trie)
                    trie.append([{},False])
                node=trie[node][0][ch]
            trie[node][1]=True
        res=[]
        def find_word(word):
            n=len(word)
            dp=[False]*(n+1)
            dp[0]=True
            for i in range(n):
                if not dp[i]:
                    continue
                j=i
                node=0
                while j<n and word[j] in trie[node][0]:
                    node=trie[node][0][word[j]]
                    j+=1
                    if trie[node][1]:
                        dp[j]=True
            return dp[n]
        words.sort(key=len)
        for w in words:
            if w and find_word(w):
                res.append(w)
            insert(w)
        return res
```

Trie+dfs

```
def new_code():
    return {'end':False,'next':{}}
def insert(root,word):
    node=root
    for ch in word:
        if ch not in node['next']:
            node['next'][ch]=new_code()
        node=node['next'][ch]
    node['end']=True
def dfs(node):
    res=1
    for ch in node['next'].values():
        res*=dfs(ch)
    if node['end']:
        return 1+res
    else:
        return res
n=int(input())
root=new_code()
for i in range(n):
    insert(root,input())
print(dfs(root))
```

单调栈 -monotonic stack#

1.下一个更大/更小元素

```
class Solution:
    def nextGreaterElement(self, nums1: List[int], nums2: List[int]) -> 
List[int]:
        next_greater={}
        stack=[]
        for num in nums2:
            while stack and num>stack[-1]:
                next_greater[stack.pop()]=num
            stack.append(num)
        while stack:
            next_greater[stack.pop()]=-1
        return [next_greater[num] for num in nums1]
```

2.区间贡献类
用单调栈去找一个数周围符合条件的左右端点
子数组最小值之和（对于每个数算其左边第一个严格小于它的位置和右边严格小于等于它的位置）
右边边界点的算法很巧妙

```
class Solution:
    def sumSubarrayMins(self, arr: List[int]) -> int:
        n=len(arr)
        left=[-1]*n
        right=[n]*n
        MOD=10**9+7
        stack=[]
        ans=0
        for i in range(n):
            while stack and arr[stack[-1]]>=arr[i]:
                stack.pop()
            left[i]=stack[-1] if stack else -1
            stack.append(i)
        stack=[]
        for i in range(n):
            while stack and arr[stack[-1]]>=arr[i]:
                idx=stack.pop()
                right[idx]=i
            stack.append(i)
        for i in range(n):
            l=i-left[i]
            r=right[i]-i
            ans=(ans+arr[i]*l*r)%MOD
        return ans
```

3.环形数组
2n循环

```
class Solution:
    def nextGreaterElements(self, nums: List[int]) -> List[int]:
        n=len(nums)
        res=[-1]*n
        stack=[]
```

```python
        for i in range(2*n):
            idx=i%n
            while stack and nums[idx]>nums[stack[-1]]:
                index=stack.pop()
                res[index]=nums[idx]
            if i<n:
                stack.append(i)
        return res
```

4.最大宽度坡

建单调递减栈，再返回遍历

```python
class Solution:
    def maxWidthRamp(self, nums: List[int]) -> int:
        n=len(nums)
        stack=[]
        for i in range(n):
            if not stack or nums[i]<nums[stack[-1]]:
                stack.append(i)
        ans=[]
        for j in range(n-1,-1,-1):
            while stack and nums[j]>=nums[stack[-1]]:
                i=stack.pop()
                minus=j-i
                ans.append(minus)
        return max(ans)
```

5.存储跨度

```python
prices=list(map(int,input().split()))
n=len(prices)
stack=[]
res=[1]*n
for i in range(n):
    span=1
    while stack and prices[stack[-1][0]]<=prices[i]:
        span+=stack.pop()[1]
    res[i]=span
    stack.append((i,span))
print(res)
```

6.接雨水

凹槽式计数

```
class Solution:
    def trap(self, height: List[int]) -> int:
        n=len(height)
        stack=[]
        water=0
        for i in range(n):
            while stack and height[i]>height[stack[-1]]:
                mid=stack.pop()
                if not stack:
                    break
                left=stack[-1]
                h=min(height[left],height[i])-height[mid]
                w=i-left-1
                water+=h*w
            stack.append(i)
        return water
```

动态规划-dp

1.递推写法
数字三角形-从下往上算

```
n=int(input())
tri=[]
for _ in range(n):
    tri.append([int(i) for i in input().split()])
dp=[[0 for _ in range(n)] for _ in range(n)]
for j in range(n):
    dp[n-1][j]=tri[n-1][j]
for i in range(n-2,-1,-1):
    for j in range(i+1):
        dp[i][j]=max(dp[i+1][j],dp[i+1][j+1])+tri[i][j]
print(dp[0][0])
```

2.最大连续子序列和，Kadane算法
考虑方案

```
n=int(input())
nums=list(map(int,input().split()))
dp=[0]*n
start=[0]*n
dp[0]=nums[0]
for i in range(1,n):
    if dp[i-1]>=0:
        dp[i]=dp[i-1]+nums[i]
        start[i]=start[i-1]
    else:
        dp[i]=nums[i]
        start[i]=i
max_num=max(dp)
pos=dp.index(max_num)
print(max_num,start[pos]+1,pos+1)
```

Kadane算法-找出连续子数组的最大值

```
def kadane(arr):
    max_current=max_global=arr[0]
    for i in range(1,len(arr)):
        max_current=max(arr[i],max_current+arr[i])
        max_global=max(max_global,max_current)
    return max_global
```

3.最长上升子序列
核心部分

```
for i in range(1,len(arr)):
    for j in range(i):
        if nums[i]>nums[j]:#算最长非递增子序列换成<=即可
            dp[i]=max(dp[i],dp[j]+1)
```

4.背包问题
1.0-1背包
未压缩的二维dp

```
N,B=map(int,input().split())
p=list(map(int,input().split()))
w=list(map(int,input().split()))
dp=[[0 for _ in range(B+1)] for _ in range(N+1)]
for i in range(1,N+1):
    for j in range(1,B+1):
        if j>=w[i-1]:
            dp[i][j]=max(dp[i-1][j],p[i-1]+dp[i-1][j-w[i-1]])
        else:
            dp[i][j]=dp[i-1][j]
print(dp[N][B])
```

滚动数组优化

```
for i in range(n):
    for j in range(B,w[i]-1,-1):#倒着查，不然处理dp[j]时，dp[j-w[i]]会受本轮循环中之前的处理影响
        dp[i]=max(dp[i],dp[i-w[i]]+p[i])
print(dp[-1])
```

最长回文子串
中心扩展法

```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        lon=len(s)
        start,end=0,0
        def expand(left,right):
            while left>=0 and right<lon and s[left]==s[right]:
                left-=1
                right+=1
            return right-left-1
        for i in range(lon):
            len1=expand(i,i)
            len2=expand(i,i+1)
            max_len=max(len1,len2)
            if max_len>end-start:
                start=i-(max_len-1)//2
                end=i+max_len//2
        return(s[start:end+1])
```

Manacher算法-by ChatGPT

```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        # 1. 预处理
        t = "^#" + "#".join(s) + "#$"
        n = len(t)
        P = [0] * n
        # 当前最右回文的中心和右边界
        C = 0
        R = 0
        # 2. Manacher 主过程
        for i in range(1, n - 1):
            mirror = 2 * C - i
            if i < R:
                P[i] = min(R - i, P[mirror])
            # 尝试扩展
            while t[i + P[i] + 1] == t[i - P[i] - 1]:
                P[i] += 1
            # 更新最右回文
            if i + P[i] > R:
                C = i
                R = i + P[i]
        # 3. 找到最大回文
        max_len = 0
        center = 0
        for i in range(1, n - 1):
            if P[i] > max_len:
                max_len = P[i]
                center = i
        # 4. 映射回原串
        start = (center - max_len) // 2
        return s[start:start + max_len]
```

2.完全背包
——循环计算

```
n, a, b, c = map(int, input().split())
dp = [0]+[float('-inf')]*n
for i in range(1, n+1):
    for j in (a, b, c):
        if i >= j:
            dp[i] = max(dp[i-j] + 1, dp[i])
print(dp[n])
```

3.多重背包（数量有上限）
法1：二进制拆分+0-1背包（易超时）

```
while True:
    n,m=map(int,input().split())
    if n==0 and m==0:
        break
    input_total=list(map(int,input().split()))
    A=input_total[0:n]
    C=input_total[n:]
    items=[]
    for a,c in zip(A,C):
        k=1
        while k<=c:
            items.append(k*a)
            c-=k
            k<<=1
        if c>0:
            items.append(c*a)
    dp=[False]*(m+1)
    dp[0]=True
    for w in items:
        for j in range(m+1,w-1,-1):
            if dp[j-w]:
                dp[j]=True
    print(sum(dp[1:]))
```

法2：模分组+滑动窗口

```
while True:
    n,m=map(int,input().split())
    if n==0 and m==0:
        break
    input_total=list(map(int,input().split()))
    A=input_total[0:n]
    C=input_total[n:]
    dp=[False]*(m+1)
    dp[0]=True
    for a,c in zip(A,C):
        ndp=dp[:]
        for mod in range(a):
            used=0
            j=mod
            while j<=m:
                if dp[j]:
                    used=0
                elif j>=a and used<c and ndp[j-a]:
                    ndp[j]=True
                    used+=1
                else:
                    used=c+1
                j+=a
        dp=ndp
    print(sum(dp[1:]))
```

但都AC不了，AC代码如下，太难了

```
import math

while True:
    n, m = map(int, input().split())
    if n == 0 and m == 0:
        break
    ls = list(map(int, input().split()))
    w = (1 << (m + 1)) - 1  # e.g., m=10, w=2047
    result = 1
    for i in range(n):
        number = ls[i + n] + 1  # e.g., number = 10
        limit = int(math.log(number, 2))  # limit = 3
        rest = number - (1 << limit)  # rest = 3
        # 处理 2 的幂次方部分
        for j in range(limit):
            shift = ls[i] * (1 << j)
            result = (result | (result << shift)) & w
        # 处理剩余部分
        if rest > 0:
            result = (result | (result << (ls[i] * rest))) & w
    # print(sum_2(result) - 1)
    print(bin(result).count('1') - 1)
```

4.恰好型即最优解

```
T,n=map(int,input().split())
dp=[0]+[-1]*T
for i in range(n):
    t,w=map(int,input().split())
    for j in range(T,t-1,-1):
        if dp[j-t]!=-1:
            dp[j]=max(dp[j],dp[j-t]+w)
print(dp[T] if dp[T]!=0 else -1)
```

5.最长公共子串
用字母来做dp

```
while True:
    try:
        a,b=input().split()
    except EOFError:
        break
    alen=len(a)
    blen=len(b)
    dp=[[0 for _ in range(blen+1)] for _ in range(alen+1)]
    for i in range(1,alen+1):
        for j in range(1,blen+1):
            if a[i-1]==b[j-1]:
                dp[i][j]=max(dp[i][j-1],dp[i-1][j-1]+1)
            else:
                dp[i][j]=max(dp[i-1][j],dp[i][j-1])
    print(dp[alen][blen])
```

6.多个dp数组
状态间互相影响
红蓝玫瑰

```
r=list(input())
n=len(r)
R=[0]*n
B=[0]*n
if r[0]=="R":
    R[0]=0
    B[0]=1
else:
    R[0]=1
    B[0]=0
for i in range(n-1):
    if r[i+1]=="B":
        B[i+1]=B[i]
        R[i+1]=min(R[i],B[i])+1
    else:
        R[i+1]=R[i]
        B[i+1]=min(R[i],B[i])+1
print(R[-1])
```

使序列递增的最小交换数
keep[i]：第i个不动
swap[i]：第i个交换
然后考虑nums数组的大小情况

```
class Solution:
    def minSwap(self, nums1: List[int], nums2: List[int]) -> int:
        n=len(nums1)
        keep=[float('inf')]*n
        swap=[float('inf')]*n
        keep[0]=0
        swap[0]=1
        for i in range(n-1):
            if nums1[i]<nums1[i+1] and nums2[i]<nums2[i+1]:
                keep[i+1]=keep[i]
                swap[i+1]=swap[i]+1
            if nums1[i]<nums2[i+1] and nums2[i]<nums1[i+1]:
                keep[i+1]=min(keep[i+1],swap[i])
                swap[i+1]=min(keep[i]+1,swap[i+1])
        return min(keep[-1],swap[-1])
```

递归-recursion

1.汉诺塔
经典汉诺塔

```
n,col_1,col_2,col_3=input().split()
def find_Hanoi(num,col_a,col_b,col_c):
    if num==1:
        print(f"{1}{':'}{col_a}{'->'}{col_c}")
    else:
        find_Hanoi(num-1,col_a,col_c,col_b)
        print(f"{num}{':'}{col_a}{'->'}{col_c}")
        find_Hanoi(num-1,col_b,col_a,col_c)
find_Hanoi(int(n),col_1,col_2,col_3)
```

加柱汉诺塔
f(n)=min(1<=k<n) (2*f(k)+d(n-k))
2.全排列（双集合的使用+考虑如何覆盖完整整个集合）
全排列

```
class Solution():
    def permute(self, nums):
        l=len(nums)
        ans,sol=[],[]
        def backtrack():
            if len(ans)==l:
                sol.append(ans[:])
                return
            for x in nums:
                if x not in ans:
                    ans.append(x)
                    backtrack()
                    ans.pop()
        backtrack()
        return sol
```

子集

```python
class Solution():
    def permute(self, nums):
        l=len(nums)
        ans,sol=[],[]
        def backtrack():
            if len(ans)==l:
                sol.append(ans[:])
                return
            for x in nums:
                if x not in ans:
                    ans.append(x)
                    backtrack()
                    ans.pop()
        backtrack()
        return sol
```

分割回文串

```python
from functools import lru_cache
class Solution:
    def partition(self, s: str) -> List[List[str]]:
        n=len(s)
        res,sol=[],[]
        @lru_cache(None)
        def ispal(i,j):
            return i>=j or (s[i]==s[j] and ispal(i+1,j-1))
        def find_huiwen(start):
            if start==n:
                res.append(sol[:])
                return
            for i in range(start,n):
                if ispal(start,i):
                    sol.append(s[start:i+1])
                    find_huiwen(i+1)
                    sol.pop()
        find_huiwen(0)
        return res
```

3.八皇后

学习其中对角线如何统计占据情况，其实就是比较复杂的深搜题，主要要考虑回溯

```python
def solve_eight_queens():
    solutions=[]
    def backtrack(row,col_used,diagonal_1,diagonal_2,path):
        if row==8:
            solutions.append(''.join(map(str,path)))
            return
        for i in range(8):
            if not col_used[i] and not diagonal_1[i+row] and not 
diagonal_2[7+row-i]:
                col_used[i]=True
                diagonal_1[i+row]=True
                diagonal_2[7+row-i]=True
                path.append(i+1)
                backtrack(row+1,col_used,diagonal_1,diagonal_2,path)
                path.pop()
                diagonal_1[i+row]=False
                col_used[i]=False
                diagonal_2[7+row-i]=False
    col_used=[False]*8
    diagonal_1=[False]*15
    diagonal_2=[False]*15
    backtrack(0,col_used,diagonal_1,diagonal_2,[])
    solutions.sort()
    return solutions
positions=solve_eight_queens()
p=int(input())
for _ in range(p):
    i=int(input())
    print(positions[i-1])
```

深度优先搜索-dfs

nonlocal：函数内变量
global：全局变量

```
x = 1
def outer():
    x = 2
    def inner():
        global x
        x = 3
    inner()
    print(x)
outer()
print(x)
```

输出2,3

```python
x = 1
def outer():
    x = 2
    def inner():
        nonlocal x
        x = 3
    inner()
    print(x)
outer()
print(x)
```

输出3,1
path要深拷贝，因为后续有回溯
要对dfs进行剪枝，防止走回原来的位置

```python
n,m=map(int,input().split())
matrix=[]
for _ in range(n):
    matrix.append(list(map(int,input().split())))
max_path=[]
max_sum=float('-inf')
dir=[(0,1),(0,-1),(-1,0),(1,0)]
def dfs(x,y,path,current):
    global max_sum,max_path
    if x==n-1 and y==m-1:
        if current>max_sum:
            max_sum=current
            max_path=path[:]
        return
    for dx,dy in dir:
        nx,ny=x+dx,y+dy
        if 0<=nx<n and 0<=ny<m and matrix[nx][ny]!='#':
            p=matrix[nx][ny]
            matrix[nx][ny]='#'
            path.append((nx+1,ny+1))
            current+=p
            dfs(nx,ny,path,current)
            matrix[nx][ny]=p
            current-=p
            path.pop()
l=matrix[0][0]
matrix[0][0]='#'
dfs(0,0,[(1,1)],l)
```

```
for a,b in max_path:
    print(a,b)
```

广度优先搜索-bfs

第一次到达某个状态
经典模版

```
from collections import deque
def bfs():
    q=deque()
    q.append((sx,sy))
    visited[sx][sy]=True
    while q:
        x,y=q.popleft()
        for dx,dy in dirs:
            nx,ny=x+dx,y+dy
            if 合法 and not visited[nx][ny]:
                visited[nx][ny]=True
                q.append((nx,ny))
```

1.多源bfs
为什么从0扩散?
bfs中的起点只能是距离为0的点

```
from collections import deque
class Solution:
    def updateMatrix(self, mat: List[List[int]]) -> List[List[int]]:
        n,m=len(mat),len(mat[0])
        res=[[float('inf') for _ in range(m)] for _ in range(n)]
        pos=deque()
        for i in range(n):
            for j in range(m):
                if mat[i][j]==0:
                    res[i][j]=0
                    pos.append((i,j))
        dir=[(0,1),(0,-1),(1,0),(-1,0)]
        while pos:
            x,y=pos.popleft()
            for dx,dy in dir:
                nx=x+dx
                ny=y+dy
                if 0<=nx<n and 0<=ny<m:
                    if res[nx][ny]>res[x][y]+1:
                        res[nx][ny]=res[x][y]+1
                        pos.append((nx,ny))
        return res
```

2.带步数的bfs
存储消除次数+三维visited-防止在不同的破墙次数下重合
step和bfs层绑定-扩散一层总共加一步

```
from collections import deque
class Solution:
    def shortestPath(self, grid: List[List[int]], k: int) -> int:
        q=deque()
        n,m=len(grid),len(grid[0])
        visited=[[[False]*(k+1) for _ in range(m)] for _ in range(n)]
        q.append((0,0,0))
        visited[0][0][0]=True
        step=0
        dir=[(0,1),(0,-1),(1,0),(-1,0)]
        while q:
            for _ in range(len(q)):
                x,y,used=q.popleft()
                if x==n-1 and y==m-1:
                    return step
                for dx,dy in dir:
                    nx,ny=x+dx,y+dy
                    if 0<=nx<n and 0<=ny<m:
                        nused=used+grid[nx][ny]
                        if nused<=k and not visited[nx][ny][nused]:
                            visited[nx][ny][nused]=True
                            q.append((nx,ny,nused))
            step+=1
        return -1
```

3.0-1bfs
空地代价-0
障碍代价-1
求最小代价到达目标位置
代价为0放左边，代价为1放右边-让代价最小的始终在队首
本题将dirs中的序数与数字表达的箭头方向一一对应，值得学习

```
from collections import deque
class Solution:
    def minCost(self, grid: List[List[int]]) -> int:
        n,m=len(grid),len(grid[0])
        INF=float('inf')
        dist=[[INF]*m for _ in range(n)]
        dist[0][0]=0
        dq=deque()
        dq.append((0,0))
        dirs=[(0,1),(0,-1),(1,0),(-1,0)]
        while dq:
            x,y=dq.popleft()
            if x==n-1 and y==m-1:
                return dist[x][y]
            for i,(dx,dy) in enumerate(dirs):
                nx,ny=x+dx,y+dy
                if 0<=nx<n and 0<=ny<m:
                    cost=dist[x][y]+(0 if grid[x][y]==i+1 else 1)
                    if cost<dist[nx][ny]:
                        dist[nx][ny]=cost
                        if grid[x][y]==i+1:
                            dq.appendleft((nx,ny))
                        else:
                            dq.append((nx,ny))
        return dist[n-1][m-1]
```

4.双向bfs

```
class Solution:
    def openLock(self, deadends: List[str], target: str) -> int:
        dead=set(deadends)
        if "0000" in dead:
            return -1
        begin={'0000'}
        end={target}
        visited=set()
        step=0
        while begin and end:
            if len(begin)>len(end):
                begin,end=end,begin
            next_level=set()
            for state in begin:
                if state in end:
                    return step
                if state in dead or state in visited:
                    continue
                visited.add(state)
                for i in range(4):
                    d=int(state[i])
                    for move in (1,-1):
                        nd=(d+move)%10
                        nxt=state[:i]+str(nd)+state[i+1:]
                        next_level.add(nxt)
            begin=next_level
            step+=1
        return -1
```

单词接龙

```
class Solution:
    def ladderLength(self, beginWord: str, endWord: str, wordList: List[str])
-> int:
        begin={beginWord}
        words=set(wordList)
        end={endWord}
        step=1
        L=len(beginWord)
        if endWord not in words:
            return 0
        while begin and end:
            if len(begin)>len(end):
                begin,end=end,begin
            next_begin=set()
            for word in begin:
                for i in range(L):
                    for c in "abcdefghijklmnopqrstuvwxyz":
                        if word[i]==c:
                            continue
                        nxt=word[:i]+c+word[(i+1):]
                        if nxt in end:
                            return step+1
                        if nxt in words:
                            words.remove(nxt)
                            next_begin.add(nxt)
            begin=next_begin
            step+=1
        return 0
```

Dijsktra算法

权重不等

```
import heapq
from collections import defaultdict
```

```python
from math import inf
class Solution:
    def networkDelayTime(self, times: List[List[int]], n: int, k: int) -> int:
        dst=[inf]*(n+1)
        dst[0]=0
        dst[k]=0
        pq=[(0,k)]
        dct=defaultdict(list)
        for u,v,w in times:
            dct[u].append((v,w))
        while pq:
            cur_dst,u=heapq.heappop(pq)
            if cur_dst>dst[u]:
                continue
            for v,w in dct[u]:
                if dst[v]>dst[u]+w:
                    dst[v]=dst[u]+w
                    heapq.heappush(pq,(dst[v],v))
        ans=max(dst[1:])
        return ans if ans<inf else -1
```

前缀和-prefix

prefix-k in cnt的思路很关键

```python
class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
        n=len(nums)
        prefix=0
        cnt={0:1}
        ans=0
        for x in nums:
            prefix+=x
            if prefix-k in cnt:
                ans+=cnt[prefix-k]
            cnt[prefix]=cnt.get(prefix,0)+1
        return ans
```

问题转化：
0/1 数量相同：0时pre-1，1时pre+1，最后找pre相等的区间
可被k整除：按余数存字典，最后用排列组合
二维前缀和
top，bottom压缩

```
class Solution:
    def numSubmatrixSumTarget(self, matrix: List[List[int]], target: int) -> 
int:
        m,n=len(matrix),len(matrix[0])
        aim=0
        for top in range(m):
            colume=[0]*n
            for bottom in range(top,m):
                for j in range(n):
                    colume[j]+=matrix[bottom][j]
                pre=0
                prefix={}
                for i in colume:
                    pre+=i
                    if pre==target:
                        aim+=1
                    if pre-target in prefix:
                        aim+=prefix[pre-target]
                    if pre in prefix:
                        prefix[pre]+=1
                    else:
                        prefix[pre]=1
        return aim
```

二维前缀和+二分查找

```
from bisect import bisect_left,insort
class Solution:
    def maxSumSubmatrix(self, matrix: List[List[int]], k: int) -> int:
        m,n=len(matrix),len(matrix[0])
        ans=float('-inf')
        for top in range(m):
            colume=[0]*n
            for bottom in range(top,m):
                for j in range(n):
                    colume[j]+=matrix[bottom][j]
                prefix=0
                prefix_sorted=[0]
                for i in colume:
                    prefix+=i
                    idx=bisect_left(prefix_sorted,prefix-k)
                    if idx<len(prefix_sorted):
                        ans=max(ans,prefix-prefix_sorted[idx])
                        if ans==k:
                            return k
                    insort(prefix_sorted,prefix)
        return ans
```

滑动窗口

1.无重复字符的最长子串

```
def lengthOfLongestSubstring(s: str) -> int:
    char_set = set()
    left = 0
    max_len = 0
    for right in range(len(s)):
        while s[right] in char_set:
            char_set.remove(s[left])
            left += 1
        char_set.add(s[right])
        max_len = max(max_len, right - left + 1)
    return max_len
```

2.最小覆盖子串

```
class Solution:
    def minWindow(self, s: str, t: str) -> str:
        from collections import defaultdict
        t_match=defaultdict(int)
        for c in t:
            t_match[c]+=1
        window=defaultdict(int)
        match=0
        left=0
        result=""
        min_len=float('inf')
        for right in range(len(s)):
            c=s[right]
            if c in t:
                window[c]+=1
                if window[c]==t_match[c]:
                    match+=1
            while match==len(t_match):
                current_len=right-left+1
                if current_len<min_len:
                    min_len=current_len
                    result=s[left:right+1]
                left_c=s[left]
                if left_c in t:
                    window[left_c]-=1
                    if window[left_c]<t_match[left_c]:
                        match-=1
                left+=1
        return(result if min_len!=float('inf') else "")
```

3.长度为K的子数组最大和

```
def findMaxAverage(nums: list[int], k: int) -> float:
    n = len(nums)
    window_sum = sum(nums[:k])
    max_sum = window_sum
    for right in range(k, n):
        window_sum += nums[right] - nums[right - k]
        max_sum = max(max_sum, window_sum)
    return max_sum
```

筛法

1.埃氏筛

1~n的全部质数

```
def sieve(n: int):
    is_prime = [True] * (n + 1)
    is_prime[0] = is_prime[1] = False
    for i in range(2, int(n ** 0.5) + 1):
        if is_prime[i]:
            for j in range(i * i, n + 1, i):
                is_prime[j] = False
    primes = [i for i in range(2, n + 1) if is_prime[i]]
    return primes, is_prime
```

2.欧拉筛

```
def linear_sieve(n: int):
    is_prime = [True] * (n + 1)
    primes = []
    is_prime[0] = is_prime[1] = False
    for i in range(2, n + 1):
        if is_prime[i]:
            primes.append(i)
        for p in primes:
            if i * p > n:
                break
            is_prime[i * p] = False
            if i % p == 0:
                break
    return primes, is_prime
```

3.最小质因子筛

```
def spf_sieve(n: int):
    spf = list(range(n + 1))  # spf[x] = x 的最小质因子
    for i in range(2, int(n ** 0.5) + 1):
        if spf[i] == i:  # i 是质数
            for j in range(i * i, n + 1, i):
                if spf[j] == j:
                    spf[j] = i
    return spf
```

分解质因数

```
def factorize(x: int, spf):
    res = []
    while x > 1:
        p = spf[x]
        cnt = 0
        while x % p == 0:
            x //= p
            cnt += 1
        res.append((p, cnt))
    return res
```

4.区间筛

[L,R]内的质数

```
def segmented_sieve(L, R):
    import math
    limit = int(math.isqrt(R))
    primes, _ = sieve(limit)
    is_prime = [True] * (R - L + 1)
    if L == 1:
        is_prime[0] = False
    for p in primes:
        start = max(p * p, ((L + p - 1) // p) * p)
        for x in range(start, R + 1, p):
                is_prime[x - L] = False
    return [L + i for i, v in enumerate(is_prime) if v]
```

import的用法

1.math

gcd(a,b)：最大公约数

lcm(a,b)：最小公倍数

sqrt(x)：平方根（浮点）

isqrt(x)：整数平方根

ceil(x)：向上取整

floor(x)：向下取整

fabs(x)：绝对值（浮点）

log2(x)：对数

inf：正无穷

factorial：阶乘

2.collections

deque

defaultdict(int/set/list)

3.intertools

combinations

```
from itertools import combinations
# 从 [1,2,3] 中选 2 个元素的所有组合
res = combinations([1,2,3], 2)
print(list(res))  # 转列表查看：[(1, 2), (1, 3), (2, 3)]
# 字符串也适用（按字符组合）
res2 = combinations("abc", 2)
print(list(res2))  # [('a', 'b'), ('a', 'c'), ('b', 'c')]
```

permutations

```
from itertools import permutations
# 从 [1,2,3] 中选 2 个元素的所有排列
res = permutations([1,2,3], 2)
print(list(res))  # [(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
# 不传 k 则默认选所有元素（全排列）
res2 = permutations([1,2])
print(list(res2))  # [(1, 2), (2, 1)]
```

accumulate

```
from itertools import accumulate
import operator  # 提供内置运算函数
# 默认：前缀和
arr = [1,2,3,4]
res = accumulate(arr)
print(list(res))  # [1, 3, 6, 10]（1; 1+2; 1+2+3; 1+2+3+4）
# 自定义：前缀乘积
res2 = accumulate(arr, func=operator.mul)
print(list(res2))  # [1, 2, 6, 24]（1; 1×2; 1×2×3; 1×2×3×4）
# 自定义：前缀最大值
res3 = accumulate([3,1,4,2], func=max)
print(list(res3))  # [3, 3, 4, 4]
```

groupby

```
from itertools import groupby

# 1. 未排序：仅分组连续相同元素
arr = [1,1,2,2,2,3,1]
for key, group in groupby(arr):
    print(f"键：{key}，分组元素：{list(group)}")
# 输出：
# 键：1，分组元素：[1, 1]
# 键：2，分组元素：[2, 2, 2]
# 键：3，分组元素：[3]
# 键：1，分组元素：[1]（非连续，单独分组）
# 2. 排序后：全局分组
arr_sorted = sorted(arr)
for key, group in groupby(arr_sorted):
    print(f"键：{key}，分组元素：{list(group)}")
# 输出：
# 键：1，分组元素：[1, 1, 1]
# 键：2，分组元素：[2, 2, 2]
# 键：3，分组元素：[3]
# 3. 自定义 key（按奇偶分组）
arr2 = [1,3,2,4,5,7]
for key, group in groupby(arr2, key=lambda x: x%2):
    print(f"{'奇数' if key else '偶数'}：{list(group)}")
# 输出：
# 奇数：[1, 3]
# 偶数：[2, 4]
# 奇数：[5, 7]（非连续，单独分组）
```

chain

```
from itertools import chain
# 拼接列表、元组、字符串
res = chain([1,2], (3,4), "56")
print(list(res))  # [1, 2, 3, 4, '5', '6']
# 拼接生成器（高效，无内存占用）
gen1 = (x for x in range(3))
gen2 = (x for x in range(3,6))
res2 = chain(gen1, gen2)
print(list(res2))  # [0, 1, 2, 3, 4, 5]
```

4.heapq

heappush(h, x) 入堆

heappop(h) 弹出最小值

heapify(arr) 原地建堆

nlargest(k, arr) 前 k 大

nsmallest(k, arr) 前 k 小

5.bisect

bisect_left(a, x) 第一个 ≥ x

bisect_right(a, x) 第一个 > x

insort(a, x) 保序插入

6.random

randint(l, r) 随机整数

shuffle(arr) 打乱数组

choice(arr) 随机选取

7.functools

lru_cache

cmp_to_key()

拼接最大整数

```
import sys
from functools import cmp_to_key
def cmp(a: str, b: str) -> int:
    # 若 a+b 更大，则 a 排前
    if a + b > b + a:
        return -1
    elif a + b < b + a:
        return 1
    else:
        return 0
def main():
    m = int(sys.stdin.readline().strip())
    n = int(sys.stdin.readline().strip())
```

```python
    nums = sys.stdin.readline().strip().split()
    # 排序
    nums.sort(key=cmp_to_key(cmp))
    res = []
    total_len = 0
    for x in nums:
        if total_len + len(x) <= m:
            res.append(x)
            total_len += len(x)
    # 若什么都选不了，输出 0（视题意，也可输出空）
    if not res:
        print("0")
    else:
        print("".join(res))
if __name__ == "__main__":
    main()
```

不定行输入

```python
while True:
    try:
        line = input()
    except EOFError:
        break
```

```python
import sys
for line in sys.stdin:
    line = line.strip()
```

```python
import sys
data=sys.stdin.read()
lines=data.splitlines()
for line in lines:
```

Greedy

```python
import math
rest = [0,5,3,1]
while True:
    a,b,c,d,e,f = map(int,input().split())
    if a + b + c + d + e + f == 0:
        break
    boxes = d + e + f #装4*4, 5*5, 6*6
    boxes += math.ceil(c/4) #填3*3
    spaceforb = 5*d + rest[c%4] #能和4*4 3*3 一起放的2*2
    if b > spaceforb:
        boxes += math.ceil((b - spaceforb)/9)
    spacefora = boxes*36 - (36*f + 25*e + 16*d + 9*c + 4*b) #和其他箱子一起的填的
1*1

    if a > spacefora:
        boxes += math.ceil((a - spacefora)/36)
print(boxes)
```