```
import bisect
bisect.bisect_left(lst,x)
#使用bisect_left查找插入点，若x∈lst，返回最左侧x的索引；否则返回最左侧的使x若插入后能位于其左侧的元素的当前索引。
bisect.bisect_right(lst,x)
#使用bisect_right查找插入点，若x∈lst，返回最右侧x的索引；否则返回最右侧的使x若插入后能位于其右侧的元素的当前索引。
bisect.insort(lst,x)
#使用insort插入元素，返回插入后的lst
```

```
from itertools import permutations,combinations
l = [1,2,3]
print(list(permutations(l))) # 输出：[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3,1), (3, 1, 2), (3, 2, 1)]
print(list(combinations(l,2))) # 输出：[(1, 2), (1, 3), (2, 3)]
```

```
print(list(enumerate(['a','b','c']))) # 输出：[(0, 'a'), (1, 'b'), (2, 'c')]
```

string

```
1.前后缀判定：
if str1.startwith(str2):
if str1.endwith(str2):
2.子字符串 sub 在字符串中首次出现的索引，如果未找到，则返回-1
str.find(sub)
3.判定类型：
str.isupper()#是否全为大写
str.islower()#是否全为小写
str.isdigit()#是否全为数字
str.isnumeric()#是否为整数
str.isalnum()#是否全为字母或汉字或数字
str.isalpha()#是否全为字母或汉字
4.将字符串中的 old 子字符串替换为 new
str.replace(old, new)
5.移除字符串左侧/右侧的空白字符：
str.lstrip() / str.rstrip()
6.列表
list(str)
str4 = "".join(list3)
str5 = ".".join(list3)
str6 = " ".join(list3)
```

deque

```
from collections import deque
list = [0, 1, 2, 3]
ex = (1, "h", 3)
d = deque(list)
d.append("k")
d.appendleft("k")
d.extend(ex)
d.extendleft(ex)
```

```
d.pop()
d.popleft()
d.count()#返回元素个数
d.insert(0,"k")
d.rotate(n)#从右侧反转n步(例如把12234rotate1就得到41223)
d.clear()#全部删除元素
d.remove("2")#移除第一次出现的元素，如果没有那么报出ValueError
dst = deque(maxlen=2)
#dst.append(1)
#dst.append(2)
#dst.append(3)
#print(d)，输出deque([2, 3], maxlen=2)
len(d)
reversed(d)
copy.deepcopy(d)
```

heapq

```
import heapq
a = []
heapq.heappush(a,1)#建立小根堆，第一个元素是最小的
#想要建立大根堆，可以用相反数
a = [1, 2, 3]
heapq.heapify(a)
heapq.heappop(a)#弹出并返回第一个元素，同时保持堆的属性
heapq.heappushpop(a, 4)#先进行push，再pop
heapq.heapreplace(a, 4)#先进行pop，再push
b = [('a',1),('b',2),('c',3)]
heapq.nlargest(1,b,key=lambda x:x[1])#获取b列表中最大的1个值，key为x[1]
```

扩栈

```
import sys
sys.setrecursionlimit(1<<30)
```

把连续的x个字符串s记为[xs]，输入由小写英文字母、数字和[]组成的字符串，输出原始的字符串。

样例：输入[2b[3a]c]，输出baaacbaaac

```
s = input()
stack = []
for i in range(len(s)):
    stack.append(s[i])
    if stack[-1] == ']':
        stack.pop()
        helpstack = [] # 利用辅助栈求括号内的原始字符串，记得每次用前要清空
        while stack[-1] != '[':
            helpstack.append(stack.pop())
        stack.pop()
        numstr = ''
        while helpstack[-1] in '0123456789':
            numstr += str(helpstack.pop())
        helpstack = helpstack*int(numstr)
        while helpstack != []:
```

```
            stack.append(helpstack.pop())
print(''.join(stack))
```

给出项数为 n 的整数数列 a1...an。

定义函数 f(i) 代表数列中第 i 个元素之后第一个大于 ai 的元素的**下标**。若不存在，则 f(i)=0。

试求出 f(1...n)。

```
n = int(input())
a = list(map(int, input().split()))
b = [0]*n
c = []
for i in range(n):
    if not c:
        c.append(n-i-1)
    else:
        for j in range(len(c)):
            if a[n-i-1]>=a[c[-1]]:
                c.pop()
            else:
                break
        c.append(n-i-1)
    if len(c) > 1:
        b[n-i-1] = c[-2]+1
print(*b)
```

八皇后, 输出第b个解

```
def queen_stack(n):
    stack = [] # 用于保存状态的栈，栈中的元素是(row,queens)的tuple
    solutions = [] # 存储所有解决方案的列表
    stack.append((0, tuple())) # 初始状态为第一行，所有列都未放置皇后
    while stack:
        now_row,pos = stack.pop() # 从栈中取出当前处理的行数和已放置的皇后位置
        if now_row == n:
            solutions.append(pos)
        else:
            for col in range(n):
                if is_valid(now_row,col,pos):
                    stack.append((now_row+1,pos+(col,))) # 将新的合法状态压入栈
    return solutions[::-1] # 由于栈的LIFO特性，得到的solutions为倒序
def is_valid(row,col,queens): # 检查当前位置是否合法
    for r,c in enumerate(queens):
        if c==col or abs(row-r)==abs(col-c):
            return False
    return True
solutions = queen_stack(8)
n = int(input())
for _ in range(n):
    b = int(input())
    queen_string = ''.join(str(col+1) for col in solutions[b-1])
    print(queen_string)
```

中序表达式转后序表达式

先考虑最简单的情况, 对于一个中序表达式a+b, 考虑一个列表存储运算符, 一个列表存储数, 在从左到右遍历表达式的过程后, 依次输出数和运算符即可.

考虑括号不完全的情况: a+b*(c+d)

1	2	3	4	5	6	7	8	9
NULL	+	+	+*	+*(	+*(	+*(+	+*(+	+*(+)→+*→+→NULL
a	a	ab	ab	ab	abc	abc	abcd	abcd→abcd+→abcd+*→abcd+*+

推广可得思路如下:

从左到右扫描字符串, 如果遇到的是操作数则放入待输出列表末尾, 如果遇到的是符号:

- 左括号: 直接压入栈中
- 右括号: 从栈中依次向外吐出运算符添加到待输出列表的末尾, 直到左括号被吐出
- 运算符: 比较与前一个运算符之间的优先级关系, 若优先级不如或等同于前一个运算符, 则吐出前一个运算符添加到待输出列表的末尾

最后将残留的运算符依次吐出添加到待输出列表的末尾

波兰表达式是一种把运算符前置的算术表达式，例如普通的表达式2 + 3的波兰表示法为+ 2 3。波兰表达式的优点是运算符之间不必有优先级关系，也不必用括号改变运算次序，例如(2 + 3) * 4的波兰表示法为* + 2 3 4。本题求解波兰表达式的值，其中运算符包括+ - * /四个。

```
s = input().split()
def cal():
    cur = s.pop(0)
    if cur in "+-*/":
        return '(' + cal() + cur + cal() + ')'
    else:
        return cur
print("%.6f" % eval(cal()))
```

细节

- eval()函数用来执行一个字符串表达式，并返回表达式的值。
  eval是Python的一个内置函数，这个函数的作用是，返回传入字符串的表达式的结果。想象一下变量赋值时，将等号右边的表达式写成字符串的格式，将这个字符串作为eval的参数，eval的返回值就是这个表达式的结果。
  python中eval函数的用法十分的灵活，但也十分危险，安全性是其最大的缺点。
- print("%.6f" % x)表示取6位小数.

输入中，push n表示叠上一头重量是n的猪；pop表示将猪堆顶的猪赶走；min表示问现在猪堆里最轻的猪多重（需输出答案）。

```
import heapq
from collections import defaultdict
out = defaultdict(int) # defaultdict用于记录删除的元素（查找时比list、set快）
pigs_heap = [] # heap用于确定最小的元素
pigs_stack = [] # stack用于确定最后的元素
```

```
while True:
    try:
        s = input()
    except EOFError:
        break
    if s == "pop":
        if pigs_stack:
            out[pigs_stack.pop()] += 1 # 代表删除了最后一个元素
    elif s == "min":
        if pigs_stack:
            while True: # 循环，如果最小的元素已经被删除，就寻找下一个最小的
                x = heapq.heappop(pigs_heap)
                if not out[x]: # 如果最小的元素还没有被删除
                    heapq.heappush(pigs_heap,x)
                    print(x)
                    break
                out[x] -= 1
    else:
        y = int(s.split()[1])
        pigs_stack.append(y)
        heapq.heappush(pigs_heap,y)
```

二叉树

```
class Node: # 定义节点，用class实现
    def __init__(self,value):
        self.value = value
        self.left = None
        self.right = None
n = int(input())
nodes = [node(_) for _ in range(1,n+1)]
for i in range(n):
    l,r = map(int,input().split())
    if l != -1: # 一定要先判断子节点是否存在
        nodes[i].left = nodes[l]
    if r != -1:
        nodes[i].right = nodes[r]
# 这一方法中，指针只能表示相邻两层之间的关系
```

前中序建树

```
def build_tree(preorder, inorder):
    if not preorder or not inorder: # 先判断是否为空树
        return None
    root_value = preorder[0]
    root = Node(root_value)
    root_index = inorder.index(root_value)
    root.left = build_tree(preorder[1:root_index+1], inorder[:root_index]) #递归
    root.right =
build_tree(preorder[root_index_inorder+1:],inorder[root_index_inorder+1:])
    return root
```

扩展前序建树

```
def build_tree(preorder):
    if not preorder: # 先判断是否为空树
        return None
    value = preorder.pop(0) # 正序处理（若给后序，则倒序处理，后面left,right顺序反一下）
    if value == '.':
        return None
    root = Node(value)
    root.left = build_tree(preorder) # 递归是树部分的关键思想
    root.right = build_tree(preorder)
    return root
```

深度

```
def depth(root):
    if root is None: # 先判断是否为空树
        return 0 # 若计算高度，则return -1
    else:
        left_depth = depth(root.left) # 递归
        right_depth = depth(root.right)
        return max(left_depth, right_depth)+1
```

叶子数目

```
def count_leaves(root):
    if root is None: # 先判断是否为空树
        return 0
    if root.left is None and root.right is None:
        return 1
    return count_leaves(root.left)+count_leaves(root.right)
```

层次遍历

```
from collections import deque
def level_order_traversal(root):
    q = deque()
    q.append(root)
    output = []
    while q:
        node = q.popleft()
        output.append(node.value)
        if node.left: # 仍然是先判断子节点是否存在
            q.append(node.left)
        if node.right:
            q.append(node.right)
    return output
```

二叉搜索树

```
def insert(root,num):
    if not root:
        return Node(num)
    if num<root.val:
        root.left=insert(root.left,num)
    else:
        root.right=insert(root.right,num)
    return root
```

每次剪断绳子时需要的开销是此段绳子的长度，输入将绳子剪成的段数n和准备剪成的各段绳子的长度，输出最小开销。

```
import heapq
n = int(input())
a = list(map(int,input().split()))
heapq.heapify(a)
ans = 0
for i in range(n-1):
    x = heapq.heappop(a)
    y = heapq.heappop(a)
    z = x+y
    heapq.heappush(a,z)
    ans += z
print(ans)
```

遍历规则：遍历到每个节点（值为互不相同的正整数）时，按照该节点和所有子节点的值从小到大进行遍历。输入的第一行为节点个数n，接下来的n行中第一个数是此节点的值，之后的数分别表示其所有子节点的值；分行输出遍历结果。

```
class Node:
    def __init__(self,value):
        self.value = value
        self.children = []
        # self.parent = None（有些题中需要，便于确定节点归属）
def traverse_print(root,nodes):
    if root.children == []: # 同理，先判断子节点是否存在
        print(root.value)
        return
    to_sort = {root.value:root} # 此处利用value来查找Node，而不是用指针（因为多叉树的指针往往只能表示相邻两层之间的关系）
    for child in root.children:
        to_sort[child] = nodes[child]
    for value in sorted(to_sort.keys()):
        if value in root.children:
            traverse_print(to_sort[value], nodes) # 递归
        else:
            print(root.value)
n = int(input())
nodes = {}
children_list = [] # 用来找根节点
for i in range(n):
    l = list(map(int,input().split()))
    nodes[l[0]] = Node(l[0])
    for child_value in l[1:]:
```

```
        nodes[l[0]].children.append(child_value)
        children_list.append(child_value)
root = nodes[[value for value in nodes.keys() if value not in children_list][0]]
traverse_print(root,nodes)
```

设输入为扩展二叉树的前序遍历，要转换为n叉树

```
nodes = {} # 用于存储n叉树的所有节点
def bi_to_n(node):
    if node.left:
        if node.left.value != '*':
            new_node = Node(node.left.value)
            nodes[node.left] = new_node
            nodes[node].child.append(new_node)
            new_node.parent = nodes[node]
            bi_to_n(node.left) # 递归
    if node.right:
        if node.right.value != '*':
            new_node = Node(node.right.value)
            nodes[node.right] = new_node
            nodes[node].parent.child.append(new_node)
            new_node.parent = nodes[node].parent
            bi_to_n(node.right)
```

拓扑排序

```
from collections import deque,defaultdict
def topological_sort(graph):
    indegree = defaultdict(int)
    order = []
    vis = set()
    for u in graph: # 统计各顶点入度
        for v in graph[u]:
            indegree[v] += 1
    q = deque()
    for u in graph:
        if indegree[u] == 0:
            q.append(u)
    while q:
        u = q.popleft()
        order.append(u)
        vis.add(u)
        for v in graph[u]:
            indegree[v] -= 1
            if indegree[v] == 0 and v not in vis:
                q.append(v)
    if len(order) == len(graph):
        return order
    else: # 说明存在环
        return None
```

trie

```
```

```
class trie:
    def __init__(self):
        self.nex = [[0 for i in range(26)] for j in range(100000)]
        self.cnt = 0
        self.exist = [False] * 100000  # 该结点结尾的字符串是否存在

    def insert(self, s):  # 插入字符串
        p = 0
        for i in s:
            c = ord(i) - ord("a")
            if not self.nex[p][c]:
                self.cnt += 1
                self.nex[p][c] = self.cnt  # 如果没有，就添加结点
            p = self.nex[p][c]
        self.exist[p] = True

    def find(self, s):  # 查找字符串
        p = 0
        for i in s:
            c = ord(i) - ord("a")
            if not self.nex[p][c]:
                return False
            p = self.nex[p][c]
        return self.exist[p]
```

prim

```
from heapq import heappop, heappush

while True:
    try:
        n = int(input())
        m = []
        for _ in range(n):
            m.append(list(map(int,input().split())))
        d = [100000 for _ in range(n)]
        v = set()
        pq = []
        result = 0
        d[0] = 0
        heappush(pq,(d[0],0))
        while pq:
            x,y = heappop(pq)
            if y not in v:
                v.add(y)
                result += d[y]
                for i in range(n):
                    if d[i] > m[y][i]:
                        d[i] = m[y][i]
                        heappush(pq, (d[i], i))
        print(result)
    except EOFError:
        break
```

最短路

```
from heapq import heappop, heappush
from copy import deepcopy

while True:
    try:
        n = int(input())
        m = []
        for _ in range(n):
            m.append(list(map(int,input().split())))
        ini, fin = map(int,input().split())
        d = [100000 for _ in range(n)]
        route = [[] for _ in range(n)]
        v = set()
        pq = []
        result = 0
        d[ini] = 0
        route[ini].append(ini)
        heappush(pq,(d[ini],ini))
        while pq:
            x,y = heappop(pq)
            if y not in v:
                v.add(y)
                for i in range(n):
                    if d[i] > d[y] + m[y][i]:
                        d[i] = d[y] + m[y][i]
                        heappush(pq, (d[i], i))
                        route[i] = deepcopy(route[y].append(i))
                        route[y].pop()
        print(d[fin])
        print(route[fin])
    except EOFError:
        break
```

Kruskal: 将边由小到大加入图中，若不成环，则保留；反之，舍弃

并查集

```
def find(x):
    if p[x]!=x:
        p[x]=find(p[x])
    return p[x]
def union(x,y):
    rootx,rooty=find(x),find(y)
    if rootx!=rooty:
        p[rootx]=p[rooty]
#单纯并查
p=list(range(n+1))
unique_parents = set(find(x) for x in range(1, n + 1)) 最后收取数据
#反向事件 用x+n储存x的反向事件，查询时直接find（x+n）
p=list(range(2*(n+1))
if tag=="Different":
    union(x,y+n)
    union(y,x+n)
```