

# 数据结构与算法——总结

---

各种算法的英文

单调栈: monotonous-stack

并查集: disjoint set

调度场算法: shunting yard (作业4)

线段树: segment tree (不考)

拓扑排序: Topological Sorting (常用kahn)

Set去重计数 (并查集连通常用)

## 输出/输入

---

同一行输出:

使用end="", 引号中间填写希望中间隔开的东西, 比如说一个空格

在下一行print()从而达到换行的目的

```
for y in range(m):
    print(long1[x][y],end=" ")
print()
```

#假设 row 是 [1, 2, 3], 那么 print(\*row) 就等价于 print(1, 2, 3), 它会打印出每个元素之间用空格分隔的内容。这种语法对于打印列表、元组等可迭代对象的内容非常方便。

```
#对于可迭代对象, 可以通过*解决
#media=[1,2,3,4]
print(*media)
#输出: 1 2 3 4
```

输出列表内所有元素:

```
print(*lst) ##把列表中元素顺序输出
```

mylist=[1,2,3,4,5]

print(mylist,sep=',')#把列表中所有元素之间用sep内部的东西分开, 输出形式仍然是一个列表[1, 2, 3, 4, 5]

保留小数的多种方式:

```
print("{:.2f}".format(3.146)) # 3.15
print(round(3.123456789,5))# 3.12346四舍六入五成双
#保留n位小数:
print(f'{x:.nf}')
```

不定行输入: 套用try-except循环

Try except 无法使用break来跳出循环，如果已知结束的特定输入（比如说输入0代表结束），那么可以用以下代码：

```
while True:
    try:
        XXXXX
        If input()=='0':
            break
    except EOFError:
        break
```

## 其他

```
float("inf")#表示正无穷，可以用于赋值

output = ','.join(my_list)
print(output)#把列表中的所有元素一串输出，条件是列表中的元素都是str类型

data = [
    {'name': 'Alice', 'age': 25},
    {'name': 'Bob', 'age': 30},
    {'name': 'Charlie', 'age': 20}
]

# 按照年龄排序
sorted_data = sorted(data, key=lambda x: x['age'])
print(sorted_data)

strings = ["apple", "banana", "cherry", "date"]

# 按照字符串的长度排序
sorted_strings = sorted(strings, key=lambda x: len(x))
print(sorted_strings)

ord() #返回对应的ASCII表值
chr() #返回ASCII值对应的字符
bin(),oct(),hex()#分别表示二进制，八进制，十六进制的转换

# 二进制转十进制
binary_str = "1010"
decimal_num = int(binary_str, 2) # 第一个参数是字符串类型的某进制数，第二个参数是他的进制，
最终转化为整数
print(decimal_num) # 输出 10

#eval

#判断完全平方数
import math
def isPerfectSquare(num):
```

```

    if num < 0:
        return False
    sqrt_num = math.isqrt(num)
    return sqrt_num * sqrt_num == num
print(isPerfectSquare(97)) # False

#年份calendar
import calendar
print(calendar.isleap(2020)) # True, 判断是否是闰年

#字符串的连接
str=str1+str2

#用同一个数填充整个列表
dp=[[0]*(i+1) for i in range(5)]
#输出[[0], [0, 0], [0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0, 0]]

#str相关
str.lstrip() / str.rstrip(): 移除字符串左侧/右侧的空白字符。

str.find(sub): 返回子字符串`sub`在字符串中首次出现的索引, 如果未找到, 则返回-1。

str.replace(old, new): 将字符串中的`old`子字符串替换为`new`。

str.isalpha() / str.isdigit() / str.isalnum(): 检查字符串是否全部由字母/数字/字母和数字组成。
isnumeric() is_integer() isalpha() isupper() islower()
str.title(): 每个单词首字母大写。

#保留小数
#1.round函数
number = 3.14159
rounded_number = round(number, 1)

print(rounded_number)
#输出3.1, 保留一位小数
#2.format格式化
number = 3.14159
formatted_number = "{:.1f}".format(number)

print(formatted_number)
#保留一位小数, 输出的类型为str

#字符串的连接
''.join()
#eg: ','.join([2,3,4,5])
#output:2,3,4,5

#eval函数
#eval() 是 python 中功能非常强大的一个函数, 将字符串当成有效的表达式来求值, 并返回计算结果, 就是实现 list、dict、tuple、与str 之间的转化
result = eval("1 + 1")
print(result) # 2

result = eval("'+' * 5")
print(result) # +++++

```

```
# 3. 将字符串转换成列表
a = "[1, 2, 3, 4]"
result = type(eval(a))
print(result) # <class 'list'>

input_number = input("请输入一个加减乘除运算公式：")
print(eval(input_number))
## 1*2 +3
## 5

dic.keys() 获得dic里面全部key并组成一个列表

for key,value in dict.items() #遍历字典的键值对。

for index,value in enumerate(list) #枚举列表，提供元素及其索引。

dic.setdefault(key, []).append(value) #常用在字典中加入元素的方式（如果没有值就建空表，有值就直接添加）

dict.get(key,default) #从字典中获取键对应的值，如果键不存在，则返回默认值`default`。

list(zip(a,b)) #将两个列表元素一一配对，生成元组的列表。

#set 集合减法
```

```
#####
```

# default+f.string+heapq语法+statistics+序列切片



ChatGPT

`defaultdict` 是 Python 标准库 `collections` 中的一个类，它是一种特殊的字典，与内置字典 `dict` 类似，但是在处理缺失键（即字典中没有的键）时有所不同。

与普通字典不同的是，`defaultdict` 在创建时需要提供一个默认值的工厂函数（可以是任何可调对象，如 `int`、`list`、`set` 等），当试图访问一个不存在的键时，会使用工厂函数创建一个默认值并将其插入到字典中。这样可以避免在访问不存在的键时抛出 `KeyError` 异常。

下面是一个示例：

python

Copy code

```
from collections import defaultdict

# 使用 int 作为默认值工厂函数，对不存在的键返回整数 0
d = defaultdict(int)

d['a'] = 1
d['b'] = 2

print(d['a']) # 输出: 1
print(d['b']) # 输出: 2
print(d['c']) # 输出: 0, 因为 'c' 不存在, 使用 int() 创建默认值为 0

# 使用 list 作为默认值工厂函数，对不存在的键返回一个空列表
d = defaultdict(list)

d['a'].append(1)
d['b'].append(2)

print(d['a']) # 输出: [1]
print(d['b']) # 输出: [2]
print(d['c']) # 输出: [], 因为 'c' 不存在, 使用 list() 创建默认值为空列表
```

这里 `defaultdict` 提供了一种便捷的方式来处理字典中的缺失键，并且可以避免在一些情况下手动检查键是否存在。



## 1. 变量插值

python

Copy code

```
name = "John"
age = 25
formatted_string = f"My name is {name} and I am {age} years old."
print(formatted_string)
# 输出: My name is John and I am 25 years old.
```

## 2. 表达式插值

python

Copy code

```
x = 10
y = 20
formatted_string = f"The sum of {x} and {y} is {x + y}."
print(formatted_string)
# 输出: The sum of 10 and 20 is 30.
```

## 3. 格式化数字

python

Copy code

```
pi = 3.141592653589793
formatted_string = f"The value of pi is approximately {pi:.2f}."
print(formatted_string)
# 输出: The value of pi is approximately 3.14.
```

## 4. 使用表达式和函数

python

Copy code

```
import math
radius = 5
formatted_string = f"The area of a circle with radius {radius} is {math.pi * radius**2}."
print(formatted_string)
# 输出: The area of a circle with radius 5 is 78.54.
```

## 5. 使用字典和列表

python

Copy code

```
person = {"name": "Alice", "age": 30}
formatted_string = f"My name is {person['name']} and I am {person['age']} years old."
print(formatted_string)
# 输出: My name is Alice and I am 30 years old.

numbers = [1, 2, 3, 4, 5]
formatted_string = f"The third element in the list is {numbers[2]}."
print(formatted_string)
# 输出: The third element in the list is 3.
```

## 1. `heapq.heappush(heap, item)`

将元素 `item` 插入堆 `heap` 中，保持堆的不变性。

```
python 复制代码

import heapq

heap = []
heapq.heappush(heap, 3)
heapq.heappush(heap, 1)
heapq.heappush(heap, 4)
heapq.heappush(heap, 2)

print(heap) # 输出: [1, 2, 4, 3]
```

## 2. `heapq.heappop(heap)`

从堆中弹出并返回最小的元素，保持堆的不变性。

```
python 复制代码

import heapq

heap = [1, 2, 4, 3]
smallest = heapq.heappop(heap)

print(smallest) # 输出: 1
print(heap)     # 输出: [2, 3, 4]
```

## 3. `heapq.heapify(x)`

将列表 `x` 转换为堆，原地操作。

```
python 复制代码

import heapq

lst = [3, 1, 4, 2]
heapq.heapify(lst)

print(lst) # 输出: [1, 2, 4, 3]
```

## 4. `heapq.heappushpop(heap, item)`

将元素 `item` 插入堆中，然后弹出并返回堆中的最小元素。这个操作比先 `heappush` 再 `heappop` 更高效。

```
python 复制代码

import heapq

heap = [1, 3, 4, 2]
result = heapq.heappushpop(heap, 5)
```

```
print(result) # 输出: 1
print(heap)   # 输出: [2, 3, 4, 5]
```

### 5. `heapq.heapreplace(heap, item)`

弹出并返回堆中的最小元素，然后将 `item` 插入堆中。这个操作比先 `heappop` 再 `heappush` 更高效。

```
python 复制代码

import heapq

heap = [1, 3, 4, 2]
result = heapq.heapreplace(heap, 5)

print(result) # 输出: 1
print(heap)   # 输出: [2, 3, 4, 5]
```

### 6. `heapq.nlargest(n, iterable, key=None)`

返回可迭代对象 `iterable` 中的 `n` 个最大元素。

```
python 复制代码

import heapq

lst = [1, 3, 4, 2, 5]
largest = heapq.nlargest(3, lst)

print(largest) # 输出: [5, 4, 3]
```

### 7. `heapq.nsmallest(n, iterable, key=None)`

返回可迭代对象 `iterable` 中的 `n` 个最小元素。

```
python 复制代码

import heapq

lst = [1, 3, 4, 2, 5]
smallest = heapq.nsmallest(3, lst)

print(smallest) # 输出: [1, 2, 3]
```

这些函数提供了基本的堆操作，方便进行优先队列等操作。





## ChatGPT

在 Python 的 `statistics` 模块中包含了一些用于统计分析的常用函数。以下是其中一些常见的函数：

1. `mean(data)`：计算数据的算术平均值。
2. `median(data)`：计算数据的中位数。
3. `mode(data)`：计算数据的众数。
4. `stdev(data)`：计算数据的标准差。
5. `variance(data)`：计算数据的方差。
6. `harmonic_mean(data)`：计算数据的调和平均值。
7. `median_low(data)`：计算数据的低位中位数。
8. `median_high(data)`：计算数据的高位中位数。

这些函数可以用于对数据集进行统计分析，计算数据的各种统计指标。需要注意的是，这些函数都要求输入的数据是一个可迭代的对象，比如列表或者元组。

📄 🔄 📄



ChatGPT

`[::-1]` 是 Python 中的切片操作，用于对序列进行反转操作。在这种情况下，它被用于对列表进行反转。

具体来说，`[::-1]` 的三个部分分别代表切片操作的起始索引、终止索引和步长。在省略起始索引和终止索引的情况下，它们会默认为序列的开始和结束位置。步长为 `-1` 表示从后向前取值，实现了序列的反转。

因此，`queue\_expression = level\_order\_traversal(root)[::-1]` 表示对 `level\_order\_traversal(root)` 返回的列表进行反转，然后将结果赋值给 `queue\_expression`。



你

还有没有什么别的相似用法



ChatGPT

除了 `[::-1]` 进行反转外，还有一些类似的切片操作可以实现不同的效果。以下是一些示例：

1. 步长不为1的切片操作：

python

Copy code

```
sequence[start:stop:step]
```

这种形式允许你在切片时指定一个步长。如果步长大于1，则会跳过一定数量的元素。

2. 只指定步长为-1进行反转：

python

Copy code

```
sequence[::-1]
```

这是用于反转列表的常见方式，因为起始和终止索引都被省略了，步长为-1表示从末尾向前遍历，实现了反转。

3. 删除元素：

python

Copy code

```
del sequence[start:stop]
```

这种形式可以删除指定范围内的元素。

4. 截取子序列：

python

Copy code

```
sequence[start:stop]
```

这种形式用于截取序列的子序列，包括起始索引对应的元素，但不包括终止索引对应的元素。

5. 复制序列：

python

Copy code

```
sequence[:]
```

这种形式用于创建序列的副本。

这些都是切片操作的常见用法，可以根据需要进行灵活组合和应用。

# 动态规划

## 0-1背包

```
#0-1背包的memory简化
f[i][l]=max(f[i-1][l],f[i-1][l-w[i]]+v[i])#这要二维数组i为进行到第i个物品，l为最大容量
for i in range(1, n + 1):#这时只需要一维，l为最大容量，通过反复更新维护
    for l in range(w, w[i] - 1, -1):#必须这样逆序，要让每个f只被更新一次
        f[l] = max(f[l], f[l - w[i]] + v[i])
```

## 完全背包

```
#完全背包（每件物品可以选择任意次）
f[i][l]=max(f[i-1][l],f[i][l-w[i]]+v[i])#这要二维数组i为进行到第i个物品，l为最大容量
for i in range(1, n + 1):#这时只需要一维，l为最大容量，通过反复更新维护
    for l in range(0, w - w[i] + 1):#此时要正序，根本原因是可以多次选择
        f[l + w[i]] = max(f[l] + v[i], f[l + w[i]])
```

## 多重背包

```
#多重背包（物品选择指定次）
#朴素想法转化为0-1背包，可能超时，因此考察二进制拆分（先尽力拆为1，2，4，8...）
import math
k=int(math.log(x,2))
for i in range(k+2):
    if x>=2**i:
        x-=2**i
        coi.append(y*(2**i))
    else:
        coi.append(x*y)
        break

#最大连续子序列和 在dp库里找最大值即可
n = int(input())
a = list(map(int, input().split()))

dp = [0]*n
dp[0] = a[0]

for i in range(1, n):
    dp[i] = max(dp[i-1]+a[i], a[i])#状态转移方程：只有两种可能，取较大的

print(max(dp))

#最长上升子序列的长度
n = int(input())
b = list(map(int, input().split()))
dp = [1]*n

for i in range(1, n):
    for j in range(i):
        if b[j] < b[i]:
```

```
dp[i] = max(dp[i], dp[j]+1)#dp对应索引存储的是以第i个数为结尾的上升子序列长度，对于i之前的数，如果小于这个，就可以更新dp[i]的值
```

```
print(max(dp))
```

#小偷背包

```
n,b=map(int, input().split())
price=[0]+[int(i) for i in input().split()]
weight=[0]+[int(i) for i in input().split()]
bag=[0]*(b+1) for _ in range(n+1)
#bag[i][j] 表示前 i 件物品放入容量为 j 的背包可以获得的最大价值。
#price[i] 表示第 i 件物品的价值。
#weight[i] 表示第 i 件物品的重量。
for i in range(1,n+1):
    for j in range(1,b+1):
        if weight[i]<=j:
            bag[i][j]=max(price[i]+bag[i-1][j-weight[i]], bag[i-1][j])
        else:
            bag[i][j]=bag[i-1][j]
```

#状态转移方程的意义是在考虑第 i 件物品时，背包的容量为 j 时的最大价值取决于两种情况的较大值

#1，第 i 件物品放入背包，其价值为 price[i]，并且需要腾出 weight[i] 的空间，此时背包容量变为 j-weight[i]，因此最大价值为 price[i] + bag[i-1][j-weight[i]]。

#2，不放入第 i 件物品，背包容量为 j 时的最大价值为 bag[i-1][j]，即前 i-1 件物品放入容量为 j 的背包时的最大价值。

#通过状态转移方程，逐步计算出放入不同数量和重量的物品时，背包在不同容量下可以获得的最大价值，最终得到的结果就是整个问题的解。

```
print(bag[-1][-1])
```

#采药 类似小偷背包

# 首先将草药按顺序存入列表，然后构造一个二维dp表，i表示第i个草药，j表示此时剩余的时间数，如果下一个草药的时间比j小，那么就分两种情况，放入或者不放入该草药。如果下一个草药的时间比t大，那么就只能不放入这个草药。对i和j进行双循环遍历，最后到第M组的第T次即为所求

```
T, M = map(int, input().split()) # T是总时间，M是草药数目
```

```
herbal = []
```

```
for _ in range(M):
    t, v = map(int, input().split())
    herbal.append([t, v])
```

```
dp = [[0] * (T + 1) for _ in range(M + 1)]
```

```
for a in range(1, M + 1):
    for b in range(1, T + 1):
        if herbal[a - 1][0] <= b: # 如果草药采摘时间小于等于剩余时间
            dp[a][b] = max(dp[a-1][b], dp[a-1][b-herbal[a-1][0]] + herbal[a-1][1])
        else:
            dp[a][b] = dp[a-1][b]
```

```
print(dp[M][T])
```

## 最大连续子序列和

```
dp = [0]*n
dp[0] = a[0]
for i in range(1, n):
    dp[i] = max(dp[i-1]+ls[i], ls[i])
print(max(dp))
```

## 最大上升子序列和

```
if ls[i] > ls[j]:
    dp[i] = max(dp[i], dp[j] + ls[i])
```

## 最长公共子序列

```
while True:
    try:
        x, y = input().split()
    except EOFError:
        break
    lx, ly = len(x), len(y)
    dp = [[0]*(ly + 1) for _ in range(lx + 1)]
    for i in range(1, lx + 1):
        for j in range(1, ly + 1):
            if x[i - 1] == y[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            dp[i][j] = max(dp[i][j], dp[i - 1][j], dp[i][j - 1])
    print(dp[-1][-1])
```

## math库

```
import math
math.sqrt() #开方
math.ceil()#取浮点数上整数
math.floor()#取浮点数的下整数
math.gcd(a,b)#取两个数的最大公约数
math.pow(2,3) # 8.0 幂运算
math.inf#表示正无穷
math.log(100,10) # 2.0
#math.log(x,base) 以base为底, x的对数
math.comb(5,3) # 组合数, C53
math.factorial(5) # 5! 阶乘
```

## 取质数

```
#欧拉筛 输出素数列表
def Euler_sieve(n):
    primes = [True for _ in range(n+1)]
    p = 2
```

```

while p*p <= n:
    if primes[p]:
        for i in range(p*p, n+1, p):
            primes[i] = False
        p += 1
    primes[0]=primes[1]=False
    return primes
print(Euler_sieve(20))
# [False, False, True, True, False, True, False, True, False, False, False, True,
False, True, False, False, False, True, False, True, False]

#埃氏筛 输出素数列表
N=20
primes = []
is_prime = [True]*N
is_prime[0] = False;is_prime[1] = False
for i in range(1,N):
    if is_prime[i]:
        primes.append(i)
        for k in range(2*i,N,i): #用素数去筛掉它的倍数
            is_prime[k] = False
print(primes)
# [2, 3, 5, 7, 11, 13, 17, 19]

#欧拉筛 直接判断某个数是否是质数
prime = []
n = c
is_prime=[True] * (n+1) # 初始化为全是素数
is_prime[0]=is_prime[1]=False#把0和1标记为非素数

def euler_sieve():

    for i in range(2, n // 2 + 1):
        if is_prime[i]:
            prime.append(i)
            for j in prime: # 将所有质数的倍数标记为非素数

                if i * j > n:
                    break
                is_prime[j * i] = False
                if i % j == 0:
                    break

# 测试
euler_sieve()

if is_prime[n]:
    return True
else:
    return False

#6k+1质数判断法即Miller-Rabin素性测试算法
import math

def is_prime(num):

```

```

if num <= 1:
    return False
elif num <= 3:
    return True
elif num % 2 == 0 or num % 3 == 0:
    return False
i = 5
while i * i <= num:
    if num % i == 0 or num % (i + 2) == 0:
        return False
    i += 6
return True

```

#其原理基于以下观察：除了2和3之外，所有的质数都可以表示为  $6k \pm 1$  的形式，其中  $k$  是一个整数

## 进制转换 ( $n \rightarrow m$ )

- 不断//m，余数入栈，商作为下一轮被除数
- 栈中数字依次出栈（倒着输出）

### 02734:十进制到八进制

```

n = int(input())
stack = []
if n == 0: # 输入为0时单独讨论
    stack = ['0']
while n > 0: # 进入下一轮的条件是大于0
    stack.append(str(n%8))
    n //= 8
stack.reverse()
print(''.join(stack))

```

04089: 电话号码

trie, <http://cs101.openjudge.cn/practice/04089/>

给你一些电话号码，请判断它们是否是一致的，即是否有某个电话是另一个电话的前缀。比如：

Emergency 911 Alice 97 625 999 Bob 91 12 54 26

在这个例子中，我们不可能拨通Bob的电话，因为Emergency的电话是它的前缀，当拨打Bob的电话时会先接通Emergency，所以这些电话号码不是一致的。

输入

第一行是一个整数 $t$ ， $1 \leq t \leq 40$ ，表示测试数据的数目。每个测试样例的第一行是一个整数 $n$ ， $1 \leq n \leq 10000$ ，其后 $n$ 行每行是一个不超过10位的电话号码。

输出

对于每个测试数据，如果是一致的输出“YES”，如果不是输出“NO”。

样例输入

```

2
3
911
97625999

```

91125426

5

113

12340

123440

12345

98346

样例输出

NO

YES

<https://www.geeksforgeeks.org/trie-insert-and-search/>

Definition: A trie (prefix tree, derived from retrieval) is a multiway tree data structure used for storing strings over an alphabet. It is used to store a large amount of strings. The pattern matching can be done efficiently using tries.

使用字典实现的字典树 (Trie)。它的主要功能是插入和搜索字符串。

```
class TrieNode:
    def __init__(self):
        self.child={}

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

    def insert(self, nums):
        curnode = self.root
        for x in nums:
            if x not in curnode.child:
                curnode.child[x] = TrieNode()
            curnode=curnode.child[x]

    def search(self, num):
        curnode = self.root
        for x in num:
            if x not in curnode.child:
                return 0
            curnode = curnode.child[x]
        return 1

t = int(input())
p = []
for _ in range(t):
    n = int(input())
    nums = []
    for _ in range(n):
        nums.append(str(input()))
    nums.sort(reverse=True)
    s = 0
```



```
trie = Trie()
for num in nums:
    s += trie.search(num)
    trie.insert(num)
if s > 0:
    print('NO')
else:
    print('YES')
```

## 排序

### 一、冒泡排序\$(Bubble\$ \$Sort)\$

冒泡排序是最简单的排序算法，它通过不断交换相邻元素以实现正确的排序结果。

时间复杂度： $O(n^2)$ ；空间复杂度： $O(1)$ 。

冒泡排序是一种原地排序算法，无需额外空间，是稳定的。

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:
            break
```

### 二、选择排序\$(Selection\$ \$Sort)\$

选择排序是一种简单且高效的排序算法，其工作原理是从列表的未排序部分反复选择最小(或最大)元素，并将其移动到列表的已排序部分。

时间复杂度： $O(n^2)$ ；空间复杂度： $O(1)$ 。

选择排序同样是一种原地排序算法，无需额外空间，但它在小的数据集下相对高效，在极端情况下会具有较大的时间复杂度，是不稳定的。

```
def selection_sort(arr):
    for p in range(len(arr)-1, 0, -1):
        position = 0
        for location in range(1, p+1):
            if arr[location] > arr[position]:
                position = location
        if p != position:
            arr[p], arr[position] = arr[position], arr[p]
```

### 三、快速排序(Quick Sort)

快速排序是一种基于分治算法的排序算法，它选择一个元素作为基准，并通过将基准放置在已排序数组中的正确位置来围绕所选择的基准对给定数组进行分区。

时间复杂度：最好时为 $O(n \log n)$ ，最差时为 $O(n^2)$ ；空间复杂度：考虑递归堆栈，为 $O(n)$ ，不考虑则为 $O(1)$ 。

快速排序相对更适用于大数据集，在某些极端情况下会显现出较差的时间复杂度，是不稳定的。

P.S: 似乎二叉搜索树的建树再加上中序遍历，就可以获得同样的结果。

```
def quick_sort(arr, left, right):
    if left < right:
        position = partition(arr, left, right)
        quick_sort(arr, left, position-1)
        quick_sort(arr, position+1, right)
def partition(arr, left, right):
    i = left
    j = right-1
    pivot = arr[right]
    while i <= j:
        while i <= right and arr[i] < pivot:
            i += 1
        while j >= left and arr[j] >= pivot:
            j -= 1
        if i < j:
            arr[i], arr[j] = arr[j], arr[i]
    if arr[i] > pivot:
        arr[i], arr[right] = arr[right], arr[i]
    return i
```

### 四、归并排序(Merge Sort)

归并排序作为一种排序算法，其原理是将数组划分为更小的子数组，对每个子数组进行排序，然后将排序后的子数组合并在一起，形成最终的排序数组。（和快速排序同样的分治思想）

时间复杂度： $O(n \log n)$ ；空间复杂度： $O(n)$ 。

归并排序是一种天然的可并行化算法，而且稳定，因此特别适合用来处理大数据集，但它需要额外空间。

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left = arr[:mid]
        right = arr[mid:]
        merge_sort(left)
        merge_sort(right)
        i, j, k = 0, 0, 0
        while i < len(left) and j < len(right):
            if left[i] <= right[j]:
                arr[k] = left[i]
                i += 1
            else:
                arr[k] = right[j]
                j += 1
            k += 1
        while i < len(left):
            arr[k] = left[i]
            i += 1
            k += 1
        while j < len(right):
            arr[k] = right[j]
            j += 1
            k += 1
```

```

        else:
            arr[k] = right[j]
            j += 1
            k += 1
    while i < len(left):
        arr[k] = left[i]
        i += 1
        k += 1
    while j > len(right):
        arr[k] = right[j]
        j += 1
        k += 1

```

## 五、插入排序\$(Insertion\$ \$Sort)\$

插入排序是一种基本的排序算法，其思想在于通过先前的已排好序的数组得到目标插入元素的插入位置，从而达到不断排序的目的。

时间复杂度： $O(n^2)$ ；空间复杂度： $O(1)$ 。

插入排序是一种稳定的原地排序算法。

```

def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key

```

## 六、希尔排序\$(Shell\$ \$Sort)\$

希尔排序可以看作是插入排序的变种，也就相当于可以交换远项。

时间复杂度：（最差） $O(n^2)$ ；空间复杂度： $O(1)$ 。

希尔排序的时间复杂度取决于算法的对象，是不稳定的。

```

def shell_sort(arr, n):
    gap = n // 2
    while gap > 0:
        j = gap
        while j < n:
            i = j - gap
            while i >= 0:
                if arr[i+gap] > arr[i]:
                    break
                else:
                    arr[i+gap], arr[i] = arr[i], arr[i+gap]
            i -= gap
            j += 1
        gap //= 2

```

## 七、堆排序(Heap Sort)

堆排序是一种基于完全二叉树（堆）的排序算法。它通过将待排序的元素构建成一个堆，然后利用堆的性质来实现排序。在堆中，每个结点的值都必须大于等于其子结点的值。

时间复杂度： $O(n \log n)$ ；空间复杂度： $O(1)$ 。

堆排序适合处理大型数据集，采取原地排序，但不稳定，因为可能会交换相同元素。

```
def heapify(arr, n, i):
    largest = i
    l = 2*i + 1
    r = 2*i + 2
    if l < n and arr[l] > arr[largest]:
        largest = l
    if r < n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heapsort(arr):
    n = len(arr)
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```

## 八、计数排序(Counting Sort)

计数排序是一种非比较性的整数排序算法。它的基本思想是对于给定的输入序列中的每一个元素  $x$ ，确定该序列中值小于  $x$  的元素的个数。利用这个信息，可以直接确定  $x$  在输出序列中的位置，从而实现排序。

时间复杂度： $O(n+k)$ ，其中  $n$  为序列长度， $k$  为序列数据范围。空间复杂度也为  $O(n+k)$ 。

计数排序是一种稳定的排序算法，在数据集数值范围较小的情况下表现优越，但倘若数据离散程度较大则效率将会大大降低。

```
def counting_sort(arr):
    max_val = max(arr)
    min_val = min(arr)
    counts = [0 for _ in range(max_val-min_val+1)]
    for num in arr:
        counts[num-min_val] += 1
    for i in range(1, max_val-min_val+1):
        counts[i] += counts[i-1]
    result = [0 for _ in range(len(arr))]
    for num in reversed(arr):
        result[counts[num-min_val]-1] = num
```

```
counts[num-min_val] -= 1
return result
```

## 九、桶排序\$(Bucket\$ \$Sort)\$

桶排序将数组分割成几个部分（桶），然后对每个桶进行排序，最后将所有桶中的元素合并起来得到最终的有序数组。它的基本思想是将待排序数组分割成若干个较小的数组（桶），每个桶再分别排序，最后按照顺序依次取出各个桶中的元素即可。

时间复杂度：理想情况下为 $O(n+k)$ ，极端情况下为 $O(n^2)$ 。空间复杂度同样。

桶排序适合相对均匀稠密的数字序列，在处理小数序列时相对更具有优势（小数更适合用于建立桶的索引与数字之间的映射）。

P.S: 桶排序中每个桶使用的排序方法时其它的排序方法，所以桶更适理解作为一种思想，同样的，代码不在此展示。

## 十、基数排序\$(Radix\$ \$Sort)\$

基数排序是一种非比较性的整数排序算法，它根据键值的每位数字来进行排序。基数排序的核心思想是将待排序的元素按照位数切割成不同的数字，然后按照这些位数分别进行排序。这种排序算法属于分配式排序，它的性能取决于桶的使用方法。

时间复杂度： $O(d*(n+k))$ ，空间复杂度： $O(n+k)$ 。

基数排序不依赖于比较操作，适用于整数等固定长度的数据类型，并且在某些情况下具有稳定性。但它需要额外的空间，而且对于位数较大的数据可能不太实用。

```
def radix_sort(arr):
    max_num = max(arr)
    max_length = len(str(max_num))
    buckets = [[] for _ in range(10)]
    for digit in range(max_length):
        for num in arr:
            bucket_idx = (num // 10**digit) % 10
            buckets[bucket_idx].append(num)
        arr = [num for bucket in buckets for num in bucket]
        buckets = [[] for _ in range(10)]
    return arr
```

## 3、重建二叉树

<http://cs101.openjudge.cn/dsapre/02255/>

最经典的题目，通过两种遍历来得到另一种遍历。

```
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

def build(prefix, infix):
```

```

if len(prefix) == 0:
    return None
v = prefix[0]
root = TreeNode(v)
idx = infix.index(v)
root.left = build(prefix[1:idx+1], infix[:idx])
root.right = build(prefix[idx+1:], infix[idx+1:])
return root

def postfix(root):
    if root is None:
        return ''
    return postfix(root.left)+postfix(root.right)+root.val

while True:
    try:
        prefix, infix = input().split()
        print(postfix(build(prefix, infix)))
    except EOFError:
        break

```

## 4、四分树

<http://cs101.openjudge.cn/practice/01610/>

这次树的子节点大于二，因此我们用列表来存储。

```

class TreeNode:
    def __init__(self, val):
        self.val = val
        self.children = []

def build(n, matrix):
    check = sum(sum(row) for row in matrix)
    if check == 0:
        return TreeNode('00')
    elif check == n**2:
        return TreeNode('01')
    else:
        a = [matrix[i][:n//2] for i in range(n//2)]
        b = [matrix[i][n//2:] for i in range(n//2)]
        c = [matrix[i][:n//2] for i in range(n//2, n)]
        d = [matrix[i][n//2:] for i in range(n//2, n)]
        root = TreeNode('1')
        for p in [a, b, c, d]:
            root.children.append(build(n//2, p))
        return root

def get(root):
    result = ''
    queue = [root]
    while queue:
        node = queue.pop(0)
        result += node.val

```

```

        queue += node.children
    return result

for _ in range(int(input())):
    n = int(input())
    matrix = [list(map(int, input().split())) for _ in range(n)]
    ans = get(build(n, matrix))
    p = int(ans, 2)
    print(hex(p)[2:].upper())

```

## 5、表达式·表达式树·表达式求值

<http://cs101.openjudge.cn/practice/05430/>

有一说一，这道题能顺着做下来，感觉树这方面就没啥问题了？首先使用栈进行中序转后序，然后建树，之后需要注意树的打印，最后再进行一个带入求值。

```

class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

def trans(infix):
    stack = []
    result = []
    operator = {'+': 1, '-': 1, '*': 2, '/': 2}
    for token in infix:
        if token.isalpha():
            result.append(token)
        elif token == '(':
            stack.append(token)
        elif token == ')':
            while stack and stack[-1] != '(':
                result.append(stack.pop())
            stack.pop()
        else:
            while stack and stack[-1] in operator and operator[token] <= operator[stack[-1]]:
                result.append(stack.pop())
            stack.append(token)
    while stack:
        result.append(stack.pop())
    return result

def build(postfix):
    stack = []
    for char in postfix:
        if char in '+-*/':
            node = TreeNode(char)
            node.right = stack.pop()
            node.left = stack.pop()
        else:
            node = TreeNode(char)
            stack.append(node)
    return stack.pop()

```

```

        stack.append(node)
    return stack[0]

def depth(root):
    if root is None:
        return 0
    return max(depth(root.left), depth(root.right))+1

def Print(root, d):
    if d == 0:
        return root.val
    graph = [' '*(2**d-1)+root.val+' '*(2**d-1)]
    graph.append(' '*(2**d-2)+'/' if root.left is not None else ' ')
    graph.append(' '+'(\\' if root.right is not None else ' ')+ ' '*(2**d-2))
    d -= 1
    l = Print(root.left, d) if root.left is not None else [' '*(2**d-1)]*(2**d+1)
    r = Print(root.right, d) if root.right is not None else [' '*(2**d-1)]*(2**d+1)
    for i in range(2*d+1):
        graph.append(l[i]+' '+r[i])
    return graph

def cal(root):
    if root.val.isalpha():
        return my_dict[root.val]
    else:
        lv = cal(root.left)
        rv = cal(root.right)
        return int(eval(str(lv)+root.val+str(rv)))

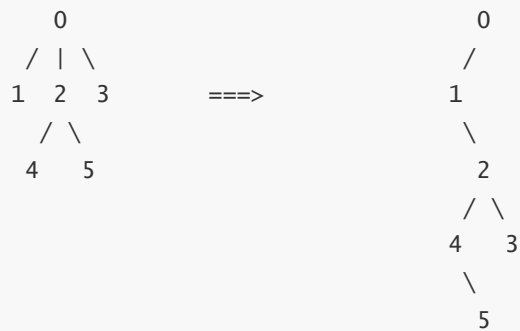
infix = list(input())
postfix = trans(infix)
root = build(postfix)
my_dict = {}
for _ in range(int(input())):
    a, b = input().split()
    my_dict[a] = b
print(''.join(postfix))
d = depth(root)
result = Print(root, d-1)
for part in result:
    print(part)
print(cal(root))

```

## 6、树的转换

我们都知道用“左儿子右兄弟”的方法可以将一棵一般的树转换为二叉树，如：





现在请你将一些一般的树用这种方法转换为二叉树，并输出转换前和转换后树的高度。

### 输入

输入是一个由“u”和“d”组成的字符串，表示一棵树的深度优先搜索信息。比如，dudduduudu可以用来表示上文中的左树，因为搜索过程为：0 Down to 1 Up to 0 Down to 2 Down to 4 Up to 2 Down to 5 Up to 2 Up to 0 Down to 3 Up to 0。你可以认为每棵树的结点数至少为2，并且不超过10000。

### 输出

按如下格式输出转换前和转换后树的高度：h1 => h2 其中，h1是转换前树的高度，h2是转换后树的高度。

dudduduudu

2 => 4

这个题不难，但是很好地诠释了什么叫做“左儿子右兄弟”，当时我理解了半天还没明白的东西hhh。

```

s = list(input())
a, b, h, nh = 0, 0, 0, 0
now = [0]
for char in s:
    if char == 'd':
        a, b = a+1, b+1
        now.append(a)
    else:
        a, b = now.pop(), b-1
    nh = max(nh, a)
    h = max(h, b)
print(f'{h} => {nh}')
  
```

## 层次遍历

- BFS

```

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

```

## 10、哈夫曼编码树

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。

1. 合并两个节点时，小的节点必须作为左子节点
2. 连接左子节点的边代表0,连接右子节点的边代表1

然后对输入的串进行编码或解码

### 输入

第一行是整数n，表示字符集有n个字符。接下来n行，每行是一个字符及其使用频率（权重）。字符都是英文字母。再接下来是若干行，有的是字母串，有的是01编码串。

### 输出

对输入中的字母串，输出该字符串的编码 对输入中的01串,将其解码，输出原始字符串

```

3
g 4
d 8
c 10
dc
110

```

```

110
dc

```

非常经典的一种算法！需要注意，我们之前的建树过程都是自上向下建立（先建立根节点，之后不断添加子树），这里我们自下向上，逐渐向上，“拼凑”出一棵完整的树。

```

import heapq

class TreeNode:
    def __init__(self, weight, val=None):
        self.weight = weight

```

```

        self.val = val
        self.left = None
        self.right = None
    def __lt__(self, other):
        if self.weight == other.weight:
            return self.val < other.val
        return self.weight < other.weight

def build(nodes):
    Q = []
    for p, char in nodes:
        heapq.heappush(Q, TreeNode(p, char))
    while len(Q) > 1:
        l = heapq.heappop(Q)
        r = heapq.heappop(Q)
        merge = TreeNode(l.weight + r.weight)
        merge.left = l
        merge.right = r
        heapq.heappush(Q, merge)
    return Q[0]

def encode(root):
    codes = {}
    def operate(node, code):
        if node.val is not None:
            codes[node.val] = code
        else:
            operate(node.left, code+'0')
            operate(node.right, code+'1')
    operate(root, '')
    return codes

def encoding(codes, s):
    ans = ''
    for p in s:
        ans += codes[p]
    return ans

def decoding(root, s):
    ans = ''
    node = root
    for p in s:
        if p == '0':
            node = node.left
        else:
            node = node.right
        if node.val is not None:
            ans += node.val
            node = root
    return ans

nodes = []
for _ in range(int(input())):
    a, b = input().split()
    nodes.append([int(b), a])
root = build(nodes)

```

```
codes = encode(root)
while True:
    try:
        s = list(input())
        if s[0].isdigit():
            print(decoding(root, s))
        else:
            print(encoding(codes, s))
    except EOFError:
        break
```