1. 泰波纳契数

设$\{T_n\}$是泰波纳契数列的第n项，令$T_0=0,T_1=1,T_2=1,T_n=T_{n-1}+T_{n-2}+T_{n-3}$.

时间复杂度：$O(n)$

```
n = int(input())

s = [0 for i in range(1000)]
s[0] = 0
s[1] = 1
s[2] = 1

for i in range(3, n + 1):
    s[i] = s[i-1] + s[i-2] + s[i-3]

print(s[n])
```

2. 整人的提词本

将字符串内容依次入栈，遇到右括号时不入栈，并依次弹出栈中的字符和一个左括号，再将这些字符反序入栈即可.

时间复杂度：$O(n)$

```
class Stack:
    def __init__(self):
        self.lst = []
        self.len = 0
    def push(self, obj):
        self.lst.append(obj)
        self.len += 1
    def pop(self):
        self.lst.pop()
        self.len -= 1
    def top(self):
        return self.lst[self.len - 1]
    def __len__(self):
        return len(self.lst)
    def empty(self):
        return (self.lst == [])

class Queue:
    def __init__(self):
        self.lst = []
        self.head = 0
    def push(self, obj):
```

```
        self.lst.append(obj)
    def pop(self):
        self.head += 1
    def top(self):  #应该叫 front
        return self.lst[self.head]
    def empty(self):
        return (self.head == len(self.lst))

str = list(input())
s1 = Stack()

for i in range(len(str)):
    if str[i] != ')':
        s1.push(str[i])
    else:
        qtmp = Queue()
        while s1.top() != '(':
            ch = s1.top()
            qtmp.push(ch)
            s1.pop()
        s1.pop()   #删掉左括号
        while not qtmp.empty():
            s1.push(qtmp.top())
            qtmp.pop()

s2 = Stack()
while not s1.empty():
    s2.push(s1.top())
    s1.pop()
while not s2.empty():
    print(s2.top(), end = '')
    s2.pop()
```

3. 两座孤岛最短距离

两座孤岛可以任选一座作为起点，计算另一座孤岛到这座孤岛的距离（答案为最小距离-1）.

第一步：任选一座孤岛并将其标记为“起点”（代码中为“X”）；

第二步：从起点开始进行广度优先搜索，直到搜索到“1”的点为止.

时间复杂度：O(地图大小)

```
class Queue:
    def __init__(self):
        self.lst = []
        self.head = 0
    def push(self, obj):
        self.lst.append(obj)
    def pop(self):
        self.head += 1
    def top(self):
        return self.lst[self.head]
    def empty(self):
        return (self.head >= len(self.lst))
class Pos:
    def __init__(self, x, y):
        self.x = x
        self.y = y

n = int(input())
map = [None for i in range(n)]  #存储地图
for i in range(n):
    str = list(input())
    map[i] = str

visit = [[-1 for i in range(n)] for j in range(n)]  #存储距离，未访问则为-1
queue = Queue()
dx = [1, 0, -1, 0]
dy = [0, 1, 0, -1]

def dfs_mark(x, y):  #选择一座岛屿作为起点并标记为“X”
    map[x][y] = 'X'
    visit[x][y] = 0
    for i in range(4):
        newx = x + dx[i]
        newy = y + dy[i]
        if newx < n and newx >= 0 and newy < n and newy >= 0:
            if map[newx][newy] == '1' and visit[newx][newy] == -1:
                dfs_mark(newx, newy)
            elif map[newx][newy] == '0' and visit[newx][newy] == -1:
                queue.push(Pos(newx, newy))
                visit[newx][newy] = 1    #与起点距离为 1 的点进入搜索队列

breakflag = 0
```

```
for i in range(n):
    for j in range(n):
        if map[i][j] == '1':
            dfs_mark(i, j)
            breakflag = 1
            break
    if breakflag:
        break

breakflag = 0
while not queue.empty(): #从距离为 1 的点开始计算距离，使用队列进行广度优先搜索
    tmp = queue.top()
    x = tmp.x
    y = tmp.y
    queue.pop()
    for i in range(4):
        newx = x + dx[i]
        newy = y + dy[i]
        if newx < n and newx >= 0 and newy < n and newy >= 0 and visit[newx][newy] == -1:
            queue.push(Pos(newx, newy))
            visit[newx][newy] = visit[x][y] + 1
            if(map[newx][newy] == '1'):
                print(visit[newx][newy] - 1)
                breakflag = 1
                break
    if breakflag:
        break
```

4. 满足合法工时的最少人数

对工人数量从 1 到（单个任务最长时间）进行二分查找，在给定工人数量时判断是否可行.

时间复杂度：$O(n\log m)$，n是任务个数，m是单个任务最长时间

```
work = input().split(',')
t = int(input())

maxtasktime = 0
for i in range(len(work)):
    work[i] = int(work[i])
    maxtasktime = max(maxtasktime, work[i])
```

```
def judge(workers: int) -> bool:
    total_time = 0
    for i in range(len(work)):
        total_time += (work[i] - 1) // workers + 1 #向上取整除法
if total_time > t:
        return False
    else:
        return True

left = 1
right = maxtasktime + 1
res = right

# 对员工数量二分查找
while left <= right:
    mid = (left + right) >> 1
    tmp = judge(mid)
    if tmp:
        res = mid
        right = mid - 1
    else:
        left = mid + 1
print(res)
```

5. 对子数列做 XOR 运算

异或运算xor满足x xor $x=0$，x xor $0=x$，因此对同一个数异或两次相当于没有进行运算．即对$v[L]-v[R]$中的数做异或相当于对$v[0]-v[R]$做异或再对$v[0]-v[L-1]$做异或．只需要提前计算出对$v[0]-v[i]$做异或的值．

时间复杂度：$O(n+m)$，n是数组长度，m是查询次数

```
num = list(input().split(' '))

prefix = []   #prefix[i]代表 num[0] xor num[1] xor... xor num[i]

for i in range(len(num)):
    num[i] = int(num[i])
    if i == 0:
        prefix.append(num[0])
    else:
        prefix.append(prefix[i - 1] ^ num[i])

for i in range(10000):
    query = list(input().split(' '))
```

```
    l = int(query[0])
    r = int(query[1])
    if l == 0:
        print(prefix[r])
    else:
        print(prefix[r] ^ prefix[l-1])
```

6. 土豪购物

土豪可以买连续的一段商品，或者连续的两段商品，且这两段商品中间恰好空出 1 个.

先考虑只买一段连续商品的情况：定义$dp[i]$为土豪买连续的一段商品，且以第i个商品结尾时的商品价值最大值，则有递推式$dp[i] = \max\{0, dp[i-1]\} + price[i]$. 所有$dp[i]$中的最大值就是土豪买一段连续商品可能买到的最大价值.

再考虑买两段商品中间空出一个的情况：若空出第i个商品，那么买的第一段商品以第$i-1$个结尾（最大价值为$dp[i-1]$），第二段商品以第$i+1$个开头. 因此只需要求买一段以第$i+1$个开头的商品时的最大价值.

可以类似定义$dp2[i]$为土豪买连续的一段商品且以第i个商品开头时的商品价值最大值. 递推式为$dp2[i] = \max\{0, dp2[i+1]\} + price[i]$.

时间复杂度：$O(n)$

```
price = input().split(',')
for i in range(len(price)):
    price[i] = int(price[i])

n = len(price)
minus_inf = -9999999999

dp = [minus_inf for i in range(n)]
res = minus_inf
for i in range(n):
    if i == 0:
        dp[0] = price[0]
    else:
        dp[i] = max(price[i], dp[i-1] + price[i])
    res = max(res, dp[i])

dp2 = [minus_inf for i in range(n)]
for i in range(n - 1, -1, -1):
    if i == n - 1:
        dp2[n-1] = price[n-1]
    else:
        dp2[i] = max(price[i], dp2[i+1] + price[i])
```

```
    res = max(res, dp2[i])

for i in range(1, n - 1):  #空出第 i 个商品
    res = max(res, dp[i-1] + dp2[i+1])

print(res)
```