如何用Python参加算法竞赛

前言

本文适合有一定c++基础且初步了解Python，并想开发自己第二竞赛用语言的人群阅读。

本文仅介绍Python3，更低版本Python请自行了解。

Python的优点在于在应对代码编写简单的题目时，在无电子板子的赛场环境可以一定缩短codeing时间。但在面对代码编写要求较高、时间限制较紧的情况，并**无法取代c++**。因此c++仍然是打算法竞赛的第一选择。

Python的适用场合有如下几种：

1. 代码简单的，如一些思维题、构造题等
2. 字符串处理，Python提供的字符串处理函数比c++丰富许多。
3. 对拍器和数据生成器

注一：

python与其他语言不同的一点在于，同样的算法，用标准库的永远比自己写的速度快。因为标准库算法大部分是用C语言写的，python由于语言限制永远到达不了C的速度，也即标准库的速度。

注二：

python的官方中文文档比起一些别的语言已经算非常好了，如果看别人代码或者题解有不懂的函数或容器，可以直接搜官方文档的对应内容。本文对于一些内容也不会讲太细，可以直接搜官方文档看

注三：

python语言并不适合递归算法，因为其递归深度，语言自身就有限制，就算去除限制，其也会开辟大量空间。蓝桥杯也会依据语言出题，python组用递归算法的题很少很少。所以平时应多注意迭代算法的积累，以及递归算法转迭代算法的方式

Python基本数据类型

python的基本数据类型有六种，数值、字符串、列表、元组、字典、集合，常用的有int，str，bool，float，list，tuple，dict，set

int（整数）

没有长度限制，大数字乘法复杂度\(O(nlogn)\)（补：因为当int达到高精度时，内部会使用FFT算法加速多项式乘法），非常方便。

float（浮点数）

大概注意一下精度就行，\([2.2250738585072014e-308, 1.7976931348623157e+308]\)

bool（布尔）

有`True`，`False`两个值（注意大小写）

list（列表）

最常用的数据类型之一，可当成C++中数组。

由于python中没有C++的栈，该结构可作栈使用

```
a = list(map(int, input().split()))#将读入创建为一个整数list
len(a)#返回a的长度
a[i]#访问a中第i个元素
```

```
a[-i]#访问a中倒数第i个元素
a[1:5]#列表切片，返回[a_1,a_2,a_3,a_4]
#列表切片还可以写成
a[:] # 常用于拷贝
a[::-1] # 翻转
a[::2]
a[:5]
a[1:]
a[1:10:2]
#三维分别代码起点，终点（左闭右开），步长。可以参照下文中的range函数介绍一起理解。
a.sort()#升序排序
a.sort(reversed = True)#降序排序
```

还可以append添加元素，pop删除尾部最后一个元素（O(1)），删除中间元素（最坏O(n)），count计数（O(n)），index定位元素（O(n)），并且重载了加法（即同维数组连接）

加法示例

```
a = [1, 2, 3]
b = [4, 5]
print(a + b)
# [1, 2, 3, 4, 5]
```

tuple（元组）

和list差不多，初始化用括号

```
a = (1, 2, 3)
```

支持list的很多操作，唯独不能对一个tuple自身进行修改

所以 dict 因为要求 key 值不可变，当想对插入一个 （list，int）键值对时，必须将 list 转为 tuple

str（字符串）

和 C++ 中字符串类似，但是无法修改其中字符，因此经常用如下方法转换为一个list再进行操作。

```
s = '1323'
s = list(s)
```

重载了加法，加法即是字符串连接

同时也有count，index，find等函数

多了一个C++没有，很好用的 split

split

默认以不可见字符进行分割，也可传入固定字符，以固定字符进行分割字符串，将子串存入list中

```
s = 'hello  my name is caidd!!!'
print(s.split())  # ['hello', 'my', 'name', 'is', 'caidd!!!']
print(s.split('my'))  # ['hello  ', ' name is caidd!!!']
```

dict（字典）

相当于C++的map容器，但是其内部是哈希表实现的，无序，大部分操作都是 \(O(1)\) 的

需要注意的是，其通过下标访问一个不存在的key时，会报异常，这点可以用后文的defaultdict解决

```
d = {'str': 1, 'int': 2, 'float': 3}
print(d['str'], d['int'], d['float'])
# 1, 2, 3
d['str'] += 1
print(d['str'])
# 2
d['list'] = 4
d.pop('str')  # 删除键值
for i in d:  # 等价于d.keys()
    print(i)
'''
int
float
list
'''
for i in d.values():
    print(i)
'''
2
3
4
'''
for k, v in d.items():
    print(k, v)
'''
int 2
float 3
list 4
'''
if 'int' in d:  # 查询
    print('yes')
# d['str'] += 1 会报错，抛异常
```

set（集合）

set是一个数学意义集合（不可重，无序）的程序实现（内部由哈希表实现）。支持各种集合操作。

```
s = set()  # 创建一个空集合
s.add(1)  # 加入元素 O(1)
s.remove(1)  # 删除1 O(1)
if 1 in s:  # 查找1是否在集合内 O(1)
```

该类型重载了位运算，可以灵活的求集合交，并，补

```
a = {1, 2, 3}
b = {3, 4, 5}
print(a & b)  # 交集 {3}
print(a | b)  # 并集 {1, 2, 3, 4, 5}
print(a - b)  # 差集 {1, 2}
print(a ^ b)  # 对称差集 {1, 2, 4, 5}
```

其可迭代，故也可以

```
for i in s:
```

这样遍历

类型转换

类型可以函数化，并互相转换

如最常用的 int，str 转换

```
a = 120
a = str(a) # 字符串 120
a = a[::-1] # 字符串 021
a = int(a) # 整型 21
```

还有 list 与 set 互相转换以实现去重操作

Python基础语法

本部分，我将直接列出c++基础语法，并给出在Python上的等价替代。

主函数体

c++main函数基础结构：

```
int main() {
    return 0;
}
```

Python主函数并不是必要的，完全直接在空文件编写代码，如：

```
int main() {
    for (int i = 0;i < 10;i ++) {
        printf("%d\n", i);
    }
    return 0;
}
```

在Python中可以直接写为：

```
for i in range(10):
    print(i)
```

当然，如果实在不习惯，想要和c++风格更加类似，可以按如下写法：

```
if __name__ == "__main__":
    for i in range(10):
        print(i)
```

第一行`if __name__ == "__main__":`的意思：字面上，这是一个if判断，而`__name__`是一个内置的特殊变量，当我们希望将一个python模块（就是写好的py文件）导入其他python模块时，就只会执行`if __name__ == "__main__":`的语句，比如：

```
print(123)
if __name__ == "__main__":
    for i in range(10):
        print(i)
```

`print(123)`就不会被执行。

但对于算法竞赛来说，一般不需要多模块操作，该写法只是为了更好的向c++代码风格靠拢。

运算符

python中新添 ** 乘方运算符

无自增 ++ 运算符，无自减运算符 --

条件连接符

python 中均以英文表示条件连接，可读性好些

python	**C**
and	&&
or	
not	!

基础语句

循环：

```
for i in range(n):
    # do something
i = 0
while i < n:
    # do something
```

需要注意的是，`for i in range(n):`实际上是对`range`生成的对象遍历，可以简单理解为对一个\([0,1,2...,n-3,n-2,n-1]\)的列表遍历，因此我们在循环中修改\(i\)并不会改变之后的循环。比如：

```
for i in range(n):
    i += 1
```

并不会让循环按照\(0 \rightarrow2\rightarrow4\cdots\)的顺序进行。

可见，Python中的for循环不如c++中的灵活，因此while的使用频率大大提高了。

关于range函数：

`range(start, end, offset)`

三个参数分别代表，起点，终点和步长

range返回的区间是左闭右开的，也就是\([start,end)\)

第1和第3个参数可以缺省。

给几个使用实例

```
for i in range(n): #遍历[0,n)
for i in range(1, n): #遍历[1,n)
for i in range(0, n, 2):#遍历[0,n)，步长为2
for i in range(n - 1, -1, -1):#倒遍历[0,n - 1)
```

分支：

和c++差别不大，给个实例：

```
if x % 3 == 0:
    # do something
elif x % 3 == 1:
    # do something
else:
    # do something
```

可以发现区别只在于Python将`else if`合并为了`elif`。

但因为引入了`in`这个关键字，有了一些更加方便的用法

```
# a是一个list x是一个int
if x not in a:
    # do something
#可以发现不用再用for逐个判断a是否有x，直接可以用in关键字就可以判断了，
#但复杂度仍然是O(n)并没变小
#不止是list，一些可迭代的高级数据类型也支持这种使用方法
```

函数

Python中函数定义方法很简单：

```
def func(x):
    # do something
    return
```

Python允许函数定义出现在函数内部

```
def func1():
    print(1)
    def func2():
        print(2)
    func2()
func1()
```

Output：

```
1

2
```

Python允许函数返回多个值

```
def func(x):
    return x + 1, x + 2
x, y = func(10)
print(x, y)
```

Output:

```
11 12
```

Python中函数内部如果想**修改**外部数字变量，需要使用`nonlocal`或者`global`关键字

```
t = 10
def func(x):
    global t
    t += 1
    return x + 1, x + 2
x, y = func(10)
print(x, y)
```

如果将`global t`注释掉程序会报错。

如果想要使用的变量不是被定义在全局区，而是某个函数体内部则要使用`nonlocal`关键字

```
def work():
    t = 10
    def func(x):
        nonlocal t
        t += 1
        return x + 1, x + 2
    x, y = func(10)
    print(x, y)
work()
```

“头文件”

Python除内置库外，有一些功能需要手动导入模块，有如下几种方法

```
import math #导入math库
from math import *#导入math库下所有变量和函数
from math import sin, cos#导入math库下的sin，cos函数
```

第一种方法和后两种调用时有所区别

第一种：

```
import math #导入math库
print(math.sin(10))
```

后两种：

```
from math import sin, cos#导入math库下的sin，cos函数
print(sin(10))
```

可见区别就是使用时是否要明确库名，一般在算法竞赛中为了代码简洁，推荐使用后者，但如果要使用`from math import *`的方法，将存在一定变量名冲突的风险。

因此，更推荐部分导入。

“宏定义”

Python中没有宏定义，但有替代可以缩短一定码量。如下：

```
def abcdefgasdas(x):
    print(x)

abcdefgasdas(10)
func = abcdefgasdas
func(10)
```

我们定义一个及其鬼畜的函数`abcdefgasdas(x)`并在之后给其“取别名”为`func`，调用`func`就等价调用了`abcdefgasdas`从而在某些要调用内置函数时，起一个更短的名字，降低码量。

附：之所以可以这样是因为python之中，一切皆对象，故可以一切皆变量（包括函数）

输入输出

输入

Python中的读入和C++还是有很大不同的，需要一定时间适应。

Python读入时都是调用`input()`其将返回标准输入中的一行数据（不包括末尾的`\n`），其返回的类型统一为字符串，因此还要对其进行变量类型转换。

在算法竞赛中，读入一行数字一般分为可数的几个整数，和一个很长的数组两种形式，我举例说明如何读入：

Input

> 5
>
> 1 3
>
> 2 4
>
> 2 5
>
> 3 2
>
> 1 2 3 4 5

```
# 上面是常用的读入一棵树，并给点赋权值的一种输入格式
# 读入方法如下
n = int(input()) # int()函数将input()获取的一行字符串转换为整数
for i in range(n - 1):
    x, y = map(int, input().split())# 因为一行有多个整数，首先对input()获取的字符串
    #用split()函数切割，该方法将返回一个以' '和'\n'为分隔符切片后的字符串列表
    #用map()函数就可以将这个列表中字符串数据全转换为整数，并赋给左值
    # do something
w = list(map(int, input().split()))
#w因为类型是列表，因此还要多一个list()，这是一个构造方法，将map对象转为list数据类型
```

字符串读入方法就很简单了

```
s = input()
```

读入优化

Python中的**真**读入优化需要码量巨大，在正式比赛中并不常用，但仍然可以使用如下方法提高一定的读入效率。

```
import sys
input = sys.stdin.readline
```

将其放在Python文件头部即可。可以提高一定效率，但没有c++那般明显。

PS：使用该方法行末的`\n`将不会被忽略，在读入字符串数据时尤其要注意

输出

使用`print()`进行输出

```
print(10) # 输出10并换行
print(1,2,3) # 输出1 2 3并在末尾换行
print(1, end = ' ') # 输出1并再末尾输出一个' '
```

将输出数据类型转换为字符串，并且将所有中间输出全部加入到此字符串中，最后一次性输出，有时可以提高一定效率，但并不明显。

文件读写

由于前几年蓝桥杯 C++ 组有需要文件读写的情况，所以在此稍微讲解常用用法，具体可见标准库

r 以只读方式打开文件。文件的指针将会放在文件的开头。

```
file = open(r'input.txt', 'r')  # 字符串前的r表示后续紧跟的字符串不进行转义，即原始字符串
a = file.readline()  # 后面调用这个方法即可如input函数一样使用
```

w+ 打开一个文件用于读写。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。

```
file = open(r'input.txt', 'w+')
file.write('hello\n')  # write方法不如print会自动换行，若要换行需要手动加换行符
```

Python库和函数

介绍一些常用的库和函数，着重和C++的STL对比。

简单函数

```
# 下列函数可以传入数字或者可迭代数据类型，常传入list
# 以list为例说明功能
max()  # 返回list的最大值，也可传入多个单体值返回最大，如max(1, 2, 3, 4)
min()  # 返回list的最小值，同max用法
sum()  # 返回list的和
any()  # 如果list中有True，返回True
all()  # 如果list中全部为True，返回True
```

数学函数

```
# 内置的，求幂取模用的是快速幂算法，O(log(n))，n为指数
pow()  # 该函数有一个内置的，math库里也有一个，需要注意的是内置的更好，支持求幂取模，求逆元（带膜，指数取-1即可）
abs()  # 绝对值
```

math 库中的

```
factorial()  # 阶乘值
ceil()       # 向上取整
floor()      # 向下取整
comb()       # 求组合值
perm()       # 求排列值
gcd()        # 最大公约数
lcm()        # 最小公倍数
exp()        # e ** x 即 e 的 x 次方
log()        # 以e为底的对数值
log2()       # 以2为底的对数值
log10()      # 以10为底的对数值
sqrt()       # 求平方根
pi           # 圆周率的常量值，精度很高
e            # 自然常数
             # 三角函数一堆，不说了，自己查
```

sort

自定义排序，Python不如C++灵活。首先它只可以对整个序列排序，而无法对部分序列排序，其次自定义方法不如C++的lamda表达式方便。

```
#自定义排序方法
import functools
a = [1,2,4,3,5]
def compare_personal(x,y):
    if x > y: return -1 #如果x>y，让x在y前面，返回-1
    return 1#否则让x在后面，返回1
a.sort(key = functools.cmp_to_key(compare_personal))
print(a)# 输出为[5, 4, 4, 3, 1]
#当然如果只是想得到降序序列，有很简单的方法:
a.sort(reversed = True)
```

当然，很多时候我们只是想让它根据列表的某一维度排序，这时也可以用python的lambda表达式，代码量少很多。

```
#a = [[6,5],[3,4],[5,6]]
a.sort(key = lambda A: (A[1],-A[0]))#按第1维升序，第二维降序排列
```

`sort`是没有返回值的原地排序，如果我们希望获取到这个排序后列表，且不想改变原来的列表时，可以用`sorted`函数。自定义

```
b = sorted(a,  key = lambda A: (A[1],-A[0]))
```

map （映射）

通过一个函数，对可迭代对象的所有值对象进行修改（创建副本，非原地修改）

```
a = [1, -2, 3, -4, 5, -6, 7]
print(list(map(abs, a)))
# [1, 2, 3, 4, 5, 6, 7]
```

collections

这个库里常用的有deque，defaultdict，Counter

deque

deque对标c++中的双端队列，可以快速在头尾弹出和加入元素。速度比普通队列快，因此在需要队列的场合，统一使用deque

```
q = deque()          # 创建空队列
q = deque([1,2,3])   # 创建元素为1,2,3的队列
q.append(1)          # 尾插
q.appendleft(1)      # 头插
q.extend([1,2])      # 尾插可迭代对象
q.extendleft([1,2]) # 头插可迭代对象
q.pop()              # 将队列尾部的数据弹出，并作为返回值。
q.popleft()          # 将队列头部的数据弹出，并作为返回值。
```

defaultdict

即当键值不存在时，有默认返回值的 dict，其余操作差不多

```
from collections import defaultdict

a = defaultdict(int)  # 默认返回int类型的默认值，若写list，则键值不存在时返回空列表
print(a['caidd'])     # 0，此时创建了一个键值对
if a['rmx']:          # 此时创建了一个键值对
    print('yes')
print(a)              # defaultdict(<class 'int'>, {'caidd': 0, 'rmx': 0})
```

由于defaultdict对于key下标运算返回的是value的引用，若不存在则只能创建一个对象再返回，因此通过下标判断存在与否的方式不推荐，建议使用 in 来判断是否存在

用下标运算来插入与修改

Counter

调用Counter函数计数，返回一个 defaultdict(int)

```
from collections import Counter

a = [1, 1, 1, 2, 3, 3, 5]
cnt = Counter(a)
print(cnt)     # Counter({1: 3, 3: 2, 2: 1, 5: 1})
print(cnt[1])  # 3
cnt[2] += 1
print(cnt[2])  # 2
print(cnt[6])  # 0
```

PS：dict、set和其子类都是用的hash实现，而不是c++中的红黑树，因此没有自动排序功能，目前没有太好的替代。

如果是非标准库的话，有Sorted Container，比较好用。让我们祝福它早日进标准库

heapq（优先队列）

Python中其实有优先队列，但是速度没有heapq快，因此用heapq代替。

heapq提供函数对一个list进行原地的小根堆的维护。

```
from heapq import heapify, heappop, heappush

q = [4, 2, 1, 4, 5]
heapify(q)  # 一次堆排序
heappush(q, 3)  # 放入1，并进行一次堆排序
print(heappop(q))  # 弹出堆头 3
```

heapq并没有提供方便的重载为大根堆的方法，如果想使用大根堆，一般的技巧是加入值取负值，弹出后再恢复。

基础操作差不多这么多，还有一些其他功能可自行了解。

zip、enumerate函数

这两个函数，都是使得枚举进一步简单化的函数。

zip函数可以同时访问不同list的同偏移量的元素

```
a = [1,2,3,4,5]
b = [5,4,3,2,1]
for x, y in zip(a, b):
    print(x, y)
```

Output:

```
1 5
2 4
3 3
4 2
5 1
```

enumerate则是在访问list中元素时，同时给出元素的下标，下标默认从0开始。

```
a = [1,2,3,4,5]
for i, x in enumerate(a):
    print(i, x)
```

Output:

```
0 1
1 2
2 3
3 4
4 5
```

itertools

这个库里的大多函数方法，都是返回一个可迭代对象，因此若要变成list还需list()转换

permutations、combinations

permutations，combinations分别是返回一个可迭代对象（一般是list）的所有排列和组合。使用时需要导入`itertools`模块

用法如下：

```
from itertools import permutations, combinations
a = [1, 2, 3]
for p in permutations(a):
    print(p)
for p in combinations(a, 2):
    print(p)
```

Output:

```
(1, 2, 3)
(1, 3, 2)
(2, 1, 3)
(2, 3, 1)
(3, 1, 2)
(3, 2, 1)
(1, 2)
(1, 3)
(2, 3)
```

accumulate（累计）

```
from itertools import accumulate

a = [2, 3, 1, 6, 5, 4]
b = list(accumulate(a))
print(b)  # [2, 5, 6, 12, 17, 21]
c = list(accumulate(a, initial=0))  # 实际情况这种更常用，用作前缀和
print(c)  # [0, 2, 5, 6, 12, 17, 21]
d = list(accumulate(a, func=lambda x, y: max(x, y), initial=0))
print(d)  # [0, 2, 3, 3, 6, 6, 6]
```

pairwise（成对遍历）

```
from itertools import pairwise

a = [2, 3, 1, 6, 5, 4]
for k, v in pairwise(a):
    print(k, v)
'''
2 3
3 1
1 6
6 5
5 4
'''
```

functools

reduce （聚合）

其位于functools库里面

对于可迭代对象进行使用，将可迭代对象里的所有值对象，两两聚合，最后返回一个值对象

```
import operator
from functools import reduce

a = [1, 2, 3, 4, 5]
print(reduce(operator.add, a))  # 15，同 sum
print(reduce(lambda x, y: max(x, y), a))  # 5，同 max
a = [[1, 2, 3, 4], [5, 6]]
print(reduce(operator.add, a))  # [1, 2, 3, 4, 5, 6] # operator调用的是加操作符，此处sum不可用
```

Python小技巧

交换

没有swap函数，但可以这么写，也很方便

```
a, b = b, a
```

列表（其实可以是任何可以迭代对象）解析式

创建列表时，可以用如下方法简化代码

```
a = [x for x in range(6)]
#生成一个[1,2,3,4,5]的列表
adj = [[] for _ in range(n)]
#生成一个二维，空列表
#等同于 vector<vector<int>> adj(n);
```

列表解析式中中括号中返回的是一个可迭代对象，这个在很多函数中都是可接受的数据类型。结合上面说的“聚合函数”，就可以这样写

```
# 求列表全体偶数和
sum(x for x in a if x % 2 == 0)
#判断列表是否严格单调递增
all(a[i - 1] < a[i] for i in range(1, n))
```

多维数组

很可惜，python自身没有天然支持固定长度的多维数组（即如C++的`int a[5][5][5];`这样的），需要numpy才能很好的使用

但是仍然可以创建，方式如下

```
a = [[0] * 2 for i in range(4)]
```

这是创建了一个 4 * 2 的 二维数组

有人可能疑惑，为什么不能这样创

```
a = [[0] * 2] * 4
```

这是后果

```
a = [[0] * 2] * 4
a[0][0] = 1
print(a)
'''
[[1, 0], [1, 0], [1, 0], [1, 0]]
'''
```

第一列全部被修改

最外层的乘4，相当于只是创建了四个引用，引用的都是一个[0] * 2

所以不行

三目表达式

和c++中的三目表达式`?:`类似，Python中也有何其类似的语法。

```
x = 10
y = 20
a = x if x == y else y
print(a)
# 输出为20，返回的是在判断x==y为假后，返回了y并赋值给a
```

快速创建一个字典

```
mp = {}
mp = dict()
mp = {a : b for a, b in lst}#将列表元素构造为键值对放入字典
```

在由任意正多边形铺设的平面网格中，计算任意两个单元格之间的最短步数。

根据平面填充原理，**只有这三种正多边形**可以完全铺满二维平面：

1. **正三角形**（每点6个相邻三角形）
2. **正方形**（四邻格）
3. **正六边形**（六邻格）

六边形

六边形格的“距离”类似于一种扭曲的曼哈顿距离

有交叉角度时，需要走对角方向，步数是 `max(dx, dy)`

否则是正常横纵之和

Distance=max(|x1−x2|,|y1−y2|,|(x1+y1)−(x2+y2)|)

如果 `x1 < x2 且 y1 > y2`（或反过来），两个坐标方向“冲突”，必须往对角线方向走，不能分开走，此时距离是 `max(dx, dy)`

否则可以“分别在 x 和 y 上”走，距离是 `dx + dy`

```
#include <bits/stdc++.h>
#define ll long long
using namespace std;

const ll N = 2e9;  // 枚举时的上限（最大编号）

// 六边形六个方向：右下 → 左下 → 左 → 左上 → 右上 → 右
int nx[6] = {1, 0, -1, -1, 0, 1};  // 每个方向 x 坐标变化量
int ny[6] = {-1, -1, 0, 1, 1, 0};  // 每个方向 y 坐标变化量（注意：y 是“斜向下”为正）

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0), cout.tie(0);

    vector<ll> cir;  // cir[i] 表示第 i 圈的起始编号

    // 构造圈起始编号数组：第 i 圈起点是 3 * i * (i - 1) + 1
    // 构造直到超过 2e9（最大编号范围）
    for (ll i = 1; 3 * i * (i - 1) + 1 <= N; i++) {
        cir.push_back(3 * i * (i - 1) + 1);
    }

    int t;
    cin >> t;  // 读入测试组数

    while (t--) {
```

```
        int x, y;
        cin >> x >> y;  // 输入两个编号（要计算最短路径的两个点）

        // ---- 第一个点编号 x 映射到坐标 (x1, y1) ----
        int p = lower_bound(cir.begin(), cir.end(), x) - cir.begin() + 1;  // 找出 x 属于第几圈（1-based）
        int x1 = 0, y1 = p - 1;  // 初始坐标：每圈起点都是在竖直方向上的 (0, p-1)

        int num = cir[p - 1] - x;  // x 到该圈起点的偏移（需反向走 num 步）
        for (int i = 0; i < 6 && num; i++) {
            int cnt = min(p - 1, num);  // 每个方向最多能走 p-1 步（该圈边长）
            num -= cnt;
            x1 += nx[i] * cnt;
            y1 += ny[i] * cnt;
        }

        // ---- 第二个点编号 y 映射到坐标 (x2, y2) ----
        p = lower_bound(cir.begin(), cir.end(), y) - cir.begin() + 1;
        int x2 = 0, y2 = p - 1;

        num = cir[p - 1] - y;
        for (int i = 0; i < 6 && num; i++) {
            int cnt = min(p - 1, num);
            num -= cnt;
            x2 += nx[i] * cnt;
            y2 += ny[i] * cnt;
        }

        // ---- 计算最短路径 ----
        int ans = 1e9;

        // 六边形最短路径的规则（六边形网格上的距离）
        if ((x1 < x2 && y1 > y2) || (x1 > x2 && y1 < y2)) {
            // 如果 x 与 y 分布在对角线上，路径可压缩为 max(dx, dy)
            ans = max(abs(x1 - x2), abs(y1 - y2)) + 1;
        } else {
            // 正常曼哈顿距离（x 与 y 坐标不交叉）
            ans = abs(x1 - x2) + abs(y1 - y2) + 1;
        }

        cout << ans << '\n';  // 输出最短路径步数
    }

    return 0;
}
```

尺取法求连续区间不同的数

```
void add(int x) {
    if (++mp[a[x]] == 1)
        cnt++;
}
void del(int x) {
```

```
    if (--mp[a[x]] == 0)
        cnt--;
}

// 尺取法，f[i] 表示区间 [i, f[i]] 中有k个不同的数
for (int l = 1, r = 1; l <= n; l++) {
    while (cnt < k && r <= n)
        add(r++);
    if (cnt == k)
        f[l] = r - 1;
    else
        f[l] = n + 1;
    del(l);
}
```

莫队算法

处理问题必须离线，不能解决在线问题

莫队算法分为 普通莫队 树上莫队 带修改莫队

分块：我们将整个序列分成sqrt(n)块

然后对于所有询问排序，对于左端点不在一个块当中的两个待排序询问，我们按照L升序排序

对于左端点在一个块当中的两个待排序询问，我们按照r升序进行排序

然后把所有区间一次扫一遍就行了

```
// 添数
inline void add(int p){
    if(cnt[A[p]] == 0)cur++;
    cnt[A[p]]++;
}
// 删数
inline void del(int p){
    cnt[A[p]]--;
    if(cnt[A[p]] == 0)cur--;
}
```

区间移动

```
while(l > Q[i].l)add(--l);
while(l < Q[i].l)del(l++);
while(r < Q[i].r)add(++r);
while(r > Q[i].r)del(r--);
```

删数是先删后移，添数是先移后添

```cpp
#include<bits/stdc++.h>
using namespace std;
const int maxn = 30005, maxq = 200005, maxm = 1000005;
int sq;

struct query{
    int l, r, id;
    // 这里只需要知道每个元素归属哪个块，而块的大小都是sqrt(n)，所以可以直接用l/sq
    bool operator<(const query &o) const{
        if(l / sq != o.l / sq)
            return l < o.l;
        if(l / sq & 1)
            return r < o.r;
        return r > o.r;
        /*
          如果是奇数块，按照r的升序排序
          如果是偶数块，按照r的降序排序
        */
    }
}Q[maxq];

int A[maxn], ans[maxq], cnt[maxm], cur, l = 1, r = 0;

inline void add(int p){
    if(cnt[A[p]] == 0)cur++;
    cnt[A[p]]++;
}
inline void del(int p){
    cnt[A[p]]--;
    if(cnt[A[p]] == 0) cur--;
}

int main(){
    int n; cin >> n;
    sq = sqrt(n);
    for(int i = 1; i <= n; ++i) cin >> A[i];
    int q; cin >> q;
    for(int i = 0; i < q; ++i)
        cin >> Q[i].l >> Q[i].r, Q[i].id = i;
    sort(Q, Q + q);
    for(int i = 0; i < q; ++i){
        while(l > Q[i].l) add(--l);
        while(r < Q[i].r) add(++r);
        while(l < Q[i].l) del(l++);
        while(r > Q[i].r) del(r--);
        ans[Q[i].id] = cur;
    }
    for(int i = 0; i < q; ++i){
        cout << ans[i] << '\n';
    }
    return 0;
}
```

组合数

```
namespace CNM {
    const int N = 3e5 + 5; // 预处理阶乘和逆元阶乘的上限（最大支持到约 3e5）

    // 快速幂：计算 x^n % mod
    ll quick(ll x, ll n) {
        ll res = 1;
        while (n) {
            if (n & 1) res = (res * x) % mod; // 如果当前位是1，乘上当前x
            x = x * x % mod;                  // 平方底数
            n >>= 1;                          // 右移一位
        }
        return res;
    }

    // 计算模逆元：inv(x) = x^(mod-2) % mod（费马小定理，要求 mod 是质数）
    ll inv(ll x) {
        return quick(x, mod - 2);
    }

    ll fac[N], invfac[N]; // fac[i] 表示 i! % mod；invfac[i] 表示 i! 的逆元

    // 初始化阶乘和逆元数组
    void init() {
        fac[0] = 1;
        // 预处理阶乘 fac[1..N-1]
        for (int i = 1; i < N; ++i)
            fac[i] = (fac[i - 1] * i) % mod;

        // 先计算最高阶的阶乘的逆元
        invfac[N - 1] = inv(fac[N - 1]);

        // 倒推计算其他阶乘的逆元（递推公式：invfac[i] = invfac[i+1] * (i+1) % mod）
        for (int i = N - 2; i >= 0; --i)
            invfac[i] = (invfac[i + 1] * (i + 1)) % mod;
    }

    // 计算组合数 C(n, m) = n! / (m! * (n-m)!) % mod
    ll C(int n, int m) {
        if (n < m || m < 0) return 0; // 非法情况返回 0
        return fac[n] * invfac[m] % mod * invfac[n - m] % mod;
    }
}
```

`CNM::C(6, 3)` 表示从 6 个不同的元素中选出 3 个元素的组合数