1 基本算法

1.1 尺取法 (双指针)

用以解决序列的区间问题, 一般有两个要求:

1. **序列是有序的, 需要先对序列进行排序**

2. 问题与序列的区间有关, 操作两个或多个指针 i, j 表示区间

在 Python 中, 用 while 实现较为方便

扫描方向:

反向扫描, 左右指针: i, j 的方向相反;

同向扫描, 快慢指针: i, j 的方向相同, 但是扫描速度一般不同, 可以形成一个大小可变的滑动窗口

1.1.1 反向扫描

找指定和的整数对

问题: 输入 n ($n <= 100000$) 个整数, 放在数组 a[] 中. 找出其中的两个数, 它们之和等于整数 m. (假定肯定有解).

输入:

第 1 行是数组 a[], 第 2 行是 m

Sample input

21 4 5 6 13 65 32 9 23

28

Sample output

5 23

```
# 哈希 复杂度为O(n), 但是需要较大的哈希空间
a = list(map(int, input().split()))
m = int(input())
s = set(a)
outed = set()
for item in s:
  if m - item in s and item not in outed:
    print(item, m - item)
    outed.add(m-item)
```

```
# 尺取法 复杂度为O(n log_2^n), 其中, 排序的复杂度为O(log_2^n), 检查的复杂度为O(n)
a = list(map(int, input().split()))
a.sort()
m = int(input())

# 双指针
i, j = 0, len(a) - 1
while (i < j):
  s = a[i] + a[j]
  # s < m: i增加1, 之后的s>=当前s
  if s < m:
    i += 1
  elif s > m:
    j -= 1
  else:
    print("{} {}".format(a[i], a[j]))
    i += 1
```

判断回文串

输入: 第 1 行输入测试实例个数, 之后每行输入一个字符串

输出: 是回文串输出 yes, 不是输出 no

```
n = int(input())
for i in range(n):
  s = str(input())
  i, j = 0, len(s) - 1
  while i < j:
    flag = False
    if s[i] == s[j]:
      flag = True
    else:
      flag = False
      break
    i += 1
    j -= 1
  if (flag):
    print('yes')
  else:
    print('no')
```

1.1.2 同向扫描

使用尺取法产生滑动窗口

寻找区间和

给定一个长度为 n 的正整数数组 a[] 和一个数 s, 在数组中找一个区间, 使得该区间的数组元素之和等于 s. 输出区间的起点和终点位置

第 1 行输入数组长度 n, 第 2 行输入数组, 第 3 行为 s

Sample input

15

6 1 2 3 4 6 4 2 8 9 10 11 12 13 14

6

Sample output

0 0

1 3

5 5

6 7

初始值 i = j = 0

如果 sum = s: 输出一个解, sum 减去 a[i], i++

如果 sum < s: j++, sum + a[j]

如果 sum > s: sum - a[i], i++

```
n = int(input())
a = list(map(int, input().split()))
s = int(input())

sum = a[0]
i, j = 0, 0
while i < n and j < n:
  if sum == s:
    print("{} {}".format(i, j))
    sum -= a[i]
    i += 1
    j += 1
    sum += a[j]
  elif sum < s:
    j += 1
    sum += a[j]
  elif sum > s:
    sum -= a[i]
    i += 1
```

数组去重

给出一个数组, 输出去除重复元素之后的数组

```
# 哈希, 数据多或者数值过大时需要占用大量的空间
a = list(map(int, input().split()))
s = set(a)
unique_a = list(s)
print(unique_a)
```

```
# 尺取法
a = list(map(int, input().split()))
# a排序, 使得相同元素排列在一起
a.sort()
n = len(a)
# 双指针均从0开始
i, j = 0, 0
# j始终指向无重复元素部分的最后一个元素
while i < n and j < n:
  # 若i和j指向的元素不同, j++, 将i指向的元素复制到j上
  # EX: 1 2(j) 3(i) 3
  # -> 1 2 3(j, i) 3
  # EX: 1 2 3(j) 3 4(i)
  # -> 1 2 3 3(j) 4(i)
  # -> 1 2 3 4(j) 4(i)
  if a[i] != a[j]:
    j += 1
    a[j] = a[i]
  i += 1
unique_a = a[0:j+1]
print(unique_a)
```

找相同数对

洛谷 P1102

给出一串数字和一个数字 C, 要求计算出所有 A - B = C 的数对的个数 (不同位置的数字一样的数对算不同的数对)

输入: 2 行, 第 1 行输入整数 n 和 C, 第 2 行输入 n 个整数

输出: 满足 A - B = C 的数对的个数

Sample Input

6 3

8 4 5 7 7 4

Sample Output

5

```
n, c = map(int, input().split())
a = list(map(int, input().split()))
a.sort()

i, j, k = 0, 0, 0
ans = 0

for i in range(n):
  # j, k指向相同元素区间的起点和终点后1个元素
  # 寻找的对象是区间内的元素 - a[i] = C
  while j < n - 1 and a[j] - a[i] < c:
    j += 1
  while k < n and a[k] - a[i] <= c:
    k += 1
  if a[j] - a[i] == c and a[k-1] - a[i] == c and k - 1 >= 0:
    ans += k - j
print(ans)
```

1.2 二分法

1.2.1 Python 二分搜索库 bisect

```
from bisect import *
def fun(find, x, bias=0):
  global a
  index = find(a, x) + bias
  print("index: {}, element: {}".format(index, a[index]))

global a
a = [1, 2, 4, 4, 4, 5]
# target: x
x = 4

# first > x
fun(bisect_right, x)
# first >= x
fun(bisect_left, x)
# first = x
fun(bisect_left, x)
# last = x
fun(bisect_right, x, bias=-1)
# last <= x
fun(bisect_right, 3, bias=-1)
# last < x
fun(bisect_left, x, bias=-1)
# count x in a monotonic array
# slow
print(a.count(x))
# fast, using binary search
print(bisect_right(a, x) - bisect_left(a, x))
```

output

```
index: 5, element: 5
index: 2, element: 4
index: 2, element: 4
index: 4, element: 4
index: 1, element: 2
index: 1, element: 2
3
3
```

1.2.2 整数二分

需要注意终止边界和左右区间问题, 避免漏解和死循环

mid 的计算

```
# 适用单调递增序列的后继问题
mid = l + (r - l) // 2 # 相当于计算出的mid向下取整, 计算的是左中位数
# 适用单调递增序列的前驱问题
mid = l + (r - l + 1) // 2 # 相当于计算出的mid向上取整, 计算的是右中位数
```

在单调递增序列中寻找 x 或 x 的后继

在单调递增序列中寻找第一个 x 的位置, 若没有 x, 则寻找比 x 大的第一个数的位置, 即寻找第一个 >= x 的位置

```
# 左闭右开[0, n)
l, r = 0, n
while l < r:
  mid = l + (r - l) // 2
  if a[mid] >= x:
    r = mid
  else:
    l = mid + 1
return l
```

在单调递增序列中寻找 x 或 x 的前驱

在单调递增序列中寻找第一个 x 的位置, 若没有 x, 则寻找比 x 小的第一个数的位置, 即寻找第一个 <= x 的位置

```
# 左开右闭(-1, n-1]
l, r = -1, n - 1
while l < r:
  mid = l + (r - l + 1) // 2
  if a[mid] <= x:
    l = mid
  else:
    r = mid - 1
return l
```

寻找 minimum

```
while l < r:
mid = l + (l - r) // 2
if check(mid):
  # reduce
  r = mid
else:
  # enlarge
  l = mid + 1
# 若r = mid - 1, 则当best=mid时可能遗漏最优解
```

寻找指定和的整数对

输入 n ($n <= 100000$) 个整数, 找出其中的两个数, 使它们之和等于整数 m, 假设肯定有解

```
from bisect import *

a = list(map(int, input().split()))
m = int(input())
n = len(a)
a.sort()

# ver. 1
for i in range(n-1):
  # bisearch, a[k] = m - a[i]
  l, r = i+1, n
  x = m - a[i]
  while l < r:
    mid = l + (r - l) // 2
    if a[mid] >= x:
      r = mid
    else:
      l = mid + 1
  if a[l] == x:
    print("{} {}".format(a[i], a[l]))

# ver. 2
for i in range(n-1):
  # bisearch, a[k] = m - a[i]
  x = m - a[i]
  # search from a[i+1] to a[end-]
  p = bisect_left(a, x, lo=i+1, hi=n)
  if a[p] == x:
    print("{} {}".format(a[i], a[p]))
```

1.2.3 整数二分 最大值最小化

序列划分：二分 + 贪心

给定一个序列, 如 {2, 2, 3, 4, 5, 1}, 将其划分为 m 个连续的子序列 S_1, S_2, S_3, 每个子序列至少有一个元素, 使得每个子序列的和的最大值最小

EX. m = 3

划分为 (2, 2, 3), (4, 5), (1) 子序列和分别为 7, 9, 1, 最大值为 9

划分为 (2, 2, 3), (4), (5, 1) 子序列和为 7, 4, 6, 最大值为 7, 优于前一个

```
# Input:
# array
# m: amount of subarray
# Output:
# x: minimum of maximum of sum(all possible subarrays)
# subarrays

from bisect import *

a = list(map(int, input().split()))
m = int(input())
n = len(a)
l, r = max(a), sum(a)
subs = []

while l < r:
  mid = l + (r - l) // 2
  # greedy divide subarray
  # each sum(subarray) <= mid
  flag = False
  idx = 0
  subs = []
  for i in range(m):
    sum = 0
    while idx < n and sum + a[idx] <= mid:
      sum += a[idx]
      idx +=1
    if idx == n or sum + a[idx] > mid:
      subs.append(idx-1)
  # judge
  if subs[-1] < n-1:
    flag = False
  else:
    flag = True
  # binary control
  if flag:
    # reduce
    r = mid
  else:
    # enlarge
    l = mid + 1

print('minimum sum: ', l)
for i in range(m):
  if i == 0:
    print(a[ : subs[0]+1])
  else:
    print(a[subs[i-1]+1 : subs[i]+1])
```

1.2.4 整数二分 最小值最大化

洛谷 P1824 进击的奶牛

在一条很长的直线上, 指定 n 个坐标点. 有 c 头牛, 每头牛占据一个坐标点, 求相邻两头牛之间距离的最大值

Input:

第 1 行输入: n c

第 2 行开始每行输入: 一个整数, 表示每个点的坐标

```
n, c = map(int, input().split())
x = []
for i in range(n):
  x.append(int(input()))
x.sort()

def check(x, dis, c):
  i = 1
  last = 0
  c -= 1
  while c > 0 and i < len(x):
    while i < len(x) and x[i] - x[last] < dis:
      i += 1
    if i < len(x) and x[i] - x[last] >= dis:
      c -= 1
      last = i
      i += 1
  if c == 0:
    return True
  else:
    return False

l, r = 0, x[-1] - x[0]
ans = 0
while l < r:
  mid = l + (r - l) // 2
  if check(x, mid, c):
    ans = mid
    l = mid + 1
  else:
    r = mid
print(ans)
```

1.2.5 实数二分

for 控制: 过大的 for 次数会超时, 过小的会导致精度不够答案错误. 一般取 100, 但是循环体内计算量大的时候容易超时, 可以缩减到 50

while 控制: while 需要设计精度 eps, 过小的 eps 会超时, 过大的会导致精度不够答案错误

```
# 精度, 可以调整
eps = 1e-7
while r - l > eps:
# for ver.
# epoch = 100 # 轮次
# for i in range(epochs):
  mid = l + (r - l) / 2.0
  if check(mid):
    # reduce range
    r = mid
  else:
    # enlarge range
    l = mid
return l
```

分蛋糕 poj 3122

m+1 个人分 n 个半径不同的蛋糕, 要求每个人分得的蛋糕重量一致, 且必须是可以切出来的一整块, 每个人能分到的最大蛋糕是多少.

Input:

第一行: 1 个整数, 表示测试用例个数

对每个测试, 第一行输入 n, m. 第二行输入 n 个整数, 表示每个蛋糕的半径

Output:

对于每个测试, 输出一个答案, 保留 4 位小数

可以将问题建模为最小值最大化问题, 用面积代替重量

保留小数

```
a = 1.13456 # float
ans = format(a, '.4f) # 四舍五入保留4位小数
```

```
def check(mid, area, f):
  sum = 0
  for i in range(len(area)):
    sum += int(area[i] / mid)
  if sum >= f:
    return True
  else:
    return False

eps = 1e-5
pi = 3.1415926
T = int(input())
for t in range(T):
  n, f = map(int, input().split())
  cakes = list(map(float, input().split()))
  maxx = 0
  area = []
  for i in range(n):
    area.append(pi * cakes[i] * cakes[i])
    if area[i] > maxx:
      maxx = area[i]
  l, r = 0, maxx
  while r - l > eps:
    mid = l + (r - l) / 2.0
    if check(mid, area, f):
      l = mid
    else:
      r = mid
  ans = format(l, '.4f')
  print(ans)
```

1.3 三分法

用于求取单峰函数的极值. 通过在 [l, r] 内取两个点 **mid1, mid2**, 将函数分为**三段**

```
k = (r - l) / 3.0
mid1, mid2 = l + k, r - k
```

三分法模板 洛谷 P3382

给出一个 N 次多项式函数, 保证在区间 [l, r] 内存在一点 x, 使得 x 是函数在区间上的极大值, 求出 x

Input:

第一行输入 n: N l r

第二行输入: N + 1 个实数, 表示从高到低各项的系数

Output:

x, 四舍五入保留 5 位小数

```
n, l, r = map(float, input().split())
n = int(n)
cons = list(map(float, input().split()))
def f(x, n, cons):
  sum = 0
  for i in range(1, n+2):
    sum += cons[-i] * pow(x, i-1)
  return sum
eps = 1e-6
while r - l > eps:
  k = (r - l) / 3.0
  mid1, mid2 = l + k, r - k
  if f(mid1, n, cons) < f(mid2, n, cons):
    l = mid1
  else:
    r = mid2
ans = format(l, '.5f')
print(ans)
```

三分法函数洛谷 P1883

给定 n 个二次函数 $f_1(x), f_2(x), \ldots, f_n(x)$（均形如 ax^2+bx+c），设 $F(x)=\max\{f_1(x), f_2(x), ..., f_n(x)\}$，求 $F(x)$ 在区间 $[0,1000]$ 上的最小值。

输入格式

输入第一行为正整数 T，表示有 T 组数据。

每组数据第一行一个正整数 n，接着 n 行，每行 3 个整数 a,b,c，用来表示每个二次函数的 3 个系数，注意二次函数有可能退化成一次。

输出格式

每组数据输出一行，表示 $F(x)$ 的在区间 $[0,1000]$ 上的最小值。答案精确到小数点后四位，四舍五入。

输入输出样例 1

输入 1

2

1

2 0 0

2

2 0 0

2 -4 2

输出 1

0.0000

0.5000

说明/提示

对于 50% 的数据，$n \le 100$。

对于 100% 的数据，$T<10$， $n \le 10^4$，$0 \le a \le 100$，$|b| \le 5\times 10^3$，$|c| \le 5\times 10^3$。

```python
def cal(cons, x):
    ans = -float('inf')
    for a, b, c in cons:
        ans = max(ans, a * x ** 2 + b * x + c)
    return ans

eps = 1e-9
T = int(input())
for t in range(T):
    n = int(input())
    cons = []
    for i in range(n):
        a, b, c = map(float, input().split())
        cons.append([a, b, c])
    l = 0
    r = 1000
    while r - l > eps:
        margin = (r - l) / 3.0
        mid1 = l + margin
        mid2 = r - margin
        f1 = cal(cons, mid1)
        f2 = cal(cons, mid2)
        if f1 <= f2:
            r = mid2
        else:
            l = mid1
    ans = cal(cons, l)
    print(format(ans, '.4f'))
```

1.4 排序与排列

1.4.1 排序

Python 自定排序 key

```
key function:
Input: x, y
Output:  1:  x > y
         0:  x = y
         -1: x < y
```

```
from functools import cmp_to_key
def cmp(x, y):
  # compare x and y
  # return result
sort_key = cmp_to_key(cmp)
```

洛谷 P1093

P1093 [NOIP 2007 普及组] 奖学金

题目背景

NOIP2007 普及组 T1

题目描述

某小学最近得到了一笔赞助，打算拿出其中一部分为学习成绩优秀的前 5 名学生发奖学金。期末，每个学生都有 3 门课的成绩：语文、数学、英语。先按总分从高到低排序，如果两个同学总分相同，再按语文成绩从高到低排序，如果两个同学总分和语文成绩都相同，那么规定学号小的同学排在前面，这样，每个学生的排序是唯一确定的。

任务：先根据输入的 3 门课的成绩计算总分，然后按上述规则排序，最后按排名顺序输出前五名名学生的学号和总分。

注意，在前 5 名同学中，每个人的奖学金都不相同，因此，你必须严格按上述规则排序。例如，在某个正确答案中，如果前两行的输出数据（每行输出两个数：学号、总分) 是：

7 279

5 279

这两行数据的含义是：总分最高的两个同学的学号依次是 7 号、5 号。这两名同学的总分都是 279 (总分等于输入的语文、数学、英语三科成绩之和) ，但学号为 7 的学生语文成绩更高一些。

如果你的前两名的输出数据是：

5 279

7 279

则按输出错误处理，不能得分。

输入格式

共 $n+1$ 行。

第 1 行为一个正整数 $n \leq 300$，表示该校参加评选的学生人数。

第 2 到 $n+1$ 行，每行有 3 个用空格隔开的数字，每个数字都在 0 到 100 之间。第 j 行的 3 个数字依次表示学号为 $j-1$ 的学生的语文、数学、英语的成绩。每个学生的学号按照输入顺序编号为 $1 \sim n$（恰好是输入数据的行号减 1)。

保证所给的数据都是正确的，不必检验。

输出格式

共 5 行，每行是两个用空格隔开的正整数，依次表示前 5 名学生的学号和总分。

输入输出样例 1

输入 1

```
6
90 67 80
87 66 91
78 89 91
88 99 77
67 89 64
78 89 98
```

输出 1

```
6 265
4 264
3 258
2 244
1 237
```

```
# 洛谷P1093

# 自定义比较函数 cmp -> key
# x: (id, c, m, e, sum)
# Input: x, y
# Output:
# 1: x > y; 0: x = y; -1: x < y
from functools import cmp_to_key
def cmp(x, y):
  if x[4] > y[4]:
    return 10
  elif x[4] < y[4]:
    return -1
  else:
    if x[1] > y[1]:
      return 1
    elif x[1] < y[1]:
      return -1
    else:
      if x[0] < y[0]:
        return 1
      else:
        return -1
sort_key = cmp_to_key(cmp)

n = int(input())
stu = []
for id in range(1, n+1):
  c, m, e = map(int, input().split())
  summ = c + m + e
  stu.append([id, c, m, e, summ])
rst = sorted(stu, key=sort_key, reverse=True)
for i in range(5):
  print('{} {}'.format(rst[i][0], rst[i][-1]))
```

1.4.2 排列

Python 生成全排列

```
from itertools import permutations
a = [1, 2, 3]
# 全排列
per = permutations(a)
for item in per:
    print(item)
# Output:
(1, 2, 3)
(1, 3, 2)
(2, 1, 3)
(2, 3, 1)
(3, 1, 2)
(3, 2, 1)

# n取m排列
per = permutations(a, 2)
for item in per:
    print(item)
# Output:
(1, 2)
(1, 3)
(2, 1)
(2, 3)
(3, 1)
(3, 2)
```

蓝桥杯 2016 省赛 Cpp A 组 T6

有加减乘除 4 种运算:

$() + () = ()$

$() - () = ()$

$() \times () = ()$

$() \div () = ()$

每个括号代表 1-13 中的某个数字, 但不能重复, 一共有多少种方案

```python
# 蓝桥杯2016省赛C++A组第6题

a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
n = len(a)
vis = [False] * n
b = [None] * n
ans = 0

def permu(s, t):
  global ans
  # 输出
  if s == 12 and b[9] == b[10] * b[11]:
    ans += 1
    return
  # 剪枝
  if s == 3 and b[0] + b[1] != b[2]:
    return
  if s == 6 and b[3] - b[4] != b[5]:
    return
  if s == 9 and b[6] * b[7] != b[8]:
    return
  # 全排列
  for i in range(t):
    if not vis[i]:
      vis[i] = True
      b[s] = a[i]
      permu(s+1, t)
      vis[i] = False

permu(0, n)
print(ans)
```

1.5 分治法

将问题分解为多个独立的子问题, 子问题的规模大致相等. 分治法常能够显著降低算法的时间复杂度.

分治法经典应用: 汉诺塔, 归并排序, 快速排序等

汉诺塔 蓝桥 1512

Input: 两个整数 N M. N 为要移动的盘子数, M 为最少移动步骤的第 M 步

Output:

第一行: #No: a->b. No 为移动的盘子编号, a->b 为从 a 杆移动到 b 杆, 取值为 {A, B, C}

第二行: 一个整数, 最少移动步数

```
step = 0
n, m = map(int, input().split())

def hanoi(x, y, z, n):
  global step, m
  if n == 1:
    step += 1
    if step == m:
      print('#{}: {}->{}'.format(m, x, z))
  else:
    hanoi(x, z, y, n-1)
    step += 1
    if step == m:
      print('#{}: {}->{}'.format(m, x, z))
    hanoi(y, x, z, n-1)

hanoi('A', 'B', 'C', n)
print(step)
```

逆序对 hdu 4911

输入一个序列, 交换任意两个元素, 不超过 k 次交换之后, 最少的逆序对有多少个.

逆序对: $a_i, a_j,\ when\ 1 <= i < j < n\ and\ a_i > a_j$

Input:

第一行: n, k

第二行: n 个整数

Output: 最少的逆序对数量

```
# hdu 4911

n, k = map(int, input().split())
a = list(map(int, input().split()))
count = 0
b = [0]*10000

def merge(l, mid, r):
  global count, a, b
  i, j = l, mid+1
  t = 0
  while i <= mid and j <= r:
    if a[i] > a[j]:
      b[t] = a[j]
      t, j = t+1, j+1
      count += mid - i + 1
    else:
      b[t] = a[i]
      t, i = t+1, i+1
  while i <= mid:
    b[t] = a[i]
    t, i = t+1, i+1
  while j <= r:
    b[t] = a[j]
    t, j = t+1, j+1
  for i in range(t):
    a[l+i] = b[i]

def merge_sort(l, r):
  if l < r:
    mid = (l + r) // 2
    merge_sort(l, mid)
    merge_sort(mid+1, r)
    merge(l, mid, r)

merge_sort(0, len(a)-1)
if count <= k:
  print(0)
else:
  print(count - k)
```

1.6 ST 算法倍增法

倍增法原理

每一步都以 2 倍扩展区间, 以此快速覆盖整个区间. 在编程上, 不使用*2, 而是利用二进制数的特性, 将一个数 N 用二进制展开 $N = a_0 2^0 + a_1 2^1 + \ldots$, 这样, 整数 n 的数位只有 $log_2 n$ 位, 以每一位作为一个跳完下一个状态的跳板, 跳板数量等同于二进制位置, 只有 $log_2 n$

ST 算法

ST 算法是用于求解 RMQ(Range Minimum/Maximum Query, 区间最值查询) 的算法, 基于倍增法原理, 适用于**静态空间**.

ST 算法的基本思想: 对于一个区间 [a, b], 区间上的最值由两个子区间 [c, d], [e, f] 决定, 且 $[c,d] \vee [e,f] = [a,b]$. 即 $min\{[a,b]\} = min\{min\{[c,d]\}, min\{[e,f]\}\}$

Procedure

1. 将数列按照倍增法划分为多个小区间.

第一组区间长度为 1; 第二组区间长度为 2; 第三组区间长度为 4, 以此类推. 每一组区间中, 第 i+1 个区间的左端点是第 i 个区间的左端点 +1.

以此可以通过不同组的区间, 递推出每一组子区间的最值. 例如, 第三组第一个子区间 {1, 2, 3, 4} 可以通过第二组的 {1, 2}{3, 4} 递推得到, 且有递推公式.

```
min: dp[s][k] = min{dp[s][k-1], dp[s + 1 << (k-1)][k-1]}
max: dp[s][k] = max{dp[s][k-1], dp[s + 1 << (k-1)][k-1]}
```

其中, dp[s][k] 表示左端点为 s, 区间长度为 2^k 的区间最值, `1 << (k-1`表示 2^{k-1}

2. 查询任意区间的最值

对于区间 [L, R], 为保证覆盖, 需要两个长度为 x 的子区间, 且满足 $x \leq len,\ 2x \geq len$.

计算 dp, 根据 dp[s][k] 定义, $2^k = x$, 因此 $k = \lfloor log_2(len) \rfloor$

区间 [L, R] 的最小值, 为覆盖其的小区间的最小值, 即

```
min = min(dp_min[L][k], dp_min[R - (1 << k) + 1][k])
max = max(dp_max[L][k], dp_max[R - (1 << k) + 1][k])
```

模版洛谷 P2880

给定一个包含 n 个整数的数列和 q 个区间查询, 查询区间内最大值和最小值的差.

Input: 第一行输入 n, q. 接下来的 n 行, 每行输入一个整数 h_i; 再加下来 q 行, 每行输入两个整数 a b, 表示一个区间查询.

```python
from math import log2
n, q = map(int, input().split())
# maximum of k
p = int(log2(n))
# RMQ table, 1-index
dp_min = [[0 for _ in range(p+1)]for _ in range(n + 1)]
dp_max = [[0 for _ in range(p+1)]for _ in range(n + 1)]
# read in list, 1-index
a = [-1]
for i in range(n):
  a.append(int(input()))

# initial dp table
# len of sub-range is 1
for i in range(1, n+1):
  dp_min[i][0] = a[i]
  dp_max[i][0] = a[i]
# len of sub-range > 1
for k in range(1, p+1):
  for s in range(1, n):
    if s + (1 << k) > n + 1:
      break
```

```
    dp_min[s][k] = min(dp_min[s][k-1], dp_min[s + (1 << (k-1))][k-1])
    dp_max[s][k] = max(dp_max[s][k-1], dp_max[s + (1 << (k-1))][k-1])

# RMQ
for _ in range(q):
  L, R = map(int, input().split())
  k = int(log2(R - L + 1))
  max_query = max(dp_max[L][k], dp_max[R - (1 << k) + 1][k])
  min_query = min(dp_min[L][k], dp_min[R - (1 << k) + 1][k])
  print(max_query - min_query)
```

1.7 离散化

离散化适用场景: 给出一个数列, 问题要求关注数字的相对大小, 数字的绝对大小不重要. 此时使用离散化, 在列表中, 使用数字的相对大小排序取代数字的数值.

Procedure

1. 排序
2. 离散化: 将排序后的数列元素从 1 开始分配数值
3. 归位: 将数列元素重新分配回原始位置

```
from random import randint

a = [] # element[0]: val, element[1]: original place
for i in range(10):
  a.append([randint(1, 5000), i])

print('original list:')
print(a)

# main function
# sort
a.sort(key=lambda x : x[0])
n = len(a)
# generate new list
newa = [0 for _ in range(n)]
for i in range(n):
  newa[a[i][1]] = i + 1

print('processed list:')
print(newa)

# IF: same value should be corresponded to the same processed value
print('---------------same value -> same order value---------------')
a = [[13, 0],[65, 1], [8, 2], [13, 3], [93, 4], [197, 5]]
print('original list')
print(a)
a.sort(key=lambda x : x[0])
n = len(a)
# generate new list
newa = [0 for _ in range(n)]
for i in range(n):
  if i > 0 and a[i][0] == a[i-1][0]:
    newa[a[i][1]] = newa[a[i-1][1]]
    continue
  newa[a[i][1]] = i + 1
print('processed list:')
print(newa)
```

```
original list:
[[2700, 0], [1266, 1], [1999, 2], [3798, 3], [1139, 4], [3214, 5],
[4171, 6], [4597, 7], [3040, 8], [1135, 9]]
processed list:
[5, 3, 4, 8, 2, 7, 9, 10, 6, 1]
---------------same value -> same order value---------------
original list
[[13, 0], [65, 1], [8, 2], [13, 3], [93, 4], [197, 5]]
processed list:
[2, 4, 1, 2, 5, 6]
```