

算法竞赛笔记

Chunyin Chan

2025 年 5 月 27 日

目录

1	基础数据结构	2
1.1	Link List in Python:	2
1.2	Queue in Python	5
1.2.1	双端队列	5
1.2.2	单调队列	6
1.2.3	单调队列与动态规划	7
1.2.4	优先队列	9
1.3	Stack in Python	9
1.3.1	单调栈	10
1.4	Binary Tree in Python	11
1.4.1	DFS 遍历	11
1.4.2	哈夫曼编码	13
1.5	Heap in Python	14
2	基本算法	16
2.1	尺取法 (双指针)	16
2.1.1	反向扫描	16
2.1.2	同向扫描	17
2.2	二分法	20
2.2.1	Python 二分搜索库 bisect	20
2.2.2	整数二分	20
2.2.3	整数二分 最大值最小化	23
2.2.4	整数二分 最小值最大化	24
2.2.5	实数二分	24
2.3	三分法	25
2.4	排序与排列	28
2.4.1	排序	28
2.4.2	排列	30
2.5	分治法	32
2.6	ST 算法倍增法	34
2.7	离散化	36
3	搜索	37
3.1	BFS 和 DFS 基础	37
3.1.1	BFS	37
3.1.2	DFS	38
3.2	剪枝与判重	40
3.2.1	BFS 判重 蓝桥 642	40
3.2.2	可行性剪枝 poj3278	41
3.2.3	最优性剪枝 洛谷 1118	42
3.2.4	优化搜索顺序 排除等效冗余 洛谷 1120	44
3.3	洪水填充	45
3.4	BFS 与最短路径	46

4 DP 基本47

4.1 DP 的两种编程方法47

4.1.1 自顶向下结合记忆化47

4.1.2 自底向上结合制表47

4.2 DP 的设计与实现48

4.2.1 dp 状态设计48

4.2.2 状态转移方程48

4.2.3 递推实现48

4.2.4 记忆化实现48

4.3 滚动数组简化优化空间占用49

4.3.1 交替滚动49

4.3.2 自我滚动49

4.4 经典线性 DP 与优化策略49

4.4.1 0/1 背包问题49

4.4.2 多重背包问题 二进制拆分优化50

1 基础数据结构

1.1 Link List in Python:

```
1 class nodes:
2     def __init__(self, val=None, pre=None, next=None):
3         self.val = val
4         self.next = next
```

EX 1

洛谷 P1996 约瑟夫问题

题目描述

n 个人围成一圈，从第一个人开始报数，数到 m 的人出列，再由下一个人重新从 1 开始报数，数到 m 的人再出圈，依次类推，直到所有的人都出圈，请输出依次出圈人的编号。

注意：本题和《深入浅出-基础篇》上例题的表述稍有不同。书上表述是给出淘汰 $n-1$ 名小朋友，而该题是全部出圈。

输入格式

输入两个整数 n, m 。

输出格式

输出一行 n 个整数，按顺序输出每个出圈人的编号。

输入输出样例

输入 #1

10 3

输出 #1

3 6 9 2 7 1 8 5 10 4

说明/提示

1 $m, n \leq 100$

```
1 # 动态链表
2 class node:
3     data = None
4     next = None
5
6 n, m = map(int, input().split())
7 # init link list
8 head = node()
9 head.data = 1
10 now = head
11 for i in range(2, n+1):
12     p = node()
13     p.data = i
14     now.next = p
15     now = p
16 now.next = head
17 prev = now
18 now = head
19 while n > 0:
20     n -= 1
21     for i in range(m-1):
22         prev = now
23         now = now.next
24     print(now.data, end=' ')
25     prev.next = now.next
26     now = now.next
```

```
1 # 列表和索引计算
2 n, m = map(int, input().split())
3 people = list(range(1, n+1))
4 rst = []
5 current = 0
6 index = 0
7 while people:
8     index = (current + m - 1) % len(people)
9     rst.append(people.pop(index))
10    current = index
11 print(' '.join(map(str, rst)))
```

EX 2

洛谷 P1160 队列安排

题目描述

一个学校里老师要将班上 N 个同学排成一列，同学被编号为 $1 \sim N$ ，他采取如下的方法：

1. 先将 1 号同学安排进队列，这时队列中只有他一个人；
2. $2 \sim N$ 号同学依次入列，编号为 i 的同学入列方式为：老师指定编号为 i 的同学站在编号为 $1 \sim (i-1)$ 中某位同学（即之前已经入列的同学）的左边或右边；
3. 从队列中去掉 M 个同学，其他同学位置顺序不变。

在所有同学按照上述方法队列排列完毕后，老师想知道从左到右所有同学的编号。

输入格式

第一行一个整数 N ，表示了有 N 个同学。

第 $2 \sim N$ 行，第 i 行包含两个整数 k, p ，其中 k 为小于 i 的正整数， p 为 0 或者 1。若 p 为 0，则表示将 i 号同学插入到 k 号同学的左边， p 为 1 则表示插入到右边。

第 $N+1$ 行为一个整数 M ，表示去掉的同学数目。

接下来 M 行，每行一个正整数 x ，表示将 x 号同学从队列中移去，如果 x 号同学已经不在队列中则忽略这一条指令。

输出格式

一行，包含最多 N 个空格隔开的整数，表示了队列从左到右所有同学的编号。

输入输出样例 #1

输入 #1

4 1 0 2 1 1 0 2 3 3

输出 #1

2 4 1

说明/提示

** 【样例解释】 **

将同学 2 插入至同学 1 左边，此时队列为：

2 1

将同学 3 插入至同学 2 右边，此时队列为：

2 3 1

将同学 4 插入至同学 1 左边，此时队列为：

2 3 4 1

将同学 3 从队列中移出，此时队列为：

2 4 1

同学 3 已经不在队列中，忽略最后一条指令

最终队列：

2 4 1

【数据范围】

对于 20% 的数据, $1 \leq N \leq 10$ 。

对于 40% 的数据, $1 \leq N \leq 1000$ 。

对于 100% 的数据, $1 < M \leq N \leq 10^5$ 。

```
1  # 链表
2  class nodes:
3      def __init__(self, val=None, next=None, prev=None):
4          self.val = val
5          self.next = next
6          self.prev = prev
7
8  N = int(input())
9  people = nodes(1)
10 head = people
11 idx = {1: people}
12 for i in range(2, N+1):
13     node = nodes(i)
14     k, p = map(int, input().split())
15     k = idx[k]
16     if p == 0:
17         if k.prev:
18             k.prev.next = node
19             node.prev = k.prev
20             node.next = k
21             k.prev = node
22         if k == head:
23             head = node
24     elif p == 1:
25         if k.next:
26             k.next.prev = node
27             node.next = k.next
28             node.prev = k
29             k.next = node
30     idx[i] = node
31
32 M = int(input())
33 for i in range(M):
34     x = int(input())
35     if x in idx.keys():
36         k = idx[x]
37         if k.prev:
38             k.prev.next = k.next
39         if k.next:
40             k.next.prev = k.prev
41         if k.next:
42             k.next.prev = k.prev
43         if k.prev:
44             k.prev.next = k.next
45         if k == head:
46             head = k.next
47     del idx[x]
48
49 while head:
50     print(head.val, end=' ')
51     head = head.next
```

1.2 Queue in Python

1.2.1 双端队列

```
1 from collections import deque # 双端队列
2
3 queue = deque(maxlen = 10) # 最大长度为10, None为无限制
4 queue = deque(iterable) # init queue by iterable obj.
5
6 queue.append(x) # add x to right side
7 queue.appendleft(x) # add x to left side
8
9 queue.extend(iterable) # extend right side by iterable obj.
10 queue.extendleft(iterable) # extend the left side by iterable obj.
11
12 queue.pop() # pop element from right side
13 queue.popleft() # pop element from left side
14
15 queue.remove(x) # remove x from Queue, raise error if not found
16 queue.clear() # clear Queue
17
18 queue.reverse() # reverse Queue
19 queue.insert(i, x) # insert x into queue at position i
20 queue.count(x) # count the number of queue elements = x
21 queue.index(x) # return first match position, raise error if not found
```

EX 1

洛谷 P1540 [NOIP 2010 提高组] 机器翻译

题目背景

NOIP2010 提高组 T1

题目描述

小晨的电脑上安装了一个机器翻译软件，他经常用这个软件来翻译英语文章。

这个翻译软件的原理很简单，它只是从头到尾，依次将每个英文单词用对应的中文含义来替换。对于每个英文单词，软件会先在内存中查找这个单词的中文含义，如果内存中有，软件就会用它进行翻译；如果内存中没有，软件就会在外存中的词典内查找，查出单词的中文含义然后翻译，并将这个单词和译义放入内存，以备后续的查找和翻译。

假设内存中有 M 个单元，每单元能存放一个单词和译义。每当软件将一个新单词存入内存前，如果当前内存中已存入的单词数不超过 $M - 1$ ，软件会将新单词存入一个未使用的内存单元；若内存中已存入 M 个单词，软件会清空最早进入内存的那个单词，腾出单元来，存放新单词。

假设一篇英语文章的长度为 N 个单词。给定这篇待译文章，翻译软件需要去外存查找多少次词典？假设在翻译开始前，内存中没有任何单词。

输入格式

共 2 行。每行中两个数之间用一个空格隔开。

第一行为两个正整数 M, N ，代表内存容量和文章的长度。

第二行为 N 个非负整数，按照文章的顺序，每个数（大小不超过 1000）代表一个英文单词。文章中两个单词是同一个单词，当且仅当它们对应的非负整数相同。

输出格式

一个整数，为软件需要查词典的次数。

输入输出样例 #1

输入 #1

3 7 1 2 1 5 4 4 1

输出 #1

5

说明/提示

样例解释

整个查字典过程如下：每行表示一个单词的翻译，冒号前为本次翻译后的内存状况：

1. 1: 查找单词 1 并调入内存。2. 1 2: 查找单词 2 并调入内存。3. 1 2: 在内存中找到单词 1。4. 1 2 5: 查找单词 5 并调入内存。5. 2 5 4: 查找单词 4 并调入内存替代单词 1。6. 2 5 4: 在内存中找到单词 4。7. 5 4 1: 查找单词 1 并调入内存替代单词 2。

共计查了 5 次词典。

数据范围

- 对于 10% 的数据有 $M = 1, N \leq 5$; - 对于 100% 的数据有 $1 \leq M \leq 100, 1 \leq N \leq 1000$ 。

```
1 # 队列 collections.deque
2 from collections import deque
3 from array import array
4
5 M, N = map(int, input().split())
6 q = deque(maxlen=M) # 双向队列
7 qs = set() # 用作哈希表
8 count = 0 # 计数
9 inp_ary = array('i', map(int, input().split())) # 输入数据
10 l = len(inp_ary)
11 for i in range(l):
12     if inp_ary[i] not in qs:
13         if len(q) == M:
14             qs.remove(q.popleft())
15             qs.add(inp_ary[i])
16             q.append(inp_ary[i])
17         count += 1
18 print(count)
```

1.2.2 单调队列

洛谷 P1886 滑动窗口 / 【模板】单调队列

题目描述

有一个长为 n 的序列 a ，以及一个大小为 k 的窗口。现在这个从左边开始向右滑动，每次滑动一个单位，求出每次滑动后窗口中的最大值和最小值。

例如，对于序列 $[1, 3, -1, -3, 5, 3, 6, 7]$ 以及 $k = 3$ ，有如下过程：

窗口位置								最小值	最大值
[1	3	-1]	-3	5	3	6	7	-1	3
1	[3	-1	-3]	5	3	6	7	-3	3
1	3	[-1	-3	5]	3	6	7	-3	5
1	3	-1	[-3	5	3]	6	7	-3	5
1	3	-1	-3	[5	3	6]	7	3	6
1	3	-1	-3	5	[3	6	7]	3	7

输入格式

输入一共有两行，第一行有两个正整数 n, k 。第二行 n 个整数，表示序列 a

输出格式

输出共两行，第一行为每次窗口滑动的最小值第二行为每次窗口滑动的最大值

输入输出样例 #1

输入 #1

8 3 1 3 -1 -3 5 3 6 7

输出 #1

-1 -3 -3 -3 3 3 3 3 5 5 6 7

说明/提示

【数据范围】对于 50% 的数据， $1 \leq n \leq 10^5$ ；对于 100% 的数据， $1 \leq k \leq n \leq 10^6$ ， $a_i \in [-2^{31}, 2^{31})$ 。

```

1 from collections import deque
2 from array import array
3 import sys
4
5 # input
6 n, k = map(int, input().split())
7 a = list(map(int, input().split()))
8
9 min_q = deque()
10 max_q = deque()
11 min_rst = []
12 max_rst = []
13
14 for i in range(n):
15     # minimum queue
16     while min_q and a[min_q[-1]] > a[i]:
17         min_q.pop()
18     min_q.append(i)
19     # maximum queue
20     while max_q and a[max_q[-1]] < a[i]:
21         max_q.pop()
22     max_q.append(i)
23
24     # 开始记录
25     if i >= k - 1:
26         min_rst.append(a[min_q[0]])
27         max_rst.append(a[max_q[0]])
28
29     # 弹出窗口外的元素
30     if min_q and min_q[0] <= i - k + 1:
31         min_q.popleft()
32     if max_q and max_q[0] <= i - k + 1:
33         max_q.popleft()
34
35 # output
36 print(' '.join(map(str, min_rst)))
37 print(' '.join(map(str, max_rst)))

```

1.2.3 单调队列与动态规划

通过前缀和 + 单调队列 + 动态规划求解子序和问题

洛谷 P1714

Problem Description

给定一个序列，给定一个最大长度 m ，求一段长度不超过 m 的连续子序列，使其子序和最大

Input

第 1 行输入 n, m 。分别为序列长度和最大长度

第 2 行输入 N 个数

Output

第 1 个数是最大子序和，第 2 和第 3 个数是开始和终止位置

Case 1:

Input

5 2

1 2 3 4 5

Output

9

Case 2:

Input

6 3

1 -2 3 -4 5 -6

Output

5

Case 3

Input

5 5

1 2 3 4 5

Output

15

```
1 # 洛谷 P1714
2
3 from collections import deque
4 n, m = map(int, input().split())
5 s = list(map(int, input().split())) # 前缀和
6 dp = deque()
7 ans = s[0]
8
9 # 计算前缀和
10 for i in range(1, n):
11     s[i] = s[i-1] + s[i]
12 # 单调队列
13 dp.append(0)
14 for i in range(1, n):
15     # 去头
16     while len(dp) > 0 and dp[0] < i - m:
17         dp.popleft()
18     # 去尾, 去除>=s[i]的元素, 使得s[j] - s[i]更大
19     while len(dp) > 0 and s[dp[-1]] >= s[i]:
20         dp.pop()
21     dp.append(i)
22     if len(dp) == n:
23         ans = max(ans, s[i])
24     elif len(dp) > 1:
25         ans = max(ans, s[i] - s[dp[0]])
26     elif len(dp) == 1:
27         ans = max(ans, s[dp[0]])
28     else:
29         ans = max(ans, s[i])
30
31 print(ans)
```

1.2.4 优先队列

```

1 from queue import PriorityQueue
2 MAXSIZE = 0
3
4 q = PriorityQueue(maxsize=MAXSIZE) # 队列大小，默认为0，<=0的队列大小为无穷
5 q1 = PriorityQueue()
6
7 q.empty() # if empty
8 q.full() # if full
9 q.qsize() # get current size
10
11 # add elem
12 # 两种添加方式不能混用
13 # 直接按照元素大小存入，元素大小越小，优先级越高
14 q.put(1)
15 q.put(2)
16 # q.put((priority number, data))
17 # priority number越小，优先级越高
18 q1.put((2, 'ele'))
19 q1.put((-3, 'ele2'))
20
21 # get element, = pop()
22 q.get()
```

1.3 Stack in Python

使用 deque 作为栈.

hdu 1062

翻转字符串

input:

olleh !dlrow

output:

hello world!

```

1 from collections import deque
2 s = deque()
3 inp = list(input().split())
4 for item in inp:
5     # enter stack
6     for i in item:
7         s.append(i)
8     while len(s) > 0:
9         print(s.pop(), end=' ')
10    print(' ', end=' ')
```

1.3.1 单调栈

洛谷 P2947 [USACO09MAR] Look Up S

题目描述

Farmer John's N ($1 \leq N \leq 100,000$) cows, conveniently numbered $1..N$, are once again standing in a row. Cow i has height H_i ($1 \leq H_i \leq 1,000,000$).

Each cow is looking to her left toward those with higher index numbers. We say that cow i 'looks up' to cow j if $i < j$ and $H_i < H_j$. For each cow i , FJ would like to know the index of the first cow in line looked up to by cow i .

Note: about 50

约翰的 $N(1 \leq N \leq 10^5)$ 头奶牛站成一排，奶牛 i 的身高是 $H_i(1 \leq H_i \leq 10^6)$ 。现在，每只奶牛都在向右看齐。对于奶牛 i ，如果奶牛 j 满足 $i < j$ 且 $H_i < H_j$ ，我们可以说奶牛 i 可以仰望奶牛 j 。求出每只奶牛离她最近的仰望对象。

Input

输入格式

1. Line 1: A single integer: N

Lines 2.. $N+1$: Line $i+1$ contains the single integer: H_i

第 1 行输入 N ，之后每行输入一个身高 H_i 。

输出格式

Lines 1.. N : Line i contains a single integer representing the smallest index of a cow up to which cow i looks.

If no such cow exists, print 0.

共 N 行，按顺序每行输出一只奶牛的最近仰望对象，如果没有仰望对象，输出 0。

输入输出样例 # 1

输入 # 1

6

3

2

6

1

1

2

输出 # 1

3

3

0

6

6

0

说明/提示

FJ has six cows of heights 3, 2, 6, 1, 1, and 2.

Cows 1 and 2 both look up to cow 3; cows 4 and 5 both look up to cow 6; and cows 3 and 6 do not look up to any cow.

【输入说明】6 头奶牛的身高分别为 3,2,6,1,1,2。

【输出说明】奶牛 #1,#2 仰望奶牛 #3，奶牛 #4,#5 仰望奶牛 #6，奶牛 #3 和 #6 没有仰望对象。

【数据规模】

对于 20% 的数据： $1 \leq N \leq 10$;

对于 50% 的数据： $1 \leq N \leq 10^3$;

对于 100% 的数据: $1 \leq N \leq 10^5, 1 \leq H_i \leq 10^6$ 。

```

1 from collections import deque
2 from array import array
3 n = int(input())
4 cows = []
5 for i in range(n):
6     cows.append(int(input()))
7 s = deque()
8 rst = array('i')
9 for i in range(n-1, -1, -1):
10     # 由于反向遍历, 栈内元素必然在当前遍历元素的右边
11     # 不比当前高的元素出栈, 保证栈底到栈顶的高度递减
12     # 目的是使得随着逐渐出栈, 栈顶元素逐渐变高, 以找到能比当前遍历元素高的元素
13     while len(s) > 0 and cows[s[-1]] <= cows[i]:
14         s.pop()
15     if len(s) == 0:
16         rst.append(0)
17     else:
18         rst.append(s[-1] + 1) # relationship between index & NO.
19     s.append(i)
20 for i in range(len(rst)-1, -1, -1):
21     print(rst[i])

```

1.4 Binary Tree in Python

```

1 class nodes:
2     def __init__(self, val=None, l=None, r=None):
3         self.val = val
4         self.l = l
5         self.r = r

```

1.4.1 DFS 遍历

```

1 # pre
2 def preorder(node):
3     if not node:
4         return
5     print(node.val, end=' ')
6     preorder(node.l)
7     preorder(node.r)
8
9 # mid
10 def inorder(node):
11     if not node:
12         return
13     inorder(node.l)
14     print(node.val, end=' ')
15     inorder(node.r)
16
17 # post
18 def postorder(node):
19     if not node:
20         return
21     postorder(node.l)
22     postorder(node.r)
23     print(node.val, end=' ')

```

```
1 # pre
2 def preorder(node):
3     if not node:
4         return
5     print(node.val, end=' ')
6     preorder(node.l)
7     preorder(node.r)
8
9 # mid
10 def inorder(node):
11     if not node:
12         return
13     inorder(node.l)
14     print(node.val, end=' ')
15     inorder(node.r)
16
17 # post
18 def postorder(node):
19     if not node:
20         return
21     postorder(node.l)
22     postorder(node.r)
23     print(node.val, end=' ')

```

```
1 # test code
2 ab = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']
3 n = list(nodes(i) for i in ab)
4 root = n[4]
5 root.l = n[1]
6 root.r = n[6]
7 root.l.l = n[0]
8 root.l.r = n[3]
9 root.l.r.l = n[2]
10 root.r.l = n[5]
11 root.r.r = n[-1]
12 root.r.r.l = n[-2]

```

```
1 // output
2 E B A D C G F I H
3 A B C D E F G H I
4 A C D B F H I G E

```

根据前序和中序遍历输出后序遍历结果

```

1 n = int(input())
2 global pre_tra
3 pre_tra = list(map(int, input().split()))
4 in_tra = list(map(int, input().split()))
5
6 root = nodes(pre_tra[0])
7 pre_tra = pre_tra[1:]
8 # split left and right subtree
9 mid = in_tra.index(root.val)
10 lp = in_tra[0:mid]
11 rp = in_tra[mid+1:]
12
13 def establish(root, lpart, rpart):
14     global pre_tra
15     # left part
16     if lpart and pre_tra:
17         # connect subtree
18         pivot = nodes(pre_tra[0])
19         pre_tra = pre_tra[1:]
20         root.l = pivot
21         # split left and right subtree
22         mid = lpart.index(pivot.val)
23         lp = lpart[0:mid]
24         rp = lpart[mid+1:]
25         establish(pivot, lp, rp)
26     # right part
27     if rpart and pre_tra:
28         # connect subtree
29         pivot = nodes(pre_tra[0])
30         pre_tra = pre_tra[1:]
31         root.r = pivot
32         # split left and right subtree
33         mid = rpart.index(pivot.val)
34         lp = rpart[0:mid]
35         rp = rpart[mid+1:]
36         establish(pivot, lp, rp)
37
38 establish(root, lp, rp)
39 postorder(root)

```

1.4.2 哈夫曼编码

poj 1521

Question: 输入一个字符串, 分别输出 ASCII(8bit / character) 和哈夫曼编码的长度, 以及压缩比

Sample Input

AAAAABCD

THE_CAT_IN_THE_HAT

END

Sample Output

64 13 4.9

144 51 2.8

```

1 from queue import PriorityQueue
2 s = input()
3 while s != 'END':
4     chaSet = set(item for item in s)
5     q = PriorityQueue()
6
7     # 只有一个字符的情况，建哈夫曼树至少需要两个节点
8     if len(s) == 1:
9         print('8 1 8')
10    else:
11        for item in chaSet:
12            q.put(s.count(item))
13        rst = 0
14        while q.qsize() > 1:
15            a = q.get()
16            b = q.get()
17            q.put(a + b)
18            # 每增加一层，编码长度加1
19            rst += a + b
20
21        # clear queue
22        q.get()
23        print('{} {} {}'.format(str(8 * len(s)),
24                                str(rst),
25                                str(8 * len(s) / rst)))
26
27    s = input()

```

1.5 Heap in Python

```

1 import heapq
2
3 # heapq创建的是小根堆，通过对元素取负可以转换为大根堆
4
5 # create a heap
6 # convert a list to heap
7 lst = [2, 8, 1, 63, 8, 1, 0, 4]
8 heapq.heapify(lst)
9 # 此后堆heap的任何操作都要通过库函数操作
10 print(lst)
11
12 # heapq.heappush(heap, item)
13 heapq.heappush(lst, 5)
14 print(lst)
15
16 # heapq.heappop(heap)
17 # -> pop and return the min element of heap
18 print(heapq.heappop(lst))
19 print(lst)
20
21 # heapq.heappushpop(heap, item)
22 # -> push item into heap, and return the min element
23 print(heapq.heappushpop(lst, 4))
24 print(lst)

```

P3378 【模板】堆

题目描述

给定一个数列，初始为空，请支持下面三种操作：

1. 给定一个整数 x ，请将 x 加入到数列中。
2. 输出数列中最小的数。

3. 删除数列中最小的数（如果有多个数最小，只删除 1 个）。

输入格式

第一行是一个整数，表示操作的次数 n 。

接下来 n 行，每行表示一次操作。每行首先有一个整数 op 表示操作类型。

- 若 $op = 1$ ，则后面有一个整数 x ，表示要将 x 加入数列。
- 若 $op = 2$ ，则表示要求输出数列中的最小数。
- 若 $op = 3$ ，则表示删除数列中的最小数。如果有多个数最小，只删除 1 个。

输出格式

对于每个操作 2，输出一行一个整数表示答案。

输入输出样例 1

输入 1

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

输出 1

```
2
5
```

说明/提示

【数据规模与约定】

- 对于 30% 的数据，保证 $n \leq 15$ 。
- 对于 70% 的数据，保证 $n \leq 10^4$ 。
- 对于 100% 的数据，保证 $1 \leq n \leq 10^6$ ， $1 \leq x < 2^{31}$ ， $op \in \{1, 2, 3\}$ 。

```
1 import heapq
2 lt = list()
3 heapq.heapify(lt)
4 n = int(input())
5 for i in range(n):
6     op = list(map(int, input().split()))
7     if len(op) == 2:
8         heapq.heappush(lt, op[1])
9     elif len(op) == 1:
10         if op[0] == 2:
11             rst = heapq.nsmallest(1, lt)
12             print(rst[0])
13         elif op[0] == 3:
14             heapq.heappop(lt)
```

2 基本算法

2.1 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

2.1.1 反向扫描

找指定和的整数对

问题: 输入 n ($n \leq 100000$) 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

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

```
1 # 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  $O(n)$ 
2 a = list(map(int, input().split()))
3 a.sort()
4 m = int(input())
5
6 # 双指针
7 i, j = 0, len(a) - 1
8 while (i < j):
9     s = a[i] + a[j]
10    #  $s < m$ :  $i$  增加 1, 之后的  $s \geq$  当前  $s$ 
11    if s < m:
12        i += 1
13    elif s > m:
14        j -= 1
15    else:
16        print("{} {}".format(a[i], a[j]))
17        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

```

1 n = int(input())
2 for i in range(n):
3     s = str(input())
4     i, j = 0, len(s) - 1
5     while i < j:
6         flag = False
7         if s[i] == s[j]:
8             flag = True
9         else:
10            flag = False
11            break
12        i += 1
13        j -= 1
14    if (flag):
15        print('yes')
16    else:
17        print('no')
```

2.1.2 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 $sum > s$: $sum - a[i]$, $i++$

```
1 n = int(input())
2 a = list(map(int, input().split()))
3 s = int(input())
4
5 sum = a[0]
6 i, j = 0, 0
7 while i < n and j < n:
8     if sum == s:
9         print("{} {}".format(i, j))
10        sum -= a[i]
11        i += 1
12        j += 1
13        sum += a[j]
14    elif sum < s:
15        j += 1
16        sum += a[j]
17    elif sum > s:
18        sum -= a[i]
19        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

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

```
1 # 尺取法
2 a = list(map(int, input().split()))
3 # a排序, 使得相同元素排列在一起
4 a.sort()
5 n = len(a)
6 # 双指针均从0开始
7 i, j = 0, 0
8 # j始终指向无重复元素部分的最后一个元素
9 while i < n and j < n:
10     # 若i和j指向的元素不同, j++, 将i指向的元素复制到j上
11     # EX: 1 2(j) 3(i) 3
12     # -> 1 2 3(j, i) 3
13     # EX: 1 2 3(j) 3 4(i)
14     # -> 1 2 3 3(j) 4(i)
15     # -> 1 2 3 4(j) 4(i)
16     if a[i] != a[j]:
17         j += 1
18         a[j] = a[i]
19     i += 1
20 unique_a = a[0:j+1]
21 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

```
1 n, c = map(int, input().split())
2 a = list(map(int, input().split()))
3 a.sort()
4
5 i, j, k = 0, 0, 0
6 ans = 0
7
8 for i in range(n):
9     # j, k指向相同元素区间的起点和终点后1个元素
10    # 寻找的对象是区间内的元素 - a[i] = C
11    while j < n - 1 and a[j] - a[i] < c:
12        j += 1
13    while k < n and a[k] - a[i] <= c:
14        k += 1
15    if a[j] - a[i] == c and a[k-1] - a[i] == c and k - 1 >= 0:
16        ans += k - j
17 print(ans)
```

2.2 二分法

2.2.1 Python 二分搜索库 bisect

```
1 from bisect import *
2 def fun(find, x, bias=0):
3     global a
4     index = find(a, x) + bias
5     print("index: {}, element: {}".format(index, a[index]))
6
7 global a
8 a = [1, 2, 4, 4, 4, 5]
9 # target: x
10 x = 4
11
12 # first > x
13 fun(bisect_right, x)
14 # first >= x
15 fun(bisect_left, x)
16 # first = x
17 fun(bisect_left, x)
18 # last = x
19 fun(bisect_right, x, bias=-1)
20 # last <= x
21 fun(bisect_right, 3, bias=-1)
22 # last < x
23 fun(bisect_left, x, bias=-1)
24 # count x in a monotonic array
25 # slow
26 print(a.count(x))
27 # fast, using binary search
28 print(bisect_right(a, x) - bisect_left(a, x))
```

output

```
1 index: 5, element: 5
2 index: 2, element: 4
3 index: 2, element: 4
4 index: 4, element: 4
5 index: 1, element: 2
6 index: 1, element: 2
7 3
8 3
```

2.2.2 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

```
1 # 适用单调递增序列的后继问题
2 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
3 # 适用单调递增序列的前驱问题
4 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 $\geq x$ 的位置

```

1 # 左闭右开 [0, n)
2 l, r = 0, n
3 while l < r:
4     mid = l + (r - l) // 2
5     if a[mid] >= x:
6         r = mid
7     else:
8         l = mid + 1
9 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 $\leq x$ 的位置

```

1 # 左开右闭 (-1, n-1]
2 l, r = -1, n - 1
3 while l < r:
4     mid = l + (r - l + 1) // 2
5     if a[mid] <= x:
6         l = mid
7     else:
8         r = mid - 1
9 return l

```

寻找 minimum

```

1 while l < r:
2     mid = l + (l - r) // 2
3     if check(mid):
4         # reduce
5         r = mid
6     else:
7         # enlarge
8         l = mid + 1
9 # 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 ($n \leq 100000$) 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

```
1 from bisect import *
2
3 a = list(map(int, input().split()))
4 m = int(input())
5 n = len(a)
6 a.sort()
7
8 # ver. 1
9 for i in range(n-1):
10     # bisearch, a[k] = m - a[i]
11     l, r = i+1, n
12     x = m - a[i]
13     while l < r:
14         mid = l + (r - l) // 2
15         if a[mid] >= x:
16             r = mid
17         else:
18             l = mid + 1
19     if a[l] == x:
20         print("{} {}".format(a[i], a[l]))
21
22 # ver. 2
23 for i in range(n-1):
24     # bisearch, a[k] = m - a[i]
25     x = m - a[i]
26     # search from a[i+1] to a[end-]
27     p = bisect_left(a, x, lo=i+1, hi=n)
28     if a[p] == x:
29         print("{} {}".format(a[i], a[p]))
```

2.2.3 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

```

1 # Input:
2 # array
3 # m: amount of subarray
4 # Output:
5 # x: minimum of maximum of sum(all possible subarrays)
6 # subarrays
7
8 from bisect import *
9
10 a = list(map(int, input().split()))
11 m = int(input())
12 n = len(a)
13 l, r = max(a), sum(a)
14 subs = []
15
16 while l < r:
17     mid = l + (r - l) // 2
18     # greedy divide subarray
19     # each sum(subarray) <= mid
20     flag = False
21     idx = 0
22     subs = []
23     for i in range(m):
24         sum = 0
25         while idx < n and sum + a[idx] <= mid:
26             sum += a[idx]
27             idx += 1
28         if idx == n or sum + a[idx] > mid:
29             subs.append(idx-1)
30     # judge
31     if subs[-1] < n-1:
32         flag = False
33     else:
34         flag = True
35     # binary control
36     if flag:
37         # reduce
38         r = mid
39     else:
40         # enlarge
41         l = mid + 1
42
43 print('minimum sum: ', l)
44 for i in range(m):
45     if i == 0:
46         print(a[ : subs[0]+1])
47     else:
48         print(a[subs[i-1]+1 : subs[i]+1])

```

2.2.4 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

```

1 n, c = map(int, input().split())
2 x = []
3 for i in range(n):
4     x.append(int(input()))
5 x.sort()
6
7 def check(x, dis, c):
8     i = 1
9     last = 0
10    c -= 1
11    while c > 0 and i < len(x):
12        while i < len(x) and x[i] - x[last] < dis:
13            i += 1
14        if i < len(x) and x[i] - x[last] >= dis:
15            c -= 1
16            last = i
17            i += 1
18    if c == 0:
19        return True
20    else:
21        return False
22
23 l, r = 0, x[-1] - x[0]
24 ans = 0
25 while l < r:
26     mid = l + (r - l) // 2
27     if check(x, mid, c):
28         ans = mid
29         l = mid + 1
30     else:
31         r = mid
32 print(ans)

```

2.2.5 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

```

1 # 精度, 可以调整
2 eps = 1e-7
3 while r - l > eps:
4     # for ver.
5     # epoch = 100 # 轮次
6     # for i in range(epochs):
7         mid = l + (r - l) / 2.0
8         if check(mid):
9             # reduce range
10            r = mid
11        else:
12            # enlarge range
13            l = mid
14 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

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

```
1 def check(mid, area, f):
2     sum = 0
3     for i in range(len(area)):
4         sum += int(area[i] / mid)
5     if sum >= f:
6         return True
7     else:
8         return False
9
10 eps = 1e-5
11 pi = 3.1415926
12 T = int(input())
13 for t in range(T):
14     n, f = map(int, input().split())
15     cakes = list(map(float, input().split()))
16     maxx = 0
17     area = []
18     for i in range(n):
19         area.append(pi * cakes[i] * cakes[i])
20         if area[i] > maxx:
21             maxx = area[i]
22     l, r = 0, maxx
23     while r - l > eps:
24         mid = l + (r - l) / 2.0
25         if check(mid, area, f):
26             l = mid
27         else:
28             r = mid
29     ans = format(l, '.4f')
30     print(ans)
```

2.3 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

```
1 k = (r - l) / 3.0
2 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 : $N \mid r$

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

```
1 n, l, r = map(float, input().split())
2 n = int(n)
3 cons = list(map(float, input().split()))
4 def f(x, n, cons):
5     sum = 0
6     for i in range(1, n+2):
7         sum += cons[-i] * pow(x, i-1)
8     return sum
9 eps = 1e-6
10 while r - l > eps:
11     k = (r - l) / 3.0
12     mid1, mid2 = l + k, r - k
13     if f(mid1, n, cons) < f(mid2, n, cons):
14         l = mid1
15     else:
16         r = mid2
17 ans = format(l, '.5f')
18 print(ans)
```

三分法函数洛谷 P1883

给定 n 个二次函数 $f_1(x), f_2(x), \dots, f_n(x)$ (均形如 $ax^2 + bx + c$) , 设 $F(x) = \max\{f_1(x), f_2(x), \dots, f_n(x)\}$, 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, $n \leq 100$ 。

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

```

1 def cal(cons, x):
2     ans = -float('inf')
3     for a, b, c in cons:
4         ans = max(ans, a * x ** 2 + b * x + c)
5     return ans
6
7 eps = 1e-9
8 T = int(input())
9 for t in range(T):
10    n = int(input())
11    cons = []
12    for i in range(n):
13        a, b, c = map(float, input().split())
14        cons.append([a, b, c])
15    l = 0
16    r = 1000
17    while r - l > eps:
18        margin = (r - l) / 3.0
19        mid1 = l + margin
20        mid2 = r - margin
21        f1 = cal(cons, mid1)
22        f2 = cal(cons, mid2)
23        if f1 <= f2:
24            r = mid2
25        else:
26            l = mid1
27    ans = cal(cons, l)
28    print(format(ans, '.4f'))

```

2.4 排序与排列

2.4.1 排序

Python 自定排序 key

key function:

Input: x, y

Output: 1: $x > y$

0: $x = y$

-1: $x < y$

```
1 from functools import cmp_to_key
2 def cmp(x, y):
3     # compare x and y
4     # return result
5 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
```

```
1 # 洛谷P1093
2
3 # 自定义比较函数 cmp -> key
4 # x: (id, c, m, e, sum)
5 # Input: x, y
6 # Output:
7 # 1: x > y; 0: x = y; -1: x < y
8 from functools import cmp_to_key
9 def cmp(x, y):
10     if x[4] > y[4]:
11         return 10
12     elif x[4] < y[4]:
13         return -1
14     else:
15         if x[1] > y[1]:
16             return 1
17         elif x[1] < y[1]:
18             return -1
19         else:
20             if x[0] < y[0]:
21                 return 1
22             else:
23                 return -1
24 sort_key = cmp_to_key(cmp)
25
26 n = int(input())
27 stu = []
28 for id in range(1, n+1):
29     c, m, e = map(int, input().split())
30     summ = c + m + e
31     stu.append([id, c, m, e, summ])
32 rst = sorted(stu, key=sort_key, reverse=True)
33 for i in range(5):
34     print('{} {}'.format(rst[i][0], rst[i][-1]))
```

2.4.2 排列

Python 生成全排列

```
1 from itertools import permutations
2 a = [1, 2, 3]
3 # 全排列
4 per = permutations(a)
5 for item in per:
6     print(item)
7 # Output:
8 (1, 2, 3)
9 (1, 3, 2)
10 (2, 1, 3)
11 (2, 3, 1)
12 (3, 1, 2)
13 (3, 2, 1)
14
15 #  $n$ 取 $m$ 排列
16 per = permutations(a, 2)
17 for item in per:
18     print(item)
19 # Output:
20 (1, 2)
21 (1, 3)
22 (2, 1)
23 (2, 3)
24 (3, 1)
25 (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

```
1 # 蓝桥杯2016省赛C++A组第6题
2
3 a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
4 n = len(a)
5 vis = [False] * n
6 b = [None] * n
7 ans = 0
8
9 def permu(s, t):
10     global ans
11     # 输出
12     if s == 12 and b[9] == b[10] * b[11]:
13         ans += 1
14         return
15     # 剪枝
16     if s == 3 and b[0] + b[1] != b[2]:
17         return
18     if s == 6 and b[3] - b[4] != b[5]:
19         return
20     if s == 9 and b[6] * b[7] != b[8]:
21         return
22     # 全排列
23     for i in range(t):
24         if not vis[i]:
25             vis[i] = True
26             b[s] = a[i]
27             permu(s+1, t)
28             vis[i] = False
29
30 permu(0, n)
31 print(ans)
```

2.5 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

```
1 step = 0
2 n, m = map(int, input().split())
3
4 def hanoi(x, y, z, n):
5     global step, m
6     if n == 1:
7         step += 1
8         if step == m:
9             print('#{}: {}->{}'.format(m, x, z))
10    else:
11        hanoi(x, z, y, n-1)
12        step += 1
13        if step == m:
14            print('#{}: {}->{}'.format(m, x, z))
15        hanoi(y, x, z, n-1)
16
17 hanoi('A', 'B', 'C', n)
18 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 $1 \leq i < j < n$ 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

```
1 # hdu 4911
2
3 n, k = map(int, input().split())
4 a = list(map(int, input().split()))
5 count = 0
6 b = [0]*10000
7
8 def merge(l, mid, r):
9     global count, a, b
10    i, j = l, mid+1
11    t = 0
12    while i <= mid and j <= r:
13        if a[i] > a[j]:
14            b[t] = a[j]
15            t, j = t+1, j+1
16            count += mid - i + 1
17        else:
18            b[t] = a[i]
19            t, i = t+1, i+1
20    while i <= mid:
21        b[t] = a[i]
22        t, i = t+1, i+1
23    while j <= r:
24        b[t] = a[j]
25        t, j = t+1, j+1
26    for i in range(t):
27        a[l+i] = b[i]
28
29 def merge_sort(l, r):
30     if l < r:
31         mid = (l + r) // 2
32         merge_sort(l, mid)
33         merge_sort(mid+1, r)
34         merge(l, mid, r)
35
36 merge_sort(0, len(a)-1)
37 if count <= k:
38     print(0)
39 else:
40     print(count - k)
```

2.6 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 $\times 2$, 而是利用二进制数的特性, 将一个数 N 用二进制展开 $N = a_0 2^0 + a_1 2^1 + \dots$, 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: $dp[s][k] = \min\{dp[s][k-1], dp[s + 1 \ll (k-1)][k-1]\}$

max: $dp[s][k] = \max\{dp[s][k-1], dp[s + 1 \ll (k-1)][k-1]\}$

其中, $dp[s][k]$ 表示左端点为 s , 区间长度为 2^k 的区间最值, $1 \ll (k-1)$ 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 = $\min(dp_min[L][k], dp_min[R - (1 \ll k) + 1][k])$

max = $\max(dp_max[L][k], dp_max[R - (1 \ll k) + 1][k])$

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.

```

1 from math import log2
2 n, q = map(int, input().split())
3 # maximum of k
4 p = int(log2(n))
5 # RMQ table, 1-index
6 dp_min = [[0 for _ in range(p+1)] for _ in range(n + 1)]
7 dp_max = [[0 for _ in range(p+1)] for _ in range(n + 1)]
8 # read in list, 1-index
9 a = [-1]
10 for i in range(n):
11     a.append(int(input()))
12
13 # initial dp table
14 # len of sub-range is 1
15 for i in range(1, n+1):
16     dp_min[i][0] = a[i]
17     dp_max[i][0] = a[i]
18 # len of sub-range > 1
19 for k in range(1, p+1):
20     for s in range(1, n):
21         if s + (1 << k) > n + 1:
22             break

```

```
23     dp_min[s][k] = min(dp_min[s][k-1], dp_min[s + (1 << (k-1))][k-1])
24     dp_max[s][k] = max(dp_max[s][k-1], dp_max[s + (1 << (k-1))][k-1])
25
26 # RMQ
27 for _ in range(q):
28     L, R = map(int, input().split())
29     k = int(log2(R - L + 1))
30     max_query = max(dp_max[L][k], dp_max[R - (1 << k) + 1][k])
31     min_query = min(dp_min[L][k], dp_min[R - (1 << k) + 1][k])
32     print(max_query - min_query)
```

2.7 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

```

1 from random import randint
2
3 a = [] # element[0]: val, element[1]: original place
4 for i in range(10):
5     a.append([randint(1, 5000), i])
6
7 print('original list:')
8 print(a)
9
10 # main function
11 # sort
12 a.sort(key=lambda x : x[0])
13 n = len(a)
14 # generate new list
15 newa = [0 for _ in range(n)]
16 for i in range(n):
17     newa[a[i][1]] = i + 1
18
19 print('processed list:')
20 print(newa)
21
22 # IF: same value should be corresponded to the same processed value
23 print('-----same value -> same order value-----')
24 a = [[13, 0], [65, 1], [8, 2], [13, 3], [93, 4], [197, 5]]
25 print('original list')
26 print(a)
27 a.sort(key=lambda x : x[0])
28 n = len(a)
29 # generate new list
30 newa = [0 for _ in range(n)]
31 for i in range(n):
32     if i > 0 and a[i][0] == a[i-1][0]:
33         newa[a[i][1]] = newa[a[i-1][1]]
34         continue
35     newa[a[i][1]] = i + 1
36 print('processed list:')
37 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]]
```

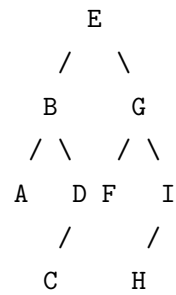
processed list:

```
[2, 4, 1, 2, 5, 6]
```

3 搜索

3.1 BFS 和 DFS 基础

构建二叉树



```
1 def build():
2     a = node('a')
3     b = node('b')
4     c = node('c')
5     d = node('d')
6     e = node('e')
7     f = node('f')
8     g = node('g')
9     h = node('h')
10    i = node('i')
11    root = e
12    root.l = b
13    root.r = g
14    b.l = a
15    b.r = d
16    d.l = c
17    g.l = f
18    g.r = i
19    i.l = h
20    return root
```

3.1.1 BFS

代码实现 BFS 可以使用**队列**. 处理第 i 层的节点 a 时, 将 a 的第 $i+1$ 层的所有孩子节点放入队尾.

```
1 class node():
2     def __init__(self, v=None, l=None, r=None):
3         self.value = v
4         self.l = l
5         self.r = r
6
7 # using queue to BFS
8 root = build()
9 q = deque()
10 q.append(root)
11 while len(q) > 0:
12     temp = q.popleft()
13     print(temp.value, end=' ')
14     if temp.l:
15         q.append(temp.l)
16     if temp.r:
17         q.append(temp.r)
```

e b g a d f i c h

BFS 遍历二叉树时, 每条边需要且只需要检查一次, 时间复杂度为 $O(m)$, m 为边的数量; 每个点只进出队列一次, 空间复杂度为 $O(n)$, n 为点的数量.

BFS 适用于寻找全局最优解. BFS 编码时需要注意去重, 待搜索的状态已经搜索过时, 不再将此状态放入队尾.

3.1.2 DFS

代码实现 DFS 可以使用递归, DFS 模板框架为

```

1 GLOBAL ans;
2 Function dfs(Layer NO., other parameters):
3   IF (exit loop):
4     ans <- updated ans;
5     return; // back to previous layer
6   ENDIF
7   剪枝
8   FOR (下一层所有可能情况):
9     IF (used[i] == false): // have not been used
10      used[i] = true // mark status
11      dfs(layer + 1, other parameters)
12      used[i] = false // recover status
13    ENDIF
14  ENDFOR
15  return; // back to previous layer

```

DFS 常用操作

```

1      # DFS
2 root = build()
3
4 # 时间戳
5 dfn = []
6 dfn_timer = 0
7 def dfn_node(root):
8   global dfn, dfn_timer
9   if root:
10    dfn_timer += 1
11    dfn.append((root.value, dfn_timer))
12    print('{0}: {1}'.format(root.value, dfn_timer), end=' ')
13    if root.l:
14      dfn_node(root.l)
15    if root.r:
16      dfn_node(root.r)
17
18 print('时间戳')
19 dfn_node(root)
20 print()
21
22 # DFS序
23 visit_timer = 0
24 def visit_order(root):
25   global visit_timer
26   if root:
27    visit_timer += 1
28    # 第一次 访问
29    print('{0}: {1}'.format(root.value, visit_timer), end=' ')
30    if root.l:
31      visit_order(root.l)
32    if root.r:
33      visit_order(root.r)
34    # 第二次 回溯
35    visit_timer += 1
36    print('{0}: {1}'.format(root.value, visit_timer), end=' ')
37

```

```

38 print('DFS序')
39 visit_order(root)
40 print()
41
42 # 树的深度
43 deep = dict()
44 deep_timer = 0
45 def deep_node(root):
46     global deep_timer, deep
47     if root:
48         deep_timer += 1
49         deep[root.value] = deep_timer
50         print('{l}: {r}'.format(root.value, deep_timer), end=' ')
51         if root.l:
52             deep_node(root.l)
53         if root.r:
54             deep_node(root.r)
55         deep_timer -= 1
56
57 print('树的深度')
58 deep_node(root)
59 print()
60
61 # 子树节点总数
62 num = dict()
63 def num_node(root):
64     global num
65     if not root:
66         return 0
67     else:
68         num[root.value] = num_node(root.l) + num_node(root.r) + 1
69         print('{l}: {r}'.format(root.value, num[root.value]), end=' ')
70         return num[root.value]
71
72 print('子树节点总数')
73 num_node(root)
74 print()

```

时间戳

e: 1 b: 2 a: 3 d: 4 c: 5 g: 6 f: 7 i: 8 h: 9

DFS序

e: 1 b: 2 a: 3 a: 4 d: 5 c: 6 c: 7 d: 8 b: 9 g: 10 \
f: 11 f: 12 i: 13 h: 14 h: 15 i: 16 g: 17 e: 18

树的深度

e: 1 b: 2 a: 3 d: 3 c: 4 g: 2 f: 3 i: 3 h: 4

子树节点总数

a: 1 c: 1 d: 2 b: 4 f: 1 h: 1 i: 2 g: 4 e: 9

DFS 适用于寻找一个可行解. 编码时需要注意剪枝, 当某一个分支已经不符合要求时, 则不再在此分支上深入搜索.

3.2 剪枝与判重

1. 可行性剪枝: 检查当前状态, 出现条件不合法则剪枝
2. 搜索顺序剪枝: 搜索树有多个层次和分支, 不同的搜索顺序会造成不同的复杂度
3. 最优性剪枝: 如果当前花费的代价已经超过前面搜索到的最优解, 退出本分支. EX. 当前路径已经长于目前搜索到的最短路径
4. 排除等效冗余: 沿着当前节点搜索全部分支, 结果一样, 退出. 一般与组合问题有关, EX. 多个数字凑成一个大数, 数字的顺序不影响结果
5. 记忆化搜索: 将已经计算出的结果保存起来, 常用于 DP.

3.2.1 BFS 判重 蓝桥 642

有 9 只盘子, 排成 1 个圆圈。其中 8 只盘子内装着 8 只蚱蜢, 有一个是空盘。我们把这些蚱蜢顺时针编号为 1~8。

每只蚱蜢都可以跳到相邻的空盘中, 也可以再用点力, 越过一个相邻的蚱蜢跳到空盘中。

请你计算一下, 如果要使得蚱蜢们的队形改为按照逆时针排列, 并且保持空盘的位置不变 (也就是 1~8 换位, 2~7 换位,...), 至少要经过多少次跳跃?

化圆为线, 将空盘子标记为 0, 从 12 点方向依顺时针将蚂蚱编号合并为字符串, 初始状态为 012345678. 若 1 跳到盘子里, 变为 1023456789; 若 7 越过 8 跳到盘子里, 变为 712345608. 目标状态为 087654321.

```

1 from collections import deque
2
3 q = deque()
4 mapp = set()
5
6 def solve():
7     global q, mapp
8     while len(q) > 0:
9         now = q[0]
10        q.popleft()
11        s = now[0]
12        step = now[1]
13        if s == '087654321':
14            print(step)
15            return
16        idx = 0
17        for i in range(9):
18            if s[i] == '0':
19                idx = i
20                break
21        for j in range(idx-2, idx+3, 1):
22            if j == idx:
23                continue
24            pos = (j + 9) % 9
25            temp_lst = list(s)
26            temp_lst[idx] = s[pos]
27            temp_lst[pos] = s[idx]
28            ns = ''.join(temp_lst)
29            if ns not in mapp:
30                mapp.add(ns)
31                q.append((ns, step+1))
32
33 s = '012345678'
34 q.append((s, 0))
35 mapp.add(s)
36 solve()

```

3.2.2 可行性剪枝 poj3278

在一条直线上, 农夫在 N 位置, 奶牛在 K 位置. 农夫要抓到牛, 有 3 种移动方法: 若位于 X , 可以移动到 $X-1$, $X+1$, $2X$. 农夫需要走多少次才能从 N 到达 K .

Input: 两个整数 N, K . $0 \leq N, K \leq 100000$

Output: 最少移动次数

使用可行性剪枝, 当 $X > K$ 时, 只能移动到 $X-1$, 不能扩大 X .

```
1 from collections import deque
2
3 n, k = map(int, input().split())
4 q = deque()
5 q.append((n, 0))
6 rst = []
7 mapp = set()
8
9 def solve():
10     global q, n, k, mapp, rst
11     while len(q) > 0:
12         now = q[0]
13         q.popleft()
14         x = now[0]
15         step = now[1]
16         if x == k:
17             rst.append(step)
18         if x < 0:
19             mapp.add(x)
20             continue
21
22         if x > k:
23             if x-1 not in mapp:
24                 q.append((x-1, step+1))
25                 mapp.add(x-1)
26         else:
27             if x-1 not in mapp:
28                 q.append((x-1, step+1))
29                 mapp.add(x-1)
30             if x+1 not in mapp:
31                 q.append((x+1, step+1))
32                 mapp.add(x+1)
33             if 2*x not in mapp:
34                 q.append((2*x, step+1))
35                 mapp.add(2*x)
36
37
38 solve()
39 rst.sort()
40 print(rst[0])
```

>> 5 17

<< 4

3.2.3 最优性剪枝 洛谷 1118

输入一个 $1 - n$ 的序列 a_i , 每次将相邻两个数相加, 形成新的序列, 直到剩下一个数字为止. EX.

```

3    1    2    4
  4    3    6
    7    9
      16

```

现在知道 n 和最终的数字 sum , 要求出最初的序列 a . 若答案有多个, 则输出字典序最小的一个.

1. 从序列计算到最后一个数, 实际上是杨辉三角的应用, 可以先计算出来杨辉三角的系数.

2. **最优性剪枝**: 若当前枚举到的序列中有一段, 这段子序列相加进行上述累加的结果已经超过 sum , 则整个序列都可以排除.

对于 Python: 生成全排列可以使用 `itertools` 中的 `permutations`; 列表可以直接进行大小比较, 放回的结果是第一个不同数字的大小比较结果; 查重可以将数字使用空格分隔再转换为字符串进行.

```

1 from itertools import permutations
2 from copy import deepcopy
3
4 # prepare
5
6 cal_tab = dict()
7 base = [0] * 12
8 base[0] = 1
9 cal_tab[1] = base
10 pass_dict = set()
11
12 def move_add(a):
13     i = 1
14     rst = deepcopy(base)
15     while i < len(a):
16         if a[i-1] == 0:
17             return rst
18         rst[i] = a[i-1] + a[i]
19         i += 1
20     return rst
21
22 def tri_sum(a):
23     global cal_tab
24     n = len(a)
25     tab = cal_tab[n]
26     rst = 0
27     for i in range(n):
28         rst += a[i] * tab[i]
29     return rst
30
31 for i in range(2, 13):
32     temp = deepcopy(cal_tab[i-1])
33     cal_tab[i] = move_add(temp)
34
35 # begin receive inputs
36 n, summ = map(int, input().split())
37 origin = [i for i in range(1, n+1)]
38 rst = []
39 for per in permutations(origin):
40     per = list(per)
41     check = ' '.join(map(str, per))
42     flag = False
43     for item in pass_dict:
44         if item in check:

```

```
45         flag = True
46         break
47     if flag:
48         continue
49     if tri_sum(per) == summ:
50         if len(rst) == 0:
51             rst = per
52         elif per < rst:
53             rst = per
54
55     elif tri_sum(per) > summ:
56         per.pop()
57         while len(per) > 0:
58             if tri_sum(per) > summ:
59                 pass_dict.add(' '.join(map(str, per)))
60                 per.pop()
61             else:
62                 break
63
64 print(' '.join(map(str, rst)))
```

```
>> 4 16
```

```
<< 3 1 2 4
```

3.2.4 优化搜索顺序 排除等效冗余 洛谷 1120

乔治有一些同样长的小木棍，他把这些木棍随意砍成几段，直到每段的长都不超过 50。现在，他想把小木棍拼接成原来的样子，但是却忘记了自己开始时有多少根木棍和它们的长度。给出每段小木棍的长度，编程帮他找出原始木棍的最小可能长度。

输入格式

第一行是一个整数 n ，表示小木棍的个数。

第二行有 n 个整数，表示各个木棍的长度 a_i

输出格式

输出一行一个整数表示答案。

1. 优化搜索顺序: 把小木棍按长度从大到小排列, 再按照从大到小的顺序尝试拼接.

2. 排除等效冗余: 即按照优化 1 进行优化, 因为在拼接中, 从大到小地拼接和从小到大的拼接一样.

3. 长度优化: 所有可行长度 D 是小木棍总长度的一个约数. 令木棍长度为 $summ$, 则 D 的范围为 $[1, summ]$. 再令 K 为原始木棍根数, 则 $summ = K \cdot D, K \in \mathbb{Z}^+ \implies D \in \{n | summ/n \in \mathbb{Z}^+\}$. 若 $summ$ 不可分为多个 D 之和, 则该 D 不可用于分割.

```

1 n = int(input())
2 a = list(map(int, input().split()))
3 a.sort(reverse=True)
4 a_mark = [False for i in range(len(a))]
5 summ = sum(a)
6
7 def reset():
8     global a_mark
9     for i in range(len(a_mark)):
10         a_mark[i] = False
11 def combine(length):
12     global a, a_mark
13     if length == 0:
14         return True
15     for i in range(len(a)):
16         flag = False
17         if a_mark[i]:
18             continue
19         else:
20             if a[i] > length:
21                 continue
22             else:
23                 a_mark[i] = True
24                 flag = combine(length - a[i])
25                 if not flag:
26                     a_mark[i] = False
27     if flag:
28         return True
29
30 for length in range(1, summ+1):
31     reset()
32     flag = True
33     if summ % length != 0:
34         continue
35     else:
36         k = summ // length
37         for i in range(k):
38             flag = combine(length)
39             if not flag:
40                 reset()
41                 break
42     if flag:
43         print(length)
44         break

```

```
>> 9
>> 5 2 1 5 2 1 5 2 1
<< 6
```

3.3 洪水填充

从一个种子点开始, 扩散到邻居, 再不断扩散到邻居的邻居的过程. 使用 BFS 和 DFS 都可以, DFS 更简单. EX.

```
1 import math
2 from time import sleep
3
4 def show(region):
5     print()
6     for i in range(15):
7         for j in range(15):
8             print(region[i][j], end=' ')
9         print()
10
11 region = [[0 for _ in range(15)]for _ in range(15)]
12 for i in range(15):
13     for j in range(15):
14         if 4.5 < math.sqrt((i-7)**2+(j-7)**2) < 5.5:
15             region[i][j] = 1
16
17 print('original')
18 show(region)
19 sleep(1)
20
21 '''
22 core
23 '''
24 def floodfill(x, y, new_color, old_color):
25     global region
26     if math.sqrt((x-7)**2+(y-7)**2) < 5.5 and region[y][x] == old_color \
27         and 0 <= x <= 14 and 0 <= y <= 14:
28         region[y][x] = new_color
29         show(region)
30         sleep(1)
31         floodfill(x+1, y, 1, 0)
32         floodfill(x-1, y, 1, 0)
33         floodfill(x, y+1, 1, 0)
34         floodfill(x, y-1, 1, 0)
35
36 floodfill(3, 5, 1, 0)
```

3.4 BFS 与最短路径

在所有相邻点的距离**相等**时, BFS 是最优的最短距离算法; 若距离**不相等**, 需要使用 Dijkstra 等通用的标准算法.

蓝桥 602 改

给出地图, 1 为障碍, 0 为可以通行的地方. 迷宫的左上角为入口, 右下角为出口. 找出一种通过迷宫的方式, 其使用的步数最少. 对每一步使用 DULR 表示, DULR 分别为下上左右.

```
010000
000100
001001
110000
```

```
1 from collections import deque
2
3 class node:
4     def __init__(self, x, y, path):
5         self.x = x
6         self.y = y
7         self.path = path
8
9 rows = 30
10 cols = 50
11 a = \
12 '01010101001011001001010110010110100100001000101010' \
13 ...(略)
14 '0011110000100001000000011011100000000100000001011' \
15 '10000001100111010111010001000110111010101101111000'
16 mapp = [[None for _ in range(cols)]for _ in range(rows)]
17 for r in range(rows):
18     for c in range(cols):
19         mapp[r][c] = int(a[c + r * cols])
20
21 vis = [[0 for _ in range(cols)]for _ in range(rows)]
22 op = [[1, 0],
23        [-1, 0],
24        [0, 1],
25        [0, -1]]
26 P = ['R', 'L', 'D', 'U']
27
28 def bfs():
29     start = node(0, 0, '')
30     vis[0][0] = 1
31     q = deque()
32     q.append(start)
33     while len(q) > 0:
34         now = q.popleft()
35         if now.x == cols-1 and now.y == rows-1:
36             print(now.path)
37             # broadcast
38             for i in range(4):
39                 next = node(now.x + op[i][0], now.y + op[i][1], now.path+P[i])
40                 # out of range
41                 if next.x < 0 or next.y < 0 or next.x > cols-1 or next.y > rows-1:
42                     continue
43                 if vis[next.y][next.x] == 1 or mapp[next.y][next.x] == 1:
44                     continue
45                 vis[next.y][next.x] = 1
46                 q.append(next)
47
48 bfs()
```

4 DP 基本

4.1 DP 的两种编程方法

以斐波那契数列计算为例: $F_1 = F_2 = 1, F_i = F_{i-1} + F_{i-2}, i \geq 3$

简单递归

```
1 def classic_fib(n):
2     if n == 1 or n == 2:
3         return 1
4     return classic_fib(n-1) + classic_fib(n-2)
```

4.1.1 自顶向下结合记忆化

依然采取递归的程序结构, 但是计算返回值前先检查待求结果是否已经被计算出. 解决每一个子问题之后存储结果, 需要时直接返回已经缓存的结果.

```
1 memo = dict()
2 def memo_fib(n):
3     global memo
4     if n == 1 or n == 2:
5         return 1
6     elif n in memo.keys():
7         return memo[n]
8     memo[n] = memo_fib(n-1) + memo_fib(n-2)
9     return memo[n]
```

4.1.2 自底向上结合制表

规避了递归编程. 在解决大问题时先解决小问题, 逐步递推到大问题. 递推过程一般需要填写多维表格 dp, 编码时会使用若干 for 循环体填表.

```
1 def dp_fib(n):
2     dp = list()
3     dp.append(1)
4     dp.append(1)
5     for i in range(2, n):
6         dp.append(dp[i-2] + dp[i-1])
7     return dp[-1]
```

```
n = 10
print(classic_fib(n))
print(memo_fib(n))
print(dp_fib(n))
>>
55
55
55
```

4.2 DP 的设计与实现

以 0/1 背包问题为例 (**hdu 2602**), 背包容量为 C , 骨头数量为 N , 已知每块骨头的价值 v_i 和体积 c_i , 求背包内骨头的最大可能价值

4.2.1 dp 状态设计

总共有 N 块骨头, 背包总体积为 C . 令二维数组 $dp[i][j]$ 为第 i 块骨头放进容量为 j 的背包后, 背包内的最大价值. 考虑到自底向上递推需要的初始化, $i \in [0, N]; j \in [0, C]$, 且 $dp[0][x] = 0, dp[x][n] = 0$ (放入 0 块物品的总价值为 0, 空间为 0 的总价值为 0)

4.2.2 状态转移方程

对于容量为 c 的物块, 当前背包的容量为 j , 放不放入背包有两种情况:

1. $c > j$, 无法放入, $dp[i][j] = dp[i-1][j]$. 直接继承第 $i-1$ 个物品放入空间为 j 的背包后, 背包的最大价值.
2. $c \leq j$, 可以放入, 此时分为两种情况

放入: 容量为 j 的背包需要腾出 c 来存放物品, 因此总价值为 $dp[i][j-c]$ (1)

不放入: 同情况 1, 总价值仍为 $dp[i-1][j]$ (2)

此情况下的最终容量应为 $\max\{(1), (2)\}$

4.2.3 递推实现

```

1 N, C = map(int, input().split())
2 vs = list(map(int, input().split()))
3 cs = list(map(int, input().split()))
4 dp = [[0 for _ in range(C+1)] for _ in range(N+1)]
5 # fill dp table, skip the first line
6 for i in range(1, N+1):
7     v = vs[i-1]
8     c = cs[i-1]
9     for j in range(C+1):
10         # judge
11         if c > j:
12             dp[i][j] = dp[i-1][j]
13         else:
14             dp[i][j] = max(dp[i-1][j], dp[i-1][j-c] + v)
15
16 for line in range(N+1):
17     print(dp[line])
18 print(f'\nfinal result: {dp[N][C]}')
```

4.2.4 记忆化实现

```

1 def cal_dp(i, j, c=0, v=0):
2     global dp
3     if i == 0 or j == 0:
4         return 0
5     if dp[i][j] != 0:
6         return dp[i][j]
7     if c > j:
8         return cal_dp(i-1, j)
9     else:
10        ans = max(cal_dp(i-1, j-c) + v, cal_dp(i-1, j))
11        dp[i][j] = ans
12        return ans
13
14 N, C = map(int, input().split())
15 vs = list(map(int, input().split()))
```

```

16 cs = list(map(int, input().split()))
17 dp = [[0 for _ in range(C+1)]for _ in range(N+1)]
18 for i in range(1, N+1):
19     c = cs[i-1]
20     v = vs[i-1]
21     for j in range(C+1):
22         dp[i][j] = cal_dp(i, j, c, v)
23
24
25 for line in range(N+1):
26     print(dp[line])
27 print(f'\nfinal result: {dp[N][C]}')
```

4.3 滚动数组简化优化空间占用

多维数组在空间占用上较大, 而在状态转移方程中, 新一行的计算只依赖于上一行, 不再需要访问更早之前计算出的结果. 因此可以复用之前的空间.

但是滚动数组损失了中间信息, 最终将无法输出具体的方案.

4.3.1 交替滚动

dp 定义为 $dp[2][j]$, 行号使用 new 和 old 分别指向当前计算的行和上一行.

hdu 2602 递推的交替滚动实现.

```

1 def swap(a, b):
2     temp = b
3     b = a
4     a = temp
5     return a, b
6
7 N, C = map(int, input().split())
8 vs = list(map(int, input().split()))
9 cs = list(map(int, input().split()))
10 dp = [[0 for _ in range(C+1)]for _ in range(2)]
11 # 此处的 01 赋值是为了在接下来的循环中, new 和 old 的取值更加符合直觉
12 new = 0
13 old = 1
14 # fill dp table, skip the first line
15 for i in range(1, N+1):
16     v = vs[i-1]
17     c = cs[i-1]
18     new, old = swap(new, old)
19     for j in range(C+1):
20         # judge
21         if c > j:
22             dp[new][j] = dp[old][j]
23         else:
24             dp[new][j] = max(dp[old][j], dp[old][j-c] + v)
25
26 print(dp[new][C])
```

4.3.2 自我滚动

缺, 待补

4.4 经典线性 DP 与优化策略

4.4.1 0/1 背包问题

同前述, 略

4.4.2 多重背包问题 二进制拆分优化

在 0/1 背包问题的基础上, 每种物体除了重量 w , 价值 v 外, 还有一个属性数量 m . 即同一种物体可以在背包中放置多个.

解法 1 转化为简单 0/1 背包: 取消数量属性, 同一种的多个物体直接视为多个物体, 只是重量和价值相等. 编码时, 这种做法会导致超时.

二进制拆分优化: 将每种物体的数量从小到大拆分为 $2^0 + 2^1 + \dots$ 的组合, 若不能恰好被 2 的指数拆分, 最后一个为余数. 假设一种物体有 15 个, 将 15 按照二进制分解为 $15 = 1 + 2 + 4 + 8$; 假如有 16 个, 拆分为 $1 + 2 + 4 + 8 + 1$, 拆分到 8 时, 当前组合的数量为 2^3 , 剩余 $1 < 2 \cdot 2^3$, 直接作为最后一个组合的数量.

将每一种物体都进行上述拆分, 得到的多组物体视作新的需要放入背包的物体, 进行简单的 0/1 背包问题求解.

原理: 任何数都可以分解为若干 2 的指数与余数的和, 即 $x = (\sum 2^i) + r$, 因此, 对于同一种物体, 在背包中放置 x 个, 等价于从经过二进制分解的多个物体组合中放置数量恰好的几个组合.

问题转化思路为: 一种物体可以放置多个 \implies 将十进制的组合问题转换为 2 进制数的组合问题以减少组合数量, 减少运行时间 \implies **二进制分解优化** \implies 新的物品 (重量, 价值) 组合 \implies 0/1 背包问题

洛谷 P1776

输入: 第一行为两个整数 n 和 W , 分别表示宝物种数和采集车的最大载重。

接下来 n 行每行三个整数: w_i, c_i, m_i , 表示第 i 个物体的价值, 体积, 数量。

输出: 输出仅一个整数, 表示在采集车不超载的情况下收集的宝物的最大价值。

```

1 def swap(a, b):
2     return b, a
3
4 n, W = map(int, input().split())
5 a = [None for _ in range(n)]
6 for i in range(n):
7     obj = list(map(int, input().split()))
8     a[i] = obj
9     new_w = []
10    new_v = []
11
12 for i in range(n):
13     v, w, m = a[i]
14     j = 1
15     while j > 0:
16         # add in new packaged objects
17         new_w.append(j*w)
18         new_v.append(j*v)
19         # update remained m
20         a[i][2] -= j
21         # update j
22         if 2*j < a[i][2]:
23             j *= 2
24         else:
25             j = a[i][2]
26
27 # 0/1 package prob. of new_w, new_v
28 n = len(new_w)
29 dp = [[0 for _ in range(W+1)] for _ in range(2)]
30 new = 0
31 old = 1
32 for i in range(n):
33     w = new_w[i]
34     v = new_v[i]
35     new, old = swap(new, old)
36     for j in range(W+1):
37         if w > j:
38             dp[new][j] = dp[old][j]
39         else:

```

```
40         dp[new][j] = max(dp[old][j], dp[old][j-w] + v)
41
42 print(dp[new][W])
```
