



# Notes for CS-101

Contributed by Stretcher , 2023.12.27

## 目录

### Notes for CS-101

#### 目录

##### 数据结构

heapq模块

queue模块

    queue.LifoQueue(maxsize=0)

    PriorityQueue(maxsize=0)

collections模块

    namedtuple

    deque

    OrderedDict

    Counter

bisect模块

array 模块

calendar模块

树状数组\*

##### 素数筛法

    欧拉筛法

    埃氏筛法

##### 贪心算法

    二分法

    Huffman编码

    最长递增子序列

        二分查找方法

        树状数组方法

    最小非减子序列数

动态规划  
寻找最大矩形  
整数划分  
背包问题  
    01背包  
    完全背包  
    多重背包  
    多重背包可行性  
    最优方案的总数  
最长公共子序列  
图搜索  
    拼接问题  
    Dijkstra算法  
    SPFA算法  
字符串  
    is判断方法  
    ASCII码  
    textwrap模块  
    re模块  
    KMP算法\*  
威佐夫博弈  
部分未提及的标准库  
    布尔运算  
    sys模块  
    itertools模块  
    decimal模块

## 数据结构

### *heapq 模块*

**heapq** 里面没有直接提供建立大根堆的方法，可以采取如下方法：每次 `push` 时给元素加一个负号（即取相反数），此时最小值变最大值

函数及参数	作用
<code>heapq.heappush(heap, item)</code>	将 <code>item</code> 的值加入 <code>heap</code> 中，保持堆的不变性
<code>heapq.heappop(heap)</code>	弹出并返回 <code>heap</code> 的最小的元素，保持堆的不变性。如果堆为空，抛出 <code>IndexError</code> 。 使用 <code>heap[0]</code> ，可以只访问最小的元素而不弹出它
<code>heapq.heappushpop(heap, item)</code>	将 <code>item</code> 放入堆中，然后弹出并返回 <code>heap</code> 的最小元素。该组合操作比先调用 <code>heappush()</code> 再调用 <code>heappop()</code> 运行起来更有效率
<code>heapq.heapify(x)</code>	将 list <code>x</code> 转换成堆，原地，线性时间内
<code>heapq.heapreplace(heap, item)</code>	弹出并返回 <code>heap</code> 中最小的一项，同时推入新的 <code>item</code> 。堆的大小不变。如果堆为空则引发 <code>IndexError</code> 。这个单步骤操作比 <code>heappop()</code> 加 <code>heappush()</code> 更高效，并且在使用固定大小的堆时更为适宜。 <code>pop/push</code> 组合总是会从堆中返回一个元素并将其替换为 <code>item</code> 。返回的值可能会比新加入的值大。如果不希望如此，可改用 <code>heappushpop()</code> 。它的 <code>push/pop</code> 组合返回两个值中较小的一个，将较大的留在堆中

该模块还提供了三个基于堆的通用目的函数。

## 函数及参数

## 作用

heapq.merge(iterables*, key=None, reverse=False)	将多个已排序的输入合并为一个已排序的输出（例如，合并来自多个日志文件的带时间戳的条目）。返回已排序值的 iterator。类似于 sorted(itertools.chain(*iterables)) 但返回一个可迭代对象，不会一次性地将数据全部放入内存，并假定每个输入流都是已排序的（从小到大）。具有两个可选参数，它们都必须指定为关键字参数。key 指定带有单个参数的 key function，用于从每个输入元素中提取比较键。默认值为 None (直接比较元素)。reverse 为一个布尔值。如果设为 True，则输入元素将按比较结果逆序进行合并。要达成与 sorted(itertools.chain(*iterables), reverse=True) 类似的行为，所有可迭代对象必须是已从大到小排序的。在 3.5 版更改：添加了可选的 key 和 reverse 形参
heapq.nlargest(n, iterable, key=None)	从 iterable 所定义的数据集中返回前 n 个最大元素组成的列表。如果提供了 key 则其应指定一个单参数的函数，用于从 iterable 的每个元素中提取比较键（例如 key=str.lower）。等价于：sorted(iterable, key=key, reverse=True)[:n]
heapq.nsmallest(n, iterable, key=None)	从 iterable 所定义的数据集中返回前 n 个最小元素组成的列表。如果提供了 key 则其应指定一个单参数的函数，用于从 iterable 的每个元素中提取比较键（例如 key=str.lower）。等价于：sorted(iterable, key=key)[:n]

后两个函数在 n 值较小时性能最好。对于更大的值，使用 sorted() 函数会更有效率。此外，当 n==1 时，使用内置的 min() 和 max() 函数会更有效率。如果需要重复使用这些函数，请考虑将可迭代对象转为真正的堆。

## queue 模块

- 1 此包中的常用方法(q = queue.queue()):
- 2 q.qsize() 返回队列的大小
- 3 q.empty() 如果队列为空，返回True，反之False
- 4 q.full() 如果队列满了，返回True，反之False
- 5 q.full 与 maxsize 大小对应
- 6 q.get([block[, timeout]]) 获取队列，timeout等待时间
- 7 q.get\_nowait() 相当q.get(False)
- 8 非阻塞 q.put(item) 写入队列，timeout等待时间
- 9 q.put\_nowait(item) 相当q.put(item, False)
- 10 q.task\_done() 在完成一项工作之后，q.task\_done() 函数向任务已经完成的队列发送一个信号
- 11 q.join() 实际上意味着等到队列为空，再执行别的操作

## queue.LifoQueue(maxsize=0)

后进先出 (Last In First Out: LIFO) 队列，最后进入队列的数据拥有出队列的优先权，就像栈一样。入参 maxsize 与先进先出队列的定义一样。

```
1 import queue
2 q = queue.LifoQueue() # 创建 LifoQueue 队列
3 for i in range(3):
4     q.put(i) # 在队列中依次插入0、1、2元素
5 for i in range(3):
6     print(q.get()) # 依次从队列中取出插入的元素，数据元素输出顺序为2、1、0
```

## PriorityQueue(maxsize=0)

优先级队列，比较队列中每个数据的大小，值最小的数据拥有出队列的优先权。数据一般以元组的形式插入，典型形式为(priority\_number, data)。如果队列中的数据没有可比性，那么数据将被包装在一个类中，忽略数据值，仅仅比较优先级数字。入参 maxsize 与先进先出队列的定义一样。

```
1 import queue
2 q = queue.PriorityQueue() # 创建 PriorityQueue 队列
3 data1 = (1, 'python')
4 data2 = (2, '-')
5 data3 = (3, '100')
6 style = (data2, data3, data1)
7 for i in style:
8     q.put(i) # 在队列中依次插入元素 data2、data3、data1
9 for i in range(3):
10    print(q.get()) # 依次从队列中取出插入的元素，数据元素输出顺序为 data1、data2、data3
```

## *collections*模块

### namedtuple

```
1 import collections
2
3 point = collections.namedtuple('Points', ['x', 'y'])
4 p1 = point(2, 3)
5 p2 = point(4, 2)
6
7 print(p1) # Points(x=2, y=3)
8 print(p2) # Points(x=4, y=2)
9
10 a= [11, 3]
11 p1._make(a)
12 print(p1) # Points(x=11, y=3)
13
14 p1._replace(x=5)
15 print(p1) # Points(x=5, y=3)
```

### deque

```
1 from collections import deque
2
3 q = deque(['a', 'b', 'c'], maxlen=10)
4 # 从右边添加一个元素
5 q.append('d')
6 print(q) # deque(['a', 'b', 'c', 'd'], maxlen=10)
7
8 # 从左边删除一个元素
9 print(q.popleft()) # a
10 print(q) # deque(['b', 'c', 'd'], maxlen=10)
11
12 # 扩展队列
13 q.extend(['i', 'j'])
14 print(q) # deque(['b', 'c', 'd', 'i', 'j'], maxlen=10)
15
16 # 查找下标
17 print(q.index('c')) # 1
18
19 # 移除第一个'd'
20 q.remove('d')
21 print(q) # deque(['b', 'c', 'i', 'j'], maxlen=10)
22
```

```

23 # 逆序
24 q.reverse()
25 print(q) # deque(['j', 'i', 'c', 'b'], maxlen=10)
26
27 # 最大长度
28 print(q maxlen) # 10

```

```

1 append(x): 添加 x 到右端。
2 appendleft(x): 添加 x 到左端。
3 clear(): 移除所有元素，使其长度为0。
4 copy(): 创建一份浅拷贝。3.5 新版功能。
5 count(x): 计算deque中个数等于 x 的元素。3.2 新版功能。
6 extend(iterable): 扩展deque的右侧，通过添加iterable参数中的元素。
7 extendleft(iterable): 扩展deque的左侧，通过添加iterable参数中的元素。注意，左添加时，在结果中
iterable参数中的顺序将被反过来添加。
8 index(x[, start[, stop]]): 返回第 x 个元素（从 start 开始计算，在 stop 之前）。返回第一个匹配，如
果没找到的话，升起 ValueError 。3.5 新版功能。
9 insert(i, x): 在位置 i 插入 x 。如果插入会导致一个限长deque超出长度 maxlen 的话，就升起一个
IndexError 。3.5 新版功能。
10 pop(): 移去并且返回一个元素，deque最右侧的那个。如果没有元素的话，就升起 IndexError 索引错误。
11 popleft(): 移去并且返回一个元素，deque最左侧的那个。如果没有元素的话，就升起 IndexError 索引错
误。
12 remove(value): 移去找到的第一个 value。如果没有的话就升起 ValueError 。
13 reverse(): 将deque逆序排列。返回 None 。3.2 新版功能。
14 rotate(n=1): 向右循环移动 n 步。如果 n 是负数，就向左循环。如果deque不是空的，向右循环移动一步就等
价于 d.appendleft(d.pop()) ， 向左循环一步就等价于 d.append(d.popleft()) 。
15 Deque对象同样提供了一个只读属性：
16 maxlen: Deque的最大尺寸，如果没有限定的话就是 None 。

```

## OrderedDict

`popitem(last=True)`：有序字典的 `popitem()` 方法移除并返回一个 `(key, value)` 键值对。如果 `last` 值为真，则按 LIFO 后进先出的顺序返回键值对，否则就按 FIFO 先进先出的顺序返回键值对。

```

1 from collections import OrderedDict
2
3 d = OrderedDict(a=1, b=2, c=3, d=4,e=5)
4 print(d) # OrderedDict([('a', 1), ('b', 2), ('c', 3), ('d', 4), ('e', 5)])
5 print(d.popitem(last=True)) # ('e', 5)
6 print(d.popitem(last=False)) # ('a', 1)
7 print(d) # OrderedDict([('b', 2), ('c', 3), ('d', 4)])
8
9
10 d = OrderedDict(a=1, b=2, c=3, d=4,e=5)
11 print(d) # OrderedDict([('a', 1), ('b', 2), ('c', 3), ('d', 4), ('e', 5)])
12
13 d.move_to_end(key='c', last=True)
14 print(d) # OrderedDict([('a', 1), ('b', 2), ('d', 4), ('e', 5), ('c', 3)])
15
16 d.move_to_end(key='b', last=False)
17 print(d) # OrderedDict([('b', 2), ('a', 1), ('d', 4), ('e', 5), ('c', 3)])

```

## Counter

```
1 from collections import Counter
2
3 c = Counter()
4 for i in 'sfsadfsdjklgsla':
5     c[i] += 1
6
7 print(isinstance(c,Counter)) # True
8 print(isinstance(c,dict)) # True
9 print(c) # Counter({'s': 4, 'd': 3, 'f': 2, 'a': 2, 'l': 2, 'j': 1, 'k': 1, 'g': 1})
10
11 c2 = Counter('asfjslfjsdlfjgkls')
12 print(c2) # Counter({'s': 4, 'd': 3, 'f': 2, 'a': 2, 'l': 2, 'j': 1, 'k': 1, 'g': 1})
13
14
15 c = Counter(['red', 'blue', 'red', 'green', 'blue', 'blue'])
16 print(c) # Counter({'blue': 3, 'red': 2, 'green': 1})
17
18 # elements() : 返回一个迭代器，其中每个元素将重复出现计数值所指定次。
19 # 元素会按首次出现的顺序返回。如果一个元素的计数值小于一，elements() 将会忽略它。
20 c = Counter(a=4, b=2, c=0, d=-2)
21 print(sorted(c.elements())) # ['a', 'a', 'a', 'a', 'b', 'b']
22
23 # most_common([n]) : 返回一个列表，其中包含 n 个最常见的元素及出现次数，按常见程度由高到低排序。
24 # 如果 n 被省略或为 None，most_common() 将返回计数器中的所有元素。计数值相等的元素按首次出现的
25 # 顺序排序：
26 c = Counter('abracadabra')
27 print(c.most_common(3)) # [('a', 5), ('b', 2), ('r', 2)]
28
29 # subtract([iterable-or-mapping]) : 从 迭代对象 或 映射对象 减去元素。
30 # 像 dict.update() 但是是减去，而不是替换。输入和输出都可以是0或者负数。
31 c = Counter(a=4, b=2, c=0, d=-2)
32 d = Counter(a=1, b=2, c=3, d=4)
33 c.subtract(d)
34 print(c) # Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

## dict.update() 方法

```
1 tinydict = {'Name': 'Zara', 'Age': 7}
2 tinydict2 = {'Sex': 'female' }
3
4 tinydict.update(tinydict2)
5 print ("Value : %s" % tinydict) # Value : {'Age': 7, 'Name': 'Zara', 'Sex': 'female'}
6
7 # 相同的键会直接替换成 update 的值:
8
9 a = {1: 2, 2: 2}
10 b = {1: 1, 3: 3}
11 b.update(a)
12 print(b) # {1: 2, 2: 2, 3: 3}
```

## bisect 模块

`bisect` 和 `bisect_right` 返回大于 `x` 的第一个下标(相当于 C++ 中的 `upper_bound`)，`bisect_left` 返回大于等于 `x` 的第一个下标(相当于 C++ 中的 `lower_bound`)

`index = bisect(ls, x) # 第1个参数是列表, 第2个参数是要查找的数, 返回值为索引`

```
1 import bisect
2 ls = [1,5,9,13,17]
3 index1 = bisect.bisect(ls,7)
4 index2 = bisect.bisect_left(ls,7)
5 index3 = bisect.bisect_right(ls,7)
6 print("index1 = {}, index2 = {}, index3 = {}".format(index1, index2, index3))
7 # index1 = 2, index2 = 2, index3 = 2
```

## array 模块

该模块定义了一个对象类型，可以紧凑地表示一个基本值的数组：字符，整数，浮点数。数组是序列类型，并且表现得非常像列表，除了存储在它们中的对象的类型是受约束的。类型在对象创建时使用 `类型代码` 指定，该类型代码是单个字符。定义了以下类型代码：

类型代码	C类型	Python类型	最小大小 (以字节为单位)	注释
'b'	signed char	INT	1	
'B'	unsigned char	INT	1	
'u'	Py_UNICODE	Unicode字符	2	(1)
'h'	signed short	INT	2	
'H'	unsigned short	INT	2	
'i'	signed int	INT	2	
'I'	unsigned int	INT	2	
'l'	signed long	INT	4	
'L'	unsigned long	INT	4	
'q'	signed long long	INT	8	(2)
'Q'	unsigned long long	INT	8	(2)
'f'	float	float	4	
'd'	double	float	8	

注释：

1. '`u`' 类型代码对应于 Python 的过时 `unicode` 字符 (`Py_UNICODE`, 即 `wchar_t`)。根据平台, 它可以是16位或32位。  
`'u'` 将与 `Py_UNICODE` API 的其余部分一起删除。

**从版本3.3开始弃用，将在4.0版中删除。**

2. The 'q' and 'Q' type codes are available only if the platform C compiler used to build Python supports C long long, or, on Windows, \_\_int64.

**版本3.3中的新功能。**

值的实际表示由机器架构（严格地说，由C实现）确定。实际大小可以通过itemsize属性访问。

```
1 import array
2
3 a = array.array("B", range(16)) # unsigned char
4 b = array.array("h", range(16)) # signed short
5
6 # 02812:恼人的青蛙，需要紧凑数组来定义布尔矩阵，否则超内存。
7 import array
8 flag = [array.array("B", [0] * (C+1)) for j in range(R+1)]
```

## calendar模块

函数及参数	作用
calendar.isleap(year)	如果 year 是闰年则返回 True,否则返回 False
calendar.leapdays(y1, y2)	返回在范围 y1 至 y2 (不包括 y2) 之间的闰年的年数，其中 y1 和 y2 是年份。此函数对于跨越世纪初的范围也适用
calendar.weekday(year, month, day)	返回某年 (1970 -- ...) , 某月 (1 -- 12) , 某日 (1 -- 31) 是星期几 (0 是星期一)
calendar.weekheader(n)	返回一个包含星期几的缩写的头。n 指定星期几缩写的字符宽度
calendar.monthrange(year, month)	返回指定 年份 的指定 月份 的第一天是星期几和这个月的天数
calendar.monthcalendar(year, month)	返回表示一个月的日历的矩阵。每一行代表一周；此月份外的日子由零表示。每周从周一 开始，除非使用 setfirstweekday() 改变设置

## 树状数组\*

```

1 # 树状数组的更新和查询
2 def update(index: int, delta: int) -> None:
3     while index <= n:
4         tree[index] += delta
5         index += index & -index
6
7
8 def query(index: int) -> int:
9     ans = 0
10    while index:
11        ans += tree[index]
12        index -= index & -index
13
14    return ans

```

## 素数筛法

### 欧拉筛法

```

1 def sieve_of_euler(x):
2     is_prime = [True] * (x + 1)
3     is_prime[1] = False
4     primes = []
5     for i in range(2, x + 1):
6         if is_prime[i]:
7             primes.append(i)
8             for j in primes:
9                 if i * j > x:
10                     break
11                 is_prime[i * j] = False
12                 if i % j == 0:
13                     break
14
15     return is_prime

```

### 埃氏筛法

```

1 def sieve_of_eratosthenes(n):
2     primes = [True] * (n + 1)
3     primes[1] = False
4     p = 2
5     while p * p <= n:
6         if primes[p] == True:
7             for i in range(p * p, n + 1, p):
8                 primes[i] = False
9             p += 1
10 prime_numbers = [p for p in range(2, n) if primes[p]]
11
12 return prime_numbers # or return primes

```

## 贪心算法

## 二分法

```
1 # 月度开销
2 n, m = map(int, input().split())
3 expense = [int(input()) for _ in range(n)]
4 l, r = max(expense), sum(expense)
5 while l < r:
6     mid = (l + r) // 2
7     cnt, tmp = 1, 0
8     for i in expense:
9         if tmp + i > mid:
10             cnt += 1
11             tmp = i
12         else:
13             tmp += i
14     if cnt > m:
15         l = mid + 1
16     else:
17         r = mid
18 print(l)
```

## Huffman 编码

```
1 # 哈夫曼编码树
2 import heapq as hq
3
4
5 class Node:
6     def __init__(self, value, weight):
7         self.value = value
8         self.weight = int(weight)
9         self.child = None
10        self.code = ''
11
12    def __lt__(self, other):
13        return (self.weight, self.value) < (other.weight, other.value)
14
15    def get_code(self):
16        codes[self.value] = self.code
17        if self.child is None: return
18        for index, child in enumerate(self.child):
19            child.code = self.code + str(index); child.get_code()
20
21
22 heap, tree, codes = [], [], dict()
23 for _ in range(int(input())): hq.heappush(heap, Node(*input().split()))
24 while len(heap) > 1:
25     tree.extend([hq.heappop(heap), hq.heappop(heap)])
26     parent = Node(tree[-2].value + tree[-1].value, tree[-2].weight + tree[-1].weight)
27     parent.child = tree[-2:]; hq.heappush(heap, parent)
28 hq.heappop(heap).get_code()
29 while True:
30     try:
31         string, res = input(), ''
32     except EOFError:
```

```

34     break
35     if string.isdigit():
36         while string:
37             for node in tree:
38                 if string.startswith(node.code):
39                     string = string[len(node.code):]
40                     res += node.value
41                     break
42             else:
43                 for char in string:
44                     res += codes[char]
45         print(res)

```

## 最长递增子序列

### 二分查找方法

```

1 # 最长上升子序列
2 from bisect import bisect_left # 如果是非递减，则改为bisect_right
3 n = int(input())
4 *seq, = map(int, input().split())
5 low = [float('-inf')] + [float('inf')] * (n + 1)
6 for x in seq:
7     low[bisect_left(low, x)] = x
8 print(low.index(float('inf')) - 1)

```

### 树状数组方法

```

1 # using segment tree to solve the LIS problem
2 def update(index: int, value: int) -> None:
3     while index <= n:
4         tree[index] = max(tree[index], value)
5         index += index & -index
6
7
8 def query(index: int) -> int:
9     ans = 0
10    while index:
11        ans = max(ans, tree[index])
12        index -= index & -index
13    return ans
14
15
16 n = int(input())
17 *seq, = map(int, input().split())
18 tree = [0] * (n + 1)
19 indexSorted = sorted(range(n), key=lambda i: (seq[i], -i)) # 如果是非递减，则去掉-i
20 for i in indexSorted:
21     update(i + 1, query(i) + 1)
22 print(query(n))

```

## 最小非减子序列数

```
1 # Wooden Sticks
2 from bisect import bisect_left
3 for _ in range(int(input())):
4     n = int(input())
5     stick = []; sticks = []
6     for i, x in enumerate(input().split()):
7         stick.append(int(x))
8         if i % 2: sticks.append(tuple(stick)); stick = []
9     sticks.sort()
10    tails, ans = [0] * n, 0
11    for stick in sticks:
12        tails[bisect_left(tails, -stick[1])] = -stick[1]
13        ans += tails[ans] != 0
14    print(ans)
```

## 动态规划

### 寻找最大矩形

```
1 # 护林员盖房子
2 m, n = map(int, input().split())
3 woods = ([[1] * (n + 2)] +
4           [[1] + list(map(int, input().split())) + [1] for _ in range(m)] +
5           [[1] * (n + 2)]) # 待求最大矩形的矩阵
6 heights = [0] * (n + 2) # 每层以上的条形柱高度
7 ans = 0
8 for i in range(1, m + 1):
9     stack = [0]
10    for j in range(1, n + 2):
11        heights[j] = (heights[j] + (woods[i][j] ^ 1)) * (woods[i][j] ^ 1)
12        while len(stack) > 1 and heights[stack[-1]] > heights[j]:
13            ans = max(ans, heights[stack.pop()] * (j - stack[-1] - 1))
14        stack.append(j)
15 print(ans)
```

## 整数划分

```
1 # 复杂的整数划分问题
2 def seg(n, k): # N划分成K个正整数之和的划分数目
3     if n < k:
4         return 0
5     if n == k:
6         return 1
7     dp = [[0 for j in range(k + 1)] for i in range(n + 1)]
8     for i in range(1, n + 1):
9         dp[i][1] = 1
10    for i in range(2, n + 1):
11        for j in range(2, min(i + 1, k + 1)):
12            if i == j:
```

```

13         dp[i][j] = 1
14     else:
15         dp[i][j] = dp[i - 1][j - 1] + dp[i - j][j]
16     return dp[n][k]
17
18
19 def seg_diff(n, k): # N划分成若干个不同正整数之和的划分数目
20     dp = [[0 for i in range(k + 1)] for j in range(n + 1)]
21     for j in range(1, k + 1):
22         dp[1][j] = 1
23     for i in range(2, n + 1):
24         for j in range(2, k + 1):
25             if i < j:
26                 dp[i][j] = dp[i][i]
27             elif i == j:
28                 dp[i][j] = 1 + dp[i][j - 1]
29             else:
30                 dp[i][j] = dp[i - j][j - 1] + dp[i][j - 1]
31     return dp[n][k]
32
33
34 def seg_odd(n, k): # N划分成若干个奇正整数之和的划分数目
35     dp = [[0 for i in range(k + 1)] for j in range(n + 1)]
36     for i in range(1, n + 1):
37         dp[i][1] = 1
38     for j in range(1, k + 1):
39         dp[1][j] = 1
40     for i in range(2, n + 1):
41         for j in range(2, k + 1):
42             if j % 2 == 0:
43                 dp[i][j] = dp[i][j - 1]
44             elif i < j:
45                 dp[i][j] = dp[i][i]
46             elif i == j:
47                 dp[i][j] = 1 + dp[i][j - 2]
48             else:
49                 dp[i][j] = dp[i - j][j] + dp[i][j - 2]
50     return dp[n][k]

```

## 背包问题

### 01背包

```

1 # 采药
2 t, m = map(int, input().split())
3 cost = []
4 value = []
5 for _ in range(m):
6     c, v = map(int, input().split())
7     cost.append(c)
8     value.append(v)
9 dp = [0] * (t + 1)
10 for i in range(m):
11     for j in range(t, cost[i] - 1, -1):
12         dp[j] = max(dp[j], dp[j - cost[i]] + value[i])
13 print(dp[t])

```

## 完全背包

```
1 # Cut Ribbon
2 from math import log2
3 n, *pieces = map(int, input().split())
4 pieces.sort()
5 for i in range(len(pieces) - 1, -1, -1):
6     for j in range(i):
7         if pieces[i] % pieces[j] == 0:
8             del pieces[i]
9             break
10 *pieces, = zip(pieces, [1] * len(pieces))
11 for i in range(len(pieces)):
12     for j in range(1, int(log2(n / pieces[i][0])) + 1):
13         pieces.append((pieces[i][0] * 2 ** j, 2 ** j))
14 dp = [0] + [float('-inf')] * n
15 for p in pieces:
16     for i in range(n, p[0] - 1, -1):
17         dp[i] = max(dp[i], dp[i - p[0]] + p[1])
18 print(dp[n])
```

## 多重背包

```
1 # NBA门票
2 prices = [1, 2, 5, 10, 20, 50, 100]
3 money = int(input())
4 *tickets, = map(int, input().split())
5 if money % 50 != 0: print('Fail'); exit()
6 money //= 50
7 dp = [0] + [float('inf')] * money
8 for i in range(7): # 每种票都在O(log(tickets[i]))的时间内处理完
9     if prices[i] * tickets[i] >= money: # 票的数量多到超过背包容量, 转换成完全背包
10        for j in range(prices[i], money + 1):
11            dp[j] = min(dp[j], dp[j - prices[i]] + 1)
12    k = 1
13    while k < tickets[i]: # 二进制优化
14        for j in range(money, prices[i] * k - 1, -1): # 01背包
15            dp[j] = min(dp[j], dp[j - prices[i] * k] + k)
16        tickets[i] -= k
17        k <= 1
18    for j in range(money, prices[i] * tickets[i] - 1, -1):
19        dp[j] = min(dp[j], dp[j - prices[i] * tickets[i]] + tickets[i])
20 print(dp[money] if dp[money] != float('inf') else 'Fail')
21
```

## 多重背包可行性

下面介绍一种实现较为简单的  $O(VN)$  复杂度解多重背包问题的算法。它的基本思想是这样的：设  $F[i, j]$  表示“用了前  $i$  种物品填满容量为  $j$  的背包后，最多还剩下几个第  $i$  种物品可用”，如果  $F[i, j] = -1$  则说明这种状态不可行，若可行应满足  $0 \leq F[i, j] \leq M_i$ 。递推求  $F[i; j]$  的伪代码如下：

```

1 F[0, 1...V] <- -1
2 F[0, 0] <- 0
3 for i <- 1 to N
4   for j <- 0 to V
5     if F[i - 1][j] ≥ 0
6       F[i][j] = M_i
7     else
8       F[i][j] = -1
9   for j <- 0 to V - C_i
10    if F[i][j] > 0
11      F[i][j + C_i] <- max{F[i][j + C_i], F[i][j] - 1}

```

最终  $F[N][0 \dots V]$  便是多重背包可行性问题的答案。

## 最优方案的总数

这里的最优方案是指物品总价值最大的方案。以 01 背包为例。结合求最大总价值和方案总数两个问题的思路，最优方案的总数可以这样求： $F[i, v]$  代表该状态的最大价值， $G[i, v]$  表示这个子问题的最优方案的总数，则在求  $F[i, v]$  的同时求  $G[i, v]$  的伪代码如下：

```

1 G[0, 0] <- 1
2 for i <- 1 to N
3   for v <- 0 to V
4     F[i, v] <- max{F[i - 1, v], F[i - 1, v - C_i] + W_i}
5     G[i, v] <- 0
6     if F[i, v] = F[i - 1, v]
7       G[i, v] <- G[i, v] + G[i - 1][v]
8     if F[i, v] = F[i - 1, v - C_i] + W_i
9       G[i, v] <- G[i, v] + G[i - 1][v - C_i]

```

## 最长公共子序列

```

1 # 公共子序列
2 while True:
3   try: s1, s2 = input().split()
4   except EOFError: break
5   dp = [[0 for _ in range(len(s2) + 1)] for _ in range(len(s1) + 1)]
6   for i in range(1, len(s1) + 1):
7     for j in range(1, len(s2) + 1):
8       dp[i][j] = max(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1] + (s1[i - 1] == s2[j - 1]))
9   print(dp[-1][-1])

```

## 图搜索

## 拼接问题

```
1 # Sticks
2 from itertools import count
3 while n := int(input()):
4     def valid(length, dep=0):
5         if dep == n and length == 0:
6             return True
7         if length == 0:
8             length = ans
9         for i in range(n):
10            if used[i] or sticks[i] > length or (i and sticks[i] == sticks[i - 1] and not
11                used[i - 1]):
12                continue
13            used[i] = True
14            if valid(length - sticks[i], dep + 1):
15                return True
16            used[i] = False
17            if sticks[i] == length or length == ans:
18                break
19        return False
20
21 *sticks, = map(int, input().split())
22 sticks.sort(reverse=True)
23 used = [False] * n
24 lenSum = sum(sticks)
25 for ans in count(sticks[0]):
26     if lenSum % ans == 0:
27         if valid(ans):
28             print(ans)
             break
```

## Dijkstra 算法

```
1 # 走山路
2 from heapq import heappush, heappop
3 di = [(0, 1), (1, 0), (0, -1), (-1, 0)]
4
5
6 class Node:
7     def __init__(self, value, cost):
8         self.value = value
9         self.cost = cost
10        self.visited = False
11    def __lt__(self, other):
12        return self.cost < other.cost
13
14
15 def dijkstra(xs, ys, xe, ye):
16     if mount[xs][ys] == '#' or mount[xe][ye] == '#':
17         return float('inf')
18     nodes = [[Node((i, j), float('inf')) for j in range(m + 2)]
19              for i in range(n + 2)]
20     nodes[xs][ys].cost = 0
21     queue = []
22     heappush(queue, Node((xs, ys), 0))
```

```

23     while queue:
24         node = heappop(queue)
25         x, y = node.value
26         if x == xe and y == ye:
27             return node.cost
28         if nodes[x][y].visited:
29             continue
30         nodes[x][y].visited = True # 注意迪杰斯特拉算法的核心是每个节点只访问一次
31         for dx, dy in di:
32             nx, ny = x + dx, y + dy
33             if mount[nx][ny].isdigit():
34                 delta = abs(int(mount[nx][ny]) - int(mount[x][y]))
35                 if node.cost + delta < nodes[nx][ny].cost:
36                     nodes[nx][ny].cost = node.cost + delta
37                     heappush(queue, Node((nx, ny), nodes[nx][ny].cost))
38             # 注意这里不能直接推入nodes[x][y], 否则queue中的对象会被修改导致乱序
39         return float('inf')
40
41
42 m, n, p = map(int, input().split())
43 mount = [[ '#' ] * (n + 2)] + \
44     [[ '#' ] + input().split() + ['#'] for _ in range(m)] + \
45     [[ '#' ] * (n + 2)]
46 for _ in range(p):
47     ans := dijkstra(*map(lambda x: int(x)+1, input().split()))
48     print(ans if ans != float('inf') else 'NO')

```

## SPFA 算法

```

1 # 最短路
2 def spfa(s):
3     dis = [float('inf') for _ in range(n)]
4     dis[s] = 0
5     queue = [s]
6     cnt = [0] * n
7     while queue:
8         u = queue.pop(0)
9         for v, w in G[u]:
10            if dis[v] > dis[u] + w:
11                dis[v] = dis[u] + w
12                if v not in queue:
13                    queue.append(v)
14                    cnt[v] += 1
15                if cnt[u] > n:
16                    return ['Error']
17    return dis
18
19
20 for _ in range(int(input())):
21     n, m, s = map(int, input().split())
22     G = [[] for _ in range(n)]
23     for _ in range(m):
24         x, y, z = map(int, input().split())
25         G[x - 1].append((y - 1, z))
26     print(*[i if i != float('inf') else 'null' for i in spfa(s - 1)])

```

# 字符串

## is判断方法

```
1  isdigit()
2  True: Unicode数字, byte数字(单字节), 全角数字(双字节)
3  False: 汉字数字, 罗马数字, 小数
4  Error: 无
5
6  isdecimal()
7  True: Unicode数字, 全角数字(双字节)
8  False: 罗马数字, 汉字数字, 小数
9  Error: byte数字(单字节)
10
11 isnumeric()
12 True: Unicode数字, 全角数字(双字节), 罗马数字, 汉字数字
13 False: 小数
14 Error: byte数字(单字节)
15
16 isalpha(): 判断给定的字符串是否全为字母
17 isalnum(): 判断给定的字符串是否只含有数字与字母
18 isupper(): 判断给定的字符串是否全为大写
19 islower(): 判断给定的字符串是否全为小写
20 istitle(): 判断给定的字符串是否符合title()
21 isspace(): 判断给定的字符串是否为空白符(空格、换行、制表符)
22 isprintable(): 判断给定的字符串是否为可打印字符(只有空格可以, 换行、制表符都不可以)
23 isidentifier(): 判断给定的字符串是否符合命名规则(只能是字母或下划线开头、不能包含除数字、字母和下划线以外的任意字符。)
```

## ASCII码

十进制	字符/缩写	解释	十进制	字符/缩写	解释
0	NUL (NULL)	空字符	64	@	
1	SOH (Start Of Headline)	标题开始	65	A	
2	STX (Start Of Text)	正文开始	66	B	
3	ETX (End Of Text)	正文结束	67	C	
4	EOT (End Of Transmission)	传输结束	68	D	
5	ENQ (Enquiry)	请求	69	E	
6	ACK (Acknowledge)	回应/响应/收到通知	70	F	
7	BEL (Bell)	响铃	71	G	
8	BS (Backspace)	退格	72	H	
9	HT (Horizontal Tab)	水平制表符	73	I	

十进制	字符/缩写	解释	十进制	字符/缩写	解释
10	LF/NL(Line Feed/New Line)	换行键	74	J	
11	VT (Vertical Tab)	垂直制表符	75	K	
12	FF/NP (Form Feed/New Page)	换页键	76	L	
13	CR (Carriage Return)	回车键	77	M	
14	SO (Shift Out)	不用切换	78	N	
15	SI (Shift In)	启用切换	79	O	
16	DLE (Data Link Escape)	数据链路转义	80	P	
17	DC1/XON (Device Control 1/Transmission On)	设备控制1/传输开始	81	Q	
18	DC2 (Device Control 2)	设备控制2	82	R	
19	DC3/XOFF (Device Control 3/Transmission Off)	设备控制3/传输中断	83	S	
20	DC4 (Device Control 4)	设备控制4	84	T	
21	NAK (Negative Acknowledge)	无响应/非正常响应/拒绝接收	85	U	
22	SYN (Synchronous Idle)	同步空闲	86	V	
23	ETB (End of Transmission Block)	传输块结束/块传输终止	87	W	
24	CAN (Cancel)	取消	88	X	
25	EM (End of Medium)	已到介质末端/介质存储已满/介质中断	89	Y	
26	SUB (Substitute)	替补/替换	90	Z	
27	ESC (Escape)	逃离/取消	91	[	
28	FS (File Separator)	文件分割符	92	\	
29	GS (Group Separator)	组分隔符/分组符	93	]	
30	RS (Record Separator)	记录分离符	94	^	
31	US (Unit Separator)	单元分隔符	95	_	
32	(Space)	空格	96	`	
33	!		97	a	
34	"		98	b	
35	#		99	c	
36	\$		100	d	

十进制	字符/缩写	解释	十进制	字符/缩写	解释
37	%		101	e	
38	&		102	f	
39	'		103	g	
40	(		104	h	
41	)		105	i	
42	*		106	j	
43	+		107	k	
44	,		108	l	
45	-		109	m	
46	.		110	n	
47	/		111	o	
48	0		112	p	
49	1		113	q	
50	2		114	r	
51	3		115	s	
52	4		116	t	
53	5		117	u	
54	6		118	v	
55	7		119	w	
56	8		120	x	
57	9		121	y	
58	:		122	z	
59	;		123	{	
60	<		124		
61	=		125	}	
62	>		126	~	
63	?		127	DEL (Delete)	删除

## *textwrap* 模块

`wrap()` 对 `text` (字符串) 中的单独段落自动换行以使每行长度最多为 `width` 个字符。 返回由输出行组成的列表，行尾不带换行符。

```
1  "\n".join(wrap(text, ...))
2  text = 'abcdefghijklmnopqrstuvwxyz'
3  textwrap.wrap(text, width=4)
4  # 返回值 ['abcd', 'efgh', 'ijkl', 'mnop', 'qrst', 'uvwxyz', 'yz']
```

`fill()` 对 `text` 中的单独段落自动换行，并返回一个包含被自动换行段落的单独字符串。

```
1 | textwrap.fill(text, width=13) # 返回值 'abcdefghijklmnopqrstuvwxyz\nabcdefghijklmnopqrstuvwxyz'
```

`fill()` 是以下语句的快捷方式

```
1 | "\n".join(wrap(text, ...))
```

## re模块

```
1 import re
2 while True:
3     try:
4         s = input()
5         reg = r'[^@\.]+(\.[^@\.]+)*@[^\@\.]+(\.[^@\.]+)+$'
6         print('YES' if re.match(reg, s) else 'NO')
7     except EOFError:
8         break
```

```
1 数字: ^[0-9]*$  
2 n位的数字: ^\d{n}$  
3 至少n位的数字: ^\d{n,}$  
4 m-n位的数字: ^\d{m,n}$  
5 零和非零开头的数字: ^(0|[1-9][0-9]*)$  
6 非零开头的最多带两位小数的数字: ^([1-9][0-9]*)(.[0-9]{1,2})?$_  
7 带1-2位小数的正数或负数: ^(\-)?\d+(\.\d{1,2})?$_  
8 正数、负数、和小数: ^(\-|\+)?\d+(\.\d+)?$_  
9 有两位小数的正实数: ^[0-9]+(.[0-9]{2})?$_  
10 有1~3位小数的正实数: ^[0-9]+(.[0-9]{1,3})?$_  
11 非零的正整数: ^[1-9]\d*$ 或 ^([1-9][0-9]*){1,3}$ 或 ^\+?[1-9][0-9]*$_  
12 非零的负整数: ^\-[1-9][]{0-9}"*$ 或 ^\-[1-9]\d*$  
13 非负整数: ^\d+$ 或 ^[1-9]\d*|0$  
14 非正整数: ^\-[1-9]\d*|0$ 或 ^((\-\d+)|(0+))$_  
15 非负浮点数: ^\d+(\.\d+)?$_ 或 ^[1-9]\d*\.\d*|0\.\d*[1-9]\d*|0?\.\0+|0$  
16 非正浮点数: ^((\-\d+(\.\d+)?)|(0+(\.\0+)?))$_ 或 ^((\-\([1-9]\d*\.\d*|0\.\d*[1-9]\d*))|0?\.\0+|0$  
17 正浮点数: ^[1-9]\d*\.\d*|0\.\d*[1-9]\d*$ 或 ^(([0-9]+).[0-9]*[1-9][0-9]*)([0-9]*[1-9][0-9]*).[0-9+]|([0-9]*[1-9][0-9]*).[0-9+]$_  
18 负浮点数: ^\-(\([1-9]\d*\.\d*|0\.\d*[1-9]\d*))$_ 或 ^\-(([0-9]+).[0-9]*[1-9][0-9]*)([0-9]*[1-9][0-9]*).[0-9+]|([0-9]*[1-9][0-9]*).0+$  
19 浮点数: ^\-(\d+)(\.\d+)?$_ 或 ^\-?([1-9]\d*\.\d*|0\.\d*[1-9]\d*|0?\.\0+|0)$
```

## re.match函数

语法: re.match(pattern, string, flags=0)

pattern	匹配的正则表达式
string	要匹配的字符串
flags	<p>标志位, 用于控制正则表达式的匹配方式, 如: 是否区分大小写, 多行匹配等等。</p> <ol style="list-style-type: none"><li>1. <b>re.I</b> 忽略大小写</li><li>2. <b>re.L</b> 表示特殊字符集 \w, \W, \b, \B, \s, \S 依赖于当前环境</li><li>3. <b>re.M</b> 多行模式</li><li>4. <b>re.S</b> 即为 . 并且包括换行符在内的任意字符 ( . 不包括换行符)</li><li>5. <b>re.U</b> 表示特殊字符集 \w, \W, \b, \B, \d, \D, \s, \S 依赖于 Unicode 字符属性数据库</li><li>6. <b>re.X</b> 为了增加可读性, 忽略空格和 # 后面的注释</li></ol>

## KMP算法\*

```
1 def get_next(pattern):
2     m = len(pattern)
3     next = [0] * m
4     j = 0
5     for i in range(1, m):
6         while j > 0 and pattern[i] != pattern[j]:
7             j = next[j-1]
8         if pattern[i] == pattern[j]:
9             j += 1
10        next[i] = j
11    return next
12
13
14 def kmp_search(text, pattern):
15     n = len(text)
16     m = len(pattern)
17     if m == 0:
18         return 0
19     next = get_next(pattern)
20     j = 0
21     for i in range(n):
22         while j > 0 and text[i] != pattern[j]:
23             j = next[j-1]
24         if text[i] == pattern[j]:
25             j += 1
26         if j == m:
27             return i - m + 1
28     return -1
29
30
31 text = "ABABABABCABABABCABABABC"
32 pattern = "ABABCABAB"
33 index = kmp_search(text, pattern)
34 print("pos matched: ", index)
35 # pos matched: 4
```

i值	a	b	x	a	b	w	a	b	x	a	d	next[i]值
0	k指针	a										0
1	k指针	i指针	a	b								0
2	k指针			i指针	a							0
3	k指针			i指针	a							1
	a	b	x	a								
		k指针		i指针								
	a	b	x	a								
4		k指针		i指针								2
	a	b	x	a	b							
		k指针		i指针								
	a	b	x	a	b							
5	k指针			i指针								0
	a	b	x	a	b	w						
							i指针					
6	k指针				i指针	a						1
	a	b	x	a	b	w	a					
		k指针			i指针							
	a	b	x	a	b	w	a	b				
7		k指针			i指针							2
	a	b	x	a	b	w	a	b				
		k指针			i指针							
	a	b	x	a	b	w	a	b				
8		k指针			i指针							3
	a	b	x	a	b	w	a	b	x			
		k指针			i指针							
	a	b	x	a	b	w	a	b	x			
9		k指针			i指针							4
	a	b	x	a	b	w	a	b	x	a		
		k指针			i指针							
	a	b	x	a	b	w	a	b	x	a		
10		k指针			i指针							0
	a	b	x	a	b	w	a	b	x	a	d	
		k指针			i指针							
	a	b	x	a	b	w	a	b	x	a	d	

## 威佐夫博弈

```

1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4 int main()
5 {
6     int n, m;
7     while(cin >> n >> m)
8     {
9         if (n < m)
10             swap(n, m);
11         //bi=ai+i
12         int i = n - m;//奇异局势的差
13         cout<<1-(m==int(i*(sqrt(5.0)+1)/2))<<endl;
14     }
15     return 0;

```

## 部分未提及的标准库

### 布尔运算

布尔运算，按优先级升序排列：

运算	结果：	备注
<code>x or y</code>	如果 $x$ 为真值，则 $x$ ，否则 $y$	(1)
<code>x and y</code>	$\text{if } x \text{ is false, then } x, \text{ else } y$	(2)
<code>not x</code>	$\text{if } x \text{ is false, then True, else False}$	(3)

注释：

1. 这是个短路运算符，因此只有在第一个参数为假值时才会对第二个参数求值。
2. 这是个短路运算符，因此只有在第一个参数为真值时才会对第二个参数求值。
3. `not` 的优先级比非布尔运算符低，因此 `not a == b` 会被解读为 `not (a == b)` 而 `a == not b` 会引发语法错误。

### sys 模块

```
1 | sys.stdout.write('hello' + '\n') # 注意'\n'
2 | print('hello') # 两种写法等价
```

**注意：**`sys.stdin.readline()` 会将标准输入全部获取，包括末尾的'\n'，因此用len计算长度时是把换行符'\n'算进去了的，但是 `input()` 获取输入时返回的结果是不包含末尾的换行符'\n'的

```
1 | import sys
2 | hi1 = input()
3 | hi2 = sys.stdin.readline()
4 | print(len(hi1))
5 | print(len(hi2)) # 输出比 print(len(hi1)) 多1
```

### itertools 模块

无限迭代器：

迭代器	参数	结果	例
<code>count()</code>	<code>start, [step]</code>	<code>start, start+step, start+2*step, ...</code>	<code>count(10) --&gt; 10 11 12 13 14 ...</code>
<code>cycle()</code>	<code>p</code>	<code>p0, p1, ... plast, p0, p1, ...</code>	<code>cycle('ABCD') --&gt; A B C D A B C D ...</code>

迭代器	参数	结果	例
repeat()	elem [,n]	elem, elem, elem, ... endlessly or up to n times	repeat(10, 3) --> 10 10 10

### 在最短输入序列上终止的迭代器：

迭代器	参数	结果	例
accumulate()	p [,func]	p0, p0 + p1, p0 + p1 + p2, ...	accumulate([1,2,3,4,5]) --> 1 3 6 10 15
chain()	p, q, ...	p0, p1, ... plast, q0, q1, ...	chain('ABC', 'DEF') --> A B C D E F
chain.from_iterable()	iterable	p0, p1, ... plast, q0, q1, ...	chain.from_iterable(['ABC', 'DEF']) --> A B C D E F
compress()	data, selectors	(d[0] if s[0]), (d[1] if s[1]), ...	compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
dropwhile()	pred, seq	seq[n], seq[n+1], starting when pred fails	dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1
filterfalse()	pred, seq	elements of seq where pred(elem) is false	filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8
groupby()	iterable[, keyfunc]	sub-iterators grouped by value of keyfunc(v)	
islice()	seq, [start,] stop [, step]	elements from seq[start:stop:step]	islice('ABCDEFG', 2, None) --> C D E F G
starmap()	func, seq	func(seq[0]), func(seq[1]), ...	starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
takewhile()	pred, seq	seq[0], seq[1], until pred fails	takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
tee()	it, n	it1, it2, ... itn splits one iterator into n	
zip_longest()	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-

### 组合生成器：

迭代器	参数	结果
product()	p, q, ... [repeat=1]	笛卡尔积, 相当于嵌套for循环
permutations()	p[, r]	r长度元组, 所有可能的顺序, 没有重复的元素
combinations()	p, r	r长度元组, 按排序顺序, 没有重复的元素
combinations_with_replacement()	p, r	r长度元组, 按排序顺序, 重复元素
product('ABCD', repeat=2)		AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD
permutations('ABCD', 2)		AB AC AD BA BC BD CA CB CD DA DB DC
combinations('ABCD', 2)		AB AC AD BC BD CD
combinations_with_replacement('ABCD', 2)		AA AB AC AD BB BC BD CC CD DD

## decimal模块

```

1  from decimal import Decimal # 引用decimal模块中的Decimal方法
2  a = "1.345"
3  #保留几位小数由像第二个括号中的几位小数决定, 即保留两位小数, 精确到0.01
4  #如果想要四舍五入保留整数, 那么第二个括号中就应该为"1."
5  a_t = Decimal(a).quantize(Decimal("0.01"), rounding = "ROUND_HALF_UP")
6  print(a_t) # 1.35

```

decimal模块中默认 rounding = ROUND\_HALF\_EVEN (可以通过 getcontext().rounding 命令查看), 即银行家式舍入法——四舍六入五成双式舍入法。

要想使用 decimal 模块中的 Decimal() + quantize() 方法实现精确的四舍五入，需要指定 rounding=ROUND\_HALF\_UP——四舍五入法。