CS101 Cheatsheet

目录

1 一句话小技巧

随机数:

```
import random
x=random.randint(a,b)#>=a,<=b的随机整数
x=random.random()#0~1随机浮点数
x=random.uniform(a,b)#a,b之间随机浮点数
x=random.choice(list)#在列表list中随机选择
```

深拷贝:

```
from copy import deepcopy
lcopy=deepcopy(l)
```

保留小数

```
  "{:.2f}".format(num)
  #或
  "%.2f" % num
```

建一个以每行输入的字符串为元素的列表:`sys.stdin.readlines()`

遍历每一项的索引和值组成的元组(类似于`dict.items()`):`for index,value in enumerate(list)`

无穷大:`float('inf')`

lru_cache(用来存储每次递归的结果,遇到相同自变量的递归时直接返回这个结果,因此它不能在此时再进行更新其中的全局变量的操作):

```
from functools import lru_cache
@lru_cache(maxsize = 128)
```

循环正常结束后执行的语句:`while...else...`

临时"弄一个"函数:lambda

```
list.sort(key = lambda list:list[0])
```

海象运算符(OJ上的py3.8用不了!!!)用来给变量赋值

```
if n:=int(input())!=0:...
```

递归可视化

```
from recviz import recviz
@recviz
def dfs(...):
```

链接数组元素与索引

```
enumerated = list(enumerate(['a', 'b', 'c']))
-> [(0, 'a'), (1, 'b'), (2, 'c')]
```

数组元素条件分离

```
filtered = list(filter(lambda x:x > 2, [1, 2, 3, 4]))
-> [3, 4]
```

递归深度限制设置:`sys.setrecursionlimit(...)`

位运算: a<<b就是在bin(a)(a的二进制表示)后加b个0

使用sys.stdin在本地调试可以按Ctrl+D作为EOF

2 数据结构与相应操作的时间复杂度

2-1 defaultdict

- 这是一种"如果访问的key不存在与字典中,就给它新建一个默认值`dict[key]=default_value`的字典

  ```
  from collections import defaultdict
  a=defaultdict(list)#以空列表为默认值的defaultdict
  #类似地,括号里是int,set也可以
  ```

- 如果要自定义默认值,用`defaultdict(lambda: xxxx)`

2-2 OrderedDict

- 这是一种"输入顺序即遍历顺序"的字典.这种字典可以像列表一样进行两端弹出

```
from collections import OrderedDict
od=OrderedDict()
od['a']=1
od['b']=2
od['c']=3
od['d']=4
```

- 删除一个元素:`del od[key]`
- 弹出一个元素:`od.pop(key(,default))#后面可加的default是当key不在字典中时返回的默认值`
  - 注意,pop的时间复杂度是O(n),而普通字典是O(1)
- 弹出末尾:`od.popitem()`
- 弹出首:`od.popitem(last=False)`

2-3 双端队列(deque)

- 可以从两端弹出元素,但弹出中间元素会很慢

```
from collections import deque
deque.popleft(item)#左弹出
deque.pop(item)#右弹出
deque.appendleft(item)#左加入
deque.append(item)#右加入
```

2-4 小顶堆(heapq)

- 这是用一个列表实现小顶堆的数据结构.小顶堆是一种儿茶素(x)(二叉树),**根节点最小,每一个节点都比它的孩子小.**
- 你可以先建立一个列表,然后对它做堆操作,这样它就被你当成了一个堆
- (python没有大顶堆数据结构,想要做大顶堆可以把heapq里面所有元素加负号)

```
import heapq
heap=[6,5,4,3,2,1]#注意现在它还只是一个普通的列表
```

- 如果它的元素并不是以堆的形式排列的,就给它"堆化":`heapq.heapify(heap)`
- 堆化后的a应为`[1,2,4,3,5,6]`(以堆的形式排列的方式显然不止一种)
- 然后可以对堆进行堆操作:

```
import heapq
heapq.heappush(heap, item)#推入元素
heapq.heappop(heap)#弹出堆顶
heapq.heappushpop(heap,item)#先推再弹,更高效
heapq.heapreplace(heap,item)#先弹再推(SN1)(bushi)
```

- 弹出和推入都是O(log n)
- 推入的原理是把新元素先加到堆底然后上滤(一层层与父节点比较,如果更大就与父节点交换).
  所以如果对一个没有堆化的列表进行堆推入,得到的不是堆(而是把这个列表当做完全二叉树做上述操作得到的结果).

2-5 列表

- 构建二维列表:`[[],[],...]`(就是列表套列表)
- 清空:`l.clear()`,O(1)
- 切片:`l[a:b]`,**O(b-a)**,不是O(1)!
- 反转:`l.reverse()`,O(n)
- **浅拷贝问题**:当列表等被拷贝时,拷贝的是列表的地址而不是列表本身

  ```
  a=[1,2,3]
  b=a
  ```

  此时a,b共用一个地址,其中任何一个被改变时,另一个也会相应改变.
  这称为"浅拷贝".(如果你知道FL Studio...这就相当于FL Studio里面复制一个pattern)

  与之相反的叫深拷贝,拷贝的是列表本身(相当于复制了pattern然后make unique)

  ```
  from copy import deepcopy
  a=[1,2,3]
  b=a[:]
  c=[[1,2],[2,3],[3,4]]
  d=c[:]
  e=deepcopy(c)
  ```

  对于一维列表,用切片复制就可以实现深拷贝,`l.copy()`与之是等价的.
  对于>=2维的列表,如果用切片或者`l.copy()`,那里面那些小列表还是浅拷贝的.如果要完全深拷贝自身,需要用deepcopy函数
- 列表化:`list(x)`,时间复杂度取决于遍历x的长度.其中x可以是以下:
  - 字符串:`list('abc')==['a','b','c']`
  - 迭代器/生成器(range啊map啊这类的):
    - `list(range(4))==[0,1,2,3]`
    - `list(map(int,input().split())==[2,5,1]` (假设输入'2 5 1')
    - `list(enumerate([2,5,1]))==[(0,2),(1,5),(2,1)]`

2-6 元组

- 跟列表差不多,除了不能增删改.

2-7 集合

- 顾名思义,集合中不能有重复的元素,否则会自动去重.
- **集合内的元素只能是"零维的"标量.**
- 添加:`s.add(x)`
- 查询:`x (not) in s`,**O(1)**
- 删除:
  - `s.remove(x)`,**O(1)**
  - `s.discard(x)`,**O(1)**:删除x,如果没找到x就无事发生

- 弹出:`s.pop()`,O(1):随弹出一项
- 清空:`s.clear()`,O(1)
- 集合运算:
  - 并:`a|b`,O(len(a)+len(b))
  - 交:`a&b`,O(len(a)+len(b))
  - 差:
    - `a-b`,O(len(a)+len(b)),注意这是表示∈a且∉b的元素的集合,
      如`{1}-{1,2}==set(),{1,2}-{1}=={2}`
    - `a^b`,对称差,等于a|b-a&b
  - `a>=b,a>b,a<=b,a<b`,O(len(小的那个))表示集合的包含关系,如果既不是子集也不是超集也返回False
- 构建集合:
  - 空集:`a=set()`
  - 集合化:set(x),x是列表\元组\字典(a是字典则会返回所有键组成的集合)

2-8 字典

- **字典的键只能是"零维"标量(int,float这种),而非"一维"以上的复合数据类型.**
- 删除:`del d[key]`
- 弹出:
  - `d.pop(key)`,O(1)
  - `d.popitem()`,O(1),随机弹出一个键值对元组
- 获取:`d.get(key)`,O(1),如果key不在字典则返回None而不报错
  - `d.get(key,dft)`可以在key不在字典时返回默认值dft
- 清空:`d.clear()`,O(1)
- 键or值们:`d.keys()`,`d.values()`,O(1).

2-9 迭代器和生成器

- range就是一种迭代器,它在迭代时,每次生成一个量.
- **可以被列表/元组/字典/集合化.**
- python中常用的迭代器/生成器有以下几种:
  - `range(n)`:生成0,1,...,(n-1)的整数序列
  - `enumerate(list)`:依次生成列表/元组中每一项索引与元素的对应:
    - `list(enumerate(['a','b','c']))=[(0,'a'),(1,'b'),(2,'c')]`
    - `dict(enumerate(['a','b','c']))={0:'a',1:'b',2:'c'}`
  - `zip(a,b)`:依次生成列表/元组a,b中每一项的对应.如果有一方有多余项,则不管
    - `list(zip([1,2,3],[11,22]))=[(1,11),(2,22)]`
    - `dict`类似

3 函数与算法

3-0 背包dp

- 01背包

```
for i in range(1, n + 1):
    for l in range(W, w[i] - 1, -1):
        f[l] = max(f[l], f[l - w[i]] + v[i])
```

- 完全背包

```
for i in range(1, n + 1):
    for l in range(0, W - w[i] + 1):
        f[l + w[i]] = max(f[l] + v[i], f[l + w[i]])
```

- 多重背包二进制分组优化

```
index = 0
for i in range(1, m + 1):
    c = 1
    p, h, k = map(int, input().split())
    while k > c:
        k -= c
        index += 1
        list[index].w = c * p
        list[index].v = c * h
        c *= 2
    index += 1
    list[index].w = p * k
    list[index].v = h * k
```

3-1 欧拉筛(ES)

- 筛出1<=i<=n所有的素数
- 输入范围n,输出1~n的素数列表
- 如果你想查询一个很大的数是不是素数,把里面prime一开始做成集合再返回用来查询

```
def ES(n):
  isprime=[True for _ in range(n+1)]
  prime=[]
  for i in range(2,n+1):
      if isprime[i]:
          prime.append(i)
      for j in range(len(prime)):
          if  i*prime[j]>n:break
          isprime[i*prime[j]]=False
          if i%prime[j]==0 :break

  return prime
```

3-2 大非负整数乘法

```
def times(a,b):
  a,b=list(str(a)),list(str(b))
  a.reverse()
  b.reverse()
  ans=[0 for i in range(10002)]
  for j in range(len(b)):
      for i in range(len(a)):
          ans[j+i]+=int(b[j])*int(a[i])%10
          ans[j+i+1]+=ans[j+i]//10
          ans[j+i]%=10
          ans[j+i+1]+=int(b[j])*int(a[i])//10
  i=0
  while i==0 and ans:
      i=ans.pop()
  ans.append(i)
  res=''
  for i in ans[-1::-1]:
      res+=str(i)
  return int(res)
```

3-3 dfs

- 直接在矩阵中进行dfs吧!
- 以下是对一个矩阵湖的dfs,其中1表示水,0表示陆地.
- 我们要找到一块连通的水域,但不能对这个湖产生影响(即:修改矩阵中的元素).
    1. 先用主函数找到一个是水的点,然后进入dfs
    2. 从这个点开始向四周探路,`for dx,dy...`
    3. 如果遇到一个符合条件的点(在矩阵范围内,是水),就继续从这个点dfs(进入下一层递归).
       但在此之前,我们为了防止回到原先这个点,在进下一层之前,得临时把原来的点填成陆地.
    4. 就这样一直往下走,如果走到死胡同(周围全是地)了,`for dx,dy in d` 这个循环就会终止,这一层的递归就结束了,就回到了上一层(即:退了一步)
    5. 退了一步,就会把上一步的那个临时填平的点恢复成水域,然后上一层的for循环继续(也就是朝其他方向走)了.
    6. 这样,我们就遍历了一片连通水域中的所有点,而且当我们遍历结束后,这个矩阵湖没有受到任何影响.
    7. 当然,如果还要找其他所有水域,我们必须在主函数里先一个个点遍历,遍历到水再进dfs.

```
def dfs(x,y):
    d=[(-1,-1),(-1,0),(-1,1),(0,-1),(0,1),(1,-1),(1,0),(1,1)]
    for dx,dy in d:
        nx,ny=x+dx,y+dy
        if 0<=nx<len(mat) and 0<=ny<len(mat[0]):
            if mat[nx][ny]==1:
                mat[x][y]=0
                dfs(nx,ny)
                mat[x][y]=1
```

- 当然,这只是最基础的dfs,而我们在dfs时通常要进行一些别的操作.例如:
  - 一边遍历一边把湖填平:把原来"临时填平"的操作改成在进入dfs时永久填平,即在`for dx,dy...`之前加上`mat[x][y]=0`
  - 一边遍历一遍记录递归深度(走了多远):在dfs的参数里多放一个`S=0`,即`dfs(x,y,S=0)`,然后进入下一个层就把S加一,即把`dfs(nx,ny)`改成`dfs(nx,ny,S+1)`
    - 路径(就是这条路怎么走的),路径权值和(比如每个点放了一定数量的金币,走这条路一共捡到几个金币)等**与某一条路径相关的量**都可以用**参数**这种方式记录.
  - 一边遍历一边记录水域面积:放一个全局变量S,每次入dfs之后`S+=1`
    - **与整个dfs而不是某一条特定路径相关的量**(比如最短路径长度,水域面积,最大权值和路径)用**全局变量**记录.
  - 如果这不是湖,这是迷宫,有终点,走到终点就跳出:在探路之前加上终点判断`if (x,y)==e:return`.

3-4 bfs

- 直接在矩阵里bfs了
- 同样,我们需要用bfs来遍历一个矩阵湖中的某片连通水域,1表示水,0表示地.步骤如下:
  1. 先在主函数里遍历矩阵湖,找到一个是水的点.
  2. 建立集合v,存储已经走过的点,防止重走.
  3. 建立

```
from collections import deque
def bfs(x0,y0,mat):
    q=deque([(x0,y0)])
    d=[(-1,0),(1,0),(0,1),(0,-1)]
    v=set()
    while q:#确定当前位置
        x,y=q.popleft()
        v.add((x,y))
        for dx,dy in d:#从当前位置开始探路
            nx,ny=x+dx,y+dy
            if 0<=nx<len(mat) and 0<=ny<len(mat[0]) and mat[nx][ny]==1:
                q.append((nx,ny))
```

- 与dfs类似,对于与某一条路径相关的量(如路径长度),可以作为参数放在q里,即把q里的元素写成(x,y,l),其中l是路径长.这种方法常用来求迷宫中两点的最短路径.
- 与全局相关的量用全局变量存储(如连通域面积),比如求面积S就每次"确定当前位置之后S+1
- 迷宫中最短路径的求法:用(x,y,l)记录点的位置和路径长,在探路前加判断:如果走到终点,即`(x,y)==e`,就return l.

3-5 Dijkstra算法

- 读作/'daikstrə/,不是dijiekestra...
- 这个算法用于以下情景:一个加权的图(图的每条连线都有一个权重),要求A到B点权重代数和的最小值.
- 例如,几座城市之间修了一些路,每条路有一个长度,要求A城到B城最短的路是多长.
- 对于一般的图,都是如下做法:

1. 构建邻接表(每个节点跟其他哪些节点连在一起,列一张表).
2. 构建小顶堆q存储即将遍历的点
3. 一边遍历一边探路一边更新最小距离(就是bfs)

- 看起来很抽象.直接上代码:

```
import heapq

def dijkstra(n, edges, s, t):
    graph = [[] for _ in range(n)]
    for u, v, w in edges:
        graph[u].append((v, w))
        graph[v].append((u, w))#构建邻接表
    pq = [(0, s)]  # (distance, node)
    visited = set()
    distances = [float('inf')] * n
    distances[s] = 0
    while pq:#遍历节点
        dist, node = heapq.heappop(pq)
        if node == t:return dist
        if node in visited:
            continue
        visited.add(node)
        for neighbor, weight in graph[node]:#走到一个节点-->探路边上的邻居
            if neighbor not in visited:
                new_dist = dist + weight
                if new_dist < distances[neighbor]:
                    new_dist = dist+weight#更新距离
                    if new_dist < distances[neighbor]:
                        distances[neighbor] = new_dist
                        heapq.heappush(pq, (new_dist, neighbor))
    return -1
```

- 这段代码更抽象了,也不够贴近生活实际.
- 在CS101中,我们见到的更多是基于矩阵的Dijkstra算法,因此我们来看看,如果遍历的是矩阵这种特殊的图,Dijkstra算法长什么样.
- Dijkstra的本质是bfs,所以我们拿bfs的代码简单改一下就好了.以下是将一个普通bfs修改为Dijkstra的过程:(注:以下的权重和/权值和就比如说走山路那题的体力消耗量)
    1. 把函数名从bfs改成Dijkstra(确信)
    2. 用小顶堆代替原先的队列q,q中的元素为(w,x,y)元组,其中w是权值和,x,y是坐标.这样每次访问的都是q中权值和最小的点.(实际上这叫优先队列)每次都访问权值和最小的点,使得第一次到终点时的权值和就是全局最小的权值和,也就是第一次到终点的时候就可以直接break而不需要再从其他路径到终点来更新最小权值和.
       注意:heapq的元素如果是元组,会依次按元组中第一,第二...个元素的大小作为该元组的大小比较的依据.所以为了让q按权重和排序,q中的点应表示为`(w,x,y)`,其中`w=weight[x][y]`,是该点到起点的最小权重和
    3. 建立一个与mat一样大的weight矩阵存储起点到每个点的最小权重和,每个点初值赋为无穷大,除了起点是0(权值和既需要存在这个矩阵里,也要存在q中的元组里,两个地方都需要用到这个权值和)
    4. 开始bfs, `while q`...这段不变

5. 探路, `for dx,dy in d`... 这段,入q条件改成:nx,ny在矩阵范围内,起点到nx,ny的权重和小于`weight[nx][ny]`(之前的路径走出的起点到nx,ny的权重和)
6. 在入堆的同时更新`weight[nx][ny]`

- 代码如下:

```
import heapq

def dijkstra(s,mat,e):#s,e分别是起点和终点.起点s=(0,x0,y0),终点e=(xe,ye).
    MAXN=float('inf')
    weight=[[MAXN]*len(mat[0]) for _ in range(len(mat))]
    q=[s]
    weight[s[1]][s[2]]=0
    d=[(-1,0),(1,0),(0,-1),(0,1)]

    #开始bfs
    while q:
        w,x,y=heapq.heappop(q)

        #先处理到终点的情况
        if (x,y)==e:
            return weight[x][y]

        #然后探路
        for dx,dy in d:
            nx,ny=x+dx,y+dy
            if 0<=nx<len(mat) and 0<=ny<len(mat[0]):#不用not in visited
                new_w=weight[x][y]+_______#这段填上点(x,y)到(nx,ny)的权重
                if new_w<weight[nx][ny]:
                    weight[nx][ny]=new_w
                    heapq.heappush(q,(new_w,nx,ny))

    return -1
```

3-6 全排列与Cantor展开

3-6-1 Cantor展开与range(1,n+1)的全排列

- 有1,2,...,n共n个数,想要枚举出它们所有的排列,按字典序从小到大,可以用Cantor展开和逆Cantor展开解决.
- **Cantor展开可以将这些排列每个对应到一个编号上.**以下是计算编号的方法:
    1. 对于每个数,看它后面有几个比它小的数,它的贡献就是几
    2. 从右到左的每一位都有一个权重,从0!一直到(n-1)!
    3. 将每个数的贡献和它所在那一位的权重相乘,求和,就是这个排列的编号.
    - 例如,有1,2,3这三个数,求2,3,1的编号:
        - 2,3,1的贡献分别为1,1,0
        - 从左到右位的权重分别是2!=2,1!=1,0!=1
        - 编号=1*2*+1*1+0*1=3
- **逆Cantor展开是通过排列的编号求出排列的方法.**做法如下:
  0.构建数组`nums=list(range(1,n+1))`,即1,2,...,n

1. 编号除以最左边一位的权重,得到商s和余数q
2. 从nums中**取出**一个数,对于这个数,nums中应有s个数比它小.将这个数放到最左边一位.
3. 对于余数q,重复1和2,直到数组取完.

- 例如,由3求出排列2,3,1:
  - s=3//2=1,从[1,2,3]取出"有一个数比它小"的数:2
  - q=3%2=1.下一轮:
  - s=1//1=1,从[1,3]取出"有一个数比它小"的数:3
  - q=1%1=0.下一轮:
  - s=0//1=0,从[1]取出最小的数:1
  - q=0%1=0,下一轮:数组空了,结束了

- Cantor和逆Cantor的代码如下:

```
import math
def Cantor(nums):
    res=0#res是最终的编号
    for i in range(len(nums)):#对于nums中的每个数
        c=0
        for j in range(i+1,len(nums)):#找到它后面有几个比它小的数
            if nums[j]<nums[i]:
                c+=1
        res+=c*math.factorial(len(nums)-1-i)
    return res

def retro_Cantor(x,length):#length是排列的长度,不然我就不知道是123的排列还是1234的排列了
    res=[]
    r=list(range(1,length+1))
    for i in range(length-1,-1,-1):
        f=math.factorial(i)
        res.append(r.pop(x//f))
        x%=f
    return res
```

- 这样,只要从第一个排列开始,Cantor-->编号-->编号+1-->retro_Cantor,就能找到下一个排列.
- 如此找n!次,就可以回到第一个排列.

3-6-2 不可Cantor展开的排列

- 2,3,6这样的排列怎么办?(**注:一个排列里不可能有两个相同的数**)同样用连续"下一个排列"的方法.
- 我们可以**将2,3,6这样'Uncantorable'的排列通过字典映射到1,2,3这样'Cantorable'的排列上**,然后通过Cantor找下一个排列,再映射回去.
- 建立字典和逆映射的字典即可.
- 例如:`cast={2:1,3:2,6:3},reversed_cast={1:2,2:3,3:6}`
- 代码如下:

```
def next_arrange(a):
    b=[0]+sorted(a)
    reversed_cast=dict(enumerate(b))#对a中的元素逆映射
    cast={v:k for k,v in reversed_cast.items()}
    cantorable_a=[cast[i] for i in a]
    _=(1+Cantor(cantorable_a))%math.factorial(len(a))
    cantorable_new_a=retro_Cantor(_,len(a))
    new_a=[reversed_cast[i] for i in cantorable_new_a]
    return new_a
```

3-7 Narayana Pandita算法求下一个排列

- **对于'Uncantorable'的排列,要求下一个排列**,可以使用这种算法--**这可比康拓简单多了**!实现方法如下:
    1. 对于排列nums,从右往左找到**第一个降序(左<右)的相邻数对** `(nums[i],nums[i+1]),nums[i]<nums[i+1]`.如果一直找不到,就说明这是整个从右到左升序,即从左到右降序的排列,即最后一个排列了.就反转nums.
    2. 找到之后,从右往左找到第一个比nums[i]大的数num[j],调换这两个数.
    3. 把 `nums[i+1:]` 反转.

```
def NP(nums):
    for i in range(len(nums)-2,-1,-1):
        if nums[i]<nums[i+1]:
            for j in range(len(nums)-1,i,-1):
                if nums[j]>nums[i]:
                    nums[j],nums[i] = nums[i],nums[j]
                    tmp=nums[len(nums)-1:i:-1]
                    nums[i+1:]=tmp
                    return nums
    else:
        nums.reverse()
        return nums
print(NP([4,2,6,3]))
```

3-8 二分查找

- 二分查找的边界条件是一个令人头疼的问题,这已经得到了官方认证(如图).
- 一般的bisect_left代码如下.这种二分对升序列表进行查找,找到的返回值lo左边的所有数都比x小,lo本身及其右边都不比x小.

```
def bisect_left(a, x, lo=0, hi=None, *, key=None):
    if hi==None:
        hi=len(a)
    if key is None:
        while lo < hi:
            mid = (lo + hi) // 2
            if a[mid] < x:
                lo = mid + 1
            else:
                hi = mid
    else:
```

```
        while lo < hi:
            mid = (lo + hi) // 2
            if key(a[mid]) < key(x):
                lo = mid + 1
            else:
                hi = mid
    return lo
```

- 由此我们可以将这段代码引申为:返回lo,**lo左边**都是**满足**条件f(x)的,而**lo及lo右边**都是**不满足**的.
  - 即判断条件`key(a[mid]) < key(x)`或`a[mid]<x`改为`f(x)==True`
    - 注意**更新lo,hi**的操作分别为**lo=mid+1**和**hi=mid**,**循环条件**是**lo<hi**
- 如果对代码进行一些改动,变成
  - **返回值及左边**都**满足**f(x),**右边**都**不满足**:
    - 其他操作不变,返回值变成lo-1.
  - **bisect_right(返回值左边**都**不大于**x,**返回值及右边**都**大于**x):
    - 判断条件改为`a[mid]<=x`,返回值为lo(不变)
- 省流:
  - **循环条件永远是lo<hi不动,只改变返回值**
  - "**返回值及左边都...**"=>返回**lo-1**
  - "**返回值左边都...**"=>返回**lo**

3-9 LIS(最长(严格)上升子序列)问题的二分解法

- 建立一个dp数组,初值全设为无穷大,其中的第k项存储"长度为k的上升子序列的最小的末尾值"
- **dp必然是递增的**,因为长度更长的上升子序列末尾值应该>它中间某项的值=长度更短的上升子序列的末尾值.这很重要!
- 遍历序列,对于第i项,找到它可以被作为长度为几的上升子序列的末尾,然后更新这个末尾.这就是二分查找找的东西.
- 最后,dp有多少项填了数,LIS长度就是多少
- 如图

```
import bisect
def lis(a):
    dp=[float('inf')]*(len(a)+2)
    for i in range(len(a)):
        dp[bisect.bisect_left(dp,a[i])]=a[i]
    print(dp)
    return bisect.bisect_left(dp,float('inf'))
```

3-A Kadane算法及衍生的最大子矩阵求法

- 给定一个序列v,其中有正数有负数有0,求其中和最大的子串(即:这个序列里连续的一些数,使之和最大).
- 基本思想是: 土豪购物,但不放回.(这是2024大雪的月考题...题目的一部分是:土豪看到一串商品,它们的价值有正有负有0,土豪要买连续的一些商品,使买到的商品价值总和最大)如下图(第二行的l[i]应是v[i])

- 即:`max_cur=max(max_cur+v[i],v[i])`.如果`max_cur`(已经买到的商品总价值)已经为负,就把它们全扔了重买第i个.否则不扔,并买第i个.
- 每次买之后都要更新一下商品的最大总价值(max_all).

```
def kadane(v):#卡丹算法求最大子序列
    max_cur=0
    max_all=0
    for i in range(len(v)):
        max_cur=max(v[i],max_cur+v[i])
        max_all=max(max_all,max_cur)
    return max_all
```

- 通过这种算法进行衍生,可以类似地求**最大子矩阵**.具体思路是:
    1. 选出矩阵的第l至r列作为一个**切片**,对于这个切片的每一行,**求出其和**,以这些**和**构建一个列表a,这样就把第l至r列的矩阵切片一维化了.
    2. 用Kadane算法求出这个**列表a的最大子串的值,这就是第l~r列的最大子矩阵的值**.如图.
    3. 遍历所有可能的l,r,然后求出全局的最大子矩阵的值.
    - 求切片的和时可以用前缀和:
        1. 构建一个前缀和矩阵qzh,每一行的第i个元素是这一行第0~i元素的和
        2. 某一行l~r列的元素的元素和就是qzh[r]-qzh[l-1].前缀和矩阵需要在左边加一列0做保护层

```
def qzh(mat,n,m):#前缀和
    qzh=[]
    for i in range(n):
        qzh.append([0])
        for j in range(m):qzh[i].append(qzh[i][-1]+mat[i][j])
    return qzh
def kadane(a):#卡丹算法求最大子序列
    max_cur=0
    max_all=0
    for i in range(len(a)):
        max_cur=max(a[i],max_cur+a[i])
        max_all=max(max_all,max_cur)
    return max_all
def max_submat(qzh,n,m):
    #现在我要算每个对于左边界为l，右边界为r的，上下浮动和伸缩的一系列子矩阵的最大值
    maxn=0
    for l in range(1,m+1):
        for r in range(l,m):
            a=list(qzh[i][r]-qzh[i][l-1] for i in range(n))
            maxn=max(maxn,kadane(a))
    return(maxn)
#主函数
n,m=map(int,input().split())#行数:n,列数:m
mat=[]
for i in range(n):
    mat.append(list(map(int,input().split())))
qzh=qzh(mat,n,m)
print(max_submat(qzh,n,m))
```

3-B 排序

3-B-1 归并排序MergeSort,O(nlogn)

- 通过将数组划分成两个小数组,每个小数组排序,然后把两个数组归并到一起(分治)
- 所以有两个操作:归并和排序.
- 是**稳定**排序,即排序完成后相同大小的元素的相对位置不变.比如有5,**5**两个5(第二个5为了清楚表示用加粗标记),排完不会变成**5**,5

```
def MergeSort(arr):
    if len(arr)<=1: return arr
    else:
        l,r=arr[:len(arr)//2],arr[len(arr)//2:]
        return Merge(MergeSort(l),MergeSort(r))
def Merge(l,r):
    res=[]
    i,j=0,0
    while i<len(l) and j<len(r):
        if l[i]<=r[j]:
            res.append(l[i])
            i+=1
        else:
            res.append(r[j])
            j+=1
    res+=l[i:]+r[j:]
    return res
```

3-B-2 快速排序QuickSort,O(nlogn)

- 先在数组arr中随便选一个值mid,不妨选数组中央的那个值`mid=arr[len(arr)//2]`
- 建立左中右(l,m,r)三个新数组,比mid小的全放到l,比x大的放到r,等于mid的放到m
- 然后对左中右分别继续排,排完合并到一起就行了
- **不稳定**

```
def quickSort(arr):
    if len(arr)<=1:
        return arr
    else:
        mid=arr[len(arr)//2]
        l,m,r=[],[],[]
        for i in arr:
            if i < mid : l.append(i)
            elif i > mid : r.append(i)
            else : m.append(i)
        return quickSort(l) + m + quickSort(r)
```