CS101 Cheat Sheet

Compiled by 韩旭

Based on the cheat sheet by 汤伟杰; I add new contents and annotate some codes; The initial version is hand-written.

0. 基础语法清单

1. **list**
   - 生成列表：`list()`，`[x for x in s if x<=a]`
   - `a.append(x)`
   - `a.pop()`（返回数组末尾的值）
   - `a.extend([4,5])`
   - `a.insert(i,x)`（在第i位插入x）
   - `a.remove(x)`（删除第一个x，无返回值）
   - `a[::-1]`（反转数组；拷贝也可以用类似的方法）
   - `a.sort(key=..., reverse=...)`
2. **dict**
   - 生成字典：`d={key:value, key2:value2}`，`d=dict()`，`{i:i*i for i in range(n)}`
   - `d.get(key,default)`
   - `key in d`，可用于查找或者遍历字典中的键
   - `for k,v in d.items()`，遍历字典中的键值对
   - `d.setdefault(k,[]).append(a)`
   - Defaultdict

     ```
     from collection import defaultdict
     d=defaultdict(int) #带默认value的字典，int的默认值是0，list的默认值是[]
     ```
3. **set**
   - 生成集合：`s=set()`，`s={1,2,3}`
   - `s.add(x)`,`s.remove(x)`，后者如果x不存在会报错
   - 集合中判断in/not in是O(1)
4. **sorting**
   - 返回新列表：`b=sorted(a,key=lambda x:(x[1],x[0]), reverse=True)`
   - 修改原列表：`a.sort()`
5. **char**

- `ord(a)`，返回97（注：a是97，A是65）
- `chr(97)`，返回'a'
- `c.isdigit()`，`c.isupper()`，`c.islower()`，`c.isalpha()`

6. **string**
   - `s.find(sub)`，寻找子串，找不到返回-1
   - `s.replace(old,new)`，把old子串替换成new
   - `s.split(sep=',')`
   - `s.strip(chars='\n')`，移除头尾的指定字符
   - `s.join(map(str,lst))`
7. **output**
   - `print(f"{x:.2f}hello")`，保留两位小数
8. **deepcopy**
   - ```
     from copy import deepcopy
     a=[[1],[2],[3]]
     b=deepcopy(a) #一层深拷贝也可以用b=a[:]
     ```
9. **math**
   - `import math`
   - `ceil(), floor(), abs(), pow(x,y) #x的y次方, sqrt(), log(x, base), gcd(a,b), lcm(a,b)`
10. **enumerate**
    - `for i,x in enumerate(iterable, start=1)`
11. **interable**
    - `it=iter(a)`，从a生成迭代器；
    - `x=next(it)`，反复调用next可以遍历it，`next(it,default)`可以设置遍历完成后的默认返回值。
12. **不定行输入**
    - ```
      while True:
          try:
              ...
          except EOFError:
              break
      ```
    - ```
      import sys
      for line in sys.stdin:
          line=line.strip() #逐行读取
          ...
      ```
    - ```
      import sys
      data=sys.stdin.read() #一次性读取
      lines=data.splitlines()
      for line in lines:
          ...
      ```

13. **tuple**（*期末机考后补充，没有特别关注tuple可能是第三题没有AC的部分原因*）
    - 生成元组：`t=(1,2,3)`，`tuple([1,2,3])`
    - tuple的`==`运算，先比较靠前的元素，因此可以用于唯一地标记坐标、序列。

I. Dynamic Programming

0.常见模版

- 最大连续子序列和：`dp[i]`是以`A[i]`结尾的连续序列个数（Kadane算法）。
- 最大上升子序列：`dp[i]`是以`A[i]`结尾的LIS的个数（bisect的用法）。
- 最大上升子序列和：结合了上面两种，遍历`j<i`，`dp[i]=max(dp[i]+a[i],dp[i]) if dp[i]>dp[j]`。
- 最长回文子串：`dp[i][j]`代表子串`s[i:j+1]`是回文，状态转移方程为`dp[i][j]=dp[i+1][j-1] and (s[i]==s[j])`。
- 最长公共子串：将两个字符串放在一块对比，`dp[i][j]`代表string1的第i个和string2的第j个，状态转移方程为`dp[i][j]=dp[i-1][j-1]+1 if str1[i-1]==str2[i-1]`。
- 最长公共子序列：允许不连续，所以需要记录上一次连续的末尾值。状态转移方程为，在最长公共子串的基础上添加`else: dp[i][j]=max(dp[i-1][j],dp[i][j-1])`。

1.背包问题

- 经典做法：`dp[i][j]`代表前i件物品放入容量为j的背包的最大价值，状态转移方程为`dp[i][j]=max(dp[i-1][j],dp[i-1][j-Wi]+Vi)`，其中W是重量，V是价值。
- **滚动数组**的改进做法：由于`dp[i-1]`这一行只会影响`dp[i]`这一行，所以直接用一维的`dp[i]`表示处理到当前物品时背包容量为i的最大价值。
- 注意dp数组的初始化：如果要求恰好装满则容量为0的时候初始化成0，其他为-inf；如果有可能装不满均为0。

1. **01背包** --> 每个物品只能拿一次

```
#小偷背包
def zero_one_bag():
    # V-总容量,n-物品个数,
    # cost=[0,     ],price=[0,     ]
    dp=[0]*(V+1)
    for i in range(1,n+1):            #每个物品
        for j in range(V,cost[i]-1,-1):     #逆向遍历每个容量
            dp[j]=max(dp[j],price[i]+dp[j-cost[i]])
    return dp[-1]
```

2. **完全背包** --> 每个物品可以拿无限次

```
#零钱找零
def total_bag():
    dp=[0]*(V+1)
    for i in range(1,n+1):            #每个物品
        for j in range(1,cost[i]+1):        #正向遍历每个容量，这样dp[j-cost[i]]就包括已经用了若干次i的情形
            dp[j]=max(dp[j],price[i]+dp[j-cost[i]])
    return dp[-1]
```

3. **多重背包** --> 每个物品的个数有限制，把每个物品的个数当成一个物品，比如1张票，2张票，4张票等，这样这些"物品"的0-1有无可以组合成任意个数的物品了（二进制编码），从而转化为01背包。

```
#NBA门票
def many_bag():
    dp=[0]*(V+1)
    for i in range(1,n+1):
        k=1
        while s[i]>0: #把每个个数当作一个物品更新一行
            cnt=min(k,s[i]) #余票为s[i]，二进制到k
            for j in range(V,cnt*cost[i]-1,-1):
                dp[j]=max(dp[j],cnt*price[i]+dp[j-cnt*cost[i]])
            s[i]-=cnt
            k*=2
    return dp[-1]
```

4. **二维费用背包**

```
def two_dimension_cost(n,V1,V2,cost1,cost2,price):
    dp=[[0]*(V2+1) for _ in range(V1+1)]
    for i in range(n):
        for c1 in range(V1-1,cost1[i]-1,-1):
            for c2 in range(V2-1,cost2[i]-1,-1):
                dp[c1][c2]=max(dp[c1][c2],dp[c1-cost1[i]][c2-cost2[i]]+price[i])
    return dp[-1][-1]
```

2.整数分割问题

1. 把n划分为若干个正整数，**不考虑顺序** --> 完全背包
   4：4=3+1=2+2=2+1+1=1+1+1+1 共5种

```
def divide1(n):
    dp=[1]+[0]*n     #把0划分只有0这一种
    for i in range(1,n+1):            #每个数字
        for j in range(i,n+1):                #正向遍历每个容量（每个n）
            dp[j]+=dp[j-i]
    return dp[-1]
```

2. 把n划分为若干个正整数，**考虑顺序**
   4：4=3+1=1+3=2+2=2+1+1=1+2+1=1+1+2=1+1+1+1 共8种

```
def divide2(n):
    dp=[1]+[0]*n
    for i in range(1,n+1):            #每个容量（每个n）
        for j in range(1,i+1):              #每个可能划分出的数字
            dp[i]+=dp[i-j]
    return dp[-1]
```

3. 把n划分为若干个不同的正整数，不考虑顺序 --> 01背包
   4：4=3+1 共1种

```
def divide3(n):
    dp=[1]+[0]*n
    for i in range(1,n+1):
        for j in range(n,i-1,-1):
            dp[j]+=dp[j-i]
    return dp[-1]
```

4. 把n划分为k个正整数，不考虑顺序

```
#放苹果
#dp[n][k]:把n分成k组
def divide4(n,k):
    dp=[[0]*(k+1) for _ in range(n+1)]
    #每个数字分成1组都是1种
    for i in range(n+1):
        dp[i][1]=1
    for i in range(1,n+1):
        for j in range(2,k+1):
            #i<j时无法划分
            #i>=j时分为两种：若分组中有1，则为dp[i-1][j-1]
            #若无1，先把每组放进去1，则为dp[i-j][j]
            if i>=j:
                dp[i][j]=dp[i-1][j-1]+dp[i-j][j]
    return dp[n][k] #dp[-1][-1]
```

3.序列dp

已知`dp[0]`到`dp[i-1]`的所有状态，求出`dp[i]`，即找出`dp[i]`与之前状态的关系。

常见定义：

- `dp[i]`:到第i个位置时的状态（最大值等）；
- `dp[i,j]`:从第i个位置到第j个位置时的状态，或到第i个位置时恰好为状态j。

4.设置多个dp数组

设置dp1和dp2两个数组记录两种状态，一般定义dp1[i]为取s[i]，dp2[i]为不取s[i]，再利用数学归纳思维找出转移方程。

例题：

题目	链接	递推式
cf-Basketball Exercise-1195C	https://codeforces.com/problemset/problem/1195/C	`dp1[i]=max(dp1[i-1],h1[i]+dp2[i-1]);dp2[i]=max(dp2[i-1],h2[i]+dp1[i-1])`
oj-红蓝玫瑰-25573	http://cs101.openjudge.cn/practice/25573	`dpr[i]=dpr[i-1] if s[i]==red else min(dpr[i-1]+1,dpb[i-1]+1);dpb[i]=dpb[i-1] if s[i]==blue else min(dpb[i-1]+1,dpr[i-1]+1)`

5.Kadane算法

oj-最大子矩阵-02766 http://cs101.openjudge.cn/practice/02766/

> Kadane算法
> 一种非常高效的算法，用于求解一维数组中 最大子数组和。它能够在 O(n) 时间复杂度内解决问题，广泛应用于许多动态规划问题中。
> 避免了计算前缀和数组

```
def kadane(s): #一维
    curr_max=total_max=s[0]
    for i in range(1,len(s)):
        curr_max=max(curr_max+s[i],s[i])
        total_max=max(total_max,curr_max)
    return total_max
```

```
def kadane(s): #二维，压缩到一维数组
    curr_max=total_max=s[0]
    for i in range(1,len(s)):
        curr_max=max(curr_max+s[i],s[i])
        total_max=max(total_max,curr_max)
    return total_max
def max_sum_matrix(mat): #上下压缩
    max_sum=-float('inf')
    row,col=len(mat),len(mat[0])
    for top in range(row):
        col_sum=[0]*col
        for bottom in range(top,row):
            for c in range(col):
                col_sum[c]+=s[bottom][c] #对每个top，只需要从上到下遍历一次bottom，求和的时候累加即可
            max_sum=max(max_sum,kadane(col_sum))
    return max_sum
```

II. Dilworth Theory

> **最少单调链个数=最长反单调链长度**
>
> （要解决的问题是：一个序列最少分几组使得每组都是单调的，它等价于找到最长反方向单调子串的长度）
>
> 找最长上升子序列的长度，用left；
> 找最长下降子序列，先reverse，再用left；
> 如果是不降，用right；
> 如果是不升，先reverse，再用right；
> 看题目要求的最终结果是否需要相同元素的考虑，需要考虑用left，不需要用right。

```
from bisect import bisect_left,bisect_right
def d(s): #求最长上升子链长度
    lst=[] #lst[k]代表长度为k+1的上升子序列中，最小的结尾值
    for i in s:
        pos=bisect_left(lst,i)
        if pos<len(lst):
            lst[pos]=i #若只能插在中间，则更新相同长度子序列的结尾
        else:
            lst.append(i)
    return len(lst)
```

注：`bisect_left(lst,i)`：默认lst已经排序，返回i应该插入的位置（即保证i的位置之后的数都大于等于i）；`bisect_right(lst,i)`则保证之前的数都小于等于i。比如`lst=[1,2,3,3,4,5]`，`i=3`，则left返回2，right返回4。

III. Prefix Sum

1.一维前缀和数组

用于处理**多次查询**从[l,r]的序列之和的问题

```
s=[int(i) for i in input().split()]
prefix=[s[0]]+[0]*(len(s)-1)
for i in range(1,len(s)):
    prefix[i]=prefix[i-1]+s[i]

distance_l_r=prefix[r]-(prefix[l-1] if l-1>=0 else 0)
```

2.二维前缀和

`prefix[i][j]`代表从(0,0)到(i-1,j-1)的元素和。

二位前缀和数组的生成：`prefix[i][j]=matrix[i-1][j-1]+prefix[i-1][j]+prefix[i][j-1]-prefix[i-1][j-1]`;

查询从(x_1,y_1)到(x_2,y_2)的矩阵区域之和：`sum=prefix[x2+1][y2+1]-prefix[x1][y2+1]-prefix[x2+1][y1]+prefix[x1][y1]`.

3.前缀和的特殊用法(哈希表)

使用prefix和prefix_map来记录已有的前缀和，从而判断子串和为0的子串个数（注：`pre[i]==pre[j]`则说明i到j-1之间的和为0；或找相同前缀和数字出现的最远位置。
例题：

题目	链接
oj-完美的爱-27141	http://cs101.openjudge.cn/practice/27141/
cf-Kousuke's Assignment-2033D	https://codeforces.com/problemset/problem/2033/D

```
#找出不重叠的和为0的子序列个数，一旦找到就将prefixed集合清空
#cf-Kousuke's Assignment-2033D
t = int(input())
for _ in range(t):
    n = int(input())
    a = list(map(int, input().split()))

    prefix = 0
    prefixed = {0}
    cnt = 0
    for i in a:
        prefix += i
        if prefix not in prefixed: #还未出现相同的前缀和
            prefixed.add(prefix)
        else:
            cnt += 1
            prefix = 0
            prefixed={0}
    print(cnt)
```

IV. SORTING

1.冒泡排序

```
def bubble_sort(s):
    n=len(s)
    f=True
    for i in range(n-1):
        f=False
        for j in range(n-i-1): #最大的一定会被交换到每一轮的最右侧（“冒泡”）
            if s[j]>s[j+1]:
                s[j],s[j+1]=s[j+1],s[j]
```

```
                f=True
        if f==False:
            break
    return s
```

2.归并排序 --> 递归

基本思路：分成左右两个list，一次比较从头开始的元素。

```
def merge_sort(s):
    if len(s)<=1:
        return s
    mid=len(s)//2
    left=merge_sort(s[:mid])
    right=merge_sort(s[mid:])    #两次递归放在一起，与 hanoi tower 的递归以及 lc-LCR085-括号生成 的递归很相似
    return merge(left,right)
def merge(l,r):
    ans=[]
    i=j=0
    while i<len(l) and j<len(r):
        if l[i]<r[j]:
            ans.append(l[i])
            i+=1
        else:
            ans.append(r[j])
            j+=1
    ans.extend(l[i:])
    ans.extend(r[j:])
    return ans
```

```
#lc-LCR085-括号生成
class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
        ans=[]
        path=['']*(n*2) #对path原地修改，然后放入ans；否则要考虑浅拷贝的问题
        def dfs(i,left): #i:填充的索引; left:左括号的个数
            if i==n*2:
                ans.append(''.join(path))
            if left<n: #如果左括号的个数小于n个(可以填一个左括号)
                path[i]='('
                dfs(i+1,left+1)
            if i-left<left: #如果右括号的个数小于左括号的(可以填一个右括号)
                path[i]=')'
                dfs(i+1,left)
        dfs(0,0)
        return ans
```

3.快速排序 --> 递归，选基准

```
def quick_sort(s):
    if len(s)<=1:
        return s
    base=s[0]
    left=[x for x in s[1:] if x<base]
    right=[x for x in s[1:] if x>=base]
    return quick_sort(left)+[base]+quick_sort(right)
```

4.lambda函数

```
sort() #--> 稳定的从小到大排序，如果列表存储的是多元元组，则依次按照每个元组的元素进行排序，且稳定
#如果想自行按照元组的元素顺序排序，可以使用lambda函数
s=[(1,2),(3,1),(4,5),(2,5)]
#按照第二个元素排序
s.sort(key=lambda x:x[1]) #[(3, 1), (1, 2), (2, 5)]
#按照第二个元素为首要升序排序，第一个元素为次要升序排序
s.sort(key=lambda x:(x[1],x[0])) #[(3, 1), (1, 2), (2, 5), (4, 5)]
#按照第二个元素为首要降序排序，第一个元素为次要升序排序
s.sort(key=lambda x:(-x[1],x[0])) #[(2, 5), (4, 5), (1, 2), (3, 1)]
#----------------------------#
#如果想对数字按照字典序组合排序，得到最大最小整数，可以冒泡可以匿名
s=[9989,998]
#冒泡
for i in range(len(s)-1):
    for j in range(len(s)-i-1):
        if str(s[j])+str(s[j+1])<str(s[j+1])+str(s[j]): #判断字典序相对大小
            s[j],s[j+1]=s[j+1],s[j]
#lambda函数
s=sorted(s,key=lambda x: str(x)*10,reverse=True) #这里是一个经验技巧：数字重复多次后排序大致能得到拼接后的最大或最小整数
#----------------------------#
#对字典的键值对进行排序，与列表存储元组差不多
d={3:34,2:23,9:33,10:33}
dd=dict(sorted(d.items(),key=lambda x:(x[1],-x[0]))) #{2: 23, 10: 33, 9: 33, 3: 34}
```

V. SEARCHING

1.DFS

dfs如果要解决枚举类的题目通常会涉及回溯操作，而在原地修改时可能无需回溯。如果有回溯操作必须要有退出条件。
防止递归深度过大，可以这样调整递归深度：

```
import sys
sys.setrecursionlimit(1 << 30)
```

如果dfs内部有类似于dp数组需要不断访问某些元素的值的时候，除了开空间创建一个dp，还可以用lru_cache。但一定要在需要进行记忆化递归的函数头顶上写，否则无效。

```
from functools import lru_cache
@lru_cache(maxsize=2048) #或者更大，如None，考虑内存因素自行调整
def dfs():
    ...
```

1. 无回溯操作
   例题：oj-lake counting-02386，原地修改

```
dx=[-1,0,1,-1,1,-1,0,1]
dy=[-1,-1,-1,0,0,1,1,1]
def dfs(x,y):
    m[x][y]='.'
    for k in range(8):
        nx=x+dx[k]
        ny=y+dy[k]
        if 0<=nx<=n-1 and 0<=ny<=s-1 and m[x][y]=='W':
            dfs(nx,ny)
```

2. 有回溯操作
   模板是：①有退出条件 ②递归之间做重复要做的事情 ③递归之后回溯为原状态

```
def dfs():
    if ...:
        return
    #do something
    dfs()
    #traceback
```

例题：

```
#oj-八皇后-02754
'''考虑以下递归步骤：
在某次递归时，curr = [1, 5, 8, 6]，此时 ans.append(curr)。
接下来，回溯修改了 curr，变为 [1, 5, 8, 7]。
由于 ans 中保存的是 curr 的引用，ans 中原本存储的 [1, 5, 8, 6] 也会变为 [1, 5, 8, 7]。
因此使用 curr[:]，创建当前列表的拷贝，确保后续对 curr 的修改不会影响已保存的解
'''
visited=[0]*8
ans=[]
def dfs(k,curr):
    global ans
    if k==9:
        ans.append(curr[:])
        return
    for i in range(1,9):
        if visited[i-1]:
            continue
        if any(abs(j-i)==abs(len(curr)-curr.index(j)) for j in curr):
```

```
            continue
        visited[i-1]=1
        curr.append(i)
        dfs(k+1,curr)
        visited[i-1]=0
        curr.pop()
dfs(1,[])
# print(ans)
for _ in range(int(input())):
    n=int(input())
    print(''.join(map(str,ans[n-1])))
```

```
#oj-有界的深度优先搜索-23558
def dfs(n,m,l,s,ans,k):
    if k==l+1 or s not in d:
        return
    for i in d[s]:
        if not visited[i] and i not in ans:
            visited[i]=1
            ans.append(i)
            dfs(n,m,l,i,ans,k+1)
            visited[i]=0

n,m,l=map(int,input().split())
d={}
for _ in range(m):
    a,b=map(int,input().split())
    if a>b: a,b=b,a
    if a not in d:
        d[a]=[]
    d[a].append(b)
    if b not in d:
        d[b]=[]
    d[b].append(a)

for v in d.values():
    v.sort()

s=int(input())
visited=[0]*n
ans=[s]
visited[s]=1
dfs(n,m,l,s,ans,1)
print(*ans)
```

2.BFS

逐层扩展，用来求最小步数，模板；如果想保留路径，可以把路径作为参数传递，其中双端队列q加入的元素可能是三维，包含坐标和时间或者步数或者路径等等。

```
from collections import deque
```

```
dx,dy=[0,-1,1,0],[-1,0,0,1]
def bfs(x,y,final):
    q=deque()
    q.append((x,y))
    inq=set()
    inq.add((x,y))
    step=1
    while q:
        for _ in range(len(q)): #遍历这一层，可以不写这一行，但是写了更清晰
            x,y=q.popleft()
            for i in range(4):
                nx,ny=x+dx[i],y+dy[i]
                if s[nx][ny]==final:
                    return step
                if 0<=nx<n and 0<=ny<m and s[nx][ny]==1 and (nx,ny) not in inq:
                    q.append((nx,ny))
                    inq.add((nx,ny))
        step+=1
    return None
```

例题:

```
#oj-体育游戏跳房子-27237
#deque中多加入一个path不断传递
from collections import deque
def bfs(n,m,path):
    step=1
    q=deque()
    q.append((n,path))
    inq=set()
    inq.add(n)
    while q:
        for _ in range(len(q)):
            x,path=q.popleft()
            if x*3>0:
                if x*3==m:
                    return step,path+['H']
                if x*3 not in inq:
                    q.append((x*3,path+['H']))
                    inq.add(x*3)
            if x//2>0:
                if x//2==m:
                    return step,path+['O']
                if x//2 not in inq:
                    q.append((x//2,path+['O']))
                    inq.add(x//2)
        step+=1

while True:
    n,m=map(int,input().split())
    if {n,m}=={0}:
        break
    step,path=bfs(n,m,[])
```

```
    print(step)
    print(''.join(path))
```

3.Dijkstra算法

解决单源最短路径问题，用于非负权图，使用`heapq`的最小堆来代替`bfs`中的`deque`，设置`dist`列表更新最短距离。
例题：

```
#oj-走山路-20106
import heapq
dx,dy=[0,-1,1,0],[-1,0,0,1]
def dijkstra(sx,sy,ex,ey):
    if s[sx][sy]=='#' or s[ex][ey]=='#':
        return 'NO'
    q=[]
    dist=[[float('inf')]*m for _ in range(n)]
    heapq.heappush(q,(0,sx,sy)) #(distance,x,y)
    dist[sx][sy]=0
    while q:
        curr,x,y=heapq.heappop(q) #heappop()
        if (x,y)==(ex,ey):
                return curr

        for i in range(4):
            nx,ny=x+dx[i],y+dy[i]
            if 0<=nx<n and 0<=ny<m and s[nx][ny]!='#':
                new=curr+abs(s[x][y]-s[nx][ny])
                if new<dist[nx][ny]:
                    heapq.heappush(q,(new,nx,ny)) #heappush()
                    dist[nx][ny]=new
    return 'NO'
```

VI.DATA STRUCTURE

0.Deque的用法

- 导入包：`from collections import deque`；
- 初始化：`deque()`，`deque([1,2,3])`；
- 函数：`append(x)`, `appendleft(x)`, `extend(iter)`, `extendleft(iter)` #逆序, `pop()`, `popleft()`, `remove(x)`；
- 注意deque的访问（`dq[i]`）是O(n)，而list是O(1)。

1.Stack

栈(stack)，使用`list`来模拟，遵循后进先出的原则。
例题：

```
#oj-快速堆猪-22067
#辅助栈
stack,min_so_far=[],[]
while True:
    try:
        s=input()
    except EOFError:
        break
    if s[-1].isdigit():
        n=int(s.split()[1])
        stack.append(n)
        if not min_so_far:
            min_so_far.append(n)
        else:
            min_so_far.append(min(min_so_far[-1],n))
    elif s=='pop' and stack:
        stack.pop()
        min_so_far.pop()
    elif s=='min' and stack:
        print(min_so_far[-1])
```

此外，还有常用的**单调栈**(monotonic stack)，其优点是：若维护了一个单调递增栈，则每次取出栈顶元素时，**新的栈顶元素**和**不符合条件而未入栈的元素**恰好是**取出的元素两侧的距离最近的比其小的元素**；单调递减栈类似。

单调栈的应用：寻找下一个更大/更小的元素。弹出某元素的过程可以视为轮到查看它的下一个最大/最小元素，或者视为删掉不符合条件的数。

例题：

```
#lc-接雨水-42 维护递减栈
#单调栈的好处是：
#由于维护的是单调下降的高度，当弹出栈顶元素的，其左侧就是左侧第一个比它高的元素
#而弹出操作也意味着右侧就是右侧第一个比它高的元素
s=list(map(int,input().split()))
stack=[]
ans=0
for i in range(len(s)):
    while stack and s[stack[-1]]<s[i]:
        curr=s[stack.pop()] #利用pop()函数的返回值很重要
        if not stack:
            break
        curr_w=i-stack[-1]-1
        curr_h=min(s[i]-curr,s[stack[-1]]-curr)
        ans+=curr_w*curr_h
    stack.append(i)
print(ans)
```

```
#oj-护林员盖房子-21577
def house(mat,n,m): #预处理，逐层计算，转换为求最大矩形面积(lc-最大矩阵-85)
    prefix=[[0]*(m+2) for _ in range(n+2)]
    for i in range(1,n+1):
        for j in range(1,m+1):
            prefix[i][j]=0 if mat[i-1][j-1]==1 else prefix[i-1][j]+1

    ans=0
    for i in range(1,n+1):
        ans_=0
        stack=[0]
        for j in range(1,m+2):
            while stack and prefix[i][j]<prefix[i][stack[-1]]:
                curr_h=prefix[i][stack.pop()]
                curr_w=j-stack[-1]-1
                ans_=max(ans_,curr_h*curr_w)
            stack.append(j)
        ans=max(ans,ans_)
    return ans
n,m=map(int,input().split())
mat=[[int(i) for i in input().split()] for _ in range(n)]
print(house(mat,n,m))
```

2.Heapq

最小堆(heapq)可以维护列表中的最小值并将其位置放在第一个，即heap[0]。如果想得到最大值，以负值形式存入。且最小堆通常涉及到内部元素的删除，而内置函数无此操作，则会利用到**懒删除**操作，使用字典记录已被删除的元素，需要取最小值时再一次性删除。

基本语法：

```
#建堆
import heapq
a=[1,2,3,4,5]
heapq.heapify(a) #在a上原地修改，会破坏a的原顺序。也可以直接从空数组开始用heapq库中的方法进行维护。

#出入堆
x=6
heapq.heappush(a,x) #把x放进堆a中
heapq.heappop(a) #弹出a的堆顶元素（即最小元素）
```

例题：

```
#懒删除 oj-快速堆猪-22067
import heapq
from collections import defaultdict
out=defaultdict(int)
stack,heap=[],[]
while True:
    try:
        s=input()
```

```
    except EOFError:
        break

    if s=='pop' and stack:
        toss=stack.pop()
        out[toss]+=1
    elif s=='min' and stack:
        while heap:
            curr_min=heapq.heappop(heap)
            if out[curr_min]==0:
                print(curr_min)
                heapq.heappush(heap,curr_min)
                break
            out[curr_min]-=1

    elif s[-1].isdigit():
        n=int(s.split()[1])
        stack.append(n)
        heapq.heappush(heap,n)
```

```
#oj-剪绳子-18164
from heapq import heappop,heappush,heapify
n=int(input())
s=list(map(int,input().split()))
ans=0
heapify(s)
while len(s)>1:
    a=heappop(s)
    b=heappop(s)
    ans+=a+b
    heappush(s,a+b)
print(ans)
```

后悔解法 cf-potions-1526C1 tags:data structure,greedy

```
import heapq
n=int(input())
s=list(map(int,input().split()))
health=0
drunk=0
heap=[]
for p in s:
    if p+health>=0:
        drunk+=1
        heapq.heappush(heap,p)
        health+=p
    elif heap and p>heap[0]:
        smallest=heapq.heappop(heap)
        health-=smallest
        heapq.heappush(heap,p)
        health+=p
print(drunk)
```

VII.INTERVAL PROBLEMS

区间合并问题常常涉及到对区间左端点或者右端点的排序。
eg：

1. 合并所有有交集的区间，返回最终个数--对左端点排序，不断更新右边界（注：判断是否视为新区间）

```
#s=[(l1,r1),(l2,r2),...,(ln,rn)]
s.sort(key=lambda x:x[0])
cnt,ans,l,r=1,[],s[0][0],s[0][1]
for i in range(1,n):
    if s[i][0]<=r:
        r=max(r,s[i][1])
    else:
        ans.append([l,r])
        l,r=s[i][0],s[i][1]
        cnt+=1
print(cnt,ans)
```

2. 选择尽量多的无交集的区间，返回最大数量--对右端点排序（注：优先保留结束早的区间）

```
#s=[(l1,r1),(l2,r2),...,(ln,rn)]
s.sort(key=lambda x:x[1])
cnt,r=1,s[0][1]
for i in range(1,n):
    if s[i][0]<=r:
        continue
    cnt+=1
    r=s[i][1]
print(cnt)
```

3. 区间选点，选取尽可能少的点，使得每个区间都至少有一个点（例子：oj-进程检测-04100）--对右端点排序

   这里解法与最大不相交区间（2）一致，每次新区间选择最右侧端点，下一个如果有交集，则跳过；如果没有，则更新区间多选一个点。

```
k=int(input())
for _ in range(k):
    n=int(input())
    curr=0
    cnt=0
    sd=[]
    for _ in range(n):
        s,d=map(int,input().split())
        sd.append([s,d])
    sd.sort(key=lambda x:x[1])
    for i in range(0,n):
        if sd[i][0]>curr:
            curr=sd[i][1]
            cnt+=1
    print(cnt)
```

4. 区间覆盖--对左端点排序，从起点开始每次选最远的右端点

```
#标准模版
intervals.sort() #按左端点排序
ans,i,covered=0,0,0
while covered<n:
    best=covered
    while i<n and intervals[i][0]<=covered+1: #所有能接上的区间，选择右端点最大的
        best+=max(best,intervals[i][1])
        i+=1
    if best==covered:
        break
    covered=best
    ans+=1

#更快写法：右端最远覆盖数组
far=[0]*(n+2)
for i,x in enumerate(intervals):
    L=max(1,i-x) #左端点
    R=min(n,i+x) #右端点
    far[L]=max(far[L],R) #以L为左端点的最大右端点

for i in range(1, n+1): #对左端点遍历
    best=max(best, far[i])
    if i>covered: #已选中区间出现断开
        ans+=1
        covered=best #选的不是L==i的，而是L<=i中最右端最远的best
```

5. 区间分组问题，最少能分成多少组使得组内区间互不相关（**最少分组=任意时刻重叠区间的最大个数**）

   标准解法：按照左端点排序，依次看某区间能否放入现有的组里（用heap维护已有分组的最小右端点）

   例子：主持人调度--对左端点排序-转为事件（在排序时增加第二个元素）

```
def min_host(n,ranges):
    events=[]
    for i in range(n):
        events.append((ranges[i][0],1)) #新添1，表示出现一个起点时主持人数加1
        events.append((ranges[i][1],-1)) #新添-1，表示出现一个终点时主持人数减1
        #最后统计主持人数聚集最多的个数，就是答案
    events.sort(key=lambda x:(x[0],x[1])) #按时间排序，时间相同优先处理“结束”事件
    min_hosts=0
    curr=0

    for time,num in events:
        curr+=num
        min_hosts=max(min_hosts,curr) #原代码有误，这里统计需要主持人最多的时段
    return min_hosts
```

类似的，将区间转换为事件，遍历事件的两个端点的例题：

```
# cf-Best Price-2051E
```

```
for _ in range(int(input())):
    n,k=map(int,input().split())
    events=[]
    for i in list(map(int,input().split())):
        events.append((i,1)) #表示下一个价格这个事件将变为bad
    for i in list(map(int,input().split())):
        events.append((i,2)) #表示下一个价格这个事件将变为无评价
    events.sort()
    i=0
    cost=0
    bad=0
    people=n
    while i<n*2:
        curr=events[i][0]
        if bad<=k:
            cost=max(cost,people*events[i][0])
        while i<n*2 and events[i][0]==curr:
            bad+=(events[i][1]==1)
            bad-=(events[i][1]==2)
            people-=(events[i][1]==2)
            i+=1
    print(cost)
```

6. 覆盖连续区间（例：购物）

   核心思路：要覆盖[1,N]上的所有整数值，就维护当前能覆盖的最大区间右端`max_reach`。在所有小于等于`max_reach+1`的硬币中（这些硬币加上后仍能构成连续区间），选择面值最大的，直到`max_reach>=N`。

VIII. Miscellaneous

1.求解或判断质数

如果是判断某个数字或者很少的数字是否为质数，可用步长为6来判断（因为质数除了2，3都满足6k-1或6k+1）；
如果是判断较多数字是否为质数，或者获取大区间内的质数，使用欧拉筛

```
#以6为步长
import math
def is_prime(n):
    if n <= 1: # 1 不是质数
        return False
    if n <= 3: # 2 和 3 是质数
        return True
    # 2 和 3 以外的偶数和能被 3 整除的数不是质数
    if n % 2 == 0 or n % 3 == 0:
        return False
    # 从 5 开始，步长为 6
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
```

```
        i += 6
    return True
```

```
#欧拉筛
def euler_sieve(n):
    primes = []
    is_prime = [True] * (n + 1)
    is_prime[0] = is_prime[1] = False  # 0和1不是质数
    for i in range(2, n + 1):
        if is_prime[i]:
            primes.append(i)  # i 是质数
        for prime in primes:
            if i * prime > n:
                break
            is_prime[i * prime] = False
            # 如果 prime 是 i 的最小质因数，停止继续筛选
            if i % prime == 0:
                break
    return primes
```

2.分解质因数

```
def pFactors(n):
    """Finds the prime factors of 'n'"""
    from math import sqrt
    pFact, limit, check, num = [], int(sqrt(n)) + 1, 2, n

    for check in range(2, limit):
        while num % check == 0:
            pFact.append(check)
            num /= check
    if num > 1:
        pFact.append(num)
    return pFact
# print(pFactors(12))
```

3.二分查找

在进行二分之前一般需要对列表进行排列。在一类特殊题中与greedy结合，如表述为**求最大值中的最小值。**
例题：

```
#oj-aggressive cows-02456
def binary_search():
    l=0
    r=(s[-1]-s[0])//(c-1)
    while l<=r: #<=
        mid=(l+r)//2
        if can_reach(mid):
            l=mid+1
```

```
        else:
            r=mid-1
    return r #r
def can_reach(mid):
    cnt=1
    curr=s[0]
    for i in range(1,n):
        if s[i]-curr>=mid:
            cnt+=1
            curr=s[i]
    return cnt>=c
```

4.排列组合

`permutations(list,r)`其中r默认是全排列，若`list=[1,2,3];r=2`，则会输出从列表任取两个数进行全排列的所有排列。

```
from itertools import permutations #时间复杂度为n!
perms=permutations([1,2,3])  #此时perms是一个迭代器，需要用for取出，或者转成列表
for perm in perms:
    print(perm) #为元组
perms_list=list(permutations([1,2,3]))
print(perms_list)
```

`combinations(list,r)`与排列类似，但第二个r参数必不可少。

```
from itertools import combinations
combs=combinations([1,2,3],3)
for c in combs:
    print(c)
```

5.zip函数

`zip()`将多个可迭代对象进行组合，成为一个一个的元组，返回值是一个zip对象，可以转为list或dict

```
a=[1,2,3]
b=['n','m','l']
zipped=zip(a,b)
z_list=list(zipped)
z_dict=dict(zipped)
####还可以进行解压####

zipped = [('Alice', 25), ('Bob', 30), ('Charlie', 35)]
# 解压
names, ages = zip(*zipped)
print(names)  # ('Alice', 'Bob', 'Charlie')
print(ages)   # (25, 30, 35)
```

6.矩阵运算

```
an,am=map(int,input().split())
a=[list(map(int,input().split())) for i in range(an)]
bn,bm=map(int,input().split())
b=[list(map(int,input().split())) for i in range(bn)]
cn,cm=map(int,input().split())
c=[list(map(int,input().split())) for i in range(cn)]
if am!=bn or an!=cn or bm!=cm:
    print('Error!')
else:
    result=[[0]*bm for i in range(an)]
    for i in range(an):
        for j in range(bm):
            result[i][j]=sum(a[i][k]*b[k][j] for k in range(am))
    for i in range(cn):
        for j in range(cm):
            result[i][j]+=c[i][j]
    for i in result:
        print(' '.join(map(str,i)))
```