Final_cheat_sheet

I.DP

1.背包问题

初始化：如果要求恰好装满则为0；如果有可能装不满为-inf

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
            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,把每个物品的个数拆成1 2 4等转化为01背包

```
#NBA门票
def many_bag():
    #s=[0,   ]为每个物品的个数
    dp=[0]*(V+1)
    for i in range(1,n+1):
        k=1
        while s[i]>0:
            cnt=min(k,s[i])
            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+数学归纳思维

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+数学归纳思维

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

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]
            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**
> 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=[]
    for i in s:
        pos=bisect_left(lst,i)
```

```
        if pos<len(lst):
            lst[pos]=i
        else:
            lst.append(i)
    return len(lst)
```

III.PREFIX SUM

1.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.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；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:
            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):
            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

```
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)
```

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：

```python
#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。

```python
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
```

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'
```

4.SPFA算法

[额外补充] `Shortest Path Faster Algorithm` 用来解决有负权图。

```
from collections import deque
def spfa(graph, start):
    # graph: 邻接表表示的图，start: 起始点
    # 返回从起始点到所有点的最短距离，若存在负权环，返回None
    n = len(graph)
    dist = [float('inf')] * n
    count = [0] * n          # 标记节点被加入队列的次数
    dist[start] = 0
    queue = deque([start])
    in_queue = set([start])  # 使用 set 来判断节点是否在队列中
    count[start] = 1
    while queue:
        node = queue.popleft()
        in_queue.remove(node)  # 从队列中移除
        # 遍历邻接节点
        for neighbor, weight in graph[node]:
            new_dist = dist[node] + weight
            if new_dist < dist[neighbor]:
                dist[neighbor] = new_dist
                # 如果节点还没在队列中，加入队列
                if neighbor not in in_queue:
                    queue.append(neighbor)
                    in_queue.add(neighbor)  # 添加到 set 中
                    count[neighbor] += 1
                    # 如果某个节点被加入队列超过 V 次，说明有负权环
                    if count[neighbor] > n:
                        print("图中存在负权环! ")
                        return None
    return dist
# 测试
graph = [
    [(1, 1), (2, 4)],  # 0 -> 1 (1), 0 -> 2 (4)
    [(2, 2), (3, 5)],  # 1 -> 2 (2), 1 -> 3 (5)
    [(3, 1)],          # 2 -> 3 (1)
    [(1, -2)], ]        # 3 -> 1 (-2)，这个负权边会形成负权环
```

```
start = 0
dist = spfa(graph, start)
if dist:
    print(f"从节点 {start} 到其他节点的最短距离: {dist}")
```

VI.DATA STRUCTURE

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。
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()]
        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)
```

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。
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)
```

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

```
#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

```
#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. oj-进程检测-04100(区间选点)--对右端点排序

```
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

```
#oj-世界杯只因-27104
def min_cameras(ranges):
    n=len(ranges)
    mx=max(ranges)
    curr=0
    num=0
    while curr<n:
        next=curr+ranges[curr]+1
```

```
        for i in range(max(0,curr-mx),min(n,curr+mx+1)):
            next=max(next,i+ranges[i]+1)
        num+=1
        curr=next
    return num
```

5. 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]))
    min_hosts=0
    curr=0

    for time,num in events:
        curr+=num
        min_hosts=min(min_hosts,curr)
    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)
```

VIII.Other trivial things

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

```python
#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
```

```python
#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))
```

BASIC GRAMMAR

1.零碎

`for index,value in enumerate(s)` 取出列表s中的索引和对应的值

`''.join(map(str,s))` 将列表s中的元素无间隔连接

`keys=[key for key,value in d.items() if value==target]` 根据值找到所有的键

`key=next((key for key,value in d.items() if value==target),None)` 根据值找到第一个对应的键

`pow(2,10000000000,10**9+7)` 内置的pow(base,exp,mod)非常快

```
from collections import defaultdict
d=defaultdict(int) #可以防止元素不存在时报错
#-------------------------------------#
#如果不确定输入的列表或元组的元素个数，可以使用*a的方式接受一段未知长度的列表
s=[1,2,3,4,5] #或者s=(1,2,3,4,5)
a,*b=s
print(a,b) #1 [2, 3, 4, 5]
```

2.二分查找

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
```

3.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)
```

4.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)
```