CHEATSHEET

```
#0-1knapsack
# n, v分别代表物品数量，背包容积
n, v = map(int, input().split())
# w为物品价值，c为物品体积（花费）
w, cost = [0], [0]
for i in range(n):
    cur_c, cur_w = map(int, input().split())
    w.append(cur_w)
    cost.append(cur_c)
#该初始化代表背包不一定要装满
dp = [0 for j in range(v+1)]
for i in range(1, n+1):
    #注意：第二层循环要逆序循环
    for j in range(v, 0, -1):       #可优化成 for j in range(v, cost[i]-1, -1):
        if j >= cost[i]:#否则j<cost[i],dp[i][j]=dp[i-1][j],也就是dp[j]无需更新
            dp[j] = max(dp[j], dp[j-cost[i]]+w[i])
print(dp[v])
#_______________________________________________________
#complete knapsack
# n, v分别代表物品数量，背包容积
n, v = map(int, input().split())
# w为物品价值，c为物品体积（花费）
w, cost = [0], [0]
for i in range(n):
    cur_c, cur_w = map(int, input().split())
    w.append(cur_w)
    cost.append(cur_c)
#该初始化代表背包不一定要装满
dp = [0 for j in range(v+1)]

for i in range(1, n+1):
    #注意：第二层循环要逆序循环
    for j in range(v, 0, -1):       #可优化成 for j in range(v, cost[i]-1, -1):
        if j >= cost[i]:#否则j<cost[i],dp[i][j]=dp[i-1][j],也就是dp[j]无需更新
            dp[j] = max(dp[j], dp[j-cost[i]]+w[i])

print(dp[v])
#________________________________________________________
#multi-knapsack
# n, v分别代表物品数量，背包容积
n, v = map(int, input().split())
# w为物品价值，c为物品体积（花费）
w, cost, s = [0], [0], [0]
for i in range(n):
    cur_c, cur_w,cur_s= map(int, input().split())
    w += [cur_w]*cur_s
    cost += [cur_c]*cur_s
n = len(w)-1
#该初始化代表背包不一定要装满
dp = [0 for j in range(v+1)]
for i in range(1, n+1):
```

```python
        for j in range(v, cost[i]-1, -1):
            if j >= cost[i]:
                dp[j] = max(dp[j], dp[j-cost[i]]+w[i])
print(dp[v])
```

```python
class DjsSet:
    def __init__(self,N):
        self.parent=[i for i in range(N+1)]
        self.rank=[0 for i in range(N+1)]
    def find(self,x):
        if self.parent[x]==x:
            return x
        else:
            result=self.find(self.parent[x])
            self.parent[x]=result
            return result
    def union(self,x,y):
        xset=self.find(x)
        yset=self.find(y)
        if xset==yset:
            return
        if self.rank[xset]>self.rank[yset]:
            self.parent[yset]=xset
        else:
            self.parent[xset]=yset
            if self.rank[xset]==self.rank[yset]:
                self.rank[yset]+=1
```

```python
def Dijskra(start,end,graph):
    heap=[(0,start,[start])]
    heapq.heapify(heap)
    has_gone=set()
    while heap:
        (length,start,path)=heapq.heappop(heap)
        if start in has_gone:
            continue
        has_gone.add(start)
        if start==end:
            return path
        for i in graph[start]:
            if i not in has_gone:
                heapq.heappush(heap,(length+graph[start][i],i,path+[i]))
```

```python
from collections import defaultdict
from heapq import *
def Prim(vertexs, edges,start='D'):
    adjacent_dict = defaultdict(list) # 注意：defaultdict(list)必须以list做为变量
    for weight,v1, v2 in edges:
        adjacent_dict[v1].append((weight, v1, v2))
        adjacent_dict[v2].append((weight, v2, v1))
    minu_tree = []  # 存储最小生成树结果
    visited = [start] # 存储访问过的顶点，注意指定起始点
    adjacent_vertexs_edges = adjacent_dict[start]
```

```python
    heapify(adjacent_vertexs_edges) # 转化为小顶堆，便于找到权重最小的边
    while adjacent_vertexs_edges:
        weight, v1, v2 = heappop(adjacent_vertexs_edges) # 权重最小的边，并同时从堆中删除。
        if v2 not in visited:
            visited.append(v2)  # 在used中有第一选定的点'A'，上面得到了距离A点最近的点'D',举例是5。将'd'追加到used中
            minu_tree.append((weight, v1, v2))
            # 再找与d相邻的点，如果没有在heap中，则应用heappush压入堆内，以加入排序行列
            for next_edge in adjacent_dict[v2]: # 找到v2相邻的边
                if next_edge[2] not in visited: # 如果v2还未被访问过，就加入堆中
                    heappush(adjacent_vertexs_edges, next_edge)
    return minu_tree
```

```python
def kruskal():
    n,m,tot_weight,graph=build()
    djsset=Dsjset(n)
    kruskal_weight=0
    cnt=0
    for edge in graph:
        if djsset.find(edge.start)!=djsset.find(edge.end):
            djsset.union(edge.start,edge.end)
            cnt+=1
            kruskal_weight+=edge.weight
        if cnt==n-1:
            break
    return kruskal_weight
```

```python
def topo_seq():
    import heapq
    n,graph=build()
    start=[]
    for i in range(n):
        if graph[i].indeg==0:
            start.append(graph[i])
    heapq.heapify(start)
    seq=[]
    while start:
        temp=heapq.heappop(start)
        seq.append(temp.name)
        for i in temp.out:
            i.indeg-=1
            if i.indeg==0:
                heapq.heappush(start,i)
        if len(seq)==n:
            return seq
seq=topo_seq()
earliest = [0] * n
for i in seq:
    for edge in G[i] :
        earliest[edge.e] = max(earliest[edge.e], earliest[i] + edge.w)
T = max(earliest)
latest = [T] * n
for j in seq[::-1]:
```

```
    for edge in H[j]:
        latest[edge.e] = min(latest[edge.e], latest[j] - edge.w)
event = []
for i in range(n):
    if earliest[i] == latest[i]:
        event.append(i)
event.sort()
print(T)
for i in event:
    G[i].sort()
    for edge in G[i]:
        if edge.e in event and abs(earliest[edge.e]-earliest[i]) == edge.w:
            print(i+1, edge.e+1)
```

```
import heapq
class Node:
    def __init__(self, weight, char=None):
        self.weight = weight
        self.char = char
        self.left = None
        self.right = None
    def __lt__(self, other):
        return self.weight<other.weight
def build_huffman_tree(characters):
    heap = []
    for char, weight in characters.items():
        heapq.heappush(heap, Node(weight, char))
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(left.weight + right.weight)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)
    return heap[0]
def build_code(root):
    codes={}
    def traverse(node,code):
        if node.char:
            codes[node.char]=code
        else:
            traverse(node.left,code+'0')
            traverse(node.right,code+'1')
    traverse(root,'')
    return codes
def encoding(codes,string):
    encoded=''
    for char in string:
        encoded+=codes[char]
       return encoded
def decoding(root,encoded_string):
    decoded=''
    node=root
    for bit in encoded_string:
        if bit==0:
```

```python
                node=node.left
            else:
                node=node.right
            if node.char:
                decoded+=node.char
                node=root
    return decoded
def external_path_length(node,depth=0):
    if node is None:
        return 0
    if node.left is None and node.right is None:
        return depth*node.weight
    return 
(external_path_length(node.left,depth+1)+external_path_length(node.right,dept
h+1))
n=int(input())
characters={}
lst=list(map(int,input().split()))
for i in range(len(lst)):
    characters[i]=lst[i]
root=build_huffman_tree(characters)
print(external_path_length(root))
```

```python
def buildtree(preorder,inorder):
    if not preorder or not inorder:
        return None
    root=Node(preorder[0])
    rootindex=inorder.index(root.val)
    root.left=buildtree(preorder[1:rootindex+1],inorder[:rootindex])
    root.right=buildtree(preorder[rootindex+1:],inorder[rootindex+1:])
    return root
def build(postorder,inorder):
    if not postorder or not inorder:
        return None
    root_val=postorder[-1]
    root=node(root_val)
    mid=inorder.index(root_val)
    root.left=build(postorder[:mid],inorder[:mid])
    root.right=build(postorder[mid:-1],inorder[mid+1:])
    return root
```

```python
class TrieNode:
    def __init__(self, char):
        self.char = char
        self.is_end = False
        self.children = {}
class Trie(object):
    def __init__(self):
        self.root = TrieNode("")
    def insert(self, word):
        node = self.root
        for char in word:
            if char in node.children:
                node = node.children[char]
```

```
            else:
                new_node = TrieNode(char)
                node.children[char] = new_node
                node = new_node
        node.is_end = True
    def dfs(self, node, pre):
        if node.is_end:
            self.output.append((pre + node.char))
        for child in node.children.values():
            self.dfs(child, pre + node.char)
    def search(self, x):
        node = self.root
        for char in x:
            if char in node.children:
                node = node.children[char]
            else:
                return []
        self.output = []
        self.dfs(node, x[:-1])
        return self.output
```

```
class BinaryTree:
    def __init__(self, rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None
    def insertLeft(self, newNode):
        if self.leftChild == None:
            self.leftChild = BinaryTree(newNode)
        else:  # 已经存在左子节点。此时，插入一个节点，并将已有的左子节点降一层。
            t = BinaryTree(newNode)
            t.leftChild = self.leftChild
            self.leftChild = t
    def insertRight(self, newNode):
        if self.rightChild == None:
            self.rightChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.rightChild = self.rightChild
            self.rightChild = t
    def getRightChild(self):
        return self.rightChild
    def getLeftChild(self):
        return self.leftChild
    def setRootVal(self, obj):
        self.key = obj
    def getRootVal(self):
        return self.key
    def traversal(self, method="preorder"):
        if method == "preorder":
            print(self.key, end=" ")
        if self.leftChild != None:
            self.leftChild.traversal(method)
        if method == "inorder":
            print(self.key, end=" ")
```

```
        if self.rightChild != None:
            self.rightChild.traversal(method)
        if method == "postorder":
            print(self.key, end=" ")
def buildParseTree(fpexp):
    fplist = fpexp.split()
    pStack = Stack()
    eTree = BinaryTree('')
    pStack.push(eTree)
    currentTree = eTree
    for i in fplist:
        if i == '(':
            currentTree.insertLeft('')
            pStack.push(currentTree)
            currentTree = currentTree.getLeftChild()
        elif i not in '+-*/)':
            currentTree.setRootVal(int(i))
            parent = pStack.pop()
            currentTree = parent
        elif i in '+-*/':
            currentTree.setRootVal(i)
            currentTree.insertRight('')
            pStack.push(currentTree)
            currentTree = currentTree.getRightChild()
        elif i == ')':
            currentTree = pStack.pop()
        else:
            raise ValueError("Unknown Operator: " + i)
    return eTree
exp = "( ( 7 + 3 ) * ( 5 - 2 ) )"
pt = buildParseTree(exp)
for mode in ["preorder", "postorder", "inorder"]:
    pt.traversal(mode)
    print()
"""
* + 7 3 - 5 2
7 3 + 5 2 - *
7 + 3 * 5 - 2
"""
import operator
def evaluate(parseTree):
    opers = {'+':operator.add, '-':operator.sub, '*':operator.mul,
'/':operator.truediv}
    leftC = parseTree.getLeftChild()
    rightC = parseTree.getRightChild()
    if leftC and rightC:
        fn = opers[parseTree.getRootVal()]
        return fn(evaluate(leftC),evaluate(rightC))
    else:
        return parseTree.getRootVal()
print(evaluate(pt))
# 30

#后序求值
def postordereval(tree):
    opers = {'+':operator.add, '-':operator.sub,
```

```python
                '*':operator.mul, '/':operator.truediv}
    res1 = None
    res2 = None
    if tree:
        res1 = postordereval(tree.getLeftChild())
        res2 = postordereval(tree.getRightChild())
        if res1 and res2:
            return opers[tree.getRootVal()](res1,res2)
        else:
            return tree.getRootVal()

print(postordereval(pt))
# 30

#中序还原完全括号表达式
def printexp(tree):
    sVal = ""
    if tree:
        sVal = '(' + printexp(tree.getLeftChild())
        sVal = sVal + str(tree.getRootVal())
        sVal = sVal + printexp(tree.getRightChild()) + ')'
    return sVal

print(printexp(pt))
# (((7)+3)*((5)-2))
```

```python
def merge_sort(lst):
    l=len(lst)
    if l<=1:
        return lst,0
    middle=l//2
    left=lst[:middle]
    right=lst[middle:]
    merged_left,left_inv=merge_sort(left)
    merged_right,right_inv=merge_sort(right)
    merged,merge_inv=merge(merged_left,merged_right)
    return merged,merge_inv+left_inv+right_inv
def merge(left,right):
    i=j=0
    merge_inv=0
    merged=[]
    while i<len(left) and j<len(right):
        if left[i]<=right[j]:
            merged.append(left[i])
            i+=1
        else:
            merged.append(right[j])
            j+=1
            merge_inv+=len(left)-i
    merged+=left[i:]
    merged+=right[j:]
    return merged,merge_inv
```

```python
def find_shortest_paths(maze, x, y, end, visited, path, shortest_paths):
```

```
    if (x, y) == end:
        shortest_paths.append(path)
        return
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < len(maze) and 0 <= ny < len(maze[0]) and maze[nx][ny] == '.' and (nx, ny) not in visited:
            visited.add((nx, ny))
            find_shortest_paths(maze, nx, ny, end, visited, path + [(nx, ny)], shortest_paths)
            visited.remove((nx, ny))
# 读取输入
n, m = map(int, input().split())
maze = [input() for _ in range(n)]
# 寻找入口和出口
start = (0, 0)
end = (n - 1, m - 1)
# 存储所有最短路径
shortest_paths = []
# 记录访问过的位置
visited = set([start])
# 当前路径
path = [start]
# 递归查找所有最短路径
find_shortest_paths(maze, start[0], start[1], end, visited, path, shortest_paths)
if shortest_paths:
    path=min(shortest_paths,key=len)
    for step in path:
        print(''.join(str(step).split()), end='')
else:
    print(0)
```

```
def infix_to_postfix(expression):
    precedence = {'+':1, '-':1, '*':2, '/':2}
    stack = []
    postfix = []
    number = ''
    for char in expression:
        if char.isnumeric() or char == '.':
            number += char
        else:
            if number:
                num = float(number)
                postfix.append(int(num) if num.is_integer() else num)
                number = ''
            if char in '+-*/':
                while stack and stack[-1] in '+-*/' and precedence[char] <= precedence[stack[-1]]:
                    postfix.append(stack.pop())
                stack.append(char)
            elif char == '(':
                stack.append(char)
            elif char == ')':
```

```
                while stack and stack[-1] != '(':
                    postfix.append(stack.pop())
                stack.pop()
    if number:
        num = float(number)
        postfix.append(int(num) if num.is_integer() else num)
    while stack:
        postfix.append(stack.pop())
    return ' '.join(str(x) for x in postfix)
n = int(input())
for _ in range(n):
    expression = input()
    print(infix_to_postfix(expression))
```

```
'''
分析过程:
(1)先考虑只有一个节点的情形,设此时的形态有f(1)种,那么很明显f(1)=1
(2)如果有两个节点呢?我们很自然想到,应该在f(1)的基础上考虑递推关系。那么,如果固定一个节点后,左右子树的分布情况为1=1+0=0+1,故有f(2) = f(1) + f(1)
(3)如果有三个节点,(我们需要考虑固定两个节点的情况么?当然不,因为当节点数量大于等于2时,无论你如何固定,其形态必然有多种)我们考虑固定一个节点,即根节点。好的,按照这个思路,还剩2个节点,那么左右子树的分布情况为2=2+0=1+1=0+2。
所以有3个节点时,递归形式为f(3)=f(2) + f(1)*f(1) + f(2)。(注意这里的乘法,因为左右子树一起组成整棵树,根据排列组合里面的乘法原理即可得出)
(4)那么有n个节点呢我们固定一个节点,那么左右子树的分布情况为n-1=n-1 + 0 = n-2 + 1 = … = 1 + n-2 = 0 + n-1。此时递归表达式为f(n) = f(n-1) + f(n-2)f(1) + f(n-3)f(2) + … + f(1)f(n-2) + f(n-1)
接下来我们定义没有节点的情况,此时也只有一种情况,即f(0)=1
那么则有:
f(0)=1,f(1)=1
f(2)=f(1)f(0)+f(0)f(1)
f(3)=f(2)f(0)+f(1)f(1)+f(0)f(2)
.
.
.
.
f(n)=f(n-1)f(0)+f(n-2)f(1)+……….+f(1)f(n-2)+f(0)f(n-1)
递推结果是卡特兰数,解见代码
'''
n=int(input())
from math import factorial
print(factorial(2*n)//(factorial(n)*factorial(n+1)))
```

```
def divide_k(n, k):
    # dp[i][j]为将i划分为j个正整数的划分方法数量
    dp = [[0]*(k+1) for _ in range(n+1)]
    for i in range(n+1):
        dp[i][1] = 1
    for i in range(1, n+1):
        for j in range(1, k+1):
            if i >= j:
                # dp[i-1][j-1]为包含1的划分的数量
                # 若不包含1，我们对每个数-1仍为正整数，划分数量为dp[i-j][j]
                dp[i][j] = dp[i-j][j]+dp[i-1][j-1]
```

```python
    return dp[n][k]


def divide_dif(n):
    # dp[i][j]表示将数字 i 划分，其中最大的数字不大于 j 的方法数量
    dp = [[0] * (n + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):
        for j in range(1, n + 1):
            # 比i大的数没用
            if i < j:
                dp[i][j] = dp[i][i]
            # 多了一种：不划分
            elif i == j:
                dp[i][j] = dp[i][j - 1] + 1
            # 用/不用j
            else:
                dp[i][j] = dp[i][j - 1] + dp[i - j][j - 1]
    return dp[n][n]


# 一个数的奇分拆总是等于互异分拆
```

```python
from collections import deque

def construct_graph(words):
    graph = {}
    for word in words:
        for i in range(len(word)):
            pattern = word[:i] + '*' + word[i + 1:]
            if pattern not in graph:
                graph[pattern] = []
            graph[pattern].append(word)
    return graph
def bfs(start, end, graph):
    queue = deque([(start, [start])])
    visited = set([start])

    while queue:
        word, path = queue.popleft()
        if word == end:
            return path
        for i in range(len(word)):
            pattern = word[:i] + '*' + word[i + 1:]
            if pattern in graph:
                neighbors = graph[pattern]
                for neighbor in neighbors:
                    if neighbor not in visited:
                        visited.add(neighbor)
                        queue.append((neighbor, path + [neighbor]))
    return None
def word_ladder(words, start, end):
    graph = construct_graph(words)
    return bfs(start, end, graph)
n = int(input())
words = [input().strip() for _ in range(n)]
```

```python
start, end = input().strip().split()
result = word_ladder(words, start, end)
if result:
    print(' '.join(result))
else:
    print("NO")
```

```python
#动态中位数
import heapq
def main():
    lst=list(map(int,input().split()))
    n=len(lst)
    ans=[]
    bigheap=[]
    smallheap=[]
    heapq.heapify(bigheap)
    heapq.heapify(smallheap)
    for i in range(n):
        if not smallheap or -smallheap[0]>=lst[i]:
            heapq.heappush(smallheap,-lst[i])
        else:
            heapq.heappush(bigheap,lst[i])
        if len(bigheap)>len(smallheap):
            heapq.heappush(smallheap,-heapq.heappop(bigheap))
        if len(smallheap)>len(bigheap)+1:
            heapq.heappush(bigheap,-heapq.heappop(smallheap))
        if i%2==0:
            ans.append(-smallheap[0])
    print(len(ans))
    print(' '.join(map(str,ans)))
t=int(input())
for i in range(t):
    main()
```