

# Cheat\_Sheet

2024 Summer Compiled by 武昱达

## 排序算法

归并排序（可用于求逆序对数）

```
1  # start--mid 和 mid+1--end 都是sorted list
2  def Merge(a,start,mid,end):
3      tmp=[]
4      l=start
5      r=mid+1
6      while l<=mid and r<=end:
7          if a[l]<=a[r]:
8              tmp.append(a[l])
9              l+=1
10         else:
11             tmp.append(a[r])
12             r+=1
13     # 以下至少有一个extend了空列表
14     tmp.extend(a[l:mid+1])
15     tmp.extend(a[r:end+1])
16     for i in range(start,end+1):
17         a[i]= tmp[i-start]
18
19     # 二分
20     def MergeSort(a,start,end):
21         if start==end:
22             return
23
24         mid=(start+end)//2
25         MergeSort(a,start,mid)
26         MergeSort(a,mid+1,end)
27         Merge(a,start,mid,end)
28
29     a=[8,5,6,4,3,7,10,2]
30     MergeSort(a,0,7)
31     print(a)
```

快速排序

```
1  def quicksort(arr, left, right):
2      if left < right:
3          partition_pos = partition(arr, left, right)
4          quicksort(arr, left, partition_pos - 1)
5          quicksort(arr, partition_pos + 1, right)
6
7
8  def partition(arr, left, right):
9      # 最右端元素作为基准元素，i,j是两个指针，通过两个元素的交换实现
10     # 基准元素左右分别小于、大于他本身。
11     i = left
```

```

12     j = right - 1
13     pivot = arr[right]
14     while i <= j:
15         while i <= right and arr[i] < pivot:
16             i += 1
17         while j >= left and arr[j] >= pivot:
18             j -= 1
19         if i < j:
20             arr[i], arr[j] = arr[j], arr[i]
21     if arr[i] > pivot:
22         arr[i], arr[right] = arr[right], arr[i]
23     return i
24
25
26 arr = [22, 11, 88, 66, 55, 77, 33, 44]
27 quicksort(arr, 0, len(arr) - 1)
28 print(arr)
29
30 # [11, 22, 33, 44, 55, 66, 77, 88]

```

## 单调栈

奶牛排队，寻找*i*右侧第一个小于*i*的索引

题目要求：N为数组长度，hi为数组元素，求最长满足条件子序列，子序列的左边界是该子序列的严格最小值，右边界是该子序列的严格最大值。

```

1  N,res=int(input()),0
2  hi=[int(input()) for _ in range(N)]
3  # left[i]是i左边第一个不小于他的元素的索引，right[i]是i右边第一个不大于他的元素的索引。
4  # 容易知道，对于指定的i，如果i作为右端点，left[i]是左端点的一个上界，反之同理。
5  left,right=[-1 for _ in range(N)],[N for _ in range(N)]
6  stack1,stack2=[],[]
7
8  for i in range(N-1,-1,-1):
9      while stack1 and hi[stack1[-1]]>hi[i]:
10         stack1.pop()
11     if stack1:right[i]=stack1[-1]
12     stack1.append(i)
13
14  for i in range(N):
15     while stack2 and hi[stack2[-1]]<hi[i]:
16         stack2.pop()
17     if stack2:left[i]=stack2[-1]
18     stack2.append(i)
19
20  for i in range(N):
21     for j in range(right[i]-1,i,-1):
22         if left[j]<i:
23             res=max(j-i+1,res)
24             break
25
26  print(res)

```

## 后序表达式求值

从左侧先后弹出两个数字a,b一个算符@，计算a@b，再放回左边。

```
1  样例输入
2      3
3      5 3.4 +
4      5 3.4 + 6 /
5      5 3.4 + 6 * 3 +
6  样例输出
7      8.40
8      1.40
9      53.40
```

```
1  def cal(a,b,operate):
2      if operate=="+":return a+b
3      if operate=="-":return a-b
4      if operate=="*":return a*b
5      if operate=="/":return a/b
6
7  from collections import deque
8  n,operators=int(input()),('+','-','*','/')
9  raw=[deque(map(str,input().split())) for _ in range(n)]
10
11 for deq in raw:
12     tmp_deq=deque()
13     while len(deq)>=1:
14         if deq[0] not in operators:
15             tmp_deq.append(float(deq.popleft()))
16         else:
17             b=tmp_deq.pop()
18             a=tmp_deq.pop()
19             operate=deq.popleft()
20             deq.appendleft(cal(a,b,operate))
21     print('{:.2f}'.format(tmp_deq[0]))
```

## 中缀转后缀

以下是 Shunting Yard 算法的基本步骤：

1. 初始化运算符栈和输出栈为空。
2. 从左到右遍历中缀表达式的每个符号。
  - 如果是操作数（数字），则将其添加到输出栈。
  - 如果是左括号，则将其推入运算符栈。
  - 如果是运算符：
    - 如果运算符的优先级大于运算符栈顶的运算符，或者运算符栈顶是左括号，则将当前运算符推入运算符栈。
    - 否则，将运算符栈顶的运算符弹出并添加到输出栈中，直到满足上述条件（或者运算符栈为空）。
    - 将当前运算符推入运算符栈。

- 如果是右括号，则将运算符栈顶的运算符弹出并添加到输出栈中，直到遇到左括号。将左括号弹出但不添加到输出栈中。
3. 如果还有剩余的运算符在运算符栈中，将它们依次弹出并添加到输出栈中。
  4. 输出栈中的元素就是转换后的后缀表达式。

```
1  样例输入
2      3
3      7+8.3
4      3+4.5*(7+2)
5      (3)*((3+4)*(2+3.5)/(4+5))
6  样例输出
7      7 8.3 +
8      3 4.5 7 2 + * +
9      3 3 4 + 2 3.5 + * 4 5 + / *
```

```
1  def infix_to_postfix(expression):
2      def get_precedence(op):
3          precedences = {'+': 1, '-': 1, '*': 2, '/': 2}
4          return precedences[op] if op in precedences else 0
5
6      def is_operator(c):
7          return c in "+-*/"
8
9      def is_number(c):
10         return c.isdigit() or c == '.'
11
12     output = []
13     stack = []
14     number_buffer = []
15
16     def flush_number_buffer():
17         if number_buffer:
18             output.append(''.join(number_buffer))
19             number_buffer.clear()
20
21     # 主体部分
22     for c in expression:
23         if is_number(c):
24             number_buffer.append(c)
25         elif c == '(':
26             flush_number_buffer()
27             stack.append(c)
28         elif c == ')':
29             flush_number_buffer()
30             while stack and stack[-1] != '(':
31                 output.append(stack.pop())
32             stack.pop() # popping '('
33         elif is_operator(c):
34             flush_number_buffer()
35             while stack and get_precedence(c) <= get_precedence(stack[-1]):
36                 output.append(stack.pop())
37             stack.append(c)
38
39     flush_number_buffer()
```

```

40     while stack:
41         output.append(stack.pop())
42
43     return ' '.join(output)
44
45 # Read number of expressions
46 n = int(input())
47 # Read each expression and convert it
48 for _ in range(n):
49     infix_expr = input()
50     postfix_expr = infix_to_postfix(infix_expr)
51     print(postfix_expr)

```

## Shunting Yard算法

中缀转后缀

```

1  operators=['+', '-', '*', '/']
2  cals=['(', ')']
3  # 预处理数据的部分已省略。
4  def pre_to_post(lst):
5      s_op, s_out=[], []
6      while lst:
7          tmp=lst.pop(0)
8          if tmp not in operators and tmp not in cals:
9              s_out.append(tmp)
10             continue
11
12         if tmp=="(":
13             s_op.append(tmp)
14             continue
15
16         if tmp==")":
17             while (a:=s_op.pop())!="(":
18                 s_out.append(a)
19
20         if tmp in operators:
21             if not s_op:
22                 s_op.append(tmp)
23                 continue
24             if is_prior(tmp, s_op[-1]) or s_op[-1]=="(":
25                 s_op.append(tmp)
26                 continue
27             while (not (is_prior(tmp, s_op[-1]) or s_op[-1]=="(")
28                    or not s_op):
29                 s_out.append(s_op.pop())
30             s_op.append(tmp)
31             continue
32
33     while len(s_op)!=0:
34         tmp=s_op.pop()
35         if tmp in operators:
36             s_out.append(tmp)
37
38     return " ".join(s_out)

```

```

39
40 def is_prior(A,B):
41     if (A=="*" or A=="/") and (B=="+" or B=="-"):
42         return True
43     return False
44
45 def input_to_lst(x):
46     tmp=list(x)
47
48 for i in range(int(input())):
49     print(pre_to_post(expProcessor(input())))

```

## 建树

```

1 class TreeNode:
2     def __init__(self,val):
3         self.val=val
4         self.left=None
5         self.right=None

```

## Huffman算法

哈夫曼编码树

- 描述

构造一个具有n个外部节点的扩充二叉树，每个外部节点 $K_i$ 有一个 $W_i$ 对应，作为该外部节点的权。使得这个扩充二叉树的叶节点带权外部路径长度总和最小：

$$\text{Min}(W_1 * L_1 + W_2 * L_2 + W_3 * L_3 + \dots + W_n * L_n)$$

$W_i$ :每个节点的权值。 $L_i$ :根节点到第 $i$ 个外部叶子节点的距离。编程计算最小外部路径长度总和。

- 输入

第一行输入一个整数n，外部节点的个数。第二行输入n个整数，代表各个外部节点的权值。

$2 \leq N \leq 100$

- 输出

输出最小外部路径长度总和。

- 样例输入

```
4 1 1 3 5
```

- 样例输出

```
17
```

```

1 import heapq
2 class HuffmanTreeNode:
3     def __init__(self,weight,char=None):
4         self.weight=weight
5         self.char=char
6         self.left=None
7         self.right=None
8
9     def __lt__(self,other):
10         return self.weight<other.weight

```

```

11
12 def BuildHuffmanTree(characters):
13     heap=[HuffmanTreeNode(weight,char) for char,weight in
characters.items()]
14     heapq.heapify(heap)
15     while len(heap)>1:
16         left=heapq.heappop(heap)
17         right=heapq.heappop(heap)
18         merged=HuffmanTreeNode(left.weight+right.weight,None)
19         merged.left=left
20         merged.right=right
21         heapq.heappush(heap,merged)
22     root=heapq.heappop(heap)
23     return root
24
25 def enpaths_huffman_tree(root):
26     # 字典形如(idx,weight):path
27     paths={}
28     def traverse(node,path):
29         if node.char:
30             paths[(node.char,node.weight)]=path
31         else:
32             traverse(node.left,path+1)
33             traverse(node.right,path+1)
34     traverse(root,0)
35     return paths
36
37 def min_weighted_path(paths):
38     return sum(tup[1]*path for tup,path in paths.items())
39
40 n,characters=int(input()),{}
41 raw=list(map(int,input().split()))
42 for char,weight in enumerate(raw):
43     characters[str(char)]=weight
44 root=BuildHuffmanTree(characters)
45 paths=enpaths_huffman_tree(root)
46 print(min_weighted_path(paths))

```

## 并查集

发现它，抓住它

```

1 class UnionFind:
2     def __init__(self, n):
3         self.parent = list(range(n))
4         self.rank = [0] * n
5
6     def find(self, x):
7         if self.parent[x] != x:
8             self.parent[x] = self.find(self.parent[x])
9         return self.parent[x]
10
11     def union(self, x, y):

```

```

12     rootX = self.find(x)
13     rootY = self.find(y)
14     if rootX != rootY:
15         if self.rank[rootX] > self.rank[rootY]:
16             self.parent[rootY] = rootX
17         elif self.rank[rootX] < self.rank[rootY]:
18             self.parent[rootX] = rootY
19         else:
20             self.parent[rootY] = rootX
21             self.rank[rootX] += 1
22
23 def solve():
24     n, m = map(int, input().split())
25     uf = UnionFind(2 * n) # 初始化并查集，每个案件对应两个节点，一个是本身，另一个是其
对立案件。
26     for _ in range(m):
27         operation, a, b = input().split()
28         a, b = int(a) - 1, int(b) - 1
29         if operation == "D":
30             uf.union(a, b + n) # a与b的对立案件合并
31             uf.union(a + n, b) # a的对立案件与b合并
32         else: # "A"
33             if uf.find(a) == uf.find(b) or uf.find(a + n) == uf.find(b + n):
34                 print("In the same gang.")
35             elif uf.find(a) == uf.find(b + n) or uf.find(a + n) ==
uf.find(b):
36                 print("In different gangs.")
37             else:
38                 print("Not sure yet.")
39
40 T = int(input())
41 for _ in range(T):
42     solve()

```

## 食物链

```

1 class DisjointSet:
2     def __init__(self, n):
3         # 设[1,n] 区间表示同类, [n+1,2*n]表示x吃的动物, [2*n+1,3*n]表示吃x的动物。
4         self.parent = [i for i in range(3 * n + 1)] # 每个动物有三种可能的类型,
用 3 * n 来表示每种类型的并查集
5         self.rank = [0] * (3 * n + 1)
6
7     def find(self, u):
8         if self.parent[u] != u:
9             self.parent[u] = self.find(self.parent[u])
10        return self.parent[u]
11
12    def union(self, u, v):
13        pu, pv = self.find(u), self.find(v)
14        if pu == pv:
15            return False
16        if self.rank[pu] > self.rank[pv]:

```



```

17         self.parent[pv] = pu
18     elif self.rank[pu] < self.rank[pv]:
19         self.parent[pu] = pv
20     else:
21         self.parent[pv] = pu
22         self.rank[pu] += 1
23     return True
24
25
26 def is_valid(n, statements):
27     dsu = DisjointSet(n)
28
29     false_count = 0
30     for d, x, y in statements:
31         if x > n or y > n:
32             false_count += 1
33             continue
34
35         if d == 1: # 同类
36             if dsu.find(x) == dsu.find(y + n) or dsu.find(x) == dsu.find(y + 2 * n):
37                 # 不是同类
38                 false_count += 1
39             else:
40                 dsu.union(x, y)
41                 dsu.union(x + n, y + n)
42                 dsu.union(x + 2 * n, y + 2 * n)
43
44         else: # x吃y
45             if dsu.find(x) == dsu.find(y) or dsu.find(x + 2 * n) ==
46                 dsu.find(y):
47                 false_count += 1
48             else: # [1, n] 区间表示同类, [n+1, 2*n]表示x吃的动物, [2*n+1, 3*n]表示吃x的
49                 # 动物
50                 dsu.union(x + n, y)
51                 dsu.union(x, y + 2 * n)
52                 dsu.union(x + 2 * n, y + n)
53
54     return false_count
55
56
57 if __name__ == "__main__":
58     N, K = map(int, input().split())
59     statements = []
60     for _ in range(K):
61         D, X, Y = map(int, input().split())
62         statements.append((D, X, Y))
63     result = is_valid(N, statements)
64     print(result)

```

## Prim算法

步骤:

1. 起点入堆。

2. 堆顶元素出堆（排序依据是到该元素的开销），如已访问过，continue；否则标记为visited。
3. 访问该节点相邻节点，（访问开销（排序依据），相邻节点）入堆。
4. 相邻节点前驱设置为当前节点（如需）。
5. 当前节点入树

**全部精要在于：每次走出下一步的开销都是当前最小的。**

Agri-net

题目：用邻接矩阵给出图，求最小生成树路径权值和。

```

1 4
2 0 4 9 21
3 4 0 8 17
4 9 8 0 16
5 21 17 16 0
6      # 注意这一步continue很关键，因为一个节点会同时很多存在于pq中（这是由出队标记决定的）
7      # 如果不设计这一步continue，则会重复加路径长。

```

```

1 from heapq import heappop, heappush
2 def prim(matrix):
3     ans=0
4     pq,visited=[(0,0)], [False for _ in range(N)]
5     while pq:
6         c,cur=heappop(pq)
7         if visited[cur]:continue
8         visited[cur]=True
9         ans+=c
10        for i in range(N):
11            if not visited[i] and matrix[cur][i]!=0:
12                heappush(pq,(matrix[cur][i],i))
13    return ans
14
15 while True:
16     try:
17         N=int(input())
18         matrix=[list(map(int,input().split())) for _ in range(N)]
19         print(prim(matrix))
20     except:break

```

## Kruskal算法（能写Prim建议写Prim）

Agri-net

```

1 class DisJointSet:
2     def __init__(self,num_vertices):
3         self.parent=list(range(num_vertices))
4         self.rank=[0 for _ in range(num_vertices)]
5
6     def find(self,x):
7         if self.parent[x]!=x:
8             self.parent[x] = self.find(self.parent[x])

```

```

9         return self.parent[x]
10
11     def union(self,x,y):
12         root_x=self.find(x)
13         root_y=self.find(y)
14         if root_x!=root_y:
15             if self.rank[root_x]<self.rank[root_y]:
16                 self.parent[root_x]=root_y
17             elif self.rank[root_x]>self.rank[root_y]:
18                 self.parent[root_y]=root_x
19             else:
20                 self.parent[root_x]=root_y
21                 self.rank[root_y]+=1
22
23     # graph是邻接表
24     def kruskal(graph:list):
25         res,edges,dsj=[],[],DisJointSet(len(graph))
26         for i in range(len(graph)):
27             for j in range(i+1,len(graph)):
28                 if graph[i][j]!=0:
29                     edges.append((i,j,graph[i][j]))
30
31         for i in sorted(edges,key=lambda x:x[2]):
32             u,v,weight=i
33             if dsj.find(u)!=dsj.find(v):
34                 dsj.union(u,v)
35                 res.append((u,v,weight))
36         return res
37
38     while True:
39         try:
40             n=int(input())
41             graph=[list(map(int,input().split())) for _ in range(n)]
42             res=kruskal(graph)
43             print(sum(i[2] for i in res))
44         except EOFError:break

```

## Kahn算法

Kahn算法的基本思想是通过不断地移除图中的入度为0的顶点，并将其添加到拓扑排序的结果中，直到图中所有的顶点都被移除。具体步骤如下：

1. 初始化一个队列，用于存储当前入度为0的顶点。
2. 遍历图中的所有顶点，计算每个顶点的入度，并将入度为0的顶点加入到队列中。
3. 不断地从队列中弹出顶点，并将其加入到拓扑排序的结果中。同时，遍历该顶点的邻居，并将其入度减1。如果某个邻居的入度减为0，则将其加入到队列中。
4. 重复步骤3，直到队列为空。

**Kahn算法的时间复杂度为 $O(V + E)$** ，其中 $V$ 是顶点数， $E$ 是边数。它是一种简单而高效的拓扑排序算法，在有向无环图（DAG）中广泛应用。

题目：给出一个图的结构，输出其拓扑排序序列，要求在同等条件下，编号小的顶点在前。

题解中graph是邻接表，形如graph[1]=[2,3,4]，由于本题要求顺序，因此不用队列而用优先队列。

```
1 from collections import defaultdict
2 from heapq import heappush,heappop
3 def Kahn(graph):
4     q,ans=[],[]
5     in_degree=defaultdict(int)
6     for lst in graph.values():
7         for vert in lst:
8             in_degree[vert]+=1
9
10    for vert in graph.keys():
11        if vert not in in_degree or in_degree[vert]==0:
12            heappush(q,vert)
13
14    while q:
15        vertex=heappop(q)
16        ans.append('v'+str(vertex))
17        for neighbor in graph[vertex]:
18            in_degree[neighbor]-=1
19            if in_degree[neighbor]==0:
20                heappush(q,neighbor)
21    return ans
22
23 v,a=map(int,input().split())
24 graph={}
25 for _ in range(a):
26     f,t=map(int,input().split())
27     if f not in graph:graph[f]=[]
28     if t not in graph:graph[t]=[]
29     graph[f].append(t)
30
31 for i in range(1,v+1):
32     if i not in graph:graph[i]=[]
33
34 res=Kahn(graph)
35 print(*res)
```

## Dijkstra算法

道路（更推荐第二种剪枝写法）

N个以1...N标号的城市通过单向的道路相连。每条道路包含两个参数：道路的长度和需要为该路付的通行费（以金币的数目来表示）。Bob从1到N。他希望能够尽可能快的到那，但是他囊中羞涩。我们希望能够帮助Bob找到从1到N最短的路径，前提是他能够付得起通行费。输出结果应该只包括一行，即从城市1到城市N所需要的最小的路径长度（花费不能超过K个金币）。如果这样的路径不存在，结果应该输出-1。

S: 起点; D: 终点; L: 道路长; T: 通行费。

```
1 from heapq import heappop,heappush
```

```

2 from collections import defaultdict
3 K,N,R=int(input()),int(input()),int(input())
4 graph=defaultdict(list)
5 for i in range(R):
6     S,D,L,T=map(int,input().split())
7     graph[S].append((D,L,T))
8 def Dijkstra(graph):
9     global K,N,R
10    q,ans=[],[]
11    heappush(q,(0,0,1,0))
12    while q:
13        l,cost,cur,step=heappop(q)
14        if cur==N:return l
15        for next,nl,nc in graph[cur]:
16            # 剪枝: 如果步数不少于N: 意味着一定走了回头路, 减掉。
17            if cost+nc<=K and step+1<N:
18                heappush(q,(l+nl,cost+nc,next,step+1))
19    return -1
20 print(Dijkstra(graph))

```

```

1 from heapq import heappop,heappush
2 from collections import defaultdict
3 K,N,R=int(input()),int(input()),int(input())
4 graph=defaultdict(list)
5 for i in range(R):
6     S,D,L,T=map(int,input().split())
7     graph[S].append((D,L,T))
8
9 def Dijkstra(graph):
10    global K,N,R
11    q,ans=[],[]
12    min_cost={i:float('inf') for i in range(1,N+1)}
13    heappush(q,(0,0,1))
14    while q:
15        l,cost,cur=heappop(q)
16        min_cost[cur]=min(min_cost[cur],cost)
17        if cur==N:return l
18        for next,nl,nc in graph[cur]:
19            # 剪枝1: 只有花费小于等于K才能入堆。
20            # 剪枝2: 只有到达下一个节点的花费比上次更小时才能入堆 (否则路程长花费大, 无意义)。
21            if cost+nc<=K and nc+cost<min_cost[next]:
22                heappush(q,(l+nl,cost+nc,next))
23    return -1
24 print(Dijkstra(graph))

```

## Kosaraju算法

```

1 def dfs1(graph, node, visited, stack):
2     visited[node] = True
3     for neighbor in graph[node]:
4         if not visited[neighbor]:

```

```

5         dfs1(graph, neighbor, visited, stack)
6     stack.append(node)
7
8     def dfs2(graph, node, visited, component):
9         visited[node] = True
10        component.append(node)
11        for neighbor in graph[node]:
12            if not visited[neighbor]:
13                dfs2(graph, neighbor, visited, component)
14
15    def kosaraju(graph):
16        # Step 1: Perform first DFS to get finishing times
17        stack = []
18        visited = [False] * len(graph)
19        for node in range(len(graph)):
20            if not visited[node]:
21                dfs1(graph, node, visited, stack)
22
23        # Step 2: Transpose the graph
24        transposed_graph = [[] for _ in range(len(graph))]
25        for node in range(len(graph)):
26            for neighbor in graph[node]:
27                transposed_graph[neighbor].append(node)
28
29        # Step 3: Perform second DFS on the transposed graph to find SCCs
30        visited = [False] * len(graph)
31        sccs = []
32        while stack:
33            node = stack.pop()
34            if not visited[node]:
35                scc = []
36                dfs2(transposed_graph, node, visited, scc)
37                sccs.append(scc)
38        return sccs
39
40    # Example
41    graph = [[1], [2, 4], [3, 5], [0, 6], [5], [4], [7], [5, 6]]
42    sccs = kosaraju(graph)
43    print("Strongly Connected Components:")
44    for scc in sccs:
45        print(scc)
46
47    """
48    Strongly Connected Components:
49    [0, 3, 2, 1]
50    [6, 7]
51    [5, 4]
52
53    """

```

## 无向图判断连通和成环

判断无向图是否连通有无回路

```
1 from collections import defaultdict, deque
```

```

2  # graph是邻接表{1:[2,3,4]}
3  def is_connected(graph,n):
4      dq=deque()
5      dq.append(0)
6      visited=set()
7      visited.add(0)
8      while dq:
9          cur_vert=dq.popleft()
10         for next_vert in graph[cur_vert]:
11             if next_vert not in visited:
12                 dq.append(next_vert)
13                 visited.add(next_vert)
14     return len(visited)==n
15
16 def is_loop(graph):
17     global_visited=set()
18     for vertex in graph:
19         if vertex not in global_visited:
20             # 以下是一个BFS函数。
21             local_visited={}
22             dq=deque()
23             dq.append((vertex,0))
24             local_visited[vertex]=0
25             global_visited.add(vertex)
26             while dq:
27                 cur_vert,steps=dq.popleft()
28                 for next_vert in graph[cur_vert]:
29                     if next_vert in local_visited:
30                         if local_visited[next_vert]>=steps:
31                             return True
32                     else:
33                         dq.append((next_vert,steps+1))
34                         local_visited[next_vert]=steps+1
35                         global_visited.add(next_vert)
36     return False
37
38 n,m=map(int,input().split())
39 graph=defaultdict(list)
40 for _ in range(m):
41     a,b=map(int,input().split())
42     graph[a].append(b)
43     graph[b].append(a)
44 print('connected:yes' if is_connected(graph,n) else 'connected:no')
45 print('loop:yes' if is_loop(graph) else 'loop:no')

```

## 二分查找算法

月度开销

```

1  n,m=map(int,input().split())
2  expend=[int(input()) for i in range(n)]
3  def check(x):
4      # 判断x作为最大月度开销是否可以实现，如果可以实现，则说明不够小或刚好符合题意。
5      # 看m个方案是否可行。
6      nums,s=1,0

```

```
7     for i in range(n):
8         if expend[i]+s>x:
9             s=expend[i]
10            nums+=1      # 求和大于设定的最大月度开销，则应该插入挡板，份数+1
11        else:s+=expend[i]
12        return nums>m    # if nums>m return True,else return False
13
14    lo,hi,res=max(expend),sum(expend)+1,1
15    while lo<hi:
16        mid=(lo+hi)//2
17        if check(mid):lo=mid+1
18        else:res,hi=mid,mid
19
20    print(res)
```

## 其他注意事项 (Debug)

1. 写Dijkstra采用“出堆标记”是肯定正确的，尽管入堆标记可能更快。
2. 抄代码的时候注意缩进。
3. 注意要把题目数据全部接收，即使程序进行一半时已经得出结果。
4. 注意字符串和int类型，尤其不要犯'1'==1这种错误。
5. 注意对于某一个类的实例，不要重复定义。