DSAA模板

Basic opeartion

```
sort :
C++  sort(begin,end,cmp)
//end不参与排序
java  Arrays.sort(int[] arr,int Indexfrom,int Indexto,Comparator<? super T> c);
//Indexto不参与排序
```

```
comparator :
C++
bool cmp (int a,int b)
{
    return a>b;  从大到小
}
java
//自定义排序，必须使用Integer
Integer[] arr= {1,2,3,6,7,13,4,6,23,12,312,153,718};
Arrays.sort(arr,new Comparator<Integer>() {
    @Override //-1在前，1在后，0默认
    public int compare(Integer o1, Integer o2) {
        if(o1>o2) return -1;
        else if(o1<o2) return 1;
        return 0;
    }
});
```

```
data structure :
C++
#include<bits/stdc++.h>          stack,queue,vector,list,set,priority_queue,pair,map
java
233
```

```
next_permutation
C++
sort(); do{ }while(next_permutation(begin,end)) //end不参与排序
java

```

```
C++ set:
set集合是c++ stl库中自带的一个容器，set具有以下两个特点:
1、set中的元素都是排好序的
2、set集合中没有重复的元素
常用操作:
begin()       返回set容器的第一个元素的地址
end()           返回set容器的最后一个元素地址
clear()       删除set容器中的所有的元素
empty()         判断set容器是否为空
max_size()     返回set容器可能包含的元素最大个数
size()           返回当前set容器中的元素个数
erase(it) 删除迭代器指针it处元素
insert(a) 插入某个元素
当set集合中的元素为结构体时，该结构体必须实现运算符‘<’的重载
//Example:
struct People{
    string name;
    int age;
    bool operator <(const People p) const  //运算符重载 {
        return age<p.age;       //按照年龄由小到大进行排序
    }
};
int main()
{
    set<People>s;
    s.insert((People){"张三",14});
    s.insert((People){"李四",16});
    s.insert((People){"王二麻子",10});
    set<People>::iterator it;
    for(it=s.begin();it!=s.end();it++)  //使用迭代器进行遍历
    {
        printf("姓名: %s 年龄: %d\n",(*it).name.c_str(),(*it).age);
    }
    return 0;
}
```

```
思维拓展:
暴力 预处理 二分答案 差分数组 离线.......
```

Template

Chapter 1、Graph:

Struct Edge

```cpp
using std::vector;
struct edge{
    bool operator <(const edge& edg) const{
        return dis < edg.dis;
    }
    int from=0,to=0,dis=0;
    edge(int from,int to,int dis){
        this->dis=dis;
        this->from=from;
        this->to=to;
    };
};
vector<edge> vecto;

vecto.clear();

for(int p=0;p<m;p++){
    int x, y, w;
    scanf("%d%d%d", &x, &y, &w);
    //取决于图有向无向
    vecto.push_back(edge(x, y, w));
    vecto.push_back(edge(y, x, w));
}

for (auto len : vecto) {}
```

Kruskal

```cpp
int father[maxn];
int find(int x);
void unite(int x,int y);
void ini();

int find(int x){
    if(x==father[x]) return x;
    return father[x] = find(father[x]);
}
void unite(int x,int y){
    int l=find(x);
    int r=find(y);
    father[r] = l;
}
void ini(){
    for (int i = 0; i < maxn;i++){
            father[i] = i;
    }
}

//每次使用kruskal之前要确保边集合是升序的
std::sort(vecto.begin(), vecto.end());

```

```
int kruskal (vector<edge> vecto){
    int ans=0;
    for (auto len : vecto){
        int lll=find(len.from);
        int rrr=find(len.to);
        if(lll!=rrr){
            unite(len.from, len.to);
            ans +=len.dis;
        }
    }
    return ans;
}

```

前向星

```
int cnt, head[maxn];
struct node
{
    int to, next; ll w;
    bool operator < (const node& a) const {
        return w < a.w;
    }
} edge[maxn of edge];

void addEdge(int u, int v, ll w) {
    edge[cnt] = (node){v,head[u],w};
    head[u] = cnt++;
}

void init() {
    cnt = 0;
    memset(head, -1, sizeof(head));
}

//遍历以x的出边edge[i]
for (int i = head[x]; i; i = edge[i].nxt) {
     node k = edge[i];
}

```

DijKstra

```
//天然获得的就是单源最短路排序后的结果

//                                  C++

```

```cpp
#include <bits/stdc++.h>
using namespace std;
/*
 * 使用优先队列优化Dijkstra算法
 * 复杂度O(ElogE)
 * 注意对vector<Edge>E[MAXN]进行初始化后加边
 */
const int INF=0x3f3f3f3f;
const int MAXN=1000010;
struct qnode{
    int v;
    int c;
    qnode(int _v=0,int _c=0):v(_v),c(_c){}
    bool operator <(const qnode &r)const {
        return c>r.c;
    }
};
struct Edge{
    int v,cost;
    Edge(int _v=0,int _cost=0):v(_v),cost(_cost){}
};
vector<Edge>E[MAXN];
bool vis[MAXN];
int dist[MAXN];
int par[MAXN];
void Dijkstra(int n,int start)//点的编号从1开始 {
    memset(vis,false,sizeof(vis));
    memset(par,0,sizeof(par));
    for(int i=1;i<=n;i++)dist[i]=INF;
    priority_queue<qnode>que;
    while(!que.empty())que.pop();
    dist[start]=0;
    que.push(qnode(start,0));
    qnode tmp;
    while(!que.empty())
    {
        tmp=que.top();
        que.pop();
        int u=tmp.v;
        if(vis[u])continue;
        vis[u]=true;
        for(int i=0;i<E[u].size();i++)  {
            int v=E[tmp.v][i].v;
            int cost=E[u][i].cost;
            if(!vis[v]&&dist[v]>dist[u]+cost)  {
                dist[v]=dist[u]+cost;
                que.push(qnode(v,dist[v]));
            }
        }
    }
}
void addedge(int u,int v,int w){
    E[u].push_back(Edge(v,w));
}
int main(){
    int n,m;
    int T;
```

```
    scanf("%d",&T);
    while(T--){
        scanf("%d%d",&n,&m);
        for(int i=1;i<=n;i++)E[i].clear();//初始化
        for(int i=0;i<m;i++) {
            int u,v,w;
            scanf("%d%d%d",&u,&v,&w);
            addedge(u,v,w);
            //addedge(v,u,w);无向图
        }
        Dijkstra(n,1);
        //单源最短路，dist[i]为从源点start到i的最短路
    }
    return 0;
}

```

```
//                                Java    包含路径
//n是点数 s是源 gra[j]是起点为j的邻接链表 返回一个长度为n+1的dis数组
    static int[] dijkstra(int n,int s,ArrayList<Node>[] gra){
        PriorityQueue<Node> minheap=new PriorityQueue();
        int[] vis=new int[n+1];
        int[] dis=new int[n+1];
        int[] par=new int[n+1];
        //String path[] = new String[n+1];//存放从start点到各点的最短路径的字符串表示
        for(int i=1;i<n+1;i++) {
            dis[i]=INF;
            par[i]=-1;
             path[i] = start+"->"+i;
        }
        dis[s]=0;
        minheap.add(new Node(s,0));
        while(!minheap.isEmpty()) {
            Node tmp=minheap.poll();
            int u=tmp.value;
            if(vis[u]==1) continue;
            vis[u]=1;
            for(Node a:gra[u]) {
                int v=a.value;
                int cost=a.wei;
                if(dis[u]+cost<dis[v]&&vis[v]==0) {
                    dis[v]=dis[u]+cost;
                    par[v]=u;
                    minheap.add(new Node(v,dis[v]));
                     //path[v] = path[u]+"->"+v;
                }

            }
        }
        return dis;
    }

    public static class Node implements Comparable<Node>{
        int value;

        int wei;
```

```
        Node next;
        public Node(int value,int wei) {
            this.value=value;
            this.wei=wei;
        }
        @Override
        public int compareTo(Node arg0) {
            // TODO Auto-generated method stub
            int num=(int)(this.wei-arg0.wei);
            return num;
        }
    }


}

```

Topological sort

```
//                                  Java
//按字典序的拓扑 可以用来判断DAG
//利用优先队列
int count=0;
for(int j=1;j<n+1;j++){
   if(vis[j]==0 && degree[j]==0){
   PriorityQueue<Integer> minheap=new PriorityQueue();
   minheap.add(j);
   vis[j]=1;
   int temp;
      while(!minheap.isEmpty()) {
         temp=minheap.poll();
         ans[count]=temp;
         count++;
         for(Integer k:arr[temp]) {
             degree[k]--;
             if(degree[k]==0&&vis[k]==0) {
                 minheap.add(k);
                 vis[k]=1;
             }
      }}}
}

//
if(count==n) {
    //说明可以进行topo，即这是一个有向无环图
    for(int j=0;j<n;j++){
        out.print(ans[j]+" "); //输出拓扑序
    }

}
else         //有环

```

```cpp
//                                  C++
int inDegree[nmax];//存每个节点的入度
vector<pair<int,int> >G[nmax];//邻接表 1st存节点，2nd存边权
int n,m;//点数，边数

void toposort(){
    queue<int>q;
    while(!q.empty()) q.pop();//清空队列
    for(int i=1;i<=n;i++){//顶点编号：1~n
        if(inDegree[i]==0){
            q.push(i);//把入度为0的顶点扔进队列
        }
    }
    while(!q.empty()){
        int u=q.front();
        q.pop();//删除该点
        for(int i=0;i<G[u].size();i++){//遍历与点u相邻的所有边，删除之。
            int v=G[u][i].first;
            inDegree[v]--;
            if(inDegree[v]==0){
                q.push(v);//队列里放点
            }
            //此处为拓扑序相应操作
        }
        //G[u].clear();
    }
}
```

BFS

```cpp
//解释：以s为起点的BFS count为visit到的个数，visit 0没扫到 1在队列内 2已操作完出队
//利用队列
int[] vis=new int[n+1];
Queue<Integer> queue=new LinkedList<Integer>();
queue.add(s);
//int[] dis=new int[n+1];//最短路
//String path[] = new String[n+1];//最短路径的字符串表示
//int[] par=new int[n+1];//BFS树
int count=1;
//for(int i=0;i<=n;i++) {
//    dis[i]=-1;    //-1代表不可达
//    path[i] = s+"->"+i;
//}
//dis[s]=0;
vis[s]=1;
O:while(!queue.isEmpty()) {
   int tmp=queue.poll();
   for(int k=head[tmp];k!=0;k=edge[k].next) {
```

```
        int f=edge[k].to;
        if(vis[f]==0) {
            queue.add(f);vis[f]=1;
            //dis[f]=dis[tmp]+1;
            //path[f]=path[tmp]+"->"+f;
            //par[f]=tmp;
            count++;
        }
    }
    vis[tmp]=2;
}
```

DFS

```
//利用递归
void dfs(int v){
    vis[v] = true;
    for(int i = 0; i < G[v].size(); i++){
        if(vis[G[v][i]] == false){
            dfs(G[v][i]); // 如果该顶点未访问，深度遍历之
        }
    }
}
//利用栈

```

Longest path

```

```

Tarjan

```
#include<stdio.h>
#include<string.h>
#include<vector>
#include<algorithm>
using namespace std;
#define maxn 1000000
vector<int >vec[maxn];
int ans[maxn];
int vis[maxn];
int dfn[maxn];
int low[maxn];
int n,m,tt,cnt,sig;
```

```

void init(){
    memset(low,0,sizeof(low));
    memset(dfn,0,sizeof(dfn));
    memset(vis,0,sizeof(vis));
    for(int i=1; i<=n; i++)vec[i].clear();
}

void Tarjan(int u){
    vis[u]=1;
    low[u]=dfn[u]=cnt++;
    for(int i=0; i<vec[u].size(); i++){
        int v=vec[u][i];
        if(vis[v]==0)Tarjan(v);
        if(vis[v]==1)low[u]=min(low[u],low[v]);
    }
    if(dfn[u]==low[u]){
        sig++;
    }
}

void Slove(){
    tt=-1;
    cnt=1;
    sig=0;
    for(int i=1; i<=n; i++){
        if(vis[i]==0){
            Tarjan(i);
        }
    }
    printf("%d\n",sig);
}

int main(){
    while(~scanf("%d",&n)){
        if(n==0)break;
        scanf("%d",&m);
        init();
        for(int i=0; i<m; i++) {
            int x,y;
            scanf("%d%d",&x,&y);
            vec[x].push_back(y);
        }
        Slove();
    }
}

```

Chapter2、Binary Search

```

//Binary Search  离散点二分查找（不一定为整数）
while (left <= right) {

```

```
    mid = (left + right) / 2;
    if (  ) {
    //操作
    break;
    }
    else if () left = mid + 1;
    else right = mid -1;
}
//Binary Find maximum 二分求极大值            Binary Find minimum 二分求极小值
while(l<=r){                                 while(l<=r){
    mid=(r+l)<<1;                                mid=(r+l)<<1;
    if(!check()) r=mid-1;                        if(check()) r=mid-1;
    else        l=mid+1;                         else        l=mid+1;
}return l-1;                                 }return l;
```

```
//连续点的Binary Search
1.使用答案误差 eps 转化成离散点:       left=mid+eps;  right=mid-eps;
2.使用函数误差 将while设为:           f (rignt) - f (left) < eps;
3.约束循环次数
```

Chapter3、String

KMP

```
//next[n] 为前n位的公共前后缀长(不包括自身)
public static int[] getNext(String s) {
        char[] c = s.toCharArray();
        int len = s.length();
        int[] next = new int[len + 1];
        int i = 0, j = -1;
        next[i] = j;
        while (i < len) {
            if (j == -1 || c[i] == c[j]) {
                i++;
                j++;
                //if(i<len&&c[i]!=c[j]) next[i] = j;  优化KMP，但会改变next数组
                //else next[i]= next[j];
                next[i] = j;
            } else j = next[j];
        }
        return next;
    }
public static boolean kmp(String pattern, String text) {
    char[] p = pattern.toCharArray();
    char[] t = text.toCharArray();
    int next[] = getNext(p);
    int i = 0, j = 0;
    while(j < p.length && i < t.length) {

        if(j == -1 || t[i] == p[j]) {
```

```
            i++;j++;
        }
        else {
            j = next[j];
        }
    }
    if(j == p.length)
    return true;    //return i-j+1 返回第一次出现位置：从1开始
    else
    return false;
}
//next性质 next[n] 为前n位的公共前后缀长(不包括自身)
//循环节长度：len-next[len]；前提：首先这必须得是一个循环的String（且只有一次循环需要特判）
//循环次数：len/(len-next[len]);
public static int kmp_count(String pattern, String text) {
        char[] p = pattern.toCharArray();
        char[] t = text.toCharArray();
        int next[] = getNext(pattern);
        int i = 0, j = 0,count = 0;
        while(i < t.length&&j<p.length) {
            if(j == -1 || t[i] == p[j]) {
                i++;j++;
            }
            else {
                j = next[j];
            }
            if(j==p.length) {
                count++;j=next[j];
            }
        }
        return count;
    }//count 是出现几次目标串
```

Chapter4、Tree

Class Treenode

```
//基于treeNode数组的建树 treeNode数组tre[0]不做初始化为NULL
public static class treeNode{
        public int p; //父节点
        public int lc;//左儿子
        public int rc;//右儿子
        public treeNode() {
            p=0, lc=0, rc=0;
        }
        public void setP(int ap) {
            p=ap;
        }
        public void setLc(int alc) {
            lc=alc;
        }
        public void setRc(int arc) {
            rc=arc;
```

```
        }
        public boolean hasLc() {
            if(lc==0) return false;
            else return true;
        }
        public boolean hasRc() {
            if(rc==0) return false;
            else return true;
        }
        public boolean hasP() {
            if(p==0) return false;
            else return true;
        }
    }
//莫忘找根：入度为0.或者没有父亲的就是根
```

Travesal

```
void preorder(treeNode T){
    if(T==NULL) return;
    //操作
    preorder(T.left);
    preorder(T.right);
}
```

```
void inorder(treeNode T){
    if(T==NULL) return;
    inorder(T.left);
    //操作
    inorder(T.right);
}
```

```
void postorder(treeNode T){
    if(T==NULL) return;
    postorder(T.left);
    postorder(T.right);
    //操作
}
```

```
void levelorder(treeNode root){     其实就是BFS
    queue q;
    q.push(root);
    while(q.isempty){
        node=q.pop();
        //操作
        if(node.left!=NULL) q.push(node.left);
        if(node.right!=NULL) q.push(node.right);
    }
```

```
Traversal by stack
inorder:
Stack<Integer>stack = new Stack<>();
            int cur=root;
            int pre=0;
            while(!stack.isEmpty()||cur!=0) {
                if(cur!=0) {
                    stack.push(cur);
                    cur=arr[cur][0];
                }else {
                    cur=stack.pop();
                     //操作
                    //if(pre!=0&&val[cur]<=val[pre]) {
                    //  bool=-1;
                    }

                    pre=cur;
```

```
                    cur=arr[cur][1];
                }
            }
preorder:           //见cheating paper2
postorder:          //见cheating paper2
```

Tree height

```
//DFS 递归实现
public static int fd(int root,int[][] arr) {
        if(root==0) {
            return 0;
        }
        int nl=fd(arr[root][0],arr);
        int nr=fd(arr[root][1],arr);
        return (nl > nr) ? (nl + 1) : (nr + 1);

    }
public static void is(int root,int[][] arr) {
        if(root==0) {
            return;
        }
        if(Math.abs(fd(arr[root][0],arr)-fd(arr[root][1],arr))>1) bool=-1;
        is(arr[root][0],arr);
        is(arr[root][1],arr);
        return;
    }

```

Heap

```
//从图论中某一道题目里摘出来的，每个Node代表路径source—>value的路径，value是到达点的序号，wei是边权
static class heap{
        Node[] he;
        int size;
        public heap(){
            he=new Node[6005];
            size=0;
        }
        public void add(Node ad){
            size=size+1;
            he[size]=ad;
            int c=size;
            int p=size/2;
            Node temp;
            while(p>0 && he[p].wei>he[c].wei){
                temp=he[p];
                he[p]=he[c];
                he[c]=temp;

                c=p;
```

```
            p=p/2;
        }
    }
    public Node del(){
        Node temp=he[1];
        he[1]=he[size];
        size=size-1;
        fixRoot(1);
        return temp;
    }
    public void fixRoot(int a){
        int ch=2*a;
        Node temp;
        if(ch>size) {
            return;
        }
        else if(ch<size) {
            if(he[ch+1].wei<he[ch].wei) {
                ch=ch+1;
            }
        }
        if(he[ch].wei<he[a].wei){
            temp=he[ch];
            he[ch]=he[a];
            he[a]=temp;
        }
        fixRoot(ch);
    }
    public void update(int value,int wei){
        Node temp=null;
        int p=0;
        int current;
        for(int i=1;i<size+1;i++){
            if(he[i].value==value){
                he[i].wei=wei;
                current=i;
                p=i/2;
                while (p > 0 && he[p].wei > he[current].wei) {
                    temp = he[p];
                    he[p] = he[current];
                    he[current] = temp;
                    current = p;
                       p = current / 2;
                }
                i=size+1;
            }
        }
    }
}
public static class Node{
    int value;
    int wei;
    Node next;
    public Node(int value,int wei) {
        this.value=value;
        this.wei=wei;
    }
```

```
    }
    public static class list{
        Node head;
        Node tail;
        public list(){
            head=null;
            tail=null;
        }
        public void add(int value,int wei) {
            Node current=new Node(value,wei);
            if(head==null) {
                head=current;
                tail=current;
            }
            else {
                tail.next=current;
                tail=tail.next;
            }
        }
    }
```

AVL Tree

```
#include<cstdio>
#include<iostream>
using namespace std;
int delta, n, m, leave;
struct Treap{
    struct Node{    //v是该节点的值，s是子树节点数，r是一个随机值
        int v, r, s;
        Node* ch[2];
        int cmp(int x) const{
            if(x == v) return -1;
            else return x < v ? 0 : 1;
        }
        void maintain(){
            s = ch[0]->s + ch[1]->s +1;
        }
    };
    Node *root, *null;//root 是AVL树的根
    Treap(){
        null = new Node();
        root = null;
    }
    void rotate(Node* &o, int d){
        Node* k = o->ch[d^1];
        o->ch[d^1] = k->ch[d];
        k->ch[d] = o;
        o->maintain();
        k->maintain();
        o = k;
    }
    void insert(Node* &o, int x){   //在O的子树中插入值为x的点
        if(o == null){
```

```
            o = new Node();
            o->ch[0] = o->ch[1] = null;
            o->v = x;
            o->r = rand();
            o->s = 1;
        }
        else{
            int d = (x < o->v ? 0 : 1);
            insert(o->ch[d], x);
            if(o->ch[d]->r > o->r) rotate(o, d^1);
        }
        o->maintain();
    }
    int del(Node* &o, int x){//从O的子树中删除值小于x的点，返回删除点的数目
        if(o == null) return 0;
        if(o->v < x){
            int t = o->ch[0]->s + 1;
            o = o->ch[1];
            return t + del(o, x);
        }
        else{
            int t = del(o->ch[0], x);
            o->s -= t;
            return t;
        }
    }
    int find(Node* o, int k){//在以O为根的子树里搜寻第k大点，返回该点value
        if(o == null || k <= 0 || k > o->s) return 0;
        int s = (o->ch[1] == null ? 0 : o->ch[1]->s);
        if(s + 1 == k) return o->v;
        if(s >= k) return find(o->ch[1], k);
        return find(o->ch[0], k-s-1);
    }
} tp;

int main(){
    //freopen("p1503.in", "r", stdin);
    scanf("%d %d\n", &n, &m);
    char p[10];
    int x;
    tp = Treap();
    while(n--){
        scanf("%s%d\n", p, &x);
        if(p[0] == 'I') if(x >= m) tp.insert(tp.root, x-delta);
        if(p[0] == 'A') delta += x;
        if(p[0] == 'S'){ delta -= x; leave += tp.del(tp.root, m-delta); }
        if(p[0] == 'F'){
            if(x > tp.root->s) printf("-1\n");
            else printf("%d\n", tp.find(tp.root, x)+delta);
        }
    }
    printf("%d\n", leave);
    return 0;
}
//Treap 每次都要记录
```

Chapter5、Sort

Mergesort

```java
static void merge(int[] arr,int [] tmp,int left,int right) {

        if(left==right) return;
        if(left+1==right) {
            if(arr[right]>=arr[left]) return;
            else {
                int temp=0;
                temp=arr[left];
                arr[left]=arr[right];
                arr[right]=temp;
                ans++;
                return;
            }
        }
        int mid=(left+right)>>1;
        merge(arr,tmp,left,mid);
        merge(arr,tmp,mid+1,right);
        int i=left,j=mid+1,now=0;

        while(i<=mid&&j<=right) {
            if(arr[i]>arr[j]) {
                ans+=(mid-i+1);
                 //if(mid-i+1>1) flag=false;
                tmp[now++]=arr[j++];
            }else {
                tmp[now++]=arr[i++];
            }

        }


        while(i<=mid)//(处理左右子区间中、指针未指到最后所剩下的元素)
            tmp[now++]=arr[i++];
        while(j<=right)
            tmp[now++]=arr[j++];


        now=0;
        for(int k=left;k<=right;k++) {
            arr[k]=tmp[now++];
        }
    }

//特殊应用:
1.逆序对个数—ans
2.判断排序交换序列是否唯一
```

Quicksort

```java
    public static void main(String []args) {
        InputStream inputStream = System.in;
        OutputStream outputStream = System.out;
        InputReader in = new InputReader(inputStream);
        PrintWriter out = new PrintWriter(outputStream);
        int u=121;
        slove(in,out);
        out.close();
    }
    static void quickSort(int[]arr, int left, int right) {
            int len = arr.length;
            int partitionIndex;
            if (left < right) {
                partitionIndex = partition(arr, left, right);
                quickSort(arr, left, partitionIndex-1);
                quickSort(arr, partitionIndex+1, right);
            }

        }

    static int partition(int[] arr,int left,int right) {
        int pivot=left;
        int index=pivot+1;
        for(int i=index;i<=right;i++) {
            if(arr[i]<arr[pivot]) {
                int tmp=arr[i];
                arr[i]=arr[index];
                arr[index]=tmp;
                index++;
            }
        }
        int tmp=arr[pivot];
        arr[pivot]=arr[index-1];
        arr[index-1]=tmp;
        return index-1;
    }
    static void slove(InputReader in, PrintWriter out){   //Slove 里的内容是真正的解题代码
        int n=in.nextInt();
        for(int i=0;i<n;i++) {
            int m=in.nextInt();
            int[] arr=new int[m];
            int k=in.nextInt();
            for(int j=0;j<m;j++){
                arr[j]=in.nextInt();
            }
            Main.quickSort(arr, 0, m-1);
            out.println(arr[k-1]);
        }

    }
```