我们的目标是,4A, 4A, 4A ！！！

# 基本操作

## C++ 常用头文件和宏定义

* 一了百了

#include <bits/stdc++.h>

* 大军来袭

#include<cstdio>  
#include<cstring>  
#include<algorithm>  
#include<iostream>  
#include<string>  
#include<vector>  
#include<stack>  
#include<bitset>  
#include<cstdlib>  
#include<cmath>  
#include<set>  
#include<list>  
#include<deque>  
#include<map>  
#include<queue>

* 宏定义

#define FF(a,b) for(int a=0;a<b;a++)  
#define F(a,b) for(int a=1;a<=b;a++)  
#define LEN 200  
#define INF ((1<<30)-1)  
#define bug(x) cout<<#x<<"="<<x<<endl;  
// 把INF定义为(INF+INF)还能保证不溢出的一个数  
/\*  
INF+INF=2147483646  
0x7FFFFFFF=2147483647  
\*/  
using namespace std;  
typedef long long ll;  
const double pi=acos(-1);

## 输入输出

* double

1. 输入：%lf
2. 输出：%f

* scanf

**返回值解析**：

1. 返回正确读入的参数个数。
2. 遇到文件结尾，返回EOF，也就是-1
3. 安全的获取一行：fgets (buf, MAX, stdin)

* printf

1. 补前导0：%02d：总宽度为2，不足用0补
2. 设置精度：%.2f保留两位小数

* 输入流输出流

1. 关同步

std::ios::sync\_with\_stdio(false)

1. 获取一行

cin.get(name,SIZE); //回车保留在输入队列中  
cin.getline(name,SIZE); //回车被清除

1. 设置输出精度

cout<<setprecision(2)<<a; //输出: 123.45

1. 设置前导零：

cout<<setfill('0')<<setw(4)<<value<<endl;

## 二分

### 原生实现lower\_bound和upper\_bound

https://www.luogu.org/problemnew/show/P3366

* lower\_bound

int l=0; //初始化 l ，为第一个合法地址  
 int r=10; //初始化 r , 地址的结束地址  
 int mid;  
 while(l<r) {  
 mid=(l+r)/2;  
 if(arr[mid]>=obj){  
 r=mid;  
 }else{  
 l=mid+1;  
 }  
 }

* upper\_bound

int l=0; //初始化 l ，为第一个合法地址  
 int r=10; //初始化 r , 地址的结束地址  
 int mid;  
 while(l<r) {  
 mid=(l+r)/2;  
 if(arr[mid]>obj){ //没有=符号是与上文算法唯一的区别  
 r=mid;  
 }else{  
 l=mid+1;  
 }  
 }

## 排序

### 结构体排序

* 结构体内重载小于符号

bool operator < (const Node& obj) const  
{  
 return d<obj.d; //从小到大排序  
}

* 结构体外定义cmp函数

int cmp(const Node&a,const Node&b){  
 return a.d<b.d;//从小到大排序  
}

### 多属性按优先级排序

int cmp(Node& a,Node& b) {  
 if(a.name!=b.name) //最高优先级  
 return a.name<b.name;  
 else if(this.id!=o.id)  
 return a.id-b.id;  
 else  
 return a.credit-b.credit;  
}

# 图论

## 基本数据结构

### 链式前向星

* 数据结构定义

VN是题目指定的最大顶点数, EN是题目指定的最大边数

//---------------------链式前向星----------------------------  
int head[VN]; //记录源点u在mp中第一个地址i=head[u] 调用完之后就可以用mp[i]访问边表mp  
int cnt=0; //边表下标，随着数据的录入而扩张  
struct edge{ //边  
 int to,next,w;  
};  
edge mp[EN\*2]; //边表  
void add(int u,int v,int w){ //增加边  
 mp[cnt].to=v;  
 mp[cnt].w=w;  
 mp[cnt].next=head[u]; //指向源点u所构成的静态链表的头结点。如果是首次构造链，head[u]=-1 ,相当于NULL  
 head[u]=cnt++; //更新当前地址  
}  
void init\_mp(){  
 fill(head,head+VN,-1);  
}  
//---------------------链式前向星----------------------------

* 使用

首先必须在录入边表数据之前进行初始化

init\_mp();

对于无向图,一定要注意对两个方向的顶点同时进行加边操作

FF(i,M){  
 int a,b,w;  
 scanf("%d%d%d",&a,&b,&w);  
 add(a,b,w);  
 add(b,a,w);  
 }

遍历: 对u的邻接点遍历, i 代表的是边表下标, to 代表邻接点, w 代表边权

for(int i=head[u];~i;i=mp[i].next){ //链式前向星遍历  
 int to=mp[i].to; //链式前向星: u 的后继点 to  
 int w=mp[i].w; //链式前向星: u -> to 边权  
 }

### 堆优化

* 数据结构定义

优先队列默认是大根堆

定义新的比较规则cmp, 重载priority\_queue, 按dist[index]的值升序排列

//---------------------堆优化----------------------------  
struct cmp{  
 bool operator () (int a,int b){  
 return dist[a]>dist[b];  
 }  
};  
priority\_queue<int,vector<int>,cmp> pq;  
//---------------------堆优化----------------------------

* 在dijkstra使用

下文所有有注释的都是相对于非堆优化, 需要增删的内容

fill(dist,dist+LEN,MAX);  
dist[s]=0;  
pq.push(s); //优先队列入队初始化  
while(!pq.empty()){ //优先队列非空  
 int u=pq.top(); //寻找最小的dist[u]  
 pq.pop();  
 if(vis[u]) continue; //寻找最小的dist[u]  
 vis[u]=1;  
 for(int i=0;i<N;i++) if(!vis[i]){  
 if(dist[u]+g[u][i]<=dist[i]){ //一定要包含等于符号！！！！  
 dist[i]=dist[u]+g[u][i];  
 pq.push(i); //优先队列入队  
 }  
 }  
}

## 最短路径

### Floyd

void print\_via\_ij(int i,int j){  
 if(via[i][j]!=-1){  
 int k=via[i][j];   
 cout<<k<<"->";  
 print\_via\_ij(k,j);  
 }else{  
 cout<<j<<"->";  
 }  
}  
  
void floyd(){  
 for(int i=0;i<N;i++){ //初始化   
 for(int j=0;j<N;j++){  
 if(g[i][j]==0) g[i][j]=INF;  
 via[i][j]=-1;  
 A[i][j]=g[i][j];  
 }  
 }  
 for(int k=0;k<N;k++){ //临时点   
 for(int i=0;i<N;i++){  
 for(int j=0;j<N;j++){  
 if(A[i][j]>(A[i][k]+A[k][j])){  
 A[i][j]=A[i][k]+A[k][j];  
 via[i][j]=k;  
 }  
 }  
 }  
 }  
 //输出0到各点距离  
 int v=0;   
 for(int i=1;i<N;i++){  
 cout<<v<<"->";  
 print\_via\_ij(v,i);  
 cout<<endl;  
 }  
}

### 原生dijkstra

int N,M;  
int dist[VN];  
int pre[VN];  
int vis[VN];

void dijkstra(int s){  
 fill(dist,dist+N,INF);  
 fill(pre,pre+N,-1);  
 dist[s]=0;  
 pq.push(s); //优先队列初始化  
 while(!pq.empty()){ //优先队列非空  
 int u=pq.top(); //找到最小u点  
 pq.pop();  
 if(vis[u]) continue;//找到最小u点  
 vis[u]=1; //找到最近点,加入S集  
 for(int i=head[u];~i;i=mp[i].next){ //链式前向星遍历  
 int to=mp[i].to; //链式前向星: u 的后继点 to  
 int w=mp[i].w; //链式前向星: u -> to 边权  
 if(!vis[to]){ //松弛  
 if(dist[to] >= dist[u]+w) { //s->to >= s->u->to，一定要包含等于符号  
 dist[to] = dist[u] + w;  
 pre[to]=u;  
 pq.push(to);  
 }  
 }  
 }  
 }  
}

### 最短路计数

int pathc[VN]; //最短路条数

void dijkstra(int s,int e){  
 pathc[s]=1; //初始化源点最短路条数  
 ...................

if(!vis[to]){ //松弛  
 if(dist[to] > dist[u]+w) { //s->to > s->u->to  
 pathc[to] = pathc[u];  
 //其余公操作  
 }else if(dist[to]==dist[u]+w){  
 pathc[to] += pathc[u];  
 //其余公操作  
 }  
}

### 在有多条最短路的情况下, 寻找点权途经点权最大的最短路

int vw[VN]; //点权 vertex weight  
int by\_vw[VN]; //途经点权 bypass vertex weight

void dijkstra(int s,int e){  
 by\_vw[s]=vw[s];//初始化源点途经点权  
 ...................

if(!vis[to]){ //松弛  
 if(dist[to] > dist[u]+w) { //s->to > s->u->to  
 by\_vw[to]=by\_vw[u]+vw[to];  
 //其余公操作  
 }else if(dist[to]==dist[u]+w){  
 if(by\_vw[u]+vw[to] > by\_vw[to]){  
 by\_vw[to]=by\_vw[u]+vw[to];  
 }  
 }  
}

## 最小生成树

### 并查集优化Kruskal

//--------------------------并查集-----------------------------  
int fa[LEN];  
int init(){  
 int i;  
 FF(i,LEN) fa[i]=i;  
}  
int findFa(int x){  
 if(x==fa[x]) return x;  
 int r=x;  
 while(r!=fa[r]){//找到根节点  
 r=fa[r];  
 }  
 int t=x;  
 while(x!=fa[x]){//路径压缩  
 t=fa[x];  
 fa[x]=r;  
 x=t;  
 }  
 return r;  
}  
void Union(int a,int b){  
 int pa=findFa(a);  
 int pb=findFa(b);  
 fa[pa]=pb;  
}  
//--------------------------边表-----------------------------  
typedef struct Edge{//边表  
 int u,v,w;//两点和边权  
 //按照边权对边表进行排序  
 bool operator < (const Edge& obj) const  
 {  
 return w<obj.w; //从小到大排序  
 }  
}Edge;  
Edge edge[200010];  
//--------------------------主函数-----------------------------  
init(); //一定要初始化，不然拉闸  
int cnt=0,mst=0;  
FF(i,M){  
 scanf("%d%d%d",&edge[i].u,&edge[i].v,&edge[i].w);  
}  
sort(edge,edge+M);  
FF(i,M){  
 Edge &e=edge[i];  
 //Union操作  
 int pa=findFa(e.u);  
 int pb=findFa(e.v);  
 if(pa==pb)  
 continue;  
 fa[pa]=pb;  
 //更新MST  
 mst+=e.w;  
 cnt++;  
}  
if(cnt==N-1){  
 printf("%d\n",mst);  
}else{  
 printf("orz\n");  
}

### 堆优化Prim

还是有问题

https://www.luogu.org/problemnew/show/P3366

/---------------------其他数据结构----------------------------  
typedef struct Node{  
 int to,w;  
 Node(int to,int w):to(to),w(w){}  
}Node;  
vector<Node> g[VN];//邻接表  
int vis[VN];  
int dist[VN];  
//---------------------堆优化----------------------------  
struct cmp{  
 bool operator () (int a,int b){  
 return dist[a]>dist[b];  
 }  
};  
priority\_queue<int,vector<int>,cmp> pq;  
//---------------------主函数----------------------------  
 FF(i,N){ //初始化  
 dist[i]=INF;//初始化dist  
 }  
 FF(i,M){  
 int u,v,w;  
 scanf("%d%d%d",&u,&v,&w);  
 u--;v--;//用于顶点序号为[1,N]  
 g[u].push\_back(Node(v,w));  
 g[v].push\_back(Node(u,w));  
 }  
 int v=0;//初始化点  
 dist[v]=0; //第一次循环会自动将源点加入S集  
 pq.push(v);  
 int cnt=0;  
 int mst=0;  
 while(!pq.empty()&&cnt<=N){  
 int u=pq.top();  
 pq.pop();  
 if(vis[u]) continue;  
 vis[u]=1;//将dist值最小的点加入S集  
 mst+=dist[u];  
 cnt++;  
 //松弛操作  
 FF(i,g[u].size()){  
 int to=g[u][i].to;  
 int w=g[u][i].w;  
 if(vis[to]!=1 && dist[to]>w){  
 dist[to]=w;  
 pq.push(to);  
 }  
 }  
 }  
 if(cnt==N)  
 printf("%d\n",mst);  
 else  
 printf("orz\n");

# 数据结构

## 并查集

int fa[LEN];  
int init(){  
 int i;  
 FF(i,LEN) fa[i]=i;  
}  
int findFa(int x){  
 if(x==fa[x]) return x;  
 int r=x;  
 while(r!=fa[r]){//找到根节点  
 r=fa[r];  
 }  
 int t=x;  
 while(x!=fa[x]){//路径压缩  
 t=fa[x];  
 fa[x]=r;  
 x=t;  
 }  
 return r;  
}  
void Union(int a,int b){  
 int pa=findFa(a);  
 int pb=findFa(b);  
 fa[pa]=pb;  
}

## 二叉树的各类题型

https://www.cnblogs.com/TQCAI/p/8546737.html

### 根据前序+中序生成树

Node\* build\_tree(int ps,int pe,int is,int ie){//Node\* root=build\_tree(0,n-1,0,n-1);  
 if(ps>pe) return NULL;  
 if(ps==pe) return new Node(in[is]);  
 int i=is;  
 while(i<=ie && in[i]!=pre[ps]) i++;  
 Node \* node=new Node(in[i]);  
 int dLeft=i-is; //左侧元素数量  
 node->l=build\_tree(ps+1,ps+dLeft,is,is+dLeft-1);  
 node->r=build\_tree(ps+dLeft+1,pe,i+1,ie);  
 return node;  
}

### 根据后序+中序生成树

Node\* build\_tree(int ps,int pe,int is,int ie){//Node\* root=build\_tree(0,n-1,0,n-1);  
 if(ps>pe) return NULL;  
 if(ps==pe) return new Node(in[is]);  
 int i=is;  
 while(i<=ie && in[i]!=post[pe]) i++;  
 Node \* node=new Node(in[i]);  
 int dLeft=i-is; //左侧元素数量   
 node->l=build\_tree(ps,ps+dLeft-1,is,is+dLeft-1);  
 node->r=build\_tree(ps+dLeft,pe-1,i+1,ie);  
 return node;  
}

### 根据前序+中序生成后序：

//caution: should using initialize code: " t=0; "  
//pre + in -> post  
// pre\_start pre\_end in\_start in\_end  
void setPost(int ps,int pe,int is,int ie){  
 if(ps>pe)return;//null  
 if(ps==pe){  
 post[t++]=pre[ps];  
 }else{  
 //find the elem is the pair of preOrder (ps)  
 int i=is;  
 while(in[i]!=pre[ps] && i<ie) i++;//redirect  
 //left  
 setPost(ps+1, ps+i-is, is, i-1);  
 //right  
 setPost(ps+i-is+1, pe, i+1, ie);  
 //root  
 post[t++]=pre[ps];  
 }  
}

### 根据后序+中序生成后序：

//caution: should using initialize code: " t=0; "  
//post + in -> pre  
// post\_start post\_end in\_start in\_end  
void setPre(int ps,int pe,int is,int ie){  
 if(ps>pe)return;//null  
 if(ps==pe){  
 pre[t++]=post[ps];  
 }else{  
 //find the elem is the pair of preOrder (ps)  
 int i=is;  
 while(in[i]!=post[pe] && i<ie) i++;//redirect  
 //root  
 pre[t++]=post[pe];  
 //left  
 setPre(ps, ps+i-is-1, is, i-1);  
 //right  
 setPre(ps+i-is, pe-1, i+1, ie);  
 }  
}

### 根据前序+后序生成中序

#### 生成任意一个二叉树

void setIn(int preS,int preE,int postS,int postE){  
 if(preS>preE) return;  
 if(preS==preE){  
 in[t++]=pre[preS];  
 return;  
 }  
 int i=postS;  
 while(i<=postE-1 && post[i]!=pre[preS+1]) i++;  
 int ln=i-postS+1; //left\_num  
 if(i==postE-1){ //more than one condition  
 yes=0;  
 //默认找到的结点都为【左结点】。（如果想设置为“右结点”，可以改变setIn递归函数的位置）   
 setIn(preS+1,preS+ln,postS,postS+ln-1);  
 in[t++]=pre[preS];  
 return;  
 }  
 setIn(preS+1,preS+ln,postS,postS+ln-1);  
 in[t++]=pre[preS];  
 setIn(preS+ln+1,preE,postS+ln,postE-1);  
}

#### 计算有多少种二叉树

int cnt;  
void calc(int preS,int preE,int postS,int postE){  
 if(preS>=preE) return;  
 int i=postS;  
 while(i<=postE-1 && post[i]!=pre[preS+1]) i++;  
 int ln=i-postS+1; //left\_num  
 if(i==postE-1) cnt++;  
 calc(preS+1,preS+ln,postS,postS+ln-1);  
 calc(preS+ln+1,preE,postS+ln,postE-1);  
}

最后结果为pow(2,cnt)种结果

### 根据中序和层序来建树

* 数据结构

typedef struct Node{  
 int d;  
 struct Node\* l;   
 struct Node\* r;   
 Node(int d):d(d){  
 l=r=NULL;  
 }  
};   
Node \* root;  
map<int,int> num2index; //求一个数在中序中的索引   
int layer[LEN];  
int in[LEN];

* 核心代码

//主程序段调用  
for(int i=1;i<=n;i++){ //循环变量 i 遍历整个 layer 数组  
 j=1;  
 while(j<=n && in[j]!=layer[i]) j++; //找到in[j]==layer[i]  
 num2index[in[j]]=j; //这个数在中序中的索引  
 build\_tree(root,j);  
}  
//建树函数  
void build\_tree(Node\* & root,int i){// index of in  
 if(!root){  
 root=new Node(in[i]);  
 return;  
 }  
 int ri=num2index[root->d]; //root\_index;  
 if(i<ri) build\_tree(root->l,i);  
 else build\_tree(root->r,i);  
}

## AVL 树

这个很少考，先占个坑

https://www.cnblogs.com/TQCAI/p/8537110.html

## Huffman树

https://www.cnblogs.com/TQCAI/p/8538920.html

* 数据结构定义

int arr[100];  
typedef struct Node{  
 int d,l,r,i;  
 Node(int d=0):d(d){  
 l=-1;  
 r=-1;  
 }  
 bool operator < (const Node& obj) const  
 {  
 return d>obj.d; //因为要装入到大根堆的优先队列中  
 }  
}Node;  
  
Node nodes[100];//静态结点

* 初始化

int i,n=0;  
 while(~scanf("%d",&i)){  
 nodes[n]=Node(i);  
 nodes[n].i=n; //类似并查集，如果没有父结点，父结点就是自己  
 pq.push(nodes[n]);  
 n++;  
 }

* 建立Huffman树

while(pq.size()>1){ //最后只剩下一个结点  
 //从优先队列中拿出两个最小的结点  
 if(pq.empty())  
 break;  
 Node a=pq.top();  
 pq.pop();  
 if(pq.empty())  
 break;  
 Node b=pq.top();  
 pq.pop();  
 //构造一个新的结点  
 Node c(a.d+b.d);  
 c.l=a.i;  
 c.r=b.i;  
 nodes[n]=c; //放到新的静态区域中，并对n更新  
 c.i=n;  
 n++;  
 //把新的结点放入优先队列中  
 pq.push(c);  
 }

## 线段树

### 更新：区间加，查询：区间和

https://www.luogu.org/problemnew/show/P3372

#define LSON LS(p),l,mid  
#define RSON RS(p),mid+1,r  
#define LS(x) x<<1  
#define RS(x) x<<1|1  
#define MS ll mid=(l+r)>>1  
  
ll a[LEN];  
ll tree[LEN\*4]; //线段树  
ll lazy[LEN\*4]; //lazy优化  
  
inline void push\_up(ll p){  
 tree[p]=tree[LS(p)]+tree[RS(p)];  
}  
  
void build(ll p, ll l, ll r){  
 lazy[p]=0;  
 if(l==r){  
 tree[p]=a[l];  
 return;  
 }  
 MS;  
 build(LSON);  
 build(RSON);  
 push\_up(p);  
}  
  
inline void f(ll p,ll l,ll r,ll d){  
 lazy[p]+=d;  
 tree[p]+=d\*(r-l+1);  
}  
  
inline void push\_down(ll p,ll l,ll r){  
 ll mid=(l+r)>>1;  
 f(LSON,lazy[p]);  
 f(RSON,lazy[p]);  
 lazy[p]=0;  
}  
  
//update(nl,nr,1,1,N,d)  
inline void update(ll nl,ll nr,/\*固定域\*/ ll p, ll l, ll r,/\*查找域\*/ ll d){  
 if(nl<=l && r<=nr){  
 tree[p]+=d\*(r-l+1);  
 lazy[p]+=d;  
 return;  
 }  
 push\_down(p,l,r);  
 MS;  
 if(nl<=mid)  
 update(nl,nr,LSON,d);  
 if(mid<nr) // (mid+1)<=nr  
 update(nl,nr,RSON,d);  
 push\_up(p);  
}  
  
//query(nl,nr,1,1,N)  
ll query(ll nl,ll nr,/\*固定域\*/ ll p, ll l, ll r/\*查找域\*/){  
 ll res=0; //add  
 if(nl<=l && r<=nr)  
 return tree[p]; //modify  
 push\_down(p,l,r);  
 MS;  
 if(nl<=mid)  
 res+=query(nl,nr,LSON); //modify  
 if(mid<nr)  
 res+=query(nl,nr,RSON); //modify  
 return res; //add  
}

# 搜索法

## 连通增量

### 4连通增量

int dx[4]={0,0,1,-1};  
int dy[4]={1,-1,0,0};

### 8连通增量

int dir[8][2];  
int n;  
  
void build\_dir()  
{  
 int k=0;  
 FF(i,3)FF(j,3){  
 if(i==1 && j==1) continue;  
 dir[k][0]=i-1;  
 dir[k][1]=j-1;  
 k++;  
 }  
}

## DFS

### for外夹紧的DFS

https://blog.csdn.net/TQCAI666/article/details/86475929

//调用  
dfs(s);  
//dfs  
void dfs(int s){//当进入dfs(s)时，0～s-1的状态都已经完成  
 //退出区  
 if(s==e){ //终态。也可以在退出区剪枝  
 return; //一定要有返回   
 }  
 //入栈区  
 path.push\_back(s);  
 vis[s]=1;  
 //遍历区  
 for(i){  
 dfs(i);  
 }   
 //出栈区  
 vis[s]=0;  
 path.pop\_back();  
}

### for内夹紧的DFS

https://blog.csdn.net/TQCAI666/article/details/86475720

//调用  
vis[s]=1;//注意,如果初态不同,可能还会有循环  
dfs(s);  
vis[s]=0;  
//dfs  
void dfs(int s){  
 if(s==e){ //终态判断也可以写在for内  
 return; //一定要有返回   
 }  
 for(i){  
 path.push\_back(i);  
 vis[i]=1;  
 dfs(i);  
 vis[i]=0;  
 path.pop\_back();  
 }   
}

### 排列树

void backtrack(int t) { //坑: 第一个结果并不是字典序最小结果  
 if (t > N){  
 //终态  
 }else  
 for (int i = t; i<=N; i++){  
 swap(x[t], x[i]);  
 if(/\*剪枝条件 && 限制条件\*/) backtrack(t+1);  
 swap(x[t], x[i]);  
 }  
}

* 自带的全排列

do{  
 //终态  
}while(next\_permutation(arr,arr+n)); //n!

* next\_permutation代码实现

https://www.cnblogs.com/eudiwffe/p/6260699.html

// STL next permutation base idea  
int next\_permutation(int \*begin, int \*end)  
{  
 int \*i=begin, \*j, \*k;  
 if (i==end || ++i==end) return 0; // 0 or 1 element, no next permutation  
 for (i=end-1; i!=begin;) {  
 j = i--; // find last increasing pair (i,j)  
 if (!(\*i < \*j)) continue;  
 // find last k which not less than i,  
 for (k=end; !(\*i < \*(--k)););  
 iter\_swap(i,k);  
 // now the range [j,end) is in descending order  
 reverse(j,end);  
 return 1;  
 }  
 // current is in descending order  
 reverse(begin,end);  
 return 0;  
}

## BFS

### 一般的BFS

void bfs(){  
 queue<node> q;  
 q.push(S);  
 vis[S]=1;//入队标记  
 bool isFind=false;//终态标记  
 while(!q.empty()){  
 int sz=q.size();  
 while(sz--){ //方便计算层数  
 node u=q.front();  
 q.pop();  
 if(u==E){ //终态  
 isFind=true;  
 goto END;  
 }  
 FF(i,N){ //对于N种情况进行分析  
 if(!vis[i]){ //入队判断  
 vis[i]=1;  
 q.push(Node[i]); //入队标记  
 }  
 }  
 }  
 step++; //bfs层数  
 }  
 END: //cpp用goto跳出多重循环,Java有其他法  
}

## 排列组合

http://www.cnblogs.com/aiguiling/p/8594023.html

https://www.cnblogs.com/TQCAI/p/8454793.html

### 排列序列A(n,m)

int a[LEN]; //排列前的序列  
int vis[LEN];  
int cur[LEN]; //完成一次排列后形成的序列  
  
void permutations(int t,int n,int m){//A(n,m)  
 if(t>=m){ //达到循环上限  
 for(int i=0;i<m;i++) //输出一次排列后的序列  
 printf("%d ",cur[i]);  
 printf("\n");  
 return;  
 }else{  
 for(int i=0;i<n;i++)if(!vis[i]){  
 vis[i]=1;  
 cur[t]=a[i];  
 permutations(t+1,n,m);  
 vis[i]=0;  
 }  
 }  
}  
//调用  
permutations(0,n,m);

### 组合序列C(n,m)

占个坑

### 笛卡尔积P(n,m)

空间复杂度略微高一些

string a="1234";//递归前  
vector<string> ans;//递归结果  
  
string ch2str(char fuck){  
 char fuckYou[2]={0};  
 fuckYou[0]=fuck;  
 return fuckYou;  
}  
  
void product(int t,int n,int m){//P(n,m)  
 if(t>=m){//达到循环上限  
 for(int i=0;i<ans.size();i++)  
 printf("%s\n",ans[i].c\_str());  
  
 return;  
 }else{  
 vector<string> tmp;  
 int sz=ans.size();  
 if(sz){  
 for(int i=0;i<n;i++){  
 for(int j=0;j<sz;j++){  
 tmp.push\_back(ans[j]+a[i]);  
 }  
 }  
 }else{  
 for(int i=0;i<n;i++){  
 tmp.push\_back(ch2str(a[i]));  
 }  
 }  
 ans=tmp;  
 product(t+1,n,m);  
 }  
}

# 常用自带库与函数

## C语言字符串函数

//拷贝  
char \*strcpy(char \*dest, const char \*src)  
//比较  
int strcmp(const char \*str1, const char \*str2)  
/\*  
如果返回值 < 0，则表示 str1 小于 str2。  
如果返回值 > 0，则表示 str2 小于 str1。  
如果返回值 = 0，则表示 str1 等于 str2。  
\*/  
//拼接  
char \*strncpy(char \*dest, const char \*src, size\_t n)  
 //->实例  
 char src[10]="1234567";  
 char dest[10];  
 int start=2;  
 int length=3;  
 strncpy(dest,src+start,length);  
 dest[length]=0;//制作终止符

## 其他

* 内存赋值

memset(void \*s,int ch,size\_t n);

==注意==：

1. 对于在源程序段声明的数组，可以用sizeof直接获得字节大小。对于函数内声明的数组不行。
2. 只能对字节进行赋值，常用的有0和1 。

## algorithm

https://blog.csdn.net/weixin\_41070218/article/details/79626549

* accumulate求和

对于数组a，对于区间[0,N)，进行求和，之后加上0

int sum=accumulate(a,a+N,0);

* distance求迭代器的下标

distance(s.begin(),it)

* 批量赋值

1. 一维数组： fill(a,a+length,value);
2. 二维数组：

FF(i,LEN)  
 fill(dp[i],dp[i]+LEN,0);

* count计数

一种用法是判断容器内是否存在某一元素

count(a, a+length, value)

**条件计数**

count\_if(a, a+length, condition\_function)

* search子串查找

int\*p=search(seq,seq+5,sub,sub+3);

**重复串查找**

seq中是否有n个value

int\*p=search\_n(seq,seq+8,n,value);

* 连续序列去重

数组arr为连续序列，重复的元素都相邻。使用下面语句后，**去重+长度更新**

n=unique(arr,arr+n)-arr;

* copy复制

1. 用法1

//将数组myints中的七个元素复制到myvector容器中  
copy ( myints, myints+7, myvector.begin() );//调用这行代码之前，必须用 myvector.resize(7)

1. 用法2 数组平移

copy(arr+1,arr+9+1,arr);//把长度为9的数组向左平移1位

* for\_each对于每个迭代器执行指定函数

1. 对于迭代器

// 3 for\_each  
 void fun(map<int,string>::reference a) //不要少了reference,不然会报错。  
{  
 a.second+="\_suffix"; //a是引用对象。  
 cout<<a.first<<" "<<a.second<<endl;  
}  
for\_each(m.begin(),m.end(),fun);

1. 对于容器

void fun1(int & p){ //回调函数的参数同样也是引用类型  
 p++;  
}

* equal判断两个序列是否完全相等

bool ans=equal(A,A+LEN,B);

* mismatch找到两个序列第一次失配的元素

pair<int\*,int\*>ans=mismatch(A,A+LEN,B);

* rotate循环平移

1. 向左移动n位

rotate(A,A+n,A+LEN);

1. 向右移动n位

rotate(A,A+LEN-n,A+LEN);

* swap\_ranges交换两个区间的值

swap\_ranges(A,A+5,B);

* remove删除容器中的某个值

// 把要删除的元素移动到容器末尾  
it=remove(v.begin(),v.end(),removedValue) //it为末尾删除区的第一个元素  
// 真正意义上的删除  
v.erase(remove(v.begin(),v.end(),removedValue),v.end());

* 对容器中的序列套接一个函数，把结果放到一个容器

transform(A,A+LEN,B,add);

* replace把序列中的某个值替换为另一个值

replace(std::begin(data), std::end(data), pre, after);

## string

### 基本操作

**初始化**：

string(int n,char c); //用n个字符c初始化

**取子串**：

string substr(int pos=0, int n=npos) const; //返回由pos开始的n个字符组成的子字符串

**查找**：

==注意！== find函数如果查找不到，就返回string::npos

int find(char c,int pos=0) const; //从pos开始查找字符c在当前字符串的位置   
int find(const char \*s, int pos=0) const; //从pos开始查找字符串s在当前字符串的位置  
int find(const string &s, int pos=0) const; //从pos开始查找字符串s在当前字符串中的位置  
find函数如果查找不到，就返回 string::npos  
int rfind(char c, int pos=npos) const; //从pos开始从后向前查找字符c在当前字符串中的位置   
int rfind(const char \*s, int pos=npos) const;  
int rfind(const string &s, int pos=npos) const;  
//rfind是反向查找的意思，如果查找不到， 返回-1

**替换**：

string &replace(int pos, int n, const char \*s);//删除从pos开始的n个字符，然后在pos处插入串s  
string &replace(int pos, int n, const string &s); //删除从pos开始的n个字符，然后在pos处插入串s  
void swap(string &s2); //交换当前字符串与s2的值

**删除**：

string &insert(int pos, const char \*s);  
string &insert(int pos, const string &s);  
//前两个函数在pos位置插入字符串s  
string &insert(int pos, int n, char c); //在pos位置 插入n个字符c  
string &erase(int pos=0, int n=npos); //删除pos开始的n个字符，返回修改后的字符串  
/\*  
注意string的erase与其他容器的erase不同。其他容器的erase删除迭代器，返回删除元素的第一个后级元素  
\*/

**大小写转换**：

transform(s.begin(), s.end(), s.begin(), ::toupper);//转大写  
transform(s.begin(), s.end(), s.begin(), ::tolower);//转小写

### 常见字符串ASCII值

'0'=48 'a'=97 'A'= 65

### 任意类型转化为str

template<typename T> string toString(const T& t){  
 ostringstream oss;  
 oss<<t;  
 return oss.str();  
}

### 将str转化为数值类型

template <class Type>  
Type stringToNum(const string& str){  
 istringstream iss(str);  
 Type num;  
 iss >> num;  
 return num;  
}

### 插入字符串

void insertStr(string &str,const string &obj,int pos=0){  
 str.replace(pos,0,obj);  
}

### 全部替换

bool replaceAll(string &str,const string &a,const string &b){  
 int al=a.length();  
 int bl=b.length();  
 int pos,pre=0;  
 bool isFind=false;  
 while((pos=str.find(a,pre))!=string::npos){  
 str.replace(pos,al,b);  
 pre=pos+bl;  
 isFind=true  
 }  
 return isFind;  
}

### 字符串拆分

* 单字符切割复杂字串

所谓复杂字串，是指用若干个空格隔开的文本。此函数使用频率不高

void split(string s,char token,vector<string>& v)  
{  
 int L = s.length();  
 int start=0;  
 string t;  
 for(int i=0; i<L; i++)  
 {  
 if(s[i] == token && i == 0)//第一个就遇到分割符  
 {  
 start += 1;  
 }  
 else if(s[i] == token)  
 {  
 t=s.substr(start,i - start);  
 if(t.length()>0)  
 v.push\_back(t);  
 start = i+1;  
 }  
 else if(i == L-1)//到达尾部  
 {  
 t=s.substr(start,i+1 - start);  
 if(t.length()>0)  
 v.push\_back(t);  
 }  
 }  
}

* 单字符切割简单字串

所谓简单字串，一般指在OJ输入的数据，例如北邮OJ的IP地址的123:45:78:9:判断IP是否合法。在这里被分割为5个字串，其中最后一个是空串

void split(string s,char token,vector<string>& v){  
 int n=s.length();  
 string t;  
 FF(i,n){  
 char c=s[i];  
 if(c!=token)  
 t+=c;  
 else{  
 v.push\_back(t);  
 t="";  
 }  
 }  
 v.push\_back(t);  
}

* 多字符切割简单字串

void split(string s,string token,vector<string>& v){  
 int n=s.length();  
 int m=token.length();  
 string t;  
 int i=0;  
 while(i+m<=n){  
 int step=1;  
 char c=s[i];  
 if(s.substr(i,m)!=token)  
 t+=c;  
 else{  
 v.push\_back(t);  
 t="";  
 step=m;  
 }  
 i+=step;  
 }  
 v.push\_back(t);  
}

## priority\_queue

### 基本操作

入队：push()

出队：pop()

队顶：top()

### 小根堆

pq默认为大根堆（见严蔚敏的大根堆和堆排序），但是如果我们想用小根堆：

priority\_queue<int,vector<int>,greater<int> > pq;

### 结构体小根堆

//根据d的值有小到大  
typedef struct Node{  
 int d;  
 bool operator < (const Node& obj) const  
 {  
 return d>obj.d; //注意  
 }  
}Node;  
priority\_queue<Node> pq;

### 外部小根堆

struct cmp{  
 bool operator () (int a,int b){  
 return dist[a]>dist[b];  
 }  
};  
priority\_queue<int,vector<int>,cmp> pq;

## set

### 基本操作

set：红黑树实现

multiset：多值集合，但能动态排序

**初始化**：

int num[]={1,2,3};  
multiset<int> s(num,num+3);

**插入**：insert()

**计数**：count() 也可以判断某个item存在与否

**清空**：clear()

**查找**：it=find(s.begin(),s.end(),v); 或者it=s.find(v)

获取下标：distance(s.begin(),it)

查找失败：it==s.end()

二分查找：

[a,b)

s.lower\_bound(v) s.upper\_bound(v)

**删除**：

s.erase(it)或s.erase(it\_begin,it\_end)

注意，set的迭代器没有实现+操作符，所以只能循环使用++或--来对下标进行定位

**遍历**：

multiset<int>::iterator it=s.begin();  
 while(it!=s.end()){  
 printf("%d",\*it);  
 it++;  
 if(it!=s.end())  
 printf(" ");  
 }

### 改为从大到小排序

使用STL自带重载

set<int,greater<int> > s;

自己写个

struct cmp{  
 bool operator() (const int& a,const int& b) const  
 {  
 return a>b;  
 }  
};  
set<int,cmp> s;

### 结构体放入

typedef struct Node{  
 int d;  
 Node(int d):d(d){}  
 bool operator < (const Node& obj) const  
 {  
 return d<obj.d; //从小到大排序  
 }  
}Node;

## vector

### 动态删除所有满足条件的元素

while(it!=v.end()){  
 if(\*it==3){//满足了某一条件，则删除  
 it=v.erase(it);  
 }else{//不满足，自增  
 it++;  
 }  
}

# 数学问题

## 表达式求值

https://blog.csdn.net/TQCAI666/article/details/88692096

* 用一个结构体记录数组或操作符

typedef struct Node{//运算结点，记录数字或运算符号  
 bool isop;  
 char op;  
 int num;  
 Node(){}  
 Node(int d){  
 num=d;  
 isop=0;  
 }  
 Node(char op\_){  
 op=op\_;  
 isop=1;  
 }  
 void output(){  
 if(isop) cout<<op;  
 else cout<<num;  
 }  
}Node;

### 中缀表达式生成后缀表达式

int getPriority(char op){  
 switch(op){  
 case '+':  
 case '-':  
 return 1;  
 case '\*':  
 case '/':  
 return 2;  
 default:  
 return 0;  
 }  
}  
  
vector<Node> yieldPostOrderExpress(vector<Node> &in){  
 vector<Node> post;  
 Node ops[LEN];  
 int top=-1;//栈顶指针  
 FF(i,in.size()){  
 Node u=in[i];  
 if(u.isop){  
 if(u.op=='('){  
 ops[++top]=u;//入栈  
 }else if(u.op==')'){  
 while(top>=0){  
 Node p=ops[top--];  
 if(p.op=='('){  
 break;  
 }else{  
 post.push\_back(p);  
 }  
 }  
 }else{  
 while(top>=0 && getPriority(u.op) <= getPriority(ops[top].op)){  
 post.push\_back(ops[top--]);//持续出栈直到满足优先级  
 }  
 ops[++top]=u;//入栈  
 }  
  
 }else{  
 post.push\_back(u);  
 }  
 }  
 //处理剩下的栈  
 while(top>=0){  
 post.push\_back(ops[top--]);  
 }  
 return post;  
}

### 后缀表达式计算实值

Node operate(Node &a,Node& b,char op){  
 int v;  
 switch(op){  
 case '+':  
 v=a.num+b.num;  
 break;  
 case '-':  
 v=a.num-b.num;  
 break;  
 case '\*':  
 v=a.num\*b.num;  
 break;  
 case '/':  
 v=a.num/b.num;  
 break;  
 }  
 return Node(v);  
}  
  
int calcPostOrderExpress(vector<Node>& post){  
 Node s[LEN];  
 int top=-1;  
 FF(i,post.size()){  
 Node& u=post[i];  
 if(u.isop){  
 Node a=s[top--];  
 Node b=s[top--];  
 Node ans=operate(a,b,u.op);  
 s[++top]=ans;  
 }else{  
 s[++top]=u;  
 }  
 }  
 return s[0].num;  
}

## 进制转换

### 实值转二进制序列

推广后可以转换为N进制序列

string getBinarySequence(int num){  
 string ans="";  
 while(num){  
 if(num&1){  
 ans="1"+ans;  
 }else{  
 ans="0"+ans;  
 }  
 num>>=1;  
 }  
 return ans;  
}

### R进制序列转实值

int getTrueValue(int r,string seq){//radix sequence  
 int n=seq.size();  
 int base=1;  
 int ans=0;  
 for(int i=n-1;i>=0;i--){  
 int v=seq[i]-'0';  
 ans+=v\*base;  
 base\*=r;  
 }  
 return ans;  
}

## 排列组合

### 记忆化搜索计算组合数

int C[50][50];  
  
int C\_(int n,int m){  
 if(C[n][m]) return C[n][m];  
 if(m==0){  
 C[n][m]=1;  
 return C[n][m];  
 }  
 if(m==n){  
 C[n][m]=1;  
 return C[n][m];  
 }  
 else{  
 C[n][m]=C\_(n-1,m-1)+C\_(n-1,m);  
 return C[n][m];  
 }  
}

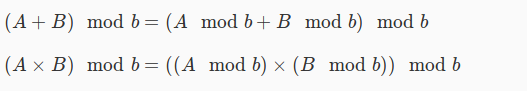
## 快速幂

### 常数快速幂

ll quickPower(ll a,ll b){  
 ll ans=1,base=a;  
 while(b>0){  
 if(b&1)  
 ans\*=base;  
 base\*=base;  
 b>>=1;  
 }  
 return ans;  
}

### 快速幂取余

性质



在这里插入图片描述

ll quickPowerMod(ll a,ll b,ll m){  
 ll ans=1,base=a;  
 while(b>0){  
 if(b&1){  
 ans\*=base;  
 ans%=m;  
 }  
 base\*=base;  
 base%=m;  
 b>>=1;  
 }  
 return ans%m;  
}

### 高精度快速幂

http://www.cnblogs.com/TQCAI/p/8410799.html

https://blog.csdn.net/TQCAI666/article/details/86599395

hp ans("1");  
 hp base("2"); //2^p  
 while(P>0){  
 if(P&1)  
 ans=multiplyh(ans,base);  
 base= multiplyh(base,base);  
 P>>=1;  
 }

### 矩阵快速幂

* 矩阵乘法

matrix\* multi(matrix& A, matrix& B){  
 matrix\* C=new matrix;  
 C->row=A.row;  
 C->col=B.col;  
 for(int i=0;i<A.row;i++)  
 for(int j=0;j<B.col;j++)  
 for(int k=0;k<A.col;k++)  
 C->m[i][j]+=A.m[i][k]\*B.m[k][j];  
 return C;  
}

* 矩阵快速幂

matrix quickPower(matrix& A,int p){ //A^p  
 int n=A.col;  
 //得到单位阵  
 matrix ans(n,n);//ans=E  
 FF(i,n)  
 FF(j,n)  
 if(i==j)  
 ans.m[i][j]=1;  
 else  
 ans.m[i][j]=0;  
 //开始快速幂  
 matrix base=A;  
 while(p){  
 if(p&1)  
 ans=\*multi(ans,base);  
 base=\*multi(base,base);  
 p>>=1;  
 }  
 return ans;  
}

## 数论

### gcd

int gcd(int a,int b){  
 while( b != 0){  
 int t = a % b;  
 a = b;  
 b = t;  
 }  
 return a;  
}

### 素数判断

bool isPrime(int x){  
 if(x<=1) return 0;  
 for(int i=2;i<=sqrt(x);i++){  
 if(x%i==0) return 0;  
 }  
 return 1;  
}

### 欧拉线性筛

#define MAXN 100005 //素数的最多个数  
#define MAXL 1299710 //要求的最大素数  
//实际操作中，只需要控制MAXL的长度，MAXN开的足够大即可  
int prime[MAXN];//素数表  
int check[MAXL];//如果n是素数，则check[n]=0，否则check[n]=1  
  
void EulerSieveMethod(){  
 int tot = 0;  
 memset(check, 0, sizeof(check));  
 for(int i=2;i<MAXL;i++){  
 if(check[i]==0){//素数  
 prime[tot++]=i;  
 }  
 for(int j=0;j<tot;j++){  
 int v=i\*prime[j];  
 if(v>MAXL) break;  
 check[v]=1;  
 if(i%prime[j]==0)//有点没搞明白  
 break;  
 }  
 }  
}

# 动态规划

## 背包问题

### 01背包

https://www.luogu.org/problemnew/show/P1048

* 对于==刚好装满==问题

1. 采用-INF对f数组进行初始化

fill(f,f+FN,-INF);

需要注意的是，INF应该采取**最大的安全值**，例如：#define INF ((1<<30)-1)

1. f[0]初始化为0
2. 对结果进行判断

if(ans<0)  
 puts("NO");  
 else  
 printf("%d\n",ans);

* 原生方法

for(int i = 1; i <= N; i++){ //i代表物品  
 for(int j = 0; j <= M; j++){ //j代表当前背包容量上限  
 if(j>=w[i]){ //当前物品能放入  
 f[i][j]=max(f[i-1][j], //不放  
 f[i-1][j-w[i]]+v[i]); //放  
 }else{  
 f[i][j]=f[i-1][j];  
 }  
 }  
 }  
 printf("%d", f[N][M]);

* 滚动数组优化

for(int i=1;i<=N;i++){  
 for(int c=M;c>=w[i];c--){  
 f[c]=max(f[c],f[c-w[i]]+v[i]);  
 }  
 }  
 printf("%d\n",f[M]);

### 完全背包

http://nyoj.top/problem/311

* 滚动数组优化

for(int i=1;i<=N;i++){  
 for(int c=w[i];c<=M;c++){  
 f[c]=max(f[c],f[c-w[i]]+v[i]);  
 }  
 }

## 线性DP

### 最长不下降子序列LIS

https://www.cnblogs.com/itlqs/p/5743114.html

* N2复杂度

void DP\_n2(){  
 int p; //LIS 结束的下标  
 int ans=0;  
 F(i,n){  
 dp[i]=1; //初始化  
 F(j,i-1){  
 if(a[j]<a[i])  
 dp[i]=max(dp[i],dp[j]+1);  
 }  
 if(dp[i]>ans){ //更新最优解  
 ans=dp[i];  
 p=i;  
 }  
 }  
}

**分析：**根据DP数组的线性递增顺序就能找到所有的LIS序列

ans=5 p=11  
//序列 :  
1 3 7 6 8 5 3 2 7 2 9   
//DP数组 :  
1 2 3 3 4 3 2 2 4 2 5

* Nlog2N复杂度

1. 不严格递增（不下降）

void DP\_nLogn1(){  
 int top=0;  
 F(i,n){  
 if(top==0 || a[i]>=dp[top-1])  
 dp[top++]=a[i];  
 else{  
 int pos=upper\_bound(dp,dp+top,a[i])-dp;  
 dp[pos]=a[i];  
 }  
 }  
}

1. 严格递增

void DP\_nLogn2(){  
 int top=0;  
 F(i,n){  
 if(top==0 || a[i]>dp[top-1])  
 dp[top++]=a[i];  
 else{  
 int pos=lower\_bound(dp,dp+top,a[i])-dp;  
 dp[pos]=a[i];  
 }  
 }  
}

### 最长公共子序列

https://www.cnblogs.com/TQCAI/p/8427862.html

void LCS(){  
 F(i,N){  
 F(j,M){  
 if(a[i]==b[j])  
 dp[i][j]=dp[i-1][j-1]+1;  
 else  
 dp[i][j]=max(dp[i-1][j],dp[i][j-1]);  
 }  
 }  
}

### 最大子段和

int i,p=0,ans=-MAX,t,n;  
 scanf("%d",&n);  
 for(i=0;i<n;i++){  
 scanf("%d",&t);  
 p+=t;  
 ans=max(ans,p);  
 if(p<0) p=0; //说明当前的t是负数， 从下个t开始初始化  
 }  
 printf("%d\n",ans);

## 区间DP

### 加分二叉树

https://blog.csdn.net/TQCAI666/article/details/86476042

F(i,N){  
 cin>>f[i][i];  
 f[i][i-1]=1; //空结点为0  
 root[i][i]=i;  
 }  
 f[N+1][N]=1;  
 for(int d=1;d<N;d++){//间隔  
 for(int i=1;i+d<=N;i++){//开始  
 int j=i+d;//结束  
 root[i][j]=i;  
 f[i][j]=f[i][i]+f[i+1][j];  
 for(int k=i+1;k<=j;k++){ //分割  
 int tmp\_fij=f[k][k]+f[i][k-1]\*f[k+1][j];  
 if(tmp\_fij>f[i][j]){  
 f[i][j]=tmp\_fij;  
 root[i][j]=k;  
 }  
 }  
 }  
 }  
 cout<<f[1][N]<<endl;

### 最长回文子串

PAT-A: https://pintia.cn/problem-sets/994805342720868352/problems/994805446102073344

同样用到了区间DP的思想，通过逐渐增大排查范围，无后效性

char buf[LEN];  
int dp[LEN][LEN];  
  
int main()  
{  
 fgets(buf,LEN,stdin);  
 int N=strlen(buf);  
 //初始化。1表示合法  
 FF(i,N) dp[i][i]=1;  
 int ans=0;  
 FF(i,N-1)  
 if(buf[i]==buf[i+1])  
 dp[i][i+1]=1;  
 for(int v=2;v<N;v++){//v表示间隔  
 for(int i=0;i+v<N;i++){//i表示开始下标  
 int j=i+v;  
 if(buf[i]==buf[j] && dp[i+1][j-1]==1){//判定为合法  
 dp[i][j]=1;  
 ans=max(ans,v);  
 }  
 }  
 }  
 printf("%d",ans+1);  
 return 0;  
}

# 分治递归

## 归并排序求逆序对

https://www.cnblogs.com/TQCAI/p/8641452.html

ll tmp[LEN],a[LEN];  
ll ans=0;  
ll arr[LEN];  
  
void merge(ll s,ll mid,ll e){  
 ll i=0,a=s,b=mid+1;  
 while(a<=mid && b<=e){  
 if(arr[a]<=arr[b]){ //巨坑,<=  
 tmp[i++]=arr[a++];  
 }else{  
 tmp[i++]=arr[b++];  
 ans+=mid-a+1;  
 }  
 }  
 while(a<=mid) tmp[i++]=arr[a++];  
 while(b<=e) tmp[i++]=arr[b++];  
 FF(j,i) arr[s+j]=tmp[j];  
}  
  
void mergeSort(ll s,ll e){  
 if(s<e){ //能够被划为。如果只有一个数就不能被划分  
 ll mid=(s+e) /2;  
 mergeSort(s,mid);  
 mergeSort(mid+1,e);  
 merge(s,mid,e);  
 }  
}

## 台阶问题

https://www.luogu.org/problemnew/show/P1192

假设有N级台阶，一个人一次可以走[1,K]级，问一共有多少种走法

相当于一次可以走1、 2步的斐波那契问题的推广

f[0]=1;//初始化第一步  
 for(int i=1;i<=N;i++){//每一级台阶  
 for(int j=1;j<=K && i-j>=0 ;j++){//走j步，j∈[1,K]  
 f[i]+=f[i-j];  
 }  
 }  
 cout<<f[N];

## 传球游戏

https://www.luogu.org/problemnew/show/P1057

有N个人，其中一个人可以从他的左边或者右边接到球。游戏开始，某人发球，求进行M次游戏后，球重新回到他的手中的传球方法数。

**实则为dp**

状态转移：f[i][k]=f[i-1][k-1]+f[i+1][k-1]，(i=1或n时，需单独处理)

f[1][0]=1;//fij表示在第j轮，球传到i手中的方法数。  
 for(int i=1;i<=m;i++)//传 m 次   
 {  
 f[1][i]=f[2][i-1]+f[n][i-1]; //第一个   
 f[n][i]=f[1][i-1]+f[n-1][i-1]; //最后一个   
 for(int j=2;j<n;j++) //中间   
 f[j][i]=f[j-1][i-1]+f[j+1][i-1];  
 }//注意第一个人和最后一个人的单独处理   
 printf("%d",f[1][m]);

## 整数划分

占个坑

https://www.cnblogs.com/TQCAI/p/8674820.html