**链式前向星：**

struct Node

{

int to,next;

};

Node edge[MAXN];

int box[MAXN];

int ent=1;

void add(int from,int to)

{

edge[ent].to=to;

edge[ent].next=box[from];

box[from]=ent++;

}

void edgefrom(int n) //输出以n为开头的所有边

{

for(int j=0; j<10; ++j)

for(int i=box[j]; i; i=edge[i].next)

{

cout<<j<<"to"<<edge[i].to<<endl;

}

}

**dijkstra（优先队列优化）：**

//如果不用优先队列，则去掉优先队列，在每次循环时找到没有被visited的点中dis值最小的

int dis[maxn2];

bool vis[maxn2];

const int Inf=;

struct FSO

{

int from,dis;

FSO(int from\_,int dis\_):from(from\_),dis(dis\_) {}

bool operator<(const FSO &n)const {return dis>n.dis;}

};

int dijkstra(const int &n,int from,int to)

{

for(int i=0; i<maxn2; ++i)

dis[i]=Inf;

memset(vis,0,sizeof(vis));

priority\_queue<FSO> que;

que.push(FSO(from,0));

dis[from]=0;

int mx,now;

while(1)

{

if(que.empty()) break;

mx=que.top().from;

if(vis[mx])

{

que.pop();

continue;

}

que.pop();

for(int j=box[mx]; j; j=edge[j].next)

{

now=edge[j].to;

if(dis[now]>dis[mx]+1)

{

dis[now]=dis[mx]+1;

que.push(FSO(edge[j].to,dis[now]));

}

}

vis[mx]=1;

}

if(dis[to]!=Inf)

return dis[to];

else return -1;

}

**迭代加深搜索：（搜分解分数式）：**

注意剪枝….

bool ids(int d,double sum,int now) //深度，和，现在的值

{

if(d==maxd)

{

if(abs(sum-val)<dev)

{

return 1;

}

else return 0;

}

int k=(maxd-d)/abs(val-sum)+1; //迭代上限，还剩下maxd-d个数,到val的值为val-sum，为能到达val现在可以填的最小的数即1/k

for(int i=now;i<=k;++i)

{

if(ids(d+1,sum+1.0/i,i+1))

{

cout<<"1/"<<i<<" ";

return 1;

}

}

return 0;

}

Int main()

{

While(ids(0,0,1))

deep++;

}

**第k短路（A\*算法）：**

首先对反图做一遍dijkstra,作为预处理

（存图的时候就存反边

void add(int from,int to,int weight)

{

edge[ent].to=to;

edge[ent].from=from;

edge[ent].next=box[from];

edge[ent].last=rebox[to];

edge[ent].weight=weight;

box[from]=ent;

rebox[to]=ent++;

}

//这个是打印路径的，如果不需要打印路径，则吧关于vector的东西去掉

struct ASO

{

int from,dis,f;

vector<int> luj;

ASO(int from\_,int dis\_,int f\_,vector<int> l):from(from\_),dis(dis\_),f(f\_),luj(l){}

bool operator<(const ASO &a)const

{

return f>a.f;

}

};

int Astar(int from,int to,int k)

{

int mx,now,ndis;

int c=0;

if(from==to)

c—

//这个判断有争议，用不用都可以试下

priority\_queue<ASO> que;

vector<int> tem;

tem.push\_back(from);

que.push(ASO(from,0,dis[from],tem));

while(1)

{

if(que.empty()) break;

mx=que.top().from;

ndis=que.top().dis;

tem=que.top().luj;

que.pop();

if(mx==to)

{

c++;

printf("%d : %d\n",c,ndis);

printf("lujing:");

for(vector<int>::iterator iter=tem.begin();iter!=tem.end();++iter)

cout<<\*iter<<" ";

cout<<endl;

}

if(c==k)

return ndis;

tem.push\_back(1);

for(int i=box[mx]; i; i=edge[i].next)

{

now=edge[i].to;

tem.pop\_back();

tem.push\_back(now);

que.push(ASO(now,ndis+edge[i].weight,ndis+edge[i].weight+dis[now],tem));

}

}

return -1;

}

**并查集：**

//两个基本功能：并/查~

int father[maxn];

int num[maxn];

int n,m;

void init()

{

for(int i=0;i<n;++i)

num[i]=1;

for(int i=0;i<n;++i)

father[i]=i;

}

int find(int a)

{

while(father[a]!=a)

{

a=father[a];

}

return a;

}

void Union(int a,int b)

{

a=find(a);

b=find(b);

if(a==b)

return;

father[a]=b;

num[b]+=num[a];

return;

}

**最小生成树：**

**krusakal：**

//把所有边按长度排序，然后从头开始加边，如果存在环路则跳过，判断环路用并查集

bool used[maxn]

void kruskal()

{

sort(edge,edge+ent);

ub=0;

for(int i=0; i<ent; ++i)

{

if(find(edge[i].u)==find(edge[i].v))

continue;

Uni(edge[i].u,edge[i].v);

used[ub++]=i;

if(ub==p-1) break;

}

}

**Prim:**

//使用邻接表来存（内存消耗很大，但速度较快）

struct FSO //原来的版本没有dis,直接用dp[p]来判断，这导致队列不能有效更新

{

int p,dis;

FSO(int p\_,int dis\_):p(p\_),dis(dis\_) {}

bool operator<(const FSO &a)const

{

return dis>a.dis;

}

};

void prim()

{

int now;

for(int i=0; i<p; ++i)

dp[i]=Inf;

memset(vis,0,sizeof(vis));

priority\_queue<FSO> que;

que.push(FSO(0,0));

while(1)

{

if(que.empty()) break;

now=que.top().p;

if(vis[now])

{

que.pop();

continue;

}

que.pop();

for(int i=0; i<p; ++i)

{

if(i==now) continue;

if(vis[i]) continue;

if(dis[now][i]<dp[i])

{

dp[i]=dis[now][i];

que.push(FSO(i,dp[i]));

}

}

vis[now]=1;

}

}

**GCD/exGCD:**

int gcd(int a,int b)

{

int t;

for(;b;t=b,b=a%b,a=t);

return a;

}

int exgcd(int a,int b,int &x,int &y)

{

int res;

if(b==0)

{

x=1;

y=0;

res=a;

}

else

{

res=exgcd(b,a%b,x,y);

int t=x;

x=y;

y=t-a/b\*y;

}

return res;

}

**大数加法：**

//这里使用千进制数：

int a4[maxn/4];

int b4[maxn/4];

char tem[maxn];

int lena,lenb;

int read(int\* a) //输入数据

{

if(scanf("%s",tem)==EOF) return -1;

int l=strlen(tem);

int len=l/4;

int p=l%4;

int k=-1;

if(p)

{

len++;

k++;

a[k]=tem[0]-'0';

for(int i=1; i<p; ++i)

{

a[k]=a[k]\*10+tem[i]-'0';

}

}

for(int i=p; i<l; ++i)

{

if(i%4==p)

{

k++;

a[k]=tem[i]-'0';

}

else

{

a[k]=a[k]\*10+tem[i]-'0';

}

}

return len;

}

void cal()

{

int t,m,ta,tb;

m=max(lena,lenb);

t=0;

lena--;

lenb--;

for(int i=m-1; i>=0; --i,--lena,--lenb)

{

ta=lena<0?0:a4[lena];

tb=lenb<0?0:b4[lenb];

a4[i]=ta+tb+t;

t=a4[i]/10000;

a4[i]%=10000;

}

int z=1;

if(t)

{

printf("1");

printf("%04d",a4[0]);

}

else

{

for(z=0; z<m; ++z)

if(a4[z]!=0)

break;

printf("%d",a4[z++]);

}

for(int i=z; i<m; ++i)

printf("%04d",a4[i]);

printf("\n");

}

int main()

{

while((lena=read(a4))!=-1)

{

lenb=read(b4);

cal();

}

}

**单调队列：**

push:平均O(1),查询:O（1）

struct Node

{

int val,t; //val是值，t是加入的时间

Node(int val\_,int t\_):val(val\_),t(t\_){}

Node(){}

};

Node maxq[maxn];

int maxf,maxr;

void push(int n,int t)

{

if(maxq[maxf].t<t-k+1) //如果头超出范围，则pop

maxf++;

for(int i=maxr-1;i>=maxf;--i)//把队列中小于要push进去的值的点都pop掉

if(maxq[i].val<n)

maxr--;

else break;

maxq[maxr++]=Node(n,t);

}

**线段树：**

struct Node

{

int l,r; //表示的线段范围

int color; //存的值，这里为二进制表示的存在color数（状态压缩）

bool lazy; //lazy标记

};

Node tree[4\*maxn];

void buildtree(int l,int r,int n)

{

tree[n].r=r;

tree[n].l=l;

tree[n].color=1;

tree[n].lazy=0;

if(r==l)

{

return;

}

int m=(l+r)>>1;

buildtree(l,m,n<<1);

buildtree(m+1,r,n<<1|1);

}

void pushDown(int n)

{

if(tree[n].lazy)

{

tree[n<<1].color=tree[n<<1|1].color=tree[n].color;

tree[n].lazy=0;

tree[n<<1].lazy=tree[n<<1|1].lazy=1;

}

}

void pushUp(int n)

{

tree[n].color=tree[n<<1].color|tree[n<<1|1].color;

}

void change(int l,int r,int col,int n)

{

if(tree[n].l==l && tree[n].r==r)

{

tree[n].color=col;

if(l!=r)

tree[n].lazy=1;

return;

}

if(tree[n].color==col) return;

pushDown(n);

int m=(tree[n].l+tree[n].r)>>1;

if(r<=m) change(l,r,col,n<<1);

else if(l<=m && r>m)

{

change(l,m,col,n<<1);

change(m+1,r,col,n<<1|1);

}

else change(l,r,col,n<<1|1);

pushUp(n);

}

inline int toBin(const int n) //把颜色转换为二进制表示

{

return 1<<(n-1);

}

int count(int n) //计算一个color里有多少种颜色

{

int res=0;

while(n)

{

if(n&1) res++;

n>>=1;

}

return res;

}