基础

1.DFS

```
bool vst[maxn][maxn];                          // 访问标记
int map[maxn][maxn];                           // 坐标范围
int dir[4][2] = {0, 1, 0, -1, 1, 0, -1, 0}; // 方向向量，(x,y)周围的四个方向

bool CheckEdge(int x, int y) // 边界条件和约束条件的判断
{
        if (!vst[x][y] &&  ... ) // 满足条件
                return 1;
        else // 与约束条件冲突
                return 0;
}
void dfs(int x, int y)
{
        vst[x][y] = 1;       // 标记该节点被访问过
        if (map[x][y] == G) // 出现目标态G
        {
                ..... // 做相应处理
                    return;
        }
        for (int i = 0; i < 4; i++)
        {
                if (CheckEdge(x + dir[i][0], y + dir[i][1])) // 按照规则生成下
一个节点
                        dfs(x + dir[i][0], y + dir[i][1]);
        }
        return; // 没有下层搜索节点，回溯
}
int main()
{
        ..... return 0;
}
```

2.BFS

```
bool vst[maxn][maxn]; // 访问标记
int dir[4][2]={0,1,0,-1,1,0,-1,0}; // 方向向量

struct State // BFS 队列中的状态数据结构
{
        int x,y; // 坐标位置
        int Step_Counter; // 搜索步数统计器
```

```
};
bool CheckState(State s) // 约束条件检验
{
        if(!vst[s.x][s.y] &&  ...) // 满足条件
                return 1;
        else // 约束条件冲突
                return 0;
}
void bfs(State st)
{
        queue<State> q;       // BFS 队列
        State now, next;      // 定义2 个状态，当前和下一个
        st.Step_Counter = 0; // 计数器清零
        q.push(st);           // 入队
        vst[st.x][st.y] = 1; // 访问标记
        while (!q.empty())
        {
                now = q.front(); // 取队首元素进行扩展
                if (now == G)    // 出现目标态，此时为Step_Counter 的最小值，可以退出即可
                {
                        ...... // 做相关处理
                            return;
                }
                for (int i = 0; i < 4; i++)
                {
                        next.x = now.x + dir[i][0]; // 按照规则生成下一个状态
                        next.y = now.y + dir[i][1];
                        next.Step_Counter = now.Step_Counter + 1; // 计数器加1
                        if (CheckState(next))                     // 如果状态满足约束条件则入队
                        {
                                q.push(next);
                                vst[next.x][next.y] = 1; //访问标记
                        }
                }
                q.pop(); // 队首元素出队
        }
        return;
}

int main()
{
        ...... return 0;
}
```

3.二分

1.找最大值中的最小

```
int main()
{
        int l;
        int r;
        while(l < r)
        {
                int mid = (l + r)/ 2;
                if(check())
                {
                        r = mid;  // 这里是 r = mid, 说明[l,mid]是合法范围
                }
                else
                {
                        l = mid + 1;    //  [l,mid]这个范围都不是合法范围，所以下一次查找直接从 l = mid + 1开始了
                }
                //最后的l,r是答案 因为 l == r ，最终就是答案。
        }

}
```

2.找最小值中的最大

```
int main()
{
        int l;
        int r;
        while(l < r)
        {
                int mid = (l + r + 1)/ 2;  // 这里要 l + r +1 要不然会死循环
                if(check())
                {
                        l = mid;          // mid这个位置 满足条件之后 查找 [mid , right]的位置， 所以l移到mid的位置
                }
                else
                {
                        r = mid - 1;      // [mid,r] 不满足条件， 所以要移到满足条件的一方， r = mid - 1
                }
        }
        //最后的l,r是答案 因为 l == r
}
```

3.找最小值最终返回r

```
int main()
{
        int l;
        int r;
        while(l ≤ r)
        {
                int mid = (l + r )/ 2;
                if(check())
                {
                        l = mid - 1;
                }
                else
                {
                        r = mid + 1;
                }
        }
        //最终 l == r + 1, 找最小值返回r，返回r位置

}
```

4.找最大值最终返回l

```
int main()
{
        int l;
        int r;
        while(l ≤ r)
        {
                int mid = (l + r )/ 2;
                if(check())
                {
                        l = mid - 1;
                }
                else
                {
                        r = mid + 1;
                }
}
        //最终 l == r + 1, 找最大值返回r
}
```

5.中间保留保存最佳值

```
int main()
{
        int l;
        int r;
        while(l ≤ r)
        {
                int mid = (l + r )/ 2;
                if(check())
                {
                        ans = mid;  // 这里需要保证 check()函数是找最佳值的位置
的。
                                                                //  这个位置已经满足一个位置了，
再往mid - 1位置找看是否还有更佳位置
                        l = mid - 1;
                }
                else
                {
                        r = mid + 1;
                }
}
        //答案是ans
}
```

4.枚举全排列

```
#include<iostream>
#include<cstdio>
using namespace std;

int N;
int a[10000],book[10000];

void dfs(int step) {
        if(step==N+1) {
                for(int i=1; i< =N; i++) {
                        printf("%5d",a[i]);
                        //cout<<a[i]<<' ';
                }
                cout<<endl;
                return ;
        }
        for(int i=1; i< =N; i++) {
                if(book[i]==0) {
                        a[step]=i;
```

```
                        book[i]=1;
                        dfs(step+1);
                        book[i]=0;
                }
        }
        return ;
}
int main() {
        cin>>N;
        for(int i=1; i< =N; i++) {
                a[i]=i;
                book[i]=0;
        }
        dfs(1);

        return 0;
}
```

5.高精度

```
#include<stdio.h>
#include<string>
#include<string.h>
#include<iostream>
using namespace std;
//compare比较函数：相等返回0，大于返回1，小于返回-1
int compare(string str1,string str2)
{
    if(str1.length()>str2.length()) return 1;
    else if(str1.length()<str2.length())  return -1;
    else return str1.compare(str2);
}
//高精度加法
//只能是两个正数相加
string add(string str1,string str2)//高精度加法
{
    string str;
    int len1=str1.length();
    int len2=str2.length();
    //前面补0，弄成长度相同
    if(len1<len2)
    {
        for(int i=1;i< =len2-len1;i++)
           str1="0"+str1;
    }
    else
    {
        for(int i=1;i< =len1-len2;i++)
```

```
            str2="0"+str2;
    }
    len1=str1.length();
    int cf=0;
    int temp;
    for(int i=len1-1;i> =0;i--)
    {
        temp=str1[i]-'0'+str2[i]-'0'+cf;
        cf=temp/10;
        temp%=10;
        str=char(temp+'0')+str;
    }
    if(cf! =0)  str=char(cf+'0')+str;
    return str;
}
//高精度减法
//只能是两个正数相减，而且要大减小
string sub(string str1,string str2)//高精度减法
{
    string str;
    int tmp=str1.length()-str2.length();
    int cf=0;
    for(int i=str2.length()-1;i> =0;i--)
    {
        if(str1[tmp+i]<str2[i]+cf)
        {
            str=char(str1[tmp+i]-str2[i]-cf+'0'+10)+str;
            cf=1;
        }
        else
        {
            str=char(str1[tmp+i]-str2[i]-cf+'0')+str;
            cf=0;
        }
    }
    for(int i=tmp-1;i> =0;i--)
    {
        if(str1[i]-cf> ='0')
        {
            str=char(str1[i]-cf)+str;
            cf=0;
        }
        else
        {
            str=char(str1[i]-cf+10)+str;
            cf=1;
        }
    }
    str.erase(0,str.find_first_not_of('0'));//去除结果中多余的前导0
    return str;
```

```
}
//高精度乘法
//只能是两个正数相乘
string mul(string str1,string str2)
{
    string str;
    int len1=str1.length();
    int len2=str2.length();
    string tempstr;
    for(int i=len2-1;i> =0;i--)
    {
        tempstr="";
        int temp=str2[i]-'0';
        int t=0;
        int cf=0;
        if(temp! =0)
        {
            for(int j=1;j< =len2-1-i;j++)
              tempstr+="0";
            for(int j=len1-1;j> =0;j--)
            {
                t=(temp*(str1[j]-'0')+cf)%10;
                cf=(temp*(str1[j]-'0')+cf)/10;
                tempstr=char(t+'0')+tempstr;
            }
            if(cf! =0) tempstr=char(cf+'0')+tempstr;
        }
        str=add(str,tempstr);
    }
    str.erase(0,str.find_first_not_of('0'));
    return str;
}
//高精度除法
//两个正数相除，商为quotient,余数为residue
//需要高精度减法和乘法
void div(string str1,string str2,string &quotient,string &residue)
{
    quotient=residue="";//清空
    if(str2=="0")//判断除数是否为0
    {
        quotient=residue="ERROR";
        return;
    }
    if(str1=="0")//判断被除数是否为0
    {
        quotient=residue="0";
        return;
    }
    int res=compare(str1,str2);
    if(res<0)
```

```
    {
        quotient="0";
        residue=str1;
        return;
    }
    else if(res==0)
    {
        quotient="1";
        residue="0";
        return;
    }
    else
    {
        int len1=str1.length();
        int len2=str2.length();
        string tempstr;
        tempstr.append(str1,0,len2-1);
        for(int i=len2-1;i<len1;i++)
        {
            tempstr=tempstr+str1[i];
            tempstr.erase(0,tempstr.find_first_not_of('0'));
            if(tempstr.empty())
              tempstr="0";
            for(char ch='9';ch> ='0';ch--)//试商
            {
                string str,tmp;
                str=str+ch;
                tmp=mul(str2,str);
                if(compare(tmp,tempstr)< =0)//试商成功
                {
                    quotient=quotient+ch;
                    tempstr=sub(tempstr,tmp);
                    break;
                }
            }
        }
        residue=tempstr;
    }
    quotient.erase(0,quotient.find_first_not_of('0'));
    if(quotient.empty()) quotient="0";
}
int main()
{
     string str1,str2;
     //string str3,str4;
     cin>>str1>>str2;
     //while()
     //{
         cout<<add(str1,str2)<<endl;
         //cout<<sub(str1,str2)<<endl;
```

```
        //cout<<mul(str1,str2)<<endl;
        //div(str1,str2,str3,str4);
        //cout<<str3<<"  "<<str4<<endl;
    //}
    return 0;
}
```

6. FFT加速高精乘

```
#include<bits/stdc++.h>
using namespace std;
//complex是stl自带的定义复数的容器
typedef complex<double> cp;
#define N 2097153
//pie表示圆周率π
const double pie=acos(-1);
int n;
cp a[N],b[N];
int rev[N],ans[N];
char s1[N],s2[N];
//读入优化
int read(){
        int sum=0,f=1;
        char ch=getchar();
        while(ch>'9'||ch<'0'){if(ch=='-')f=-1;ch=getchar();}
        while(ch> ='0'&&ch< ='9'){sum=(sum<<3)+(sum<<1)+ch-
'0';ch=getchar();}
        return sum*f;
}
//初始化每个位置最终到达的位置
{
    int len=1<<k;
        for(int i=0;i<len;i++)
        rev[i]=(rev[i>>1]>>1)|((i&1)<<(k-1));
}
//a表示要操作的系数，n表示序列长度
//若flag为1，则表示FFT，为-1则为IFFT(需要求倒数）
void fft(cp *a,int n,int flag){
    for(int i=0;i<n;i++)
        {
         //i小于rev[i]时才交换，防止同一个元素交换两次，回到它原来的位置。
          if(i<rev[i])swap(a[i],a[rev[i]]);
        }
        for(int h=1;h<n;h*=2)//h是准备合并序列的长度的二分之一
        {
        cp wn=exp(cp(0,flag*pie/h));//求单位根w_n^1
         for(int j=0;j<n;j+=h*2)//j表示合并到了哪一位
         {
```

```
            cp w(1,0);
             for(int k=j;k<j+h;k++)//只扫左半部分，得到右半部分的答案
             {
               cp x=a[k];
               cp y=w*a[k+h];
            a[k]=x+y;  //这两步是蝴蝶变换
            a[k+h]=x-y;
            w*=wn; //求w_n^k
              }
            }
            }
            //判断是否是FFT还是IFFT
            if(flag==-1)
            for(int i=0;i<n;i++)
        a[i]/=n;
}
int main(){
        n=read();
        scanf("%s%s",s1,s2);
        //读入的数的每一位看成多项式的一项，保存在复数的实部
    for(int i=0;i<n;i++)a[i]=(double)(s1[n-i-1]-'0');
        for(int i=0;i<n;i++)b[i]=(double)(s2[n-i-1]-'0');
        //k表示转化成二进制的位数
        int k=1,s=2;
        while((1<<k)<2*n-1)k++,s<<=1;
        init(k);
        //FFT 把a的系数表示转化为点值表示
    fft(a,s,1);
    //FFT 把b的系数表示转化为点值表示
    fft(b,s,1);
    //FFT 两个多项式的点值表示相乘
    for(int i=0;i<s;i++)
    a[i]*=b[i];
    //IFFT 把这个点值表示转化为系数表示
    fft(a,s,-1);
    //保存答案的每一位(注意进位）
    for(int i=0;i<s;i++)
    {
    //取实数四舍五入，此时虚数部分应当为0或由于浮点误差接近0
        ans[i]+=(int)(a[i].real()+0.5);
        ans[i+1]+=ans[i]/10;
        ans[i]%=10;
        }
        while(!ans[s]&&s> -1)s--;
        if(s==-1)printf("0");
        else
        for(int i=s;i> =0;i--)
        printf("%d",ans[i]);
        return 0;
}
```