```cpp
///二叉树相关

///知道先序、中序，求后序
 struct TreeNode
 {
    struct TreeNode* left;
    struct TreeNode* right;
    char  elem;
 };
 void BinaryTreeFromOrderings(char* inorder, char* preorder, int length)
 {
   if(length == 0)
     {
       //cout<<"invalid length";
       return;
     }
    TreeNode* node = new TreeNode;//Noice that [new] should be written out.
    node->elem = *preorder;
    int rootIndex = 0;
    for(;rootIndex < length; rootIndex++)
      {
        if(inorder[rootIndex] == *preorder)
        break;
      }
    //Left
    BinaryTreeFromOrderings(inorder, preorder +1, rootIndex);
    //Right
    BinaryTreeFromOrderings(inorder + rootIndex + 1, preorder + rootIndex + 1, length - (rootIndex + 1));
    cout<<node->elem<<endl;
    return;
 }
 int main(int argc, char* argv[])
 {
     printf("Hello World!\n");
     char* pr="GDAFEMHZ";
     char* in="ADEFGHMZ";
     BinaryTreeFromOrderings(in, pr, 8);
     printf("\n");
     return 0;
 }


///知道中序、后序，求先序
struct TreeNode
{
    struct TreeNode* left;
    struct TreeNode* right;
    char  elem;
};
TreeNode* BinaryTreeFromOrderings(char* inorder, char* aftorder, int length)
{
    if(length == 0)
    {
        return NULL;
    }
    TreeNode* node = new TreeNode;//Noice that [new] should be written out.
    node->elem = *(aftorder+length-1);
    std::cout<<node->elem<<std::endl;
    int rootIndex = 0;
    for(;rootIndex < length; rootIndex++)//a variation of the loop
    {
        if(inorder[rootIndex] ==  *(aftorder+length-1))
            break;
    }
    node->left = BinaryTreeFromOrderings(inorder, aftorder , rootIndex);
    node->right = BinaryTreeFromOrderings(inorder + rootIndex + 1, aftorder + rootIndex , length - (rootIndex + 1));

    return node;
}
int main(int argc, char** argv)
{
    char* af="AEFDHZMG";
    char* in="ADEFGHMZ";
    BinaryTreeFromOrderings(in, af, 8);
    printf("\n");
```

```
    return 0;
}


///层序遍历
/*二叉树结构体，并且构建了有参构造函数*/
struct BinaryTree
{
    int vec;
    BinaryTree* left;
    BinaryTree* right;
    BinaryTree(int data)
        :vec(data), left(nullptr), right(nullptr){}
};
/*队列实现层序遍历*/
void printTree(BinaryTree* arr[])
{
    queue<BinaryTree*> rel; //定义一个队列，数据类型是二叉树指针，不要仅是int！！不然无法遍历
    rel.push(arr[0]);
    while (!rel.empty())
    {
        BinaryTree* front = rel.front();
        printf("%d\n", front->vec);
        rel.pop();                    //删除最前面的节点
        if (front->left != nullptr) //判断最前面的左节点是否为空，不是则放入队列
            rel.push(front->left);
        if (front->right != nullptr)//判断最前面的右节点是否为空，不是则放入队列
            rel.push(front->right);
    }
}
int main()
{
    /*构建二叉树*/
    BinaryTree* s_arr[6];
    s_arr[0] = new BinaryTree(0);
    s_arr[1] = new BinaryTree(1);
    s_arr[2] = new BinaryTree(2);
    s_arr[3] = new BinaryTree(3);
    s_arr[4] = new BinaryTree(4);
    s_arr[5] = new BinaryTree(5);
    s_arr[0]->left = s_arr[1];  //   0
    s_arr[0]->right = s_arr[2]; //  1  2
    s_arr[1]->left = s_arr[3];  // 3     5
    s_arr[3]->left = s_arr[4];  //4
    s_arr[2]->right = s_arr[5]; //所以层序遍历的结果为：0 1 2 3 5 4
    /*层次遍历打印所有节点*/
    printTree(s_arr);
    /*释放所有空间*/
    for (int i = 0; i < 6; i++)
        delete s_arr[i];
    return 0;
}


///镜面反转
void MirrorRecursively(TreeNode *pRoot)
{
    if((pRoot == NULL) || (pRoot->left == NULL && pRoot->right))
        return;

    TreeNode *pTemp = pRoot->left;
    pRoot->left = pRoot->right;
    pRoot->right = pTemp;

    if(pRoot->left)
        MirrorRecursively(pRoot->left);

    if(pRoot->right)
        MirrorRecursively(pRoot->right);
}

```

```cpp
///搜索相关

/*
迷宫的最短路径：
    输入迷宫的长宽，S为起点、G为终点，#是墙。
*/

#include <bits/stdc++.h>
using namespace std;

const int INF=100000000;
const int MAX=1000;

typedef pair<int, int> P;
int N, M;
int sx, sy;
int gx, gy;
char maze[MAX][MAX];
//设置数组，表示到起点的最近距离
int d[MAX][MAX];
int dx[]={1,0,-1,0};
int dy[]={0,1,0,-1};

int bfs()
{
    queue<P> que;
    for(int i=0; i<N; i++)
        for(int j=0; j<M; j++)
            d[i][j]=INF;
    //先把起点放进去，距离设置为零
    que.push(P(sx, sy));
    d[sx][sy]=0;

    //直到队列没有元素，就无路可走了
    while(que.size())
    {
        P p = que.front();
        que.pop();
        if(p.first==gx&&p.second==gy)
            break;

        for(int i=0; i<4; i++)
        {
            int nx=p.first+dx[i];
            int ny=p.second+dy[i];
            if(nx>=0&&nx<N&&ny>=0&&ny<M&&maze[nx][ny]!='#'&&d[nx][ny]==INF)
            {
                que.push(P(nx,ny));
                d[nx][ny] = d[p.first][p.second]+1;
            }
        }
    }
    return d[gx][gy];
}

int main()
{
    scanf("%d%d", &N, &M);
    getchar();
    for(int i=0; i<N; i++)
    {
        for(int j=0; j<M; j++)
            scanf("%c", &maze[i][j]);
        getchar();
    }
    for(int i=0; i<N; i++)
        for(int j=0; j<M; j++)
        {
            if(maze[i][j]=='S')
            {
                sx=i;
                sy=j;
            }
            if(maze[i][j]=='G')
            {
                gx=i;
                gy=j;
```

```
            }
        }

    int asw = bfs();
    printf("%d\n", asw);


    return 0;
}


/*
部分和问题：
    从整数a1, a2...中判断是否可以从中选出若干数，使得他们的和恰好为k。
*/

#include <bits/stdc++.h>
using namespace std;


int a[1000];
int n, k;

bool dfs(int i, int sum)
{
    //已经遍历到末尾
    if(i == n)
        return sum==k;
    //不加入a[i]
    if(dfs(i+1, sum))
        return true;//同对同错
    //加入a[i]
    if(dfs(i+1, sum+a[i]))
        return true;//同对同错
    //不管怎样都不行
    return false;

}

int main()
{
    scanf("%d", &n);
    for(int i=0; i<n; i++)
        scanf("%d", &a[i]);
    scanf("%d", &k);

    if(dfs(0, 0))
        printf("YES\n");
    else
        printf("NO\n");

    return 0;
}

```

```cpp
///数据结构相关

///并查集
int par[MAX_N];//父亲
int ran[MAX_N];//树的高度
//初始化n个元素
void init(int n)
{
    for(int i=0; i<n; i++)
    {
        par[i]=i;
        ran[i]=0;
    }
}
//查询树的根
int find(int x)
{
    if(par[x]==x)
        return x;
    else
        return par[x]=find(par[x]);
}
//合并x和y所属的集合
void unite(int x, int y)
{
    x=find(x);
    y=find(y);
    if(x==y)
        return ;
    if(ran[x]<ran[y])
        par[x]=y;
    else
    {
        par[y]=x;
        if(ran[x]==ran[y])
            ran[x]++;
    }
}
//判断x和y是否属于同一个集合
bool same(int x, int y)
{
    return find(x)==find(y);
}


///优先队列
priority_queue<int, vector<int>, greater<int> > que;
//pair也可以这么玩


///二分
//注意返回的是指针
upper_bound(a, a+n, k) - lower_bound(a, a+n, k) //a中等于k的个数
//假定一个解判断是否可行
bool C()
{

}
void solve()
{
    int lb=0, ub=INF;

    for(int i=0; i<100; i++)
    while(ub-lb>1)
    {
        int mid=(lb+ub)/2;
        if(C(mid))
            lb=mid;
        else
            ub=mid;
    }

    printf("%d\n", lb);
}


///迭代器
```

```
78  Template::iterator ite;
79  //注意set和map进行迭代都是按运算符<进行排序过的
80
81
82  ///set(multiset)
83  s.begin()       //  返回指向第一个元素的迭代器
84  s.clear()       //  清除所有元素
85  s.count()       //  返回某个值元素的个数
86  s.empty()       //  如果集合为空，返回true(真）
87  s.end()         //  返回指向最后一个元素之后的迭代器，不是最后一个元素
88  s.erase()       //  删除集合中的元素
89  s.find()        //  返回一个指向被查找到元素的迭代器
90  s.insert()      //  在集合中插入元素
91  s.lower_bound() //  返回指向大于（或等于）某值的第一个元素的迭代器
92  s.size()        //  集合中元素的数目
93  s.upper_bound() //  返回大于某个值元素的迭代器
94
95
96  ///map(multimap)
97  /*  向map中插入元素  */
98  m[key] = value; //  [key]操作是map很有特色的操作,如果在map中存在键值为key的元素对,
    则返回该元素对的值域部分,否则将会创建一个键值为key的元素对,值域为默认值。所以可以用该操作向map中插入
    元素对或修改已经存在的元素对的值域部分。
99  m.insert(make_pair(key, value));    //
    也可以直接调用insert方法插入元素对,insert操作会返回一个pair,当map中没有与key相匹配的键值时,其first是
    指向插入元素对的迭代器,其second为true;若map中已经存在与key相等的键值时,其first是指向该元素对的迭代器
    ,second为false。
100
101 /*  查找元素  */
102 int i = m[key]; //
    要注意的是,当与该键值相匹配的元素对不存在时,会创建键值为key（当另一个元素是整形时，m[key]=0）的元素
    对。
103 map<string, int>::iterator it = m.find(key);    //
    如果map中存在与key相匹配的键值时,find操作将返回指向该元素对的迭代器,否则,返回的迭代器等于map的end()(
    参见vector中提到的begin()和end()操作)。
104
105 /*  删除元素  */
106 m.erase(key);   //  删除与指定key键值相匹配的元素对,并返回被删除的元素的个数。
107 m.erase(it);    //  删除由迭代器it所指定的元素对,并返回指向下一个元素对的迭代器。
108
109 /*  其他操作  */
110 m.size();       //  返回元素个数
111 m.empty();      //  判断是否为空
112 m.clear();      //  清空所有元素
113
114 ///vector
115 s[i]                //  直接以下标方式访问容器中的元素
116 s.front()           //  返回首元素
117 s.back()            //  返回尾元素
118 s.push_back(x)      //  向表尾插入元素x
119 s.size()            //  返回表长
120 s.empty()           //  表为空时，返回真，否则返回假
121 s.pop_back()        //  删除表尾元素
122 s.begin()           //  返回指向首元素的随机存取迭代器
123 s.end()             //  返回指向尾元素的下一个位置的随机存取迭代器
124 reverse(s.begin(), s.end()); //  反转
125 s.insert(it, val)   //  向迭代器it指向的元素前插入新元素val
126 s.insert(it, n, val)//  向迭代器it指向的元素前插入n个新元素val
127 s.insert(it, first, last)
128 //  将由迭代器first和last所指定的序列[first, last)插入到迭代器it指向的元素前面
129 s.erase(it)         //  删除由迭代器it所指向的元素
130 s.erase(first, last)//  删除由迭代器first和last所指定的序列[first, last)
131 s.clear()           //  删除容器中的所有元素
132 s.swap(v)           //  将s与另一个vector对象进行交换
133
```

```cpp
///数学相关


///辗转相除法
int gcd(int a, int b)
{
    if(b==0)
        return a;
    return gcd(b, a%b);
}


///杨辉三角
long long int MOD = 1e9+7;
long long int c[2010][2010];


    for(int i = 0; i < 2005; i++)
    {
        c[i][0] = 1;
        c[i][i] = 1;
    }
    for(int i = 1; i < 2005; i++)
        for(int j = 1; j < i; j++)
            c[i][j] = (c[i-1][j-1] + c[i-1][j]) % MOD;


///快速幂运算
typedef long long ll;
ll mod_pow(ll x, ll n, ll mod)
{
    ll res=1;
    while(n>0)
    {
        if(n&1)
            res=res*x%mod; //如果二进制最低位为1，则乘上x^(2^i)
        x=x*x%mod; //将x平方
        n>>=1;
    }
    return res;
}


///埃氏筛法(Sieve of Eratosthenes)
int prime[MAX_N]; //第i个素数，此为存放素数的数组
bool is_prime[MAX_N]; //is_prime[]为true时为素数

//返回n以内素数的个数
int sieveErato(int n)
{
    int p=0;
    for(int i=0; i<=n; i++)
        is_prime[i]=true;
    is_prime[0]=false;
    is_prime[1]=false;
    for(int i=2; i<=n; i++)
    {
        if(is_prime[i])
        {
            prime[p++]=i;
            for(int j=2*i; j<=n; j+=i)
                is_prime[j]=false;
        }
    }
    return p;
}


///素性测试:
bool is_prime(int n)
{
    for(int i=2; i*i<=n; i++)
    {
        if(n%i==0)
            return false;
    }
    return n!=1; //1是例外
```

```
}

///约数枚举：将约数存入一个vector中
vector<int> divisor(int n)
{
    vector<int> res;
    for(int i=1; i*i<=n; i++)
    {
        if(n%i==0)
        {
            res.push_back(i);
            if(i!=n/i)
                res.push_back(n/i);
        }
    }
    return res;
}

///整数分解：将分解的数存入map的下标中，map值为因子的个数
map<int, int> prime_factor(int n)
{
    map<int, int> res;
    for(int i=2; i*i<=n; i++)
    {
        while(n%i==0)
        {
            ++res[i];
            n/=i;
        }
    }
    if(n!=1)
        res[n]=1;
    return res;
}

/*
多重集组合数问题：
    有n种物品，第i种物品有ai个。不同种类的物品可以互相区分但相同种类的不行，
    从其中取出m个，有多少种取法？求方法数模M的值。
*/

#include <cstdio>
#include <algorithm>
using namespace std;

int n, m;
int a[MAX_N];

int dp[MAX_N][MAX_M];

void solve()
{
    //一个都不取得方法总是只有一种
    for(int i=0; i<=n; i++)
        dp[i][0]=1;
    for(int i=0; i<n; i++)
        for(int j=1; j<=m; j++)
            if(j-1-a[i+1]>=0)
            //在有取余的情况下，要避免减法运算的结果出现负数
                dp[i+1][j]=(dp[i+1][j-1]+dp[i][j]-dp[i][j-1-a[i+1]]+M)%M;
            else
                dp[i+1][j]=(dp[i+1][j-1]+dp[i][j])%M;
}


/*
划分数问题：
    有n个无区别的物品，将它们划分成不超过m组，求划分方法数模M的余数。
    1<=m<=n<=1000, 2<=M<=10000
    (被称为n的m划分)
*/
#include <cstdio>
#include <algorithm>
using namespace std;
const int MAX_M=1007, MAX_N=1007;
const int M=1e9+7;
```

```

int n, m;
int dp[MAX_M][MAX_N];

void solve()
{
    dp[0][0]=1;
    for(int i=1; i<=m; i++)
        for(int j=0; j<=n; j++)
        {
            if(j-i>=0)
                dp[i][j]=(dp[i-1][j]+dp[i][j-1])%M;
            else
                dp[i][j]=dp[i-1][j];
        }
}

int main()
{
    while(scanf("%d%d", &n, &m)!=EOF)
    {
        solve();
        printf("%d %d\n", dp[m][n], dp[m-1][n]);
    }

    return 0;
}


/*
加练题：Joseph_POJ 1012
    经典的约瑟夫问题，找出递推式：
    dp[i]表示为第i轮杀掉谁，
    dp[i]=(dp[i-1]+m-1)%(n-i+1).
*/
#include <cstdio>
#include <cstring>
#include <iostream>
#include <algorithm>
using namespace std;

int J[14];
int dp[29];

int main()
{
    for(int k=1; k<14; k++)
    {
        int total=2*k;
        int m=1;
        for(int i=1; i<=k; i++)
        {
            dp[i]=(dp[i-1]+m-1)%(total-i+1);
            if(dp[i]<=k-1)
            {
                m++;
                i=0;
            }
        }
        J[k]=m;
    }

    int k;
    while(scanf("%d", &k)!=EOF)
    {
        if(k==0)
            break;

        printf("%d\n", J[k]);
    }

    return 0;
}

```

```cpp
///图论相关


/*
单源最短路问题(Bellman-Ford Algorithm):
    单源最短路是固定一个起点，求它到其他所有点之间的距离的最短路。(和两点间最短路问题复杂度一样)
    常用于求解有负边的情况。
*/
struct edge{
    int from, to, cost;
}es[MAX_E];

int d[MAX_V]; //最短距离
int V, E;

void shortest_path(int s)
{
    for(int i=0; i<V; i++)
        d[i]=INF;
    d[s]=0;
    while(true) //如果不存在负圈，复杂度是O(|V|X|E|)
    {
        bool update=false;
        for(int i=0; i<E; i++)
        {
            edge e=es[i];
            if(d[e.from!=INF && d[e.to]>d[e.from]+e.cost])
            {
                d[e.to]=d[e.from]+e.cost;
                update=true;
            }
        }
        if(!update)
            break;
    }
}


/*
单源最短路问题(Dijkstra Algorithm):
    由于不存在负边，找到d[i]最小的顶点就是最短距离已经确定的顶点。
*/
struct edge {
    int to, cost;
};
typedef pair<int, int> P;//利用first表示最短距离，second表示顶点的编号

int V;
vector<edge> G[MAX_V]; //下标充当from
int d[MAX_V];

void dijkstra(int s)
{
    //按照first从小到大顺序排列，先取出最短距离和对应的顶点
    priority_queue<P, vector<P>, greater<P> > que;
    fill(d, d+V, INF);
    d[s]=0;
    que.push(P(0,s));

    while(que.size()) //复杂度为O(|E|Xlog|V|)
    {
        P p = que.top();
        que.pop();
        int v=p.second;
        if(d[v]<p.first)
            continue;
        for(int i=0; i<G[v].size(); i++)
        {
            edge e=G[v][i];
            if(d[e.to]>d[v]+e.cost)
            {
                d[e.to]=d[v]+e.cost;
                que.push(P(d[e.to],e.to));
            }
        }
    }
}
```

```cpp


/*
任意两点间最短路问题(Floyd-Warshall Algorithm):
    用DP求解，递推式为：
d[0][i][j]=cost[i][j]
d[k][i][j]=min(d[k-1][i][j], d[k-1][i][k]+d[k-1][k][j])
依旧可以改进为d[i][j]的形式。
    复杂度：O(|V|^3)。
    可以处理边是负数的问题；而判断是否有负圈，只需要判断是否存在d[i][i]<0。
*/
#include <bits/stdc++.h>
using namespace std;

const int MAX_V = 107; //弗法能不能用都看的是顶点数大不大
const int INF = 100007;

int d[MAX_V][MAX_V]; //表示边的权值（不存在的时候设为INF，无负边的时候设d[i][i]=0）
int V, E;

void warshall_floyd()
{
    for(int k=0; k<V; k++)
        for(int i=0; i<V; i++)
            for(int j=0; j<V; j++)
                d[i][j]=min(d[i][j], d[i][k]+d[k][j]);
}

int main()
{
    scanf("%d%d", &V, &E);

    for(int i=0; i<V; i++)
    {
        for(int j=0; j<V; j++)
            d[i][j]=INF;
        d[i][i]=0;
    }


    for(int i=0; i<E; i++)
    {
        int buf1, buf2, buf3;
        scanf("%d%d%d", &buf1, &buf2, &buf3);
        d[buf1-1][buf2-1]=buf3;
        d[buf2-1][buf1-1]=buf3;
    }


    warshall_floyd();
    printf("%d\n", d[0][V-1]);

    return 0;
}


/*
最小生成树(Minimum Spanning Tree)问题(Prim Algorithm):
    Prim和Dijkstra很类似，都是从顶点出发不断加边的过程(直到顶点集合的数量X==V)；
    查找最小权值的边mincost[]同Dijkstra一样用优先队列维护。
*/
#include <bits/stdc++.h>
using namespace std;

int cost[MAX_V][MAX_V];
int mincost[MAX_V];
bool used[MAX_V];
int V;

int prim()
{
    for(int i=0; i<V; i++)
    {
        mincost[i]=INF;
        used[i]=false;
    }
    mincost[0]=0;
```

```
155       int res=0;
156
157       while(true)
158       {
159           int v=-1;
160           //从不属于x的顶点中选取从x到其权值最小的顶点
161           for(int u=0; u<V; u++)
162           {
163               if(!used[u] && (v==-1||mincost[u]<mincost[v]))
164                   v=u;
165           }
166
167           if(v==-1)
168               break;
169           used[v]=true; //把顶点v加入x
170           res+=mincost[v]; //把边的长度加入到结果里
171
172           for(int u=0; u<V; u++)
173           {
174               mincost[u]=min(mincost[u], cost[v][u]);
175           }
176       }
177
178       return res;
179   }
180
181
182   ///树的直径
183   bfs(1);
184   int Md=-1, Mn;
185   for(int i=0; i<V; i++)
186   {
187       if(Md<dis[i])
188       {
189           Md=dis[i];
190           Mn=i;
191       }
192   }
193   bfs(Mn);
194   int L=-1;
195   for(int i=0; i<V; i++)
196   {
197       if(L<dis[i])
198           L=dis[i];
199   }
200
```

```cpp
///博弈论相关

/**
非常重要的一种模型：
    Alice先手，Bob后手；
    N堆石子Xi，
    K种取法Ai，
    利用Grundy数转化为Nim问题，异或为零必败。
*/
#include <cstdio>
#include <set>
#include <algorithm>
using namespace std;

const int MAX_N=1000007, MAX_K=107, MAX_X=10007;

int N, K, X[MAX_N], A[MAX_K];

//利用动态规划求grundy值的数组
int grundy[MAX_X];

int main()
{
    //输入
    scanf("%d%d", &N, &K);
    for(int i=0; i<K; i++)
        scanf("%d", &A[i]);
    for(int i=0; i<N; i++)
        scanf("%d", &X[i]);

    //轮到自己时剩余0必败
    grundy[0]=0;

    //计算grundy值
    int max_x=*max_element(X, X+N);
    for(int j=1; j<=max_x; j++)
    {
        set<int> s;
        for(int i=0; i<K; i++)
            if(A[i]<=j)
                s.insert(grundy[j-A[i]]);
        int g=0;
        while(s.count(g)!=0)
            g++;
        grundy[j]=g;
    }

    //Nim判断胜负
    int x=0;
    for(int i=0; i<N; i++)
        x^=grundy[X[i]];
    if(x!=0)
        puts("Alice");
    else
        puts("Bob");

    return 0;
}

```

```cpp
///计算几何相关


/** 多边形的面积：
 *  要求按照逆时针方向输入多边形顶点
 *  可以是凸多边形或凹多边形
 */
double area_of_polygon(int vcount, Lpoint plg[])
{
    int i;
    double s;
    if (vcount < 3)
    {
        return 0;
    }
    s = plg[0].y * (plg[vcount - 1].x - plg[1].x);
    for (i = 1; i < vcount; i++)
    {
        s += plg[i].y * (plg[(i - 1)].x - plg[(i + 1) % vcount].x);
    }
    return s / 2;
}


///求圆相交的面积
double Area_of_overlap(Point c1, double r1, Point c2, double r2)
{
    double d = dist(c1, c2);
    if (r1 + r2 < d + eps)
    {
        return 0;
    }
    if (d < fabs(r1 - r2) + eps)
    {
        double r = min(r1, r2);
        return PI * r * r;
    }
    double x = (d * d + r1 * r1 - r2 * r2) / (2 * d);
    double t1 = acos(x / r1);
    double t2 = acos((d - x) / r2);
    return r1 * r1 * t1 + r2 * r2 * t2 - d * r1 * sin(t1);
}



/** 求矩形相交的面积
 *  x[]、y[]存储矩阵对角线顶点（只需要任意一条）
 */
double Area_of_overlap_rec(double x[], double y[])
{
    //  将两个矩形全部统一为主对角线
    sort(x, x + 2);
    sort(x + 2, x + 4);
    sort(y , y + 2);
    sort(y + 2, y + 4);

    if (x[1] <= x[2] || x[0] >= x[3] || y[0] >= y[3] || y[1] <= y[2])   //  相离
    {
        return 0.0;
    }
    else
    {
        sort(x, x + 4);
        sort(y, y + 4);
        return (x[2] - x[1]) * (y[2] - y[1]);
    }
}

```

```cpp
///字符串相关

///匹配
char str[]="1234xyz";
char *str1=strstr(str,"34");
cout << str1 << endl;


/*
最长公共子序列问题(LCS):
    注：求的是子数列，不是连续子数列；而且只要求求了长度。
    复杂度：O(s.size() X t.size())
*/
#include <bits/stdc++.h>
using namespace std;

string s, t;

int dp[1007][1007];

void solve()
{
    for(int i=0; i<s.size(); i++)
        for(int j=0; j<t.size(); j++)
            if(s[i]==t[j])
                dp[i+1][j+1]=dp[i][j]+1;
            else
                dp[i+1][j+1]=max(dp[i+1][j], dp[i][j+1]);

    printf("%d\n", dp[s.size()][t.size()]);

}

int main()
{
    getline(cin, s);
    getline(cin, t);

    solve();

    return 0;

}


/*
最长上升子序列(LIS, Longest Increasing Subsequence)问题:
    solve1(): 定义dp[i]为以ai为末尾的最长上升子数列的长度，O(n^2);
    solve2(): 定义dp[i]为长度为i+1的上升子数列中末尾元素的最小值;
        由于知道这种dp[]除了INF之外是单调递增的，可以利用二分搜索得到更快的O(nlogn)。
*/
#include <bits/stdc++.h>
using namespace std;

const int MAX_N=1007, INF=0x3f3f3f3f;

int n;
int a[MAX_N];

int dp[MAX_N];

void solve1()
{
    int res=0;
    for(int i=0; i<n; i++)
    {
        dp[i]=1;
        for(int j=0; j<i; j++)
        {
            if(a[j]<a[i])
                dp[i]=max(dp[i], dp[j]+1);
        }
        res=max(res, dp[i]);
    }
    printf("%d\n", res);
}

```

```cpp
void solve2()
{
    fill(dp, dp+n, INF); //INF 改为 -1 变为LDS
    for(int i=0; i<n; i++)
    {
        *lower_bound(dp, dp+n, a[i]) = a[i]; //, greater<int>() 变为LDS; 改为upper变成最长不下降。
    }
    printf("%d\n", lower_bound(dp, dp+n, INF)-dp); //, greater<int>(); INF 改为 -1 变为LDS
}

int main()
{
    scanf("%d", &n);
    for(int i=0; i<n; i++)
        scanf("%d", &a[i]);

    //solve1();
    solve2();

    return 0;
}


/*
最长公共子序列问题(LCS):
    输出LCS
*/
#include <cstdio>
#include <cstring>
#include <iostream>
using namespace std;

#define MAX 1001
#define max(x,y) ((x)>(y)?(x):(y))

char s1[MAX], s2[MAX];
int maxLen[MAX][MAX];
char ans[MAX];

int main()
{
    int k = 0;
    scanf("%s%s", s1,s2);
    int len1 = strlen(s1);
    int len2 = strlen(s2);

    for (int i = 1; i <= len1; i++)
    {
        for (int j = 1; j <= len2; j++)
        {
            if (s1[i-1] == s2[j-1])
                maxLen[i][j] = maxLen[i - 1][j - 1] + 1;
            else maxLen[i][j] = max(maxLen[i][j - 1], maxLen[i-1][j]);
        }
    }

    int i = len1;
    int j = len2;
    while(i)
    {
        if (maxLen[i][j] > maxLen[i - 1][j])
        {
            if (maxLen[i][j] > maxLen[i][j - 1])
                ans[maxLen[i][j] - 1] = s1[i - 1];
            else i++;    //左平移
            j--;    //减小一个规模
        }
        i--;    //上平移
    }

    printf("%s\n", ans);
    return 0;
}


/*
最长回文子串:
```

```
    Manacher算法
*/
#include<iostream>
#include<string.h>
#include<algorithm>
using namespace std;

char s[1000];
char s_new[2000];
int p[2000];

int Init()
{
    int len = strlen(s);
    s_new[0] = '$';
    s_new[1] = '#';
    int j = 2;

    for (int i = 0; i < len; i++)
    {
        s_new[j++] = s[i];
        s_new[j++] = '#';
    }

    s_new[j] = '\0';  //别忘了哦

    return j;  //返回s_new的长度
}

int Manacher()
{
    int len = Init();  //取得新字符串长度并完成向s_new的转换
    int maxLen = -1;   //最长回文长度

    int id;
    int mx = 0;

    for (int i = 1; i < len; i++)
    {
        if (i < mx)
            p[i] = min(p[2 * id - i], mx - i);
        else
            p[i] = 1;

        while (s_new[i - p[i]] == s_new[i + p[i]])  //不需边界判断，因为左有'$',右有'\0'
            p[i]++;


        if (mx < i + p[i])
//我们每走一步i，都要和mx比较，我们希望mx尽可能的远，这样才能更有机会执行if (i <
mx)这句代码，从而提高效率
        {
            id = i;
            mx = i + p[i];
        }

        maxLen = max(maxLen, p[i] - 1);
    }

    return maxLen;
}

int main()
{
    while (printf("请输入字符串: \n"))
    {
        scanf("%s", s);
        printf("最长回文长度为 %d\n\n", Manacher());
    }

    return 0;
}
```