1.

喜串可以先将两个字符串进行比较($O(n)$)，若是不同，能拆分的话，拆分成两个子串，分两种情况，每种情况分别匹配，若是有一种情况满足喜串的定义，则返回true， 两种都不符合，返回false,

最坏的情况下需要子情况进行4次匹配，即a1b1, a2b2, a1b2, a2b1四种情况， 则 $T(n) = 4T(n/2) + O(n)$, 最坏复杂度为 $O(n^2)$， 但是每次的比较相同时间最坏情况才是n，并且每次都要4次匹配也不是几乎都发生，奇数也不会继续递归下去，所以这是一个比较宽松的界限。

以下是AC代码

```
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;

string a, b;

bool isEqual(int l1, int l2, int len)
{
    int lim = l1 + len;
    for(; l1 < lim; l1++, l2++)
        if(a[l1] != b[l2])
            return false;
    return true;
}
bool check(int l1, int l2, int len)
{
    if(isEqual(l1, l2, len)) return true;
    if(len & 1) return false;
    if(check(l1, l2 + len / 2, len / 2) && check(l1 + len / 2, l2, len / 2)) return
true;
    if(check(l1, l2, len / 2) && check(l1 + len / 2, l2 + len / 2, len / 2)) return
true;
    return false;
}

int main()
{
    cin>>a>>b;
    int len = a.size();
    if(check(0, 0, len)) printf("Yes\n");
    else printf("No\n");
}
```

喜串和最近点对已在EOJ通过

2.

(a) 充分性

当某个m×n array 为Monge arrays时，可以取 $k = i + 1,\ l = j + 1$

得到 $A[i, j] + A[i + 1, j + 1] \le A[i + 1, j] + A[i, j + 1]$， 充分性成立

必要性

先证明对任意 $i,\ j,\ d_1,\ 1 \le j < d1 \le n$, 都有 $A[i, j] + A[i + 1, d1] \le A[i, d1] + A[i + 1, j]$ ①

当$d_1=i+1$时，有 $A[i,j]+A[i+1,j+1]\le A[i+1,j]+A[i,j+1]$

假设当 $d1=k$ 时，等式成立， $A[i,j]+A[i+1,k]\le A[i,k]+A[i+1,j]$

又有$A[i,k]+A[i+1,k+1]\le A[i,k+1]+A[i+1,k]$

将两式合并，有$A[i,j]+A[i+1,k+1]\le A[i,k+1]+A[i+1,j]$

所以 $d1=k+1$ 成立

由数学归纳法得证明 ① 成立

再证明对于任意i，j，k，l，$1\le i<k\le n, 1\le j<l\le n$，都有$A[i,j]+A[k,l]\le A[i,l]+A[k,j]$

当 $k=i+1$ 时， 由 ① 知$A[i,j]+A[i+1,l]\le A[i,l]+A[i+1,j]$

假设当 $k=m$ 时，等式成立， $A[i,j]+A[m,l]\le A[i,l]+A[m,j]$

又有$A[m,j]+A[m+1,l]\le A[m,l]+A[m+1,j]$

将两式合并，有$A[i,j]+A[m+1,l]\le A[i,l]+A[m+1,j]$

所以 $k=m+1$ 成立，必要性成立

(b) 可知 $A[1,2]+A[2,3]=23+7=30$，$A[1,3]+A[2,2]=22+6=28$， 这里不满足定义

可以将$A[2,3]=5$, 这样改便满足a的性质

(c) 假设存在行i, 有$f(i)>f(i+1)$

则$A[i,f(i+1)]+A[i+1,f(i)]\le A[i,f(i)]+A[i+1,f(i+1)]$，由于f(i)表示最小值出现的最左下标，则$A[i,f(i+1)]$ > $A[i,f(i)]$(等号取不到)， 则$A[i+1,f(i)]$ < $A[i+1,f(i+1)]$, 则f(i+1)下标对应的值并不是该行的最小值，产生矛盾

所以$f(1)\le f(2)\le\ldots\le f(m)$

(d)当偶数情况已经知道了之后，如知道了f(0), f(2), 那么f(1)只需要在[f(0), f(2)]之间，那么这些奇数列所需要检索的范围其实最后就不会超过n，然后这个过程中将奇偶合并需要m步， 所以需要时间为$O(m+n)$

(e)由(e)， 其递推式为 $T(m)=T(m/2)+O(m+n)$

进行迭代：
$T(m)=T(m/2)+O(m+n)=T(m/4)+O(m/2+n)+O(m+n)=\ldots=O(2m+nlogm)=O(m+nlogm)$

3.

定义了一个矩阵类，里面定义了加法减法乘法的运算来便于Strassen算法的实现

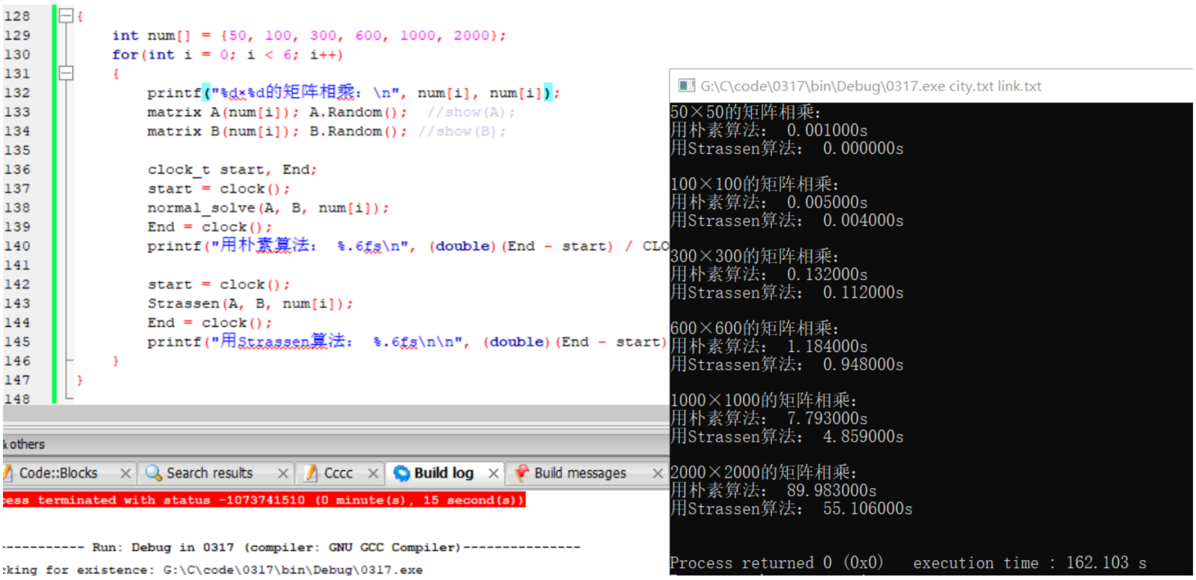
相应大小的矩阵使用随机数生成，根据书上的公式，首先将矩阵拆分成8个子矩阵，然后进行7次乘法操作，并合并成结果矩阵，具体实现展示在以下代码中：

```
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;

class matrix
{
```

```
public:
    int** arr;                  //存储矩阵的指针
    int n;
    matrix(int n0)                    //构造函数
    {
       n = n0;
       arr = new int*[n];
       for( int i = 0; i < n; i++ )
          arr[i] = new int[n];
    }

    ~matrix()                         //析构函数
    {
        for(int i = 0; i < n; i++)
            delete [] arr[i];
    }

    matrix operator+(const matrix &B)        //加法操作
    {
        matrix tmp(n);
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++)
                tmp.arr[i][j] = arr[i][j] + B.arr[i][j];
        return tmp;
    }

    matrix operator-(const matrix &B)           //减法操作
    {
        matrix tmp(n);
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++)
                tmp.arr[i][j] = arr[i][j] - B.arr[i][j];
        return tmp;
    }

    matrix operator*(const matrix &B)         //乘法操作
    {
        matrix tmp(n);
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++)
            {
                tmp.arr[i][j] = 0;
                for(int k = 0; k < n; k++)
                    tmp.arr[i][j] += arr[i][k] * B.arr[k][j];
            }
        return tmp;
    }

    matrix extract(int x, int y, int len)        //提取出子矩阵
    {
        matrix tmp(len);
        for(int i = 0; i < len; i++)
            for(int j = 0; j < len; j++)
                tmp.arr[i][j] = arr[x+i][y+j];
        return tmp;
    }

    void Fill(int x, int y, int len, matrix m)       //合并成大矩阵
    {
        for(int i = 0; i < len; i++)
```

```
            for(int j = 0; j < len; j++)
                arr[x+i][y+j] = m.arr[i][j];
    }

    void Random()                                   //随机数生成矩阵
    {
        srand((int)time(NULL));
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++)
                arr[i][j] = rand() % 1000;
    }
};

void show(matrix& m)                           //展示结果
{
    for(int i = 0; i < m.n; i++)
    {
        for(int j = 0; j < m.n; j++)
            printf("%-8d", m.arr[i][j]);
        putchar('\n');
    }
    putchar('\n');
}

void normal_solve(matrix& A, matrix& B, int n)    //朴素方法
{
    matrix ans = A*B;
    //show(ans);
}

void Strassen(matrix& A, matrix& B, int n)
{
    matrix A11 = A.extract(0, 0, n / 2);           //先拆分成8个子矩阵
    matrix A12 = A.extract(0, n / 2, n / 2);
    matrix A21 = A.extract(n / 2, 0, n / 2);
    matrix A22 = A.extract(n / 2, n / 2, n / 2);

    matrix B11 = B.extract(0, 0, n / 2);
    matrix B12 = B.extract(0, n / 2, n / 2);
    matrix B21 = B.extract(n / 2, 0, n / 2);
    matrix B22 = B.extract(n / 2, n / 2, n / 2);

    matrix M1 = A11*(B12 - B22);                    //进行7次乘法运算
    matrix M2 = (A11 + A12)*B22;
    matrix M3 = (A21 + A22)*B11;
    matrix M4 = A22*(B21 - B11);
    matrix M5 = (A11 + A22)*(B11 + B22);
    matrix M6 = (A12 - A22)*(B21 + B22);
    matrix M7 = (A11 - A21)*(B11 + B12);

    matrix ans(n);                                    //合并
    ans.Fill(0, 0, n / 2, M5 + M4 - M2 + M6);
    ans.Fill(0, n / 2, n / 2, M1 + M2);
    ans.Fill(n / 2, 0, n / 2, M3 + M4);
    ans.Fill(n / 2, n / 2, n / 2, M5 + M1 - M3 - M7);

   // show(ans);
}

int main()
```

```
{
    int num[] = {50, 100, 300, 600, 1000, 2000};
    for(int i = 0; i < 6; i++)
    {
        printf("%dx%d的矩阵相乘：\n", num[i], num[i]);
        matrix A(num[i]); A.Random();  //show(A);
        matrix B(num[i]); B.Random(); //show(B);

        clock_t start, End;
        start = clock();
        normal_solve(A, B, num[i]);
        End = clock();
        printf("用朴素算法： %.6fs\n", (double)(End - start) / CLOCKS_PER_SEC);
 //记录时间

        start = clock();
        Strassen(A, B, num[i]);
        End = clock();
        printf("用Strassen算法： %.6fs\n\n", (double)(End - start) / CLOCKS_PER_SEC);
    }
}
```



4.

(1) 当确定了一个平民的身份后，则可以让这个平民在线性时间里鉴别所有人的身份(依次与其他人进行一次query)，所以我们的目标即寻找到一个平民

而也可以在线性时间立确定某个人的身份：若是他是平民，则至少有超过一半的平民会说他是平民，若他是狼人，则至少有超过一半的平民说他是狼人

若两人配对，则有如下结果：

①互说对方是平民，这种情况下要么两人都是平民，两人都是狼人

②一人说对方是狼一人说对方是平民，这种情况要么两人都是狼人，要么一个人是狼人一个人是平民

③互说对方是狼人，这种情况的可能和第②中相同

因此，若是两人的信息中如果有一条说对方是狼人，则这两个人至少有一个人是狼人

我们可以将所有玩家(若此时有t人)两两配对，若是把第二种情况和第三种情况出现的配对去掉，则被去掉的组合都至少有一个狼人，则剩下的组合中平民数量还是超过狼人；

若t为奇数，则那个未被配对的人可以在线性时间内确定身份，若他是平民，则目标达成，若他是狼人，则也去掉；这样的话剩下每一组再挑一个人出来，则就有t/2个人，这样一直递归操作下去....当t ≤4时，由于这样操作狼人数量肯定少于平民，则剩下的必为平民。

由上可得： $T(t) = T(t/2) + O(t/2 + n)$ (t/2是指再挑t个人，n表示若t为奇数，在线性时间内确定未配对那个)

最坏情况下递归 $logn$ 层，经过迭代，最坏复杂度为 O（ $nlogn)$

(2) 其实(1)中确定那个未配对的人，未必要让它和所有n个人配对，而只要与剩下的t个人配对就可以

这时候 $T(t) = T(t/2) + O(t/2 + t)$， 复杂度这样就最坏是 O（ n）

或者可以用随机算法，可以在$O(n)$ 时间里确定一个人是否为平民，而平民占比是超过一半的，随机挑一个人，若是10次操作的话，一次都没挑到平民的概率小于 $(0.5)^{10} < 0.1\%$, 多进行几次随机则概率更低，因此只要够随机，就可以经过 $O(cn)$ 次确定出一个平民的身份(最坏情况可能会$O(n^2)$)， 但是几乎不会出现)，所以随机算法也是线性的。