目录

第 5 课 KMP 算法 2
1. KMP 算法 2
1.1. 原理 2
2. 基础题目 11
2.1. Oulipo（poj3461） 11
2.2. Power Strings（poj2406） 12
3. 强化练习 13
3.1. 最大的数 13
4. 拓展练习 14
4.1. 最大的数 14
5. 课后作业 15
5.1. 复习作业 15
5.2. 课后训练 16
5.2.1. 彩票程序 16
5.2.2. 趣味跳跃 16

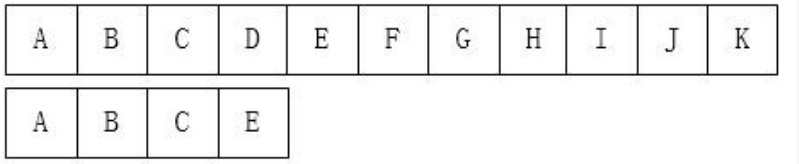
第 5 课 KMP 算法

1. KMP 算法

1.1. 原理

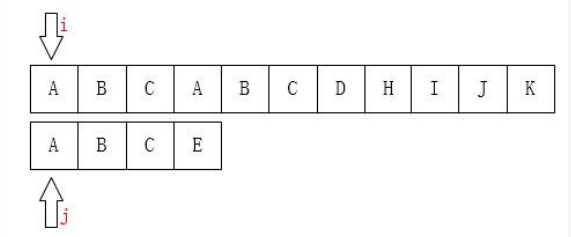
“KMP 算法是一种改进的字符串匹配算法，由 D.E.Knuth，J.H.Morris 和 V.R.Pratt 提出的，因此人们称它为克努特—莫里斯—普拉特操作（简称 KMP 算法）。KMP 算法的核心是利用匹配失败后的信息，尽量减少模式串与主串的匹配次数以达到快速匹配的目的。

KMP 算法要解决的问题就是在字符串（也叫主串）中的模式（pattern）定位问题。说简单点就是我们平时常说的关键字搜索。模式串就是关键字（接下来称它为 P），如果它在一个主串（接下来称为 T）中出现，就返回它的具体位置，否则返回-1（常用手段）。

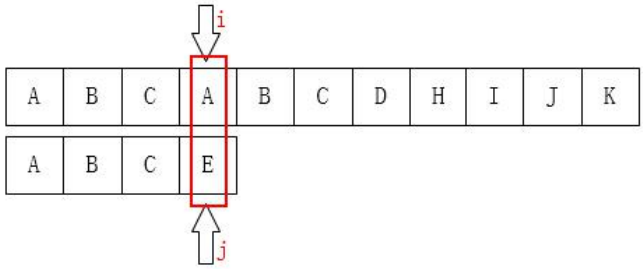


首先，对于这个问题有一个很单纯的想法：从左到右一个个匹配，如果这个过程中有某个字符不匹配，就跳回去，将模式串向右移动一位。这有什么难的？

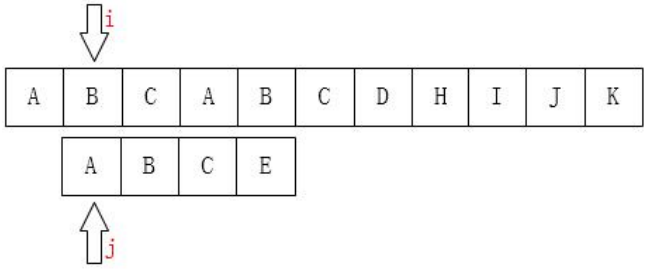
我们可以这样初始化：



之后我们只需要比较 i 指针指向的字符和 j 指针指向的字符是否一致。如果一致就都向后移动，如果不一致，如下图：

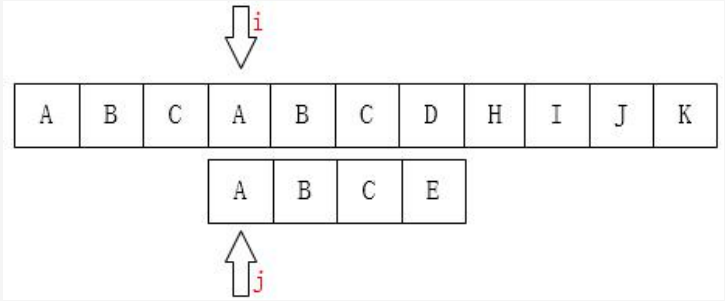


A 和 E 不相等，那就把 i 指针移回第 1 位（假设下标从 0 开始），j 移动到模式串的第 0 位，然后又重新开始这个步骤：



上面的程序是没有问题的，但不够好！

如果是人为来寻找的话，肯定不会再把 i 移动回第 1 位，**因为主串匹配失败的位置前面除了第一个 A 之外再也没有 A 了**，我们为什么能知道主串前面只有一个 A？**因为我们已经知道前面三个字符都是匹配的！（这很重要）**。移动过去肯定也是不匹配的！有一个想法，**i 可以不动，我们只需要移动 j** 即可，如下图：

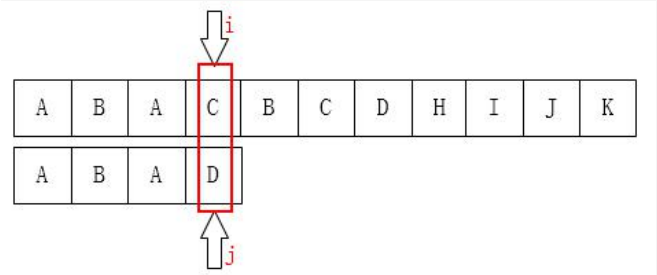


上面的这种情况还是比较理想的情况，我们最多也就多比较了几次。但假如是在主串“SSSSSSSSSSSSSA”中查找“SSSSB”，比较到最后一个才知道不匹配，然后 i 回溯，这个的效率是显然是最低的。

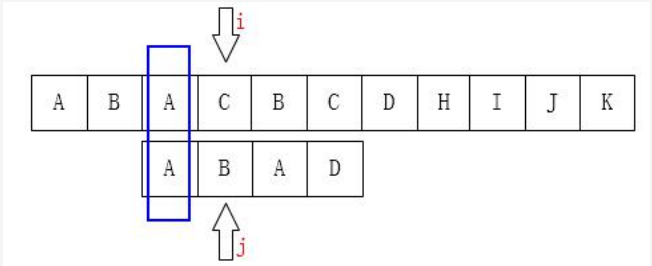
大牛们是无法忍受“暴力破解”这种低效的手段的，于是他们三个研究出了 KMP 算法。
其思想是：“**利用已经部分匹配这个有效信息，保持 i 指针不回溯，通过修改 j 指针，让模式串尽量地移动到有效的位置**。”

所以，整个 KMP 的重点就在于**当某一个字符与主串不匹配时，我们应该知道 j 指针要移动到哪**？

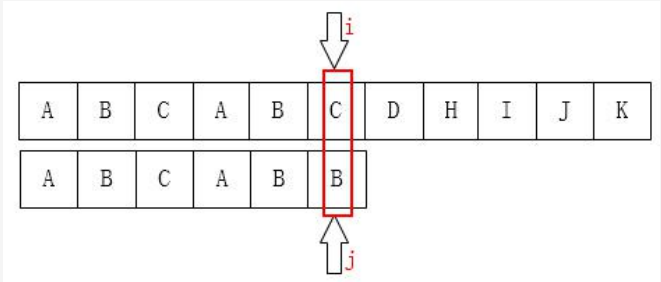
接下来我们来总结 j 的移动规律：



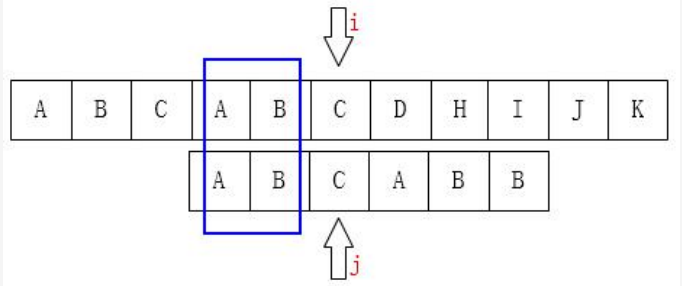
如图：C 和 D 不匹配了，我们要把 j 移动到哪？显然是第 1 位，因为前面有一个 A 相同。



如下图也是一样的情况：



可以把 j 指针移动到第 2 位，因为前面有两个字母是一样的：

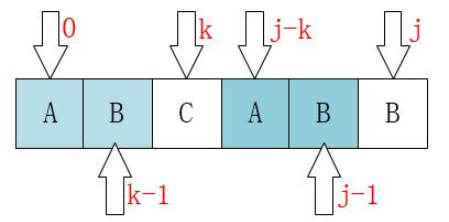


至此可以总结一下，当匹配失败时，j 要移动的下一个位置 k。存在着这样的性质：**最前面的 k 个字符和 j 之前的最后 k 个字符是一样的。**

如果用数学公式来表示是这样的

$$P[0 \sim k-1] == P[j-k \sim j-1]$$

这个相当重要，可以通过下图来理解：



弄明白了这个就应该可能明白为什么可以直接将 j 移动到 k 位置了。

因为:

> 当 T[i] != P[j]时
>
> 有 T[i-j ~ i-1] == P[0 ~ j-1]
>
> 由 P[0 ~ k-1] == P[j-k ~ j-1]
>
> 必然：T[i-k ~ i-1] == P[0 ~ k-1]

这一段只是为了证明为什么可以直接将 j 移动到 k 而无须再比较前面的 k 个字符。

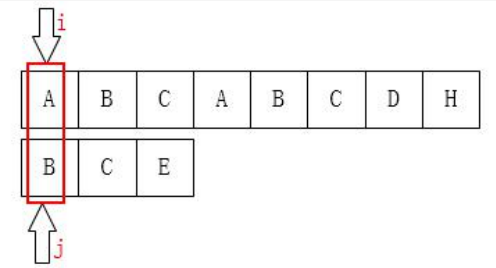
好，接下来就是重点了，怎么求这个（这些）k 呢？因为在 P 的每一个位置都可能发生不匹配，也就是说我们要计算每一个位置 j 对应的 k，所以用一个数组 next 来保存，**next[j] = k，表示当 T[i] != P[j]时，j 指针的下一个位置。**

```
const int size=100001;
int next[size];
char t[10000001];
char p[size];
void init_next()
{
    next[0]=-1;
    int j=0;
    int k=-1;
    while(j<strlen(p))
    {
        if(k==-1 || p[j]==p[k])
        {
            j++;
            next[j]=++k;
```

```
            }
            else
            {
                k=next[k];
            }
        }
}
```

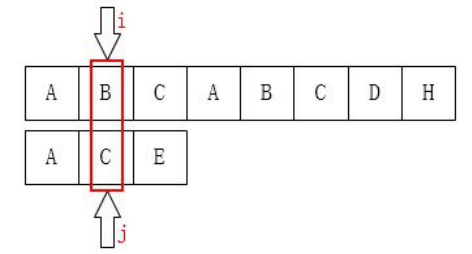
现在要始终记住一点，**next[j]的值（也就是 k）表示，当 P[j] != T[i]时，j 指针的下一步移动位置**。

先来看第一个：当 j 为 0 时，如果这时候不匹配，怎么办？



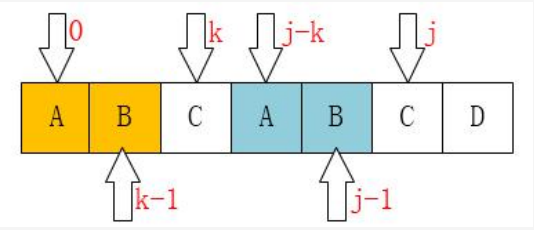
像上图这种情况，**j 已经在最左边了，不可能再移动了，这时候要应该是 i 指针后移**。所以在代码中才会有 next[0] = -1;这个初始化。

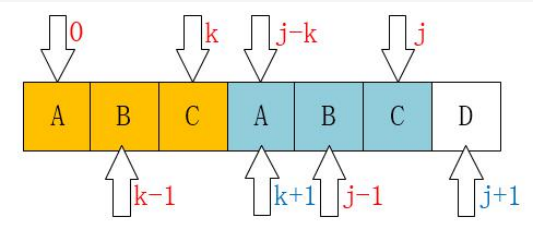
如果是当 j 为 1 的时候呢？



显然，**j 指针一定是后移到 0 位置的**。因为它前面也就只有这一个位置了~~~

下面这个是最重要的，请看如下图：





请仔细对比这两个图。我们发现一个规律：

当 P[k] == P[j]时，

有 next[j+1] == next[j] + 1

其实这个是可以证明的：

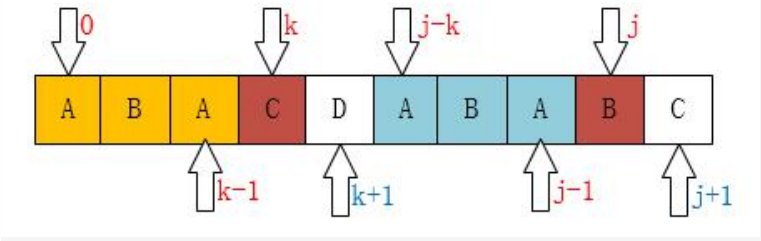
因为在 P[j]之前已经有 P[0 ~ k-1] == p[j-k ~ j-1]。（next[j] == k）

这时候现有 P[k] == P[j]，我们是不是可以得到 P[0 ~ k-1] + P[k] == p[j-k ~ j-1] + P[j]。

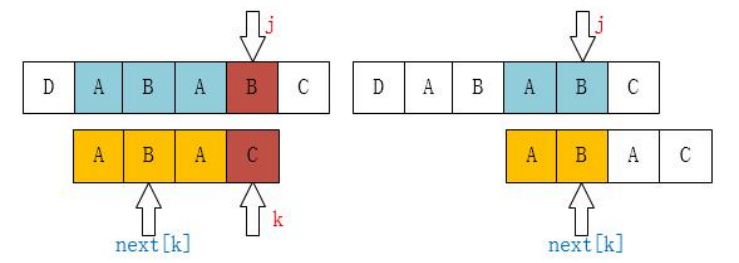
即：P[0 ~ k] == P[j-k ~ j]，即 next[j+1] == k + 1 == next[j] + 1。

这里的公式不是很好懂，还是看图会容易理解些。

那如果 P[k] != P[j]呢？比如下图所示：



像这种情况，如果你从代码上看应该是这一句：k = next[k];为什么是这样子？你看下面应该就明白了。



现在应该知道为什么要 k = next[k]了吧！像上边的例子，我们已经不可能找到[A，B，A，B]这个最长的后缀串了，但我们还是可能找到[A，B]、[B]这样的前缀串的。所以这个过程像不像

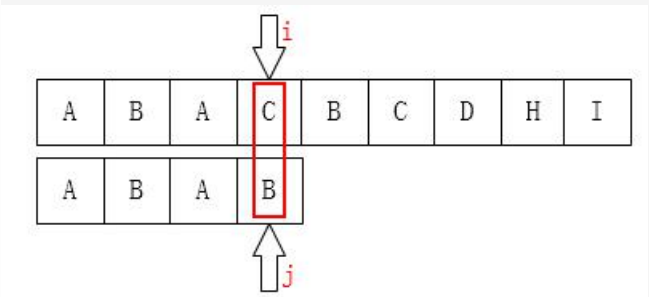
在定位[A，B，A，C]这个串，当 C 和主串不一样了（也就是 k 位置不一样了），那当然是把指针移动到 next[k]啦。

KMP 算法如下

```
#include <bits/stdc++.h>
using namespace std;
const int size=100001;
int next[size];
char t[10000001];
char p[size];
void init_next()
{
	next[0]=-1;
	int j=0;
	int k=-1;
	while(j<strlen(p))
	{
		if(k==-1 || p[j]==p[k])
		{
			j++;
			next[j]=++k;
		}
		else
		{
			k=next[k];
		}
	}
}
int kmp()
{
	int i=0;//主串的位置
	int j=0;//模式串的位置
	while(i<strlen(t) && j<strlen(p))
	{
		// 当 j 为-1 时，要移动的是 i，当然 j 也要归 0
		if(j==-1 || t[i]==p[j])
		{
			i++;
			j++;
		}
		else
		{
			//i 不需要回溯了
			j=next[j];//j 回到指定位置
		}
```

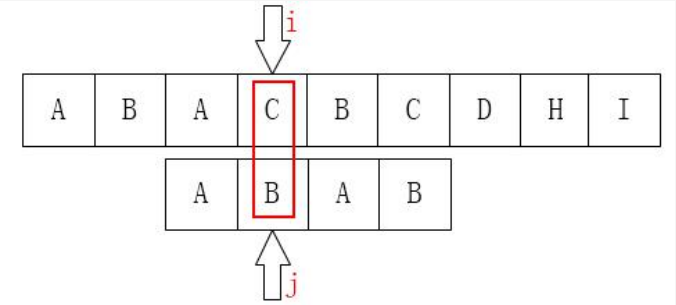
```
        }
        if(j==strlen(p))
        {
                return i-j;
        }
        else
        {
                return -1;
        }
}
int main()
{
        cin>>t;
        cin>>p;
        init_next();
        /*  可以看下 next 数组的值
        for(int i=0;i<10;i++)
        {
                cout<<next[i]<<" ";
        }
        cout<<endl;
        */
        int pos=kmp();
        cout<<pos<<endl;
        return 0;
}
```

最后，来看一下上边的算法存在的缺陷。来看第一个例子：



显然，当我们上边的算法得到的 next 数组应该是[-1，0，0，1]

所以下一步我们应该是把 j 移动到第 1 个元素咯：



不难发现，**这一步是完全没有意义的。因为后面的 B 已经不匹配了，那前面的 B 也一定是不匹配的**，同样的情况其实还发生在第 2 个元素 A 上。
显然，**发生问题的原因在于 P[j] == P[next[j]]**。

所以我们也只需要添加一个判断条件即可：

```
void init_next()
{
    next[0]=-1;
    int j=0;
    int k=-1;
    while(j<strlen(p))
    {
        if(k==-1 || p[j]==p[k])
        {
            if(p[++j]==p[++k])
            {
                //当两个字符相等时要跳过
                next[j]=next[k];
            }
            else
            {
                next[j]=k;
            }
        }
        else
        {
            k=next[k];
        }
    }
}
```

2. 基础题目

2.1. Oulipo（poj3461）

【问题描述】

给定一个字符串 s1 和一个字符串 s2，求 s2 在 s1 中的出现次数。 s1 和 s2 中的字符均为英语大写字母或小写字母。s1 中不同位置出现的 s2 可重叠。

每组数据保证 strlen（S1）≤10^6，strlen（S2）≤10^4。

【输入】

输入 T 组数据，对每组数据输出结果。

【输出】

对每组数据输出结果。

【样例输人】

```
3
BAPC
BAPC
AZAZAZAZ
AZA
AAAAAAAA
AA
```

【样例输出】

```
1
3
7
```

【参考代码】

```
#include <bits/stdc++.h>
using namespace std;
long long pw[1000010];
long long hs[1000010];
char s1[1000010],s2[1000010];
int T,base=1000000007,n,m;
int main()
{
    pw[0]=1;
    for(int i=1;i<1000000;i++)
    {
        pw[i]=pw[i-1]*base;
    }
    cin>>T;
    while(T--)
    {
        scanf("%s%s",s1+1,s2+1);
```

```
        n=strlen(s1+1);
        m=strlen(s2+1);
        hs[0]=0;
        for(int i=1;i<=n;i++)//计算主串的滚动哈希值
        {
            hs[i]=hs[i-1]*base+s1[i];
        }
        long long s=0;
        for(int i=1;i<=m;i++)//计算匹配串的哈希值
        {
            s=s*base+s2[i];
        }
        int ans=0;
        for(int i=0;i<=n-m;i++)//枚举起点为 i,长度为 n 的子串,判断与匹配串是否匹配
        {
            if(s==hs[i+m]-hs[i]*pw[m])
            {
                ans++;
            }
        }
        cout<<ans<<endl;
    }
    return 0;
}
```

2.2. Power Strings（poj2406）

【问题描述】

给定若干个长度小于等于 1000 000 000 的字符串，询问每个字符串最多是由多少个相同的子字符串重复连接而成的。如：**ababab** 则最多有 3 个 **ab** 连接而成。

【输入】

输入若干行，每行有一个字符串。特别的，字符串可能为 **.** 即一个半角句号，此时输入结束。

【输出】

输出相应的整数。

【样例输人】

```
abcd
aaaa
ababab
.
```

【样例输出】

```
1
4
3
```

【参考代码】

```
#include <iostream>
using namespace std;
int main()
{
    int n,i,a[100];
    cin>>n;
    for (i=0;i<n;i=i+1)
      cin>>a[i];
    for (i=n-1;i>=0;i=i-1)
      cout<<a[i]<<" ";
    return 0;
}
```

【补充】

如有需要，此处写入补充内容 XXXXXXXXXx。

注意 此处写入注意事项，此处写入注意事项，此处写入注意事项，此处写入注意事项，此处写入注意事项，

3. 强化练习

3.1. 最大的数

【问题描述】

输入 n 个整数，存放在数组 a 中，输出该数组的最大元素的值。

【输入】

输入包含两行：

第一行为 n，表示整数序列的长度(N <= 100)；

第二行为 n 个整数，整数之间以一个空格分开；

【输出】

输出为最大数。

【样例输入】

```
5
3 8 7 3 5
```

【样例输出】

```
8
```

【算法分析】

如有需要，此处写入算法分析描述。

【参考代码】

```
#include <iostream>
```

```
using namespace std;
int main()
{
    int a[101];
    int i=0,n,mmax;
    cin>>n;
    for(i=1;i<=n;i++)
    {
        cin>>a[i];
    }
    mmax=a[1];
    for(i=2;i<=n;i++)//打擂台求最大值
    {
        if(a[i]>mmax)
        {
            mmax=a[i];
        }
    }
    cout<<mmax<<endl;
    return 0;
}
```

【注意】

如有需要，此处写入注意事项 XXXXXXXXXXXXXXXx。

【补充】

如有需要，此处写入补充内容 XXXXXXXXXx。

4. 拓展练习

4.1. 最大的数

【问题描述】

输入 n 个整数，存放在数组 a 中，输出该数组的最大元素的值。

【输入】

输入包含两行：

第一行为 n，表示整数序列的长度(N <= 100)；

第二行为 n 个整数，整数之间以一个空格分开；

【输出】

输出为最大数。

【样例输人】

```
5
3 8 7 3 5
```

【样例输出】

```
8
```

【算法分析】

此处写入算法分析语句 XXXXXX。

【参考代码】

```
#include <iostream>
using namespace std;
int main()
{
    int a[101];
    int i=0,n,mmax;
    cin>>n;
    for(i=1;i<=n;i++)
    {
        cin>>a[i];
    }
    mmax=a[1];
    for(i=2;i<=n;i++)//打擂台求最大值
    {
        if(a[i]>mmax)
        {
            mmax=a[i];
        }
    }
    cout<<mmax<<endl;
    return 0;
}
```

【补充】

此处写入补充内容 XXXXXXXXXx。

注意 此处写入注意事项，此处写入注意事项，此处写入注意事项，此处写入注意事项，此处写入注意事项，

5. 课后作业

5.1. 复习作业

【要求】

1. 复习本次课讲过的题目；
2. 能在不看答案的情况下，将上课讲过的题目（基础题目和典型题目）自行编写实现；
3. 题目列表：

1) 删除相同的数 1
2) 删除相同的数 2
3) 删除相同的数 3
4) 移动 K 位
5) 冒泡排序
6) 统计各数字的个数

5.2. 课后训练

5.2.1. 彩票程序

【问题描述】

定义一个程序，要求实现双色球选号功能：

1. 能根据用户输入的 n，输出 n 注双色球随机号码；
2. 一注双色球号码由 6 个红号（号码范围为 1~33）和 1 个蓝号（号码范围为 1~16）组成；
3. 每注双色球的红色号码不存在重复；
4. 每注双色球红号要求按由小到大顺序输出；
5. 多注双色要求红球、蓝球对齐输出；

【样例输入 #1】

```
1
```

【样例输出 #1】

```
第 1 注：R：1  3  5  18  22  32  B：12
```

【样例输入 #2】

```
2
```

【样例输出 #2】

```
第 1 注：R：1  3  5  18  22  32  B：12
第 2 注：R：9  12  15  20  27  29  B：8
```

5.2.2. 趣味跳跃

【问题描述】

一个长度为 n（n>0）的序列中存在“有趣的跳跃”当前仅当相邻元素的差的绝对值经过排序后正好是从 1 到(n-1)。例如，1 4 2 3 存在“有趣的跳跃”，因为差的绝对值分别为 3,2,1。当然，任何只包含单个元素的序列一定存在“有趣的跳跃”。你需要写一个程序判定给定序列是否存在“有趣的跳跃”。

【输入】

一行，第一个数是 n（0 < n < 3000），为序列长度，接下来有 n 个整数，依次为序列中各元素，各元素的绝对值均不超过 1,000,000,000。

【输出】

一行，若该序列存在“有趣的跳跃”，输出"Jolly"，否则输出"Not jolly"。

【样例输入 #1】

```
4 1 4 2 3
```

【样例输出 #1】

```
Jolly
```

【样例输入 #2】

```
98 101 103 102 106
```

【样例输出 #2】

```
Jolly
```

【来源】

noi.openjudge.cn　1.6 一维数组

注意　所本题为一个比较有意思的题目，同学们可以多加思考，找规律，选择合适的算法技巧