法律声明

- 本课件包括演示文稿、示例、代码、题库、视频和声音等内容，小象学院和主讲老师拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意及内容，我们保留一切通过法律手段追究违反者的权利。

- 课程详情请咨询
  - 微信公众号：小象
  - 新浪微博：ChinaHadoop

分治和递归

小象学院 ChinaHadoop.cn 邹博

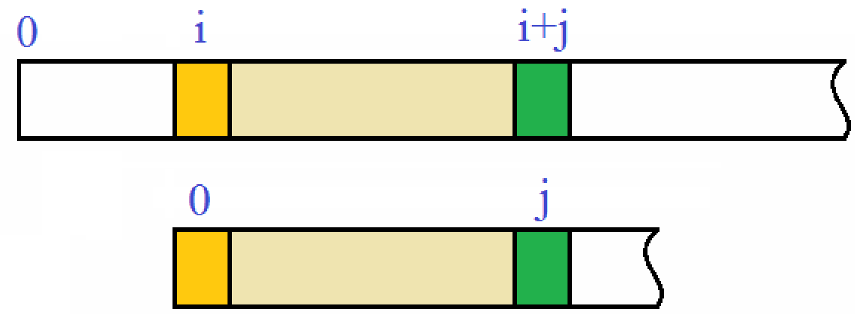
主要内容

- 迭代/分治/递归
  - 围棋中的正方形
  - 牛顿平方根公式
  - Callatz猜想问题
  - Eratosthenes筛法求素数
  - 循环染色方案
  - Hanoi塔及进阶
  - 实数的整数次幂
  - Strassen矩阵乘法/Karatsuba算法
  - 老鼠吃奶酪问题
  - 百数问题

KMP算法

- 字符串查找问题
  - 给定文本串text和模式串pattern，从文本串text中找出模式串pattern第一次出现的位置。
- 最基本的字符串匹配算法
  - 暴力求解(Brute Force)：时间复杂度O(m*n)
- KMP算法是一种线性时间复杂度的字符串匹配算法，它是对BF算法改进。

- 记：文本串长度为N，模式串长度为M
  - BF算法的时间复杂度O(M*N)，空间复杂度O(1)
  - KMP算法的时间复杂度O(M+N)，空间复杂度O(M)



暴力求解

```
//查找s中首次出现p的位置
int BruteForceSearch(const char* s, const char* p)
{
    int i = 0;  //当前匹配到的原始串首位
    int j = 0;  //模式串的匹配位置
    int size = (int)strlen(p);
    int nLast = (int)strlen(s) - size;
    while((i <= nLast) && (j < size))
    {
        if(s[i+j] == p[j])  //若匹配，则模式串匹配位置后移
        {
            j++;
        }
        else  //不匹配，则比对下一个位置，模式串回溯到首位
        {
            i++;
            j = 0;
        }
    }
    if(j >= size)
        return i;
    return -1;
}
```

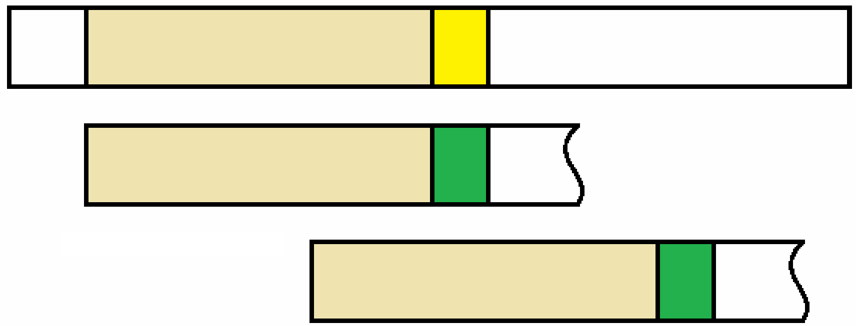
分析BF与KMP的区别

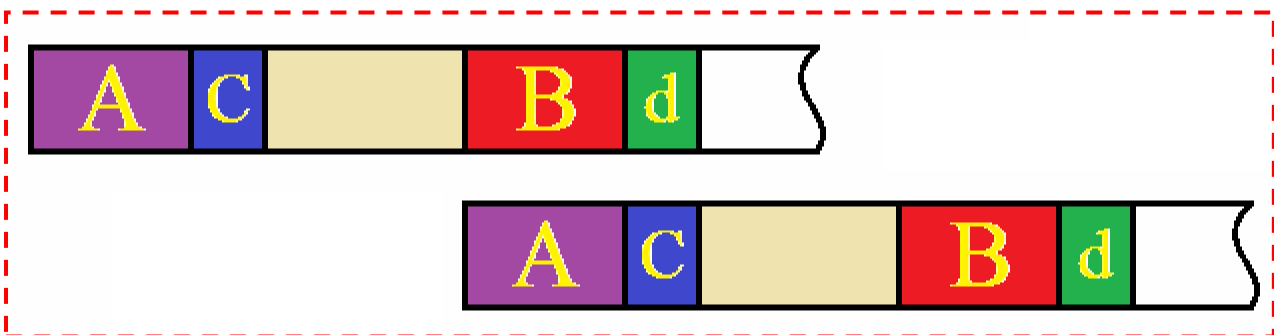
- 假设当前文本串text匹配到i位置，模式串pattern串匹配到j位置。
- BF算法中，如果当前字符匹配成功，即text[i+j]== pattern[j]，令j++，继续匹配下一个字符；
  - 若失配，即text[i+j]≠pattern[j]，令i++，j=0，即匹配失败时，模式串pattern相对于文本串text向右移动了一位。
- KMP算法中，若当前字符匹配成功，即text[i+j]== pattern[j]，令j++，继续匹配下一个字符；
  - 若失配，即text[i+j]≠pattern[j]，令j=next[j](next[j]≤j-1)，即模式串pattern相对于文本串text向右移动至少1位(实际移动位数为：j-next[j]≥1)

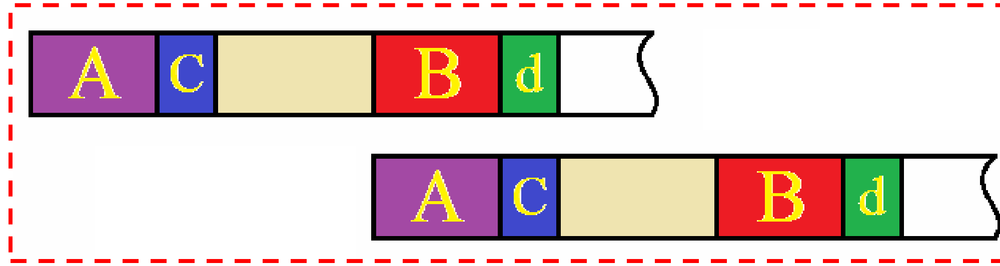
描述性说法

- 在暴力求解中，为什么模式串的索引会回溯？
  - 因为模式串存在重复字符
  - 思考：如果模式串的字符两两不相等呢？
    - 可以方便快速的编写线性时间的代码
  - 更弱一些的条件：如果模式串的首字符和其他字符不相等呢？

挖掘字符串比较的机制







分析后的结论

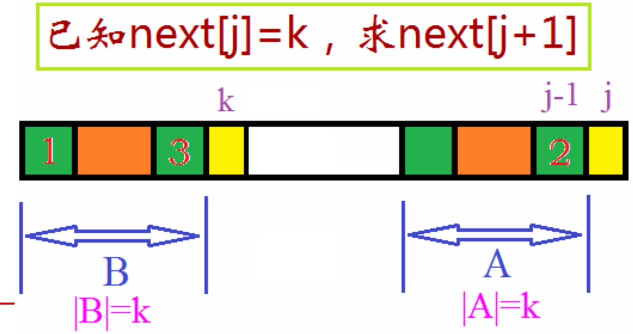
- 对于模式串的位置j，考察$Pattern_{j-1} = p_0p_1 \ldots p_{j-2}p_{j-1}$，查找字符串$Pattern_{j-1}$的最大相等k前缀和k后缀。
  - 注：计算next[j]时，考察的字符串是模式串的前j-1个字符，与pattern[j]无关。
- 即：查找满足条件的最大的k，使得
  - $p_0p_1 \ldots p_{k-2}p_{k-1} = p_{j-k}p_{j-k+1} \ldots p_{j-2}p_{j-1}$

求模式串的next

模式串	a	b	a	a	b	c	a	b	a
next	-1	0	0	1	1	2	0	1	2

- 如：j=5时，考察字符串“abaab”的最大相等k前缀和k后缀

前缀串	后缀串
a	b
ab	ab
aba	aab
abaa	baab
abaab	abaab



next的递推关系

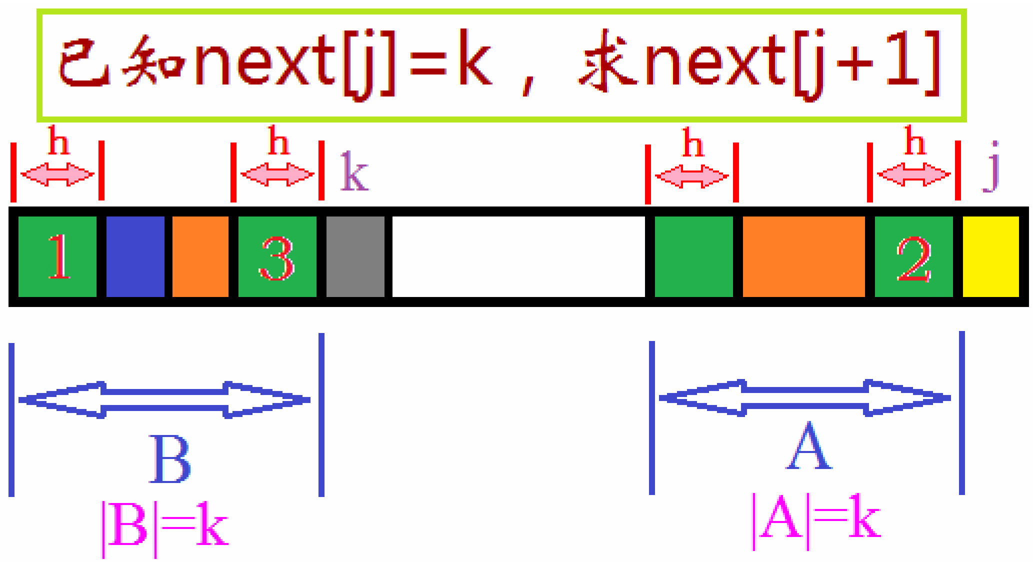
- 对于模式串的位置j，有next[j]=k，即：

  $p_0 p_1 \cdots p_{k-2} p_{k-1} = p_{j-k} p_{j-k+1} \cdots p_{j-2} p_{j-1}$

- 则，对于模式串的位置j+1，考察p_j：
- 若p[k]==p[j]
  - next[j+1]=next[j]+1
- 若p[k]≠p[j]
  - 记h=next[k]；如果p[h]==p[j],则next[j+1]=h+1，否则重复此过程。

考察不相等时，为何可以递归下去

- 若p[k]≠p[j]
  - 记h=next[k]；如果p[h]==p[j],则next[j+1]=h+1，否则重复此过程

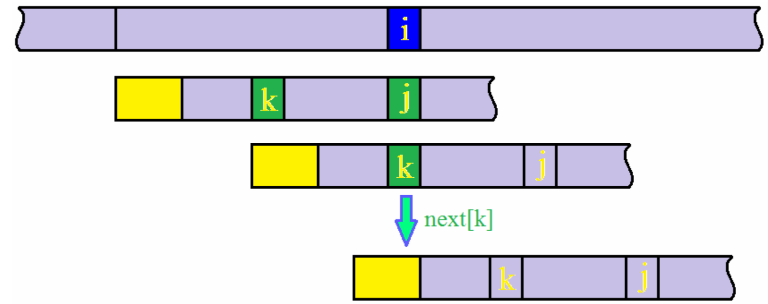


计算Next数组

```
void GetNext(char* p, int next[])
{
    int nLen = (int)strlen(p);
    next[0] = -1;
    int k = -1;
    int j = 0;
    while (j < nLen - 1)
    {
        //此刻，k即next[j-1]，且p[k]表示前缀，p[j]表示后缀
        //注：k==-1表示未找到k前缀与k后缀相等，首次分析可先忽略
        if (k == -1 || p[j] == p[k])
        {
            ++j;
            ++k;
            next[j] = k;
        }
        else    //p[j]与p[k]失配，则继续递归计算前缀p[next[k]]
        {
            k = next[k];
        }
    }
}
```

KMP Code

```
int KMP(int n)
{
    int ans = -1;
    int i = 0;
    int j = 0;
    int pattern_len = strlen(g_pattern);
    while(i < n)
    {
        if(j == -1 || g_s[i] == g_pattern[j])
        {
            ++i; ++j;
        }
        else
            j = g_next[j];
        if(j == pattern_len)
        {
            ans = i - pattern_len;
            break;
        }
    }
    return ans;
}
```



进一步分析next

- 文本串匹配到i，模式串匹配到j，此刻，若text[i]≠pattern[j]，即失配的情况：
- 若next[j]=k，说明模式串应该从j滑动到k位置；
- 若此时满足pattern[j]==pattern[k]，因为text[i]≠pattern[j]，所以， text[i] ≠pattern[k]
  - 即i和k没有匹配，应该继续滑动到next[k]。
  - 换句话说：在原始的next数组中，若next[j]=k并且pattern[j]==pattern[k]，next[j]可以直接等于next[k]。

Code2

```
void GetNext2(char* p, int next[])
{
    int nLen = (int)strlen(p);
    next[0] = -1;
    int k = -1;
    int j = 0;
    while (j < nLen - 1)
    {
        if (k == -1 || p[j] == p[k])
        {
            ++j;
            ++k;
            if(p[j] == p[k])
                next[j] = next[k];
            else
                next[j] = k;
        }
        else
        {
            k = next[k];
        }
    }
}
```

求模式串的next——变种

模式串	a	b	a	a	b	c	a	b	a
原始next	-1	0	0	1	1	2	0	1	2
新next	-1	0	-1	1	0	2	-1	0	-1

理解KMP的时间复杂度

- 我们考察模式串的“串头”和主串的对应位置(也就是暴力算法中的i)。
- 不匹配：串头后移，保证尽快结束算法；
- 匹配：串头保持不动(仅仅是i++、j++，但串头和主串的对应位置没变)，但一旦发现不匹配，会跳过匹配过的字符(next[j])。
- 最坏的情况，当串头位于N-M的位置，算法结束
- 因此，匹配的时间复杂度为O(N)，算上计算next的O(M)时间，整体时间复杂度为O(M+N)。

考察KMP的时间复杂度

- 最好情况：当模式串的首字符和其他字符都不相等时，模式串不存在相等的k前缀和k后缀，next数组全为-1
  - 一旦匹配失效，模式串直接跳过已经比较的字符。比较次数为N
- 最差情况：当模式串的首字符和其他字符全都相等时，模式串存在最长的k前缀和k后缀，next数组呈现递增样式：-1,0,1,2…
  - 举例说明

KMP最差情况

- next:-1 0 1 2 3
- 比较次数：5 1 1 1 1
- 周期：n/5
- 总次数：1.8n

- 每个周期中: m 1 1 1…
- 周期：n/m
- 总次数：$\left(2-\frac{1}{M}\right)\cdot N < 2N$

aaaabaaaabaaaabaaaabaaaab

aaaaa

aaaaa

aaaaa

aaaaa

aaaaa

aaaaa

aaaaa

aaaaa

aaaaa

aaaaa

aaaaa

最差情况下，变种KMP的运行情况

aaaabaaaabaaaabaaaabaaaab

aaaaa

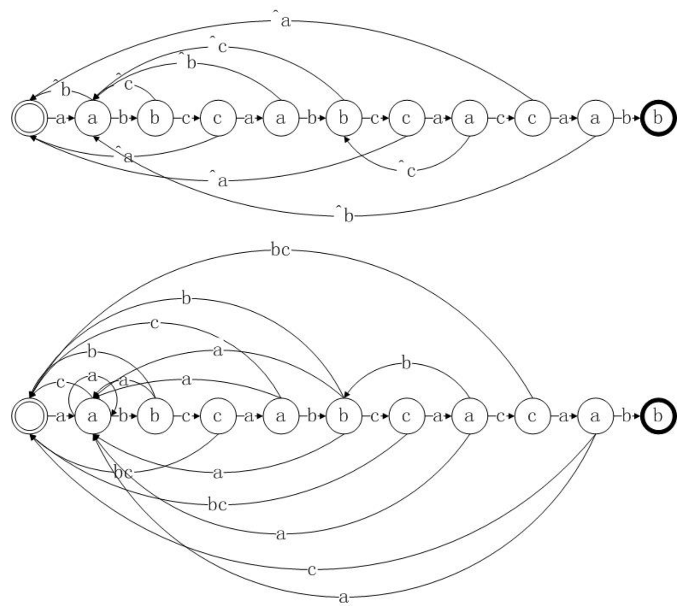
aaaaa

aaaaa

- next:-1 -1 -1 -1 -1
- 比较次数：5
- 周期：n/5
- 总次数：n

KMP的next，实际上是建立了DFA

- 以当前位置为DFA的状态，以模式串的字符为DFA的转移条件，建立确定有穷自动机
  - Deterministic Finite Automaton



图片来自网络

附：DFA和NFA

- DFA的五要素
  - 非空有限的状态集合Q
  - 输入字母表Σ
  - 转移函数δ
  - 开始状态S
  - 结束状态F
- 对于一个给定的DFA，存在唯一一个对应的有向图；有向图的每个结点对应一个状态，每条有向边对应一种转移。习惯上将结点画成两个圈表示接受状态，一个圈表示拒绝状态。用一条没有起点的边指向起始状态。
- 如果从某个状态，在确定的输入条件下，状态转移是多个状态，则这样的自动机是非确定有穷自动机。
- 可以证明，DFA和NFA是等价的，它们识别的语言成为正则语言。

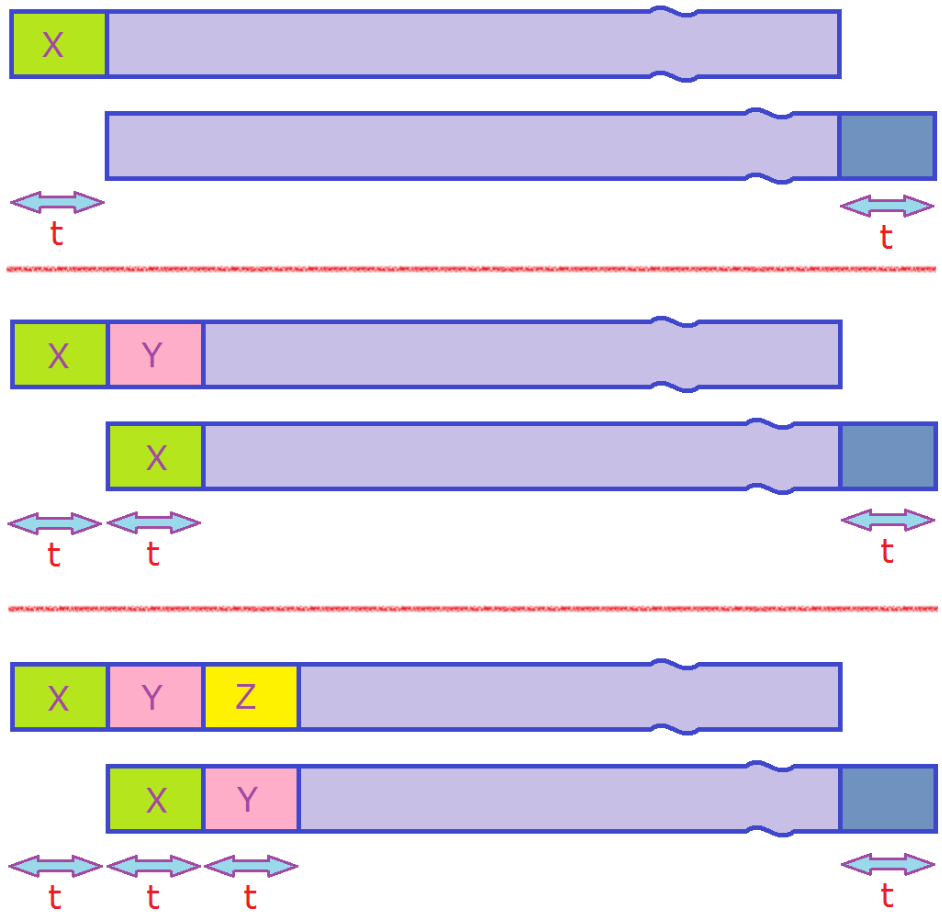
KMP应用：PowerString问题

- 给定一个长度为n的字符串S，如果存在一个字符串T，重复若干次T能够得到S，那么，S叫做周期串，T叫做S的一个周期。
- 如：字符串abababab是周期串，abab、ab都是它的周期，其中，ab是它的最小周期。
- 设计一个算法，计算S的最小周期。如果S不存在周期，返回空串。

使用next，线性时间解决问题

- 计算S的next数组；
  - 记k=next[len]，p=len-k；
  - 若len%p==0，则p为最小周期长度，前p个字符就是最小周期。
- 说明：
  - 使用的是经典KMP的next算法，非变种KMP的next算法；
  - 要“多”计算到len，即next[len]。
- 思考：如何证明？
  - 考察字符串S的k前缀first和k后缀tail：
  - 1、first和tail的前p个字符
  - 2、first和tail的前2*p个字符
  - 3、first和tail的前3*p个字符
  - ……

序号	0	1	2	3	4	5	6	7	8	9	10	11	12
字符串	a	b	c	a	b	c	a	b	c	a	b	c	\0
next	-1	0	0	0	1	2	3	4	5	6	7	8	9



Code

```
int MinPeriod(char* p)
{
    int nLen = (int)strlen(p);
    if(nLen == 0)
        return -1;
    int* next = new int[nLen];  //仿照KMP求"伪next"
    next[0] = -1;   //哨兵：串首标志
    int k = -1;
    int j = 0;
    while (j < nLen - 1)
    {
        if((k == -1) || (p[j+1] == p[k]))
        {
            ++k;
            ++j;
            next[j] = k;
        }
        else
        {
            k = next[k];
        }
    }
    next[0] = 0;    //恢复成逻辑上的0

    int nLast = next[nLen-1];
    delete[] next;
    if(nLast == 0)
        return -1;
    if(nLen % (nLen-nLast) == 0)
        return nLen-nLast;
    return -1;
}
```

时间复杂度

- 假定函数MyFunc()的时间复杂度为O(1)，则下列代码的时间复杂度关于整数n是多少?
  - Θ(NlogN) /Θ(N)

```
void CalcTime()
{
    int i,j;
    for(i = 1; i < n; i *= 3)
    {
        for(j = i/3; j < i; j++)
        {
            MyFunc();
        }
    }
}
```

- 注：Θ表示复杂度是紧的，
- 如堆排序中，建堆的时间复杂度为Θ (N)，而非Θ (NlogN)；
- 当然，可以说建堆的时间复杂度为O(NlogN)，因为O记号不要求上确界。

时间复杂度分析

```
void CalcTime()
{
	int i,j;
	for(i = 1; i < n; i *= 3)
	{
		for(j = i/3; j < i; j++)
		{
			MyFunc();
		}
	}
}
```

- 内层循环中，对于给定的i，j从$\frac{i}{3}$累加到i，循环次数为$\frac{2}{3}\cdot i$
- 外层循环中，i从1到n遍历，每次变成当前值的3倍，即1,3,9,27…，通项为$3^k, \left(k=0,1,2\cdots,3^k<N\right)$
- 将内层循环次数按照递增3倍做累加后，得循环总次数：$Time=\frac{2}{3}\cdot 1+\frac{2}{3}\cdot 3+\frac{2}{3}\cdot 9+\frac{2}{3}\cdot 27+\frac{2}{3}\cdot 81+\cdots+\frac{2}{3}\cdot 3^k$

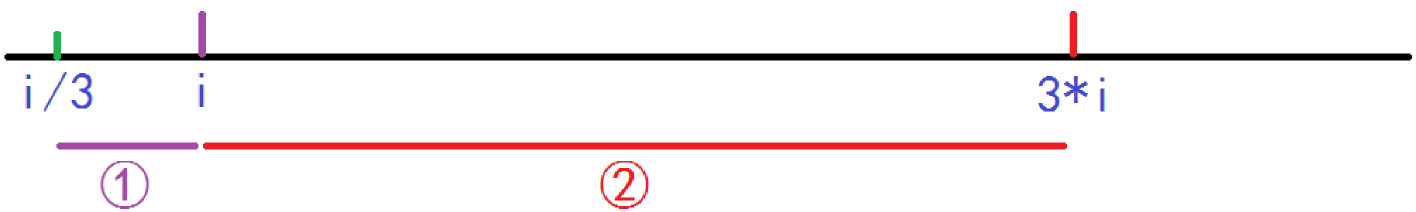
$$=\frac{2}{3}\cdot\left(1+3+9+27+\cdots+3^k\right)$$

$$=\frac{2}{3}\cdot\frac{1-3^{k+1}}{1-3}=3^k-\frac{1}{3}<N$$

图示分析

```
void CalcTime()
{
    int i,j;
    for(i = 1; i < n; i *= 3)
    {
        for(j = i/3; j < i; j++)
        {
            MyFunc();
        }
    }
}
```

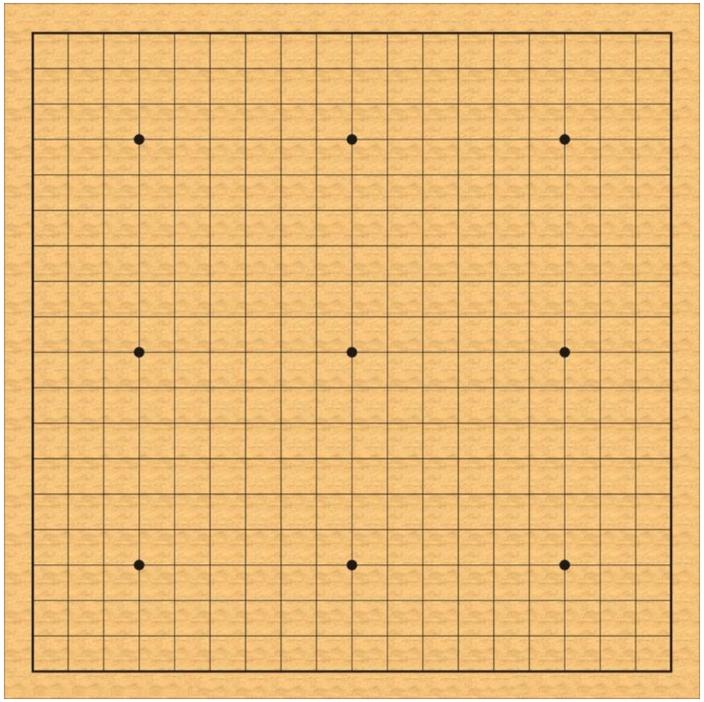
- 从下面的图示能够清楚的反映这一问题：



- 上图中，当外层循环的i位于紫色位置时，内层循环执行的是紫色的①；下次循环，当外层循环的i位于红色位置3*i时，内层循环执行的是红色的②，依次类推。所以，循环次数的上限为N。从而，时间复杂度为O(N)。

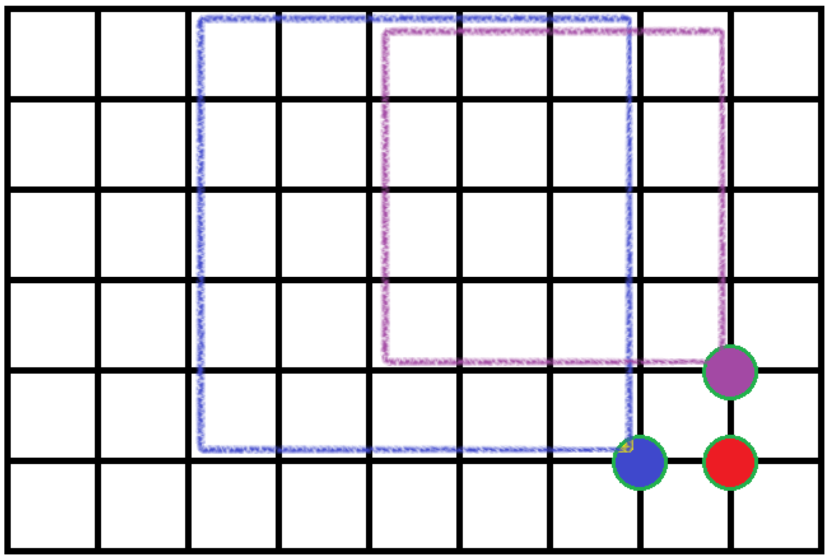
围棋中的正方形

- 围棋棋盘由横纵19*19条线组成，请问这些线共组成多少个正方形？假定只考虑横纵方向，忽略倾斜方向。

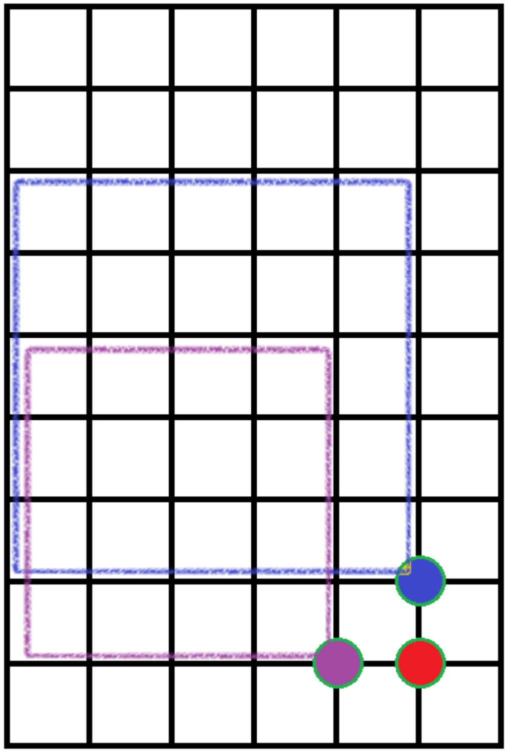


算法分析 $f(i,j)=\max\left(f(i-1,j),f(i,j-1)\right),\ i\neq j$

- 以(i,j)为右下角的正方形数目f(i,j)

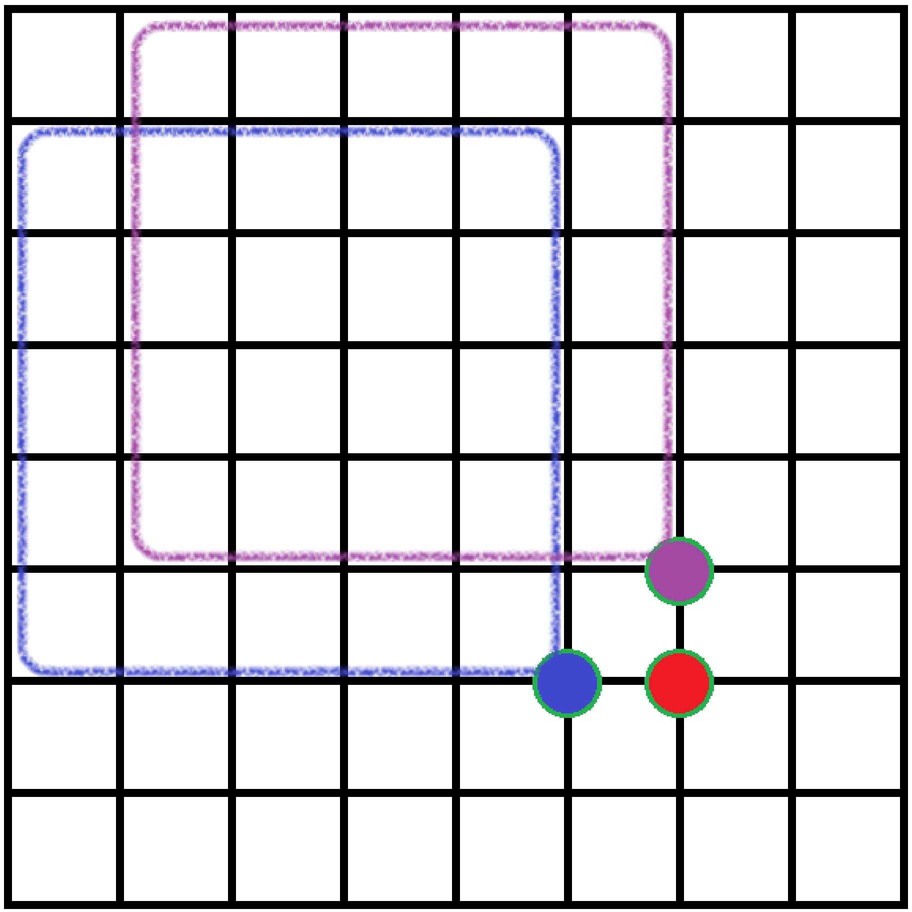


$f(i,j)=f(i,j-1),\quad i<j$



$f(i,j)=f(i-1,j),\quad i>j$

算法分析 $f(i,j)=f(i-1,j)+1, \quad i=j$



算法结论

- 综上，得出递推关系：

$$f(i,j)=\begin{cases}\max\left(f(i-1,j),f(i,j-1)\right), & i\neq j\\ f(i-1,j)+1, & i=j\end{cases}$$

- 显然有初始关系：

$$\begin{cases}f(i,0)=0\\ f(0,j)=0\end{cases}$$

Code

```
int _tmain(int argc, _TCHAR* argv[])
{
    int M = 19;
    int N = 19;
    vector<vector<int> > chess(M, vector<int>(N));

    //初值
    int i,j;
    for(i = 0; i < M; i++)
        chess[i][0] = 0;
    for(j = 0; j < N; j++)
        chess[0][j] = 0;

    //递推关系
    int count = 0;
    for(i = 1; i < M; i++)
    {
        for(j = 1; j < N; j++)
        {
            if(i != j)
                chess[i][j] = max(chess[i-1][j], chess[i][j-1]);
            else
                chess[i][j] = chess[i-1][j]+1;
            count += chess[i][j];
        }
    }
    Print(chess, M, N);
    cout << "总数为: " << count << endl;
    return 0;
}
```

思考

$$f(i,j)=\begin{cases}\max\left(f(i-1,j),f(i,j-1)\right), & i\neq j\\ f(i-1,j)+1, & i=j\end{cases}$$

- 本题的关键是如何将问题分解成更小规模的问题，从而解决问题。
  - 这个思想比题目本身更重要。
- 事实上，

0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1	1	1	1
0	1	2	2	2	2	2	2	2	2	2
0	1	2	3	3	3	3	3	3	3	3
0	1	2	3	4	4	4	4	4	4	4
0	1	2	3	4	5	5	5	5	5	5
0	1	2	3	4	5	6	6	6	6	6
0	1	2	3	4	5	6	7	7	7	7
0	1	2	3	4	5	6	7	8	8	8
0	1	2	3	4	5	6	7	8	9	9
0	1	2	3	4	5	6	7	8	9	X

$$f(i,j)=\min(i,j)$$

Code2

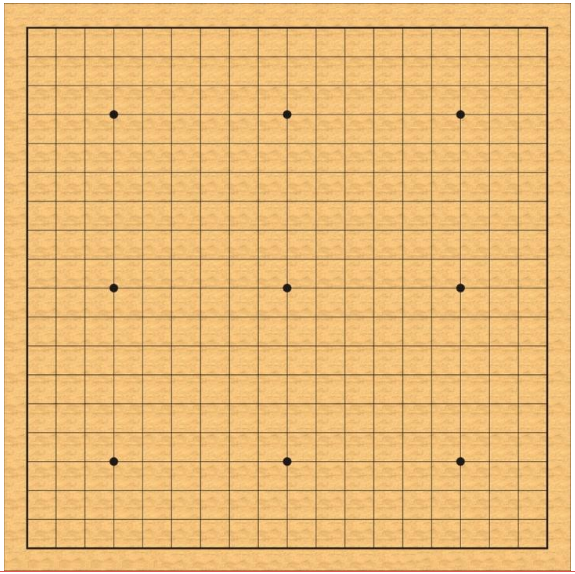
0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1	1	1	1
0	1	2	2	2	2	2	2	2	2	2
0	1	2	3	3	3	3	3	3	3	3
0	1	2	3	4	4	4	4	4	4	4
0	1	2	3	4	5	5	5	5	5	5
0	1	2	3	4	5	6	6	6	6	6
0	1	2	3	4	5	6	7	7	7	7
0	1	2	3	4	5	6	7	8	8	8
0	1	2	3	4	5	6	7	8	9	9
0	1	2	3	4	5	6	7	8	9	X

```
int _tmain(int argc, _TCHAR* argv[])
{
    int M = 19;
    int N = 19;
    int count = 0;
    int i, j;
    for(i = 1; i < M; i++)
    {
        for(j = 1; j < N; j++)
        {
            count += min(i, j);
        }
    }
    cout << "总数为: " << count << endl;
    return 0;
}
```

如果手头没有编译器呢？

- 如果数一下边长是1、2、3……18的正方形各有多少个，能够很快得到结论。

$$1^2+2^2+\cdots+18^2=\frac{18\times19\times37}{6}=2109$$



0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1	1	1	1
0	1	2	2	2	2	2	2	2	2	2
0	1	2	3	3	3	3	3	3	3	3
0	1	2	3	4	4	4	4	4	4	4
0	1	2	3	4	5	5	5	5	5	5
0	1	2	3	4	5	6	6	6	6	6
0	1	2	3	4	5	6	7	7	7	7
0	1	2	3	4	5	6	7	8	8	8
0	1	2	3	4	5	6	7	8	9	9
0	1	2	3	4	5	6	7	8	9	X

这段代码在做什么？

```
float Calc(float x);

int _tmain(int argc, _TCHAR* argv[])
{
    for(int i = 0; i <= 10; i++)
        cout << Calc((float)i) << '\n';
    return 0;
}

float Calc(float a)
{
    if(a < 1e-6)    //负数或者0，则直接返回0
        return 0;

    float x = a / 2;
    float t = a;
    while(fabs(x - t) > 1e-6)
    {
        t = x;
        x = (x + a/x) / 2;
    }
    return x;
}
```

平方根算法

- 在任意点x_0处Taylor展开

令$f(x)=x^2-a$，即$f(x)=0$

$$\Rightarrow f(x)=f(x_0)+f'(x_0)(x-x_0)+o(x)$$

$$\Rightarrow f(x)\approx f(x_0)+f'(x_0)(x-x_0)$$

$$\Rightarrow 0=f(x_0)+f'(x_0)(x-x_0)$$

$$\Rightarrow x-x_0=-\frac{f(x_0)}{f'(x_0)}$$

$$\Rightarrow x=x_0-\frac{f(x_0)}{f'(x_0)}$$

将$f(x_0)=x_0^2-a$和$f'(x_0)=2x_0$带入，

$$\Rightarrow x=x_0-\frac{f(x_0)}{f'(x_0)}$$

$$\Rightarrow x=x_0-\frac{x_0^2-a}{2x_0}$$

$$\Rightarrow x=\frac{1}{2}\left(x_0+\frac{a}{x_0}\right)$$

解决除法

- 牛顿公式：$x = x_0 - \frac{f(x_0)}{f'(x_0)}$

- 令$f(x) = \frac{1}{x^2} - a \Rightarrow \begin{cases} f(x_0) = \frac{1}{x_0^2} - a \\ f'(x_0) = -\frac{2}{x_0^3} \end{cases} \Rightarrow x = x_0 - \frac{\frac{1}{x_0^2} - a}{-\frac{2}{x_0^3}} = \frac{(3 - a x_0^2) x_0}{2}$

- 令$f(x) = \frac{1}{x} - a \Rightarrow \begin{cases} f(x_0) = \frac{1}{x_0} - a \\ f'(x_0) = -\frac{1}{x_0^2} \end{cases} \Rightarrow x = x_0 - \frac{\frac{1}{x_0} - a}{-\frac{1}{x_0^2}} = x_0 \cdot (2 - a \cdot x_0)$

Code2

```
double Reciprocal(double a)
{
    double x = 1;
    while(a * x >= 2)
    {
        if(a > 1)
            x /= 10;
        else
            x *= 10;
    }
    double t = a;
    while(fabs(x - t) > 1e-6)
    {
        t = x;
        x = x * (2 - a * x);
    }
    return x;
}
```

```
double Sqrt(double a)
{
    if(a < 1e-6)
        return 0;
    double x = 1;
    while(a * x * x >= 3)
        x *= 0.1;
    double t = a;
    while(fabs(x - t) > 1e-6)
    {
        t = x;
        x = (3 - a * t * t) * t / 2;
    }
    return Reciprocal(x);
}
```

Callatz猜想问题

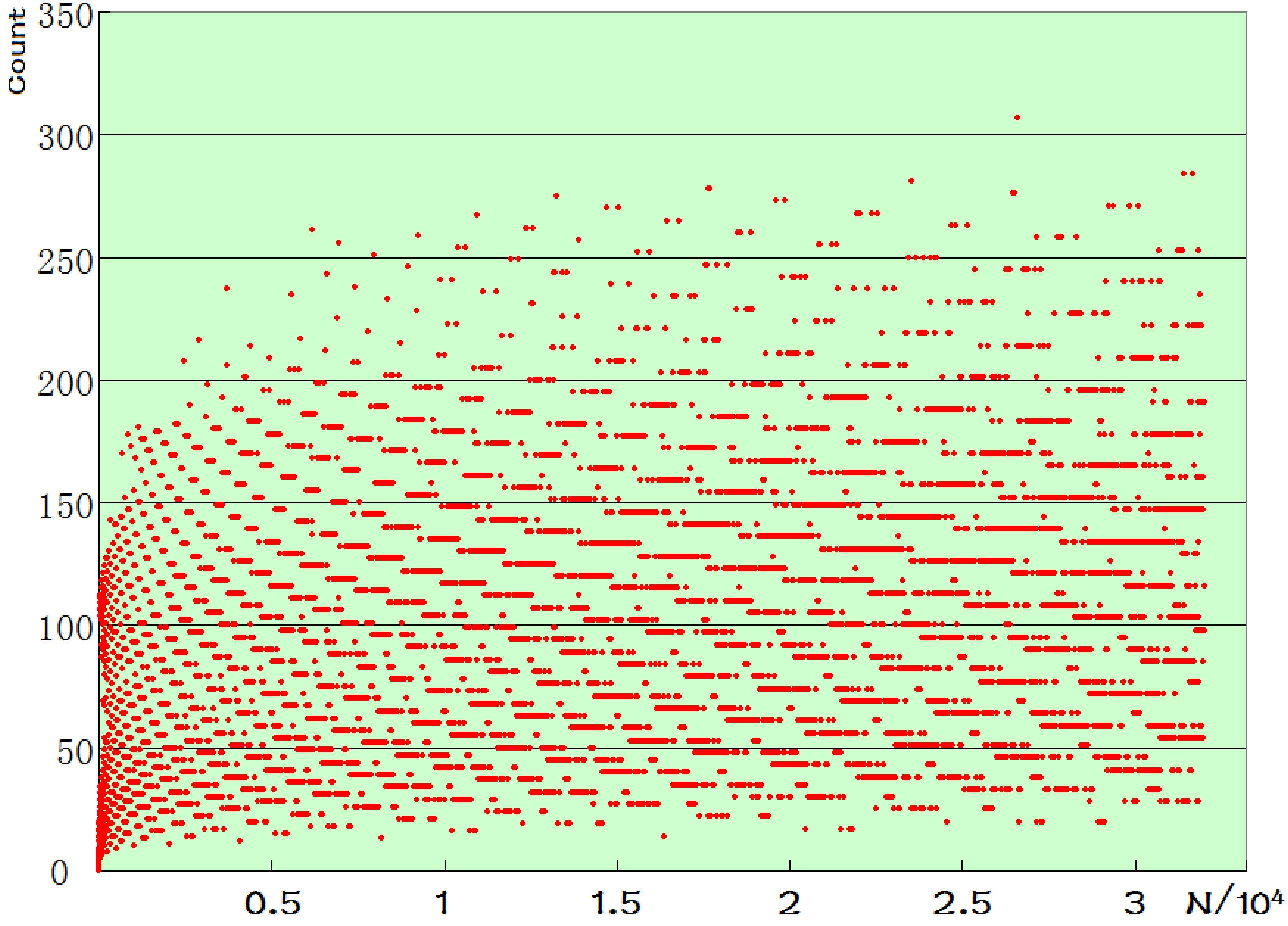
- 该问题又称3n+1猜想、角谷猜想、哈塞猜想、乌拉姆猜想、叙拉古猜想等。
- 过程非常简单：给定某正整数N，若为偶数，则N被更新为N/2；否则，N被更新为3*N+1；问：(1)经过多少步N变成1？(2)是否存在某整数X无法变成1？
- 思考：
  - 如果已经计算得到1~N-1的变换次数，如何计算N的变换次数？

Code

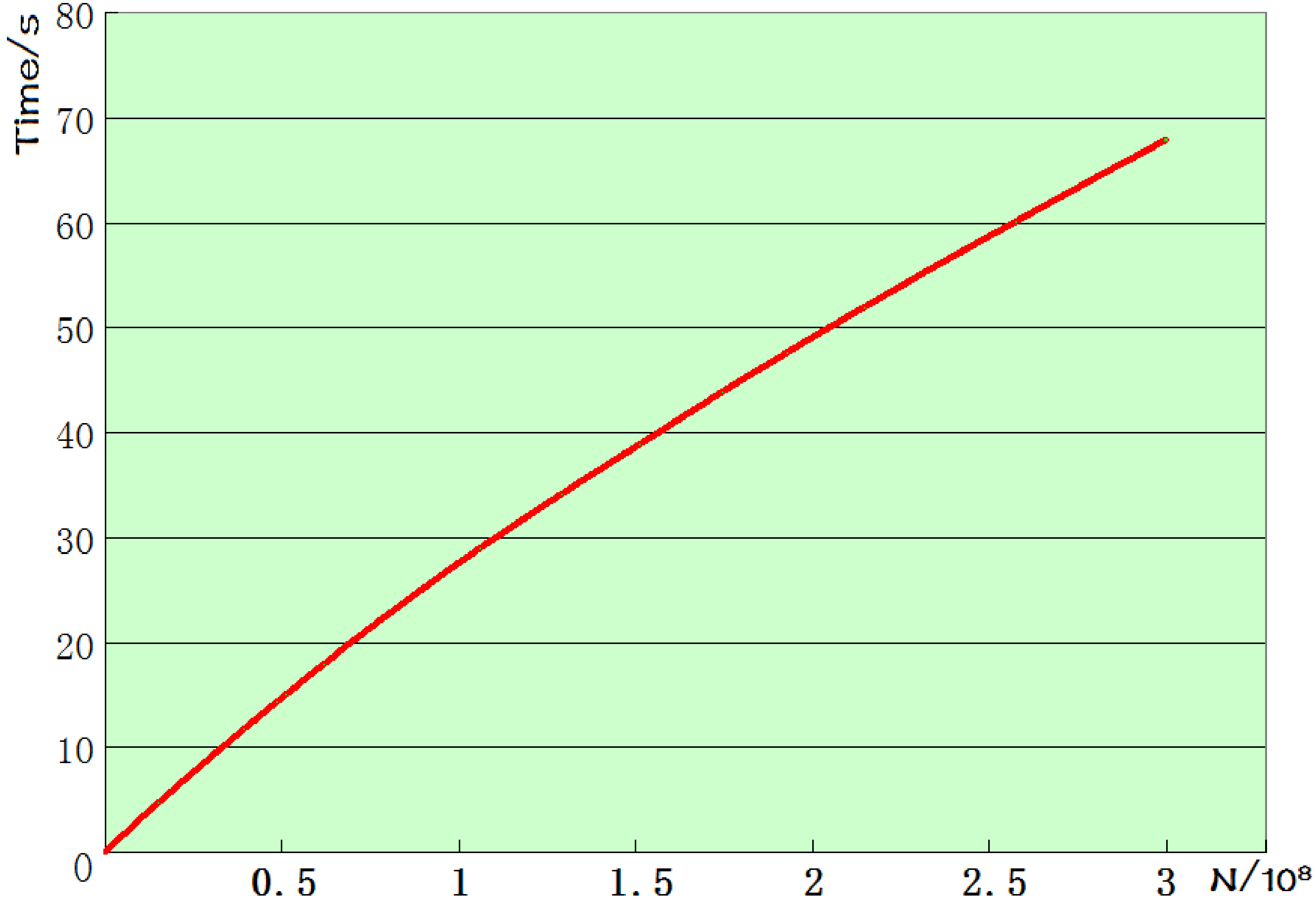
```
void Calc(long long i, int* p, int N, int timeStart)
{
    int cur = (int)i;
    int t = 0;
    while(true)
    {
        if(i % 2 == 0)
        {
            i /= 2;
            t++;
        }
        else
        {
            i = i * 3 + 1;
            t++;
        }
        if((i < N) && (p[(int)i] != -1))    //已经有部分值
        {
            p[cur] = p[(int)i] + t;
            break;
        }
    }

    if(cur % 10000 == 0)    //顺便记录时间
    {
        tt.push_back(GetTickCount() - timeStart);
    }
}
```

N-count图像



Code运行时间



Eratosthenes筛法求素数

- 给定正整数N，求小于等于N的全部素数。
- Eratosthenes筛法
  - 将2到N写成一排；
  - 记排头元素为x，则x是素数；除x以外，将x的倍数全部划去；
  - 重复以上操作，直到没有元素被划去，则剩余的即小于等于N的全部素数。
  - 为表述方面，将排头元素称为“筛数”。

Eratosthenes筛计算100以内的素数

- 2 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99
- 2 3 5 7 11 13 17 19 23 25 29 31 35 37 41 43 47 49 53 55 59 61 65 67 71 73 77 79 83 85 89 91 95 97
- 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 49 53 59 61 67 71 73 77 79 83 89 91 97
- 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

算法改进：筛选α以内的素数

- α以内素数的最大筛数为 $\sqrt{\alpha}$，记 $x=\sqrt{\alpha}$
- 对于 $\beta<\alpha$
- 若β为合数，即： $\beta=v\cdot u$
- 显然，u、v不能同时大于x，不妨 $v<u$，将它们记录在数轴上：

- 在使用x作为筛数之前，β已经被v筛掉。

Code

```
void Eratosthenes(bool* a, int n)
{
    a[1] = false;//a[0]不用
    int i;
    for(i = 2; i <= n; i++) //筛法，默认是素数
        a[i] = true;
    int p = 2;  //第一个筛孔
    int j = p*p;
    int c = 0;
    while(j <= n)
    {
        while(j <= n)
        {
            a[j] = false;
            j += p;
        }
        p++;
        while(!a[p])    //查找下一个素数
            p++;
        j = p*p;
    }
}
```

循环染色方案

- 用红、蓝两种颜色将围成一圈的8个棋子染色。规定：若某两种染色方案通过旋转的方式可以重合，则只算一种。问：一共有多少种不同的染色方案？

问题分析

- 用0、1表示颜色方案，8个棋子对应8位二进制数。
- “旋转重合则只算一种”即：将x循环左移一位得到y，则x和y属于相同类别(等价类)：
  - 若y<x，则删除x；
  - 若y>x，则删除y；
  - 该算法借鉴求素数的Eratosthenes筛法。
- 由于每个方案只需要计算1次，时间复杂度为O(N)
  - 棋子数目记为n，则$N=2^n$
  - 除了全0和全1以外，所有偶数都是重复的，所有最高位为1都是重复的，根据这两个结论可以适当优化。

Code

```
//循环左移一位
int RotateShiftLeft(int x, int N)
{
    int high = (x >> (N-1));
    x &= ((1<<(N-1)) -1);
    x <<= 1;
    x |= high;
    return x;
}
```

```
int Polya(int N, list<int>& answer)
{
    int i,j;
    int k1,k2;
    int m = (1 << N);
    int* p = new int[m];    //记录每种方案
    fill(p, p+m, 1);
    for(i = 0; i < m; i++)  //遍历所有染色方案
    {
        if(p[i] == 1)   //尚未删掉
        {
            k1 = i;
            for(j = 0; j <= N; j++)
            {
                k2 = RotateShiftLeft(k1, N);
                if(k2 == i) //说明完成了循环
                    break;
                if(k2 > i)  //后面的k2无效
                    p[k2] = 0;
                else //if(k2 < i)
                {
                    p[i] = 0;   //i无效
                    break;      //前面必然遍历过
                }
                k1 = k2;
            }
        }
    }

    for(i = 0; i < m; i++)
    {
        if(p[i] == 1)
            answer.push_back(i);
    }
    delete[] p;
    return (int)answer.size();
}
```

Code *snippets*

```
//循环左移一位
int RotateShiftLeft(int x, int N)
{
    int high = (x >> (N-1));
    x &= ((1<<(N-1)) -1);
    x <<= 1;
    x |= high;
    return x;
}
```

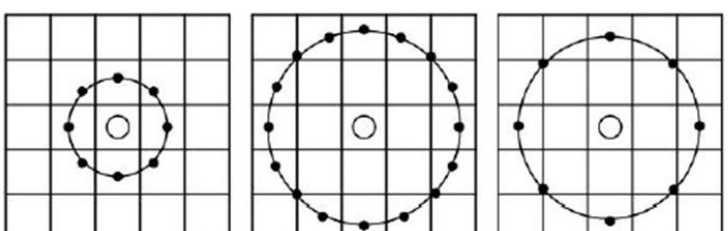
```
int Polya(int N, list<int>& answer)
{
    int i,j;
    int k1,k2;
    int m = (1 << N);
    int* p = new int[m];     //记录每种方案
    fill(p, p+m, 1);
    for(i = 0; i < m; i++)   //遍历所有染色方案
    {
        if(p[i] == 1)   //尚未删掉
        {
            k1 = i;
            for(j = 0; j <= N; j++)
            {
                k2 = RotateShiftLeft(k1, N);
                if(k2 == i) //说明完成了循环
                    break;
                if(k2 > i)  //后面的k2无效
                    p[k2] = 0;
                else //if(k2 < i)
                {
                    p[i] = 0;   //i无效
                    break;      //前面必然遍历过
                }
                k1 = k2;
            }
        }
    }
```

实验结果

- 6个棋子，共14种情况：
  - 0，1，3，5，7，9，11，13，15，21，23，27，31，63
- 7个棋子，共20种情况：
  - 0，1，3，5，7，9，11，13，15，19，21，23，27，29，31，43，47，55，63，127
- 8个棋子，共36种情况：
  - 0， 1，3，5，7，9，11，13，15，17，19，21，23，25，27，29，31，37，39，43，45，47，51，53，55，59，61，63，85，87，91，95，111，119，127，255

总结与思考

- Burnside定理和Polya计数以置换群为基础给出了该问题的分析过程(N个棋子c种颜色)。
  - 用红蓝两色对正方体六面染色有多少种方案？
- 该问题也可以看做LBP算子(Local Binary Pattern)的附属题目：
  - 根据图像上某指定像素p和周围像素(如8邻域)的相对强弱赋值为0/1，得到该像素点p的LBP值。
  - 如果循环计算LBP的最小值，则为旋转不变LBP算子。
  - 应用于：指纹识别、字符识别、人脸识别、车牌识别等



LBP_8^1 LBP_{16}^2 LBP_8^2

小象学院 ChinaHadoop.cn

Hanoi塔

- 有三根相邻的柱子，标号为A,B,C，A柱子上按从小到大叠放着n个不同大小的盘子，要求把所有盘子每次移动一个，最终放到C柱子上；移动过程中可以借助B柱子，但要求每次移动中必须保持小盘子在大盘子的上面。比如n=10，请给出最少次数的移动方案。

Code

```
void MoveOne(char from, char to)
{
    cout << from << " -> " << to << endl;
}

void Move(char from, char to, char aux, int n)
{
    if(n == 1)
    {
        MoveOne(from, to);
        return;
    }
    Move(from, aux, to, n-1);
    MoveOne(from, to);
    Move(aux, to, from, n-1);
}

int _tmain(int argc, _TCHAR* argv[])
{
    int n = 3;
    Move('A', 'C', 'B', n);
}
```

思考

- N个盘子的Hanoi塔，至少需要几次移动？

$$T(n)=2T(n-1)+1$$

$$\Rightarrow T(n)+1=2T(n-1)+2$$

$$\Rightarrow \frac{T(n)+1}{T(n-1)+1}=2$$

$$\begin{cases} T(n)=2T(n-1)+1 \\ T(1)=1 \end{cases} \Rightarrow T(n)=2^n-1$$

Hanoi塔的状态

- 给定从小到大的N个盘子，它们散乱的位于A、B、C柱上，问这一状态是否是将这N个盘子从A借助B移动到C的必经状态？如果是，返回是第几个状态，如果不是，返回-1
  - 初始状态记为0。
  - 根据从小到大这N个盘子位于哪个柱子上，形成一个只能取’A’、’B’、’C’三种可能的字符串：如“ABCCABCA”、‘

Hanoi塔递归代码分析

- N个盘子看做前N-1个盘子和最后一个盘子组成
  - 将前N-1个盘子移动到aux柱上：$2^{n-1}-1$
  - 将最大的盘子移动到to柱上：1
  - 将前N-1个盘子移动到to柱上：$2^{n-1}-1$

```
void Move(char from, char to, char aux, int n)
{
    if(n == 1)
    {
        MoveOne(from, to);
        return;
    }
    Move(from, aux, to, n-1);
    MoveOne(from, to);
    Move(aux, to, from, n-1);
}
```

Code

```
int Calc(const char* str, int size, char from, char to, char aux)
{
    if(size == 0)
        return 0;
    if(str[size-1] == aux)
        return -1;

    if(str[size-1] == to)
    {
        int n = Calc(str, size-1, aux, to, from);
        if(n == -1)
            return -1;
        return (1 << (size-1)) + n;
    }
    return Calc(str, size-1, from, aux, to); //str[size-1] == from
}

int _tmain(int argc, _TCHAR* argv[])
{
    char str[] = "ABC";
    cout << Calc(str, 3, 'A', 'C', 'B') << endl;

    strcpy(str,"AAC");
    cout << Calc(str, 3, 'A', 'C', 'B') << endl;
}
```

实数的整数次幂

- 给定实数x和整数n，求x^n。
- 分析：令$pow(x,n)= x^n$，如果能够求出$y=pow(x,n/2)$，只需要返回$y*y$即可，节省一半的时间。因此，可以递归下去。
  - 时间复杂度O(logN)
  - 需要考虑的：如果n是奇数呢？
  - 如果n是负数呢？

Code

```
double Pow(double x, int n)
{
    if(n == 0)
        return 1;
    if(n == 1)
        return x;
    if(n == 2)
        return x*x;
    double p = Pow(x, n/2);
    p *= p;
    return (n % 2 == 0) ? p : p*x;
}

double Power(double x, int n)
{
    if(n < 0)
        return 1/Pow(x, -n);
    return Pow(x, n);
}
int _tmain(int argc, _TCHAR* argv[])
{
    cout << Power(1.01, 365) << endl;
    return 0;
}
```

矩阵的乘法

- A为m×s阶的矩阵，B为s×n阶的矩阵，那么，C=A×B是m×n阶的矩阵，其中，

$$c_{ij} = \sum_{k=1}^{s} a_{ik} b_{kj}$$

- 根据定义来计算 C=A×B，需要m*n*s次乘法。
  - 即：若A、B都是n阶方阵，C的计算时间复杂度为$O(n^3)$
  - 问：可否设计更快的算法？

分治

- 矩阵分块

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$\begin{cases} C_{11} = A_{11}B_{11} + A_{12}B_{21} \\ C_{12} = A_{11}B_{12} + A_{12}B_{22} \\ C_{21} = A_{21}B_{11} + A_{22}B_{21} \\ C_{22} = A_{21}B_{12} + A_{22}B_{22} \end{cases}$$

- 按照定义：计算n/2阶矩阵的乘积：$O(n^3/8)$
- 这里需要计算8个：总时间复杂度： $O(n^3)$
  - 没有任何效果。

Strassen矩阵乘法：由8到7

- 目标

$$
\begin{cases}
C_{11} = A_{11}B_{11} + A_{12}B_{21} \\
C_{12} = A_{11}B_{12} + A_{12}B_{22} \\
C_{21} = A_{21}B_{11} + A_{22}B_{21} \\
C_{22} = A_{21}B_{12} + A_{22}B_{22}
\end{cases}
$$

$$
\begin{cases}
P = (A_{11} + A_{22})(B_{11} + B_{22}) \\
Q = (A_{21} + A_{22})B_{11} \\
R = A_{11}(B_{12} - B_{22}) \\
S = A_{22}(B_{21} - B_{11}) \\
T = (A_{11} + A_{12})B_{22} \\
U = (A_{21} - A_{11})(B_{11} + B_{12}) \\
V = (A_{12} - A_{22})(B_{21} + B_{22})
\end{cases}
\Longrightarrow
\begin{cases}
C_{11} = P + S - T + V \\
C_{12} = R + T \\
C_{21} = Q + S \\
C_{22} = P + S - Q + U
\end{cases}
$$

Strassen矩阵乘法的说明

- 时间复杂度降为$O(n^{\log 7})=O(n^{2.81})$
  - 理论意义：<u>由定义出发直接得出的算法未必是最好的</u>。
- Hopcroft与Kerr已经证明，两个2×2矩阵相乘必须要用7次乘法，如果需要进一步改进，应考虑3×3、4×4或者更高阶数的分块子矩阵——或者采用其他设计策略。
- 当n很大时，实际效果比直接定义求解的$O(n^3)$好。
- 根据矩阵乘法的定义可知，C_{ij}只与A的第i行、B的第j列相关，在实践中若遇到大矩阵，应考虑并行计算。

$$c_{ij}=\sum_{k=1}^{s}a_{ik}b_{kj}$$

思考

- 将矩阵分治乘法的思想，用于两个大整数的乘法呢？
- 根据定义，两个大整数A、B相乘，应该遍历B从低到高的数字，依次与大整数A相乘，最后将这些值相加。
  - 时间复杂度$O(n^2)$。
  - 可否将A、B写成两个规模小一半的整数A1,A2,B1,B2，然后计算它们的积呢？

大整数乘法

- 取大整数x和y的长度较大者的一半，记为k，则有：

$$\begin{cases} x = x_1 M^k + x_0 \\ y = y_1 M^k + y_0 \end{cases}$$

$$\Rightarrow xy = (x_1 M^k + x_0)(y_1 M^k + y_0)$$

$$= x_1 y_1 M^{2k} + (x_1 y_0 + x_0 y_1) M^k + x_0 y_0$$

- 计算长度为n/2的两个数的积，需要乘法次数为$O(n^2/4)$，而上面的式子需要4次乘法，总时间复杂度为$O(n^2)$，没有效果。因此，需要考虑改进。

大整数乘法：Karatsuba算法

- 事实上：

$$\begin{cases} x = x_1 M^k + x_0 \\ y = y_1 M^k + y_0 \end{cases}$$

$$\Rightarrow xy = \left(x_1 M^k + x_0\right)\left(y_1 M^k + y_0\right)$$

$$xy = x_1 y_1 M^{2k} + \left(x_1 y_0 + x_0 y_1\right) M^k + x_0 y_0$$

$$= x_1 y_1 M^{2k} + \left(\left(x_1 + x_0\right)\left(y_1 + y_0\right) - x_1 y_1 + x_0 y_0\right) M^k + x_0 y_0$$

- 上式只需要3次乘法(配合若干次加法和移位)即可完成，时间复杂度为$O(n^{\log 3})=O(n^{1.585})$。

老鼠吃奶酪

- 一只老鼠位于迷宫左上角(0, 0)，迷宫中的数字9处有块大奶酪。0表示墙，1表示可通过路径。试给出一条可行的吃到奶酪的路径；若没有返回空。
  - 假定迷宫是4连通的。

1	1	0	0	0	0	0	1
1	1	1	1	1	1	1	1
1	0	0	0	1	0	0	1
1	1	1	0	1	0	0	1
0	1	0	0	1	1	1	1
0	1	0	0	0	0	0	1
0	1	0	9	1	1	1	1
0	1	1	1	0	0	1	0

算法描述

- 假定当前位于(i,j)处，则依次计算(i-1,j), (i+1,j), (i,j-1), (i,j+1)4个相邻位置，如果相邻位置不是墙，则可以通过。
  - 递归该过程

```
void MousePath(const vector<vector<int> >& chess)
{
    vector<pair<int, int> > path;
    vector<vector<bool> > visit(chess.size(),
        vector<bool>(chess[0].size(), false));

    //开始路径搜索
    path.push_back(make_pair(0, 0));
    visit[0][0] = true;
    Search(chess, 0, 0, path, visit);
}
```

Code

	0	1	2	3	4	5	6	7
0:	1	1	0	0	0	0	0	1
1:	1	1	1	1	1	1	1	1
2:	1	0	0	0	1	0	0	1
3:	1	1	1	0	1	0	0	1
4:	0	1	0	0	1	1	1	1
5:	0	1	0	0	0	0	0	1
6:	0	1	0	9	1	1	1	1
7:	0	1	1	1	0	0	1	0

0, 0
0, 1
1, 1
1, 0
2, 0
3, 0
3, 1
4, 1
5, 1
6, 1
7, 1
7, 2
7, 3
6, 3

```
bool Search(const vector<vector<int> >& chess, int i, int j,
    vector<pair<int, int> >& path, vector<vector<bool> >& visit)
{
    if(chess[i][j] == 9)
    {
        Print(path);
        return true;
    }
    int iNext[] = {0, 0, -1, 1};
    int jNext[] = {-1, 1, 0, 0};
    int iCur, jCur;
    int m = (int)chess.size();
    int n = (int)chess[0].size();
    for(int k = 0; k < 4; k++)
    {
        iCur = i + iNext[k];
        jCur = j + jNext[k];
        if ((iCur < 0) || (iCur >= m) || (jCur < 0) || (jCur >= n))
            continue;
        if(!visit[iCur][jCur] && (chess[iCur][jCur] != 0))
        {
            path.push_back(make_pair(iCur, jCur));
            visit[iCur][jCur] = true;
            if(Search(chess, iCur, jCur, path, visit))
            {
              //如果求所有路径，则将下句替换成all.push_back(path);
                return true;
            }
            path.pop_back();
            visit[iCur][jCur] = false;
        }
    }
    return false;
}
```

百数问题

- 在1,2,3,4,5,6,7,8,9(顺序不能变)数字之间插入运算符+或者运算符-或者什么都不插入，使得计算结果是100。
  - 如：1+2+34-5+67-8+9=100
- 请输出所有的可行运算符方式。

思路解析

- 因为1,2,3,4,5,6,7,8,9中一共有8个位置可以放置运算符+、－或者<空>，因此一共有3^8种不同的插入方式，枚举所有表达式，计算该表达式的值，若等于100，则输出。
  - 可否有其他解决方案？
- 假设已完成a[0…i-1]的表达式，现考察a[i]的后面可以添加哪种字符？
  - 只有三种：+、－、<空>

Code

```
int _tmain(int argc, _TCHAR* argv[])
{
    int a[] = {1,2,3,4,5,6,7,8,9};
    int size = sizeof(a)/sizeof(int);
    vector<pair<int, char> > op;
    int count = 0;
    Calc(a, size, 0, 0, 0, op, 100, count);
    return 0;
}
```

```
//考察第cur个空位，当前表达式的值是n，最后一个数是last，操作符放置于op
bool Calc(const int* a, int size, int cur, int n, int last,
    vector<pair<int, char> >& op, int sum, int& count)
{
    if(cur == size-1)    //递归结束
    {
        last = 10 * last + a[size-1];
        if((LastOperator(op, cur-1) ? (n+last) : (n-last)) == sum)//找到解
        {
            count++;
            Print(count, a, op, size);
            return true;
        }
        return false;
    }
    last = 10*last+a[cur];
    Calc(a, size, cur+1, n, last, op, sum, count);  //<空>
    bool bAdd = LastOperator(op, cur-1);
    op.push_back(make_pair(cur, '+'));        //'+'
    Calc(a, size, cur+1, bAdd ? n+last : n-last, 0, op, sum, count);
    op.back().second = '-';                   //'-'
    Calc(a, size, cur+1, bAdd ? n+last : n-last, 0, op, sum, count);
    op.pop_back();                            //回溯
    return count != 0;
}
```

1: 123 + 45 - 67 + 8 - 9
2: 123 + 4 - 5 + 67 - 89
3: 123 - 45 - 67 + 89
4: 123 - 4 - 5 - 6 - 7 + 8 - 9
5: 12 + 3 + 4 + 5 - 6 - 7 + 89
6: 12 + 3 - 4 + 5 + 67 + 8 + 9
7: 12 - 3 - 4 + 5 - 6 + 7 + 89
8: 1 + 23 - 4 + 56 + 7 + 8 + 9
9: 1 + 23 - 4 + 5 + 6 + 78 - 9
10: 1 + 2 + 34 - 5 + 67 - 8 + 9
11: 1 + 2 + 3 - 4 + 5 + 6 + 78 + 9

我们在这里

- http://wenda.ChinaHadoop.cn
  - 视频/课程/社区
- 微博
  - @ChinaHadoop
  - @邹博_机器学习
- 微信公众号
  - 小象
  - 大数据分析挖掘

感谢大家！

恳请大家批评指正！

附：BM算法

BM算法

- Boyer-Moore算法是1977年Robert S. Boyer和J Strother Moore发明的字符串匹配算法，最坏情况下的时间复杂度为O(N)，在实践中比KMP算法的实际效能高。
- BM算法不仅效率高，而且构思巧妙，容易理解。

举例说明BM算法的运行过程

字符串	HERE IS A SIMPLE EXAMPLE
搜索词	EXAMPLE

坏字符

```
HERE IS A SIMPLE EXAMPLE
EXAMPLE
```

- 首先，"字符串"与"搜索词"头部对齐，从尾部开始比较。
- 这是一个很聪明的想法，因为如果尾部字符不匹配，那么只要一次比较，就可以知道前7个字符肯定不是要找的结果。
- “S”与“E”不匹配。这时，“S”就被称为“坏字符”(bad character)，即不匹配的字符。同时，“S”不包含在搜索词“EXAMPLE”之中，这意味着可以把搜索词直接移到“S”的后一位。
  - 还记得“暴力+KMP”中谈过的“模式串的字符两两不相等”的强要求么？放松成“模式串的首字符和其他字符不相等”，这里，迁移这个结论：模式串的尾字符和其他字符不相等。

坏字符引起的模式滑动

- 依然从尾部开始比较，发现"P"与"E"不匹配，所以"P"是"坏字符"。但是，"P"包含在搜索词"EXAMPLE"之中。所以，将搜索词后移两位，两个"P"对齐。

```
HERE IS A SIMPLE EXAMPLE
          EXAMPLE

HERE IS A SIMPLE EXAMPLE
            EXAMPLE
```

坏字符规则

```
HERE IS A SIMPLE EXAMPLE
           EXAMPLE
```

```
HERE IS A SIMPLE EXAMPLE
               EXAMPLE
```

- 后移位数 = 坏字符位置 - 坏字符在搜索词中的最右出现的位置
  - 如果"坏字符"不包含在搜索词之中，则最右出现位置为-1
- 以“P”为例，它作为“坏字符”，出现在搜索词的第6位(从0开始编号)，在搜索词中的最右出现位置为4，所以后移6-4=2位。
- 以前面的"S"为例，它出现在第6位，最右出现位置是-1(即未出现)，则整个搜索词后移6-(-1)=7位。

```
HERE IS A SIMPLE EXAMPLE
EXAMPLE
```

好后缀

- 依次比较，得到 “MPLE”匹配，称为"好后缀"(good suffix)，即所有尾部匹配的字符串。注意，"MPLE"、"PLE"、"LE"、"E"都是好后缀。

```
HERE IS A SIMPLE EXAMPLE
            EXAMPLE
```

遇到坏字符

- 发现“I”与“A”不匹配；“I”是坏字符。根据坏字符规则，此时搜索词应该后移2-(-1)=3位。问题是，有没有更优的移法？

```
HERE IS A SIMPLE EXAMPLE
          EXAMPLE
```

```
HERE IS A SIMPLE EXAMPLE
             EXAMPLE
```

考虑好后缀

```
HERE IS A SIMPLE EXAMPLE
          EXAMPLE
```

```
HERE IS A SIMPLE EXAMPLE
                EXAMPLE
```

好后缀规则

- 后移位数=好后缀的位置-好后缀在搜索词其余部分中最右出现位置
  - 如果好后缀在搜索词中没有再次出现，则为-1。
- 所有的“好后缀”(MPLE、PLE、LE、E)之中，只有“E”在“EXAMPL”之中出现，所以后移6-0=6位。
- “坏字符规则”只能移3位，“好后缀规则”可以移6位。每次后移这两个规则之中的较大值。
- 这两个规则的移动位数，只与搜索词有关，与原字符串无关。
  - 注：KMP中，往往称作文本串、模式串。

坏字符

- 继续从尾部开始比较，"P"与"E"不匹配，因此"P"是"坏字符"。根据"坏字符规则"，后移 $6 - 4 = 2$ 位。
  - 因为是最后一位就失配，尚未获得好后缀。

```
HERE IS A SIMPLE EXAMPLE
                 EXAMPLE

HERE IS A SIMPLE EXAMPLE
                 EXAMPLE
```