# 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
  - 计算HammingWeight
  - 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

# 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：$Time=\frac{2}{3}\cdot 1+\frac{2}{3}\cdot 3+\frac{2}{3}\cdot 9+\frac{2}{3}\cdot 27+\frac{2}{3}\cdot 81+\cdots+\frac{2}{3}\cdot 3^k$

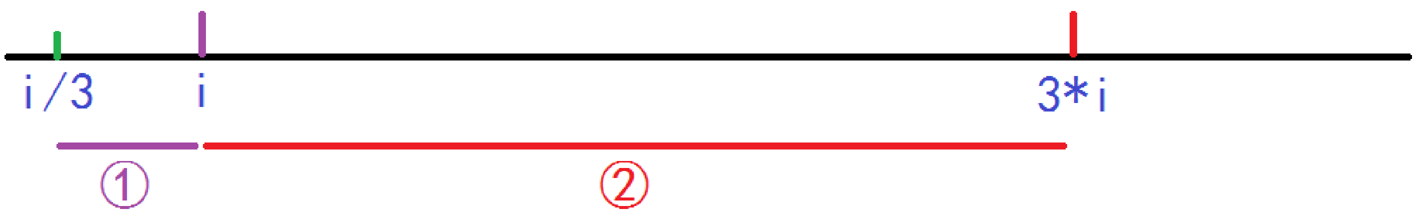
$$=\frac{2}{3}\cdot\left(1+3+9+27+\cdots+3^k\right)$$

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

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

# 图示分析

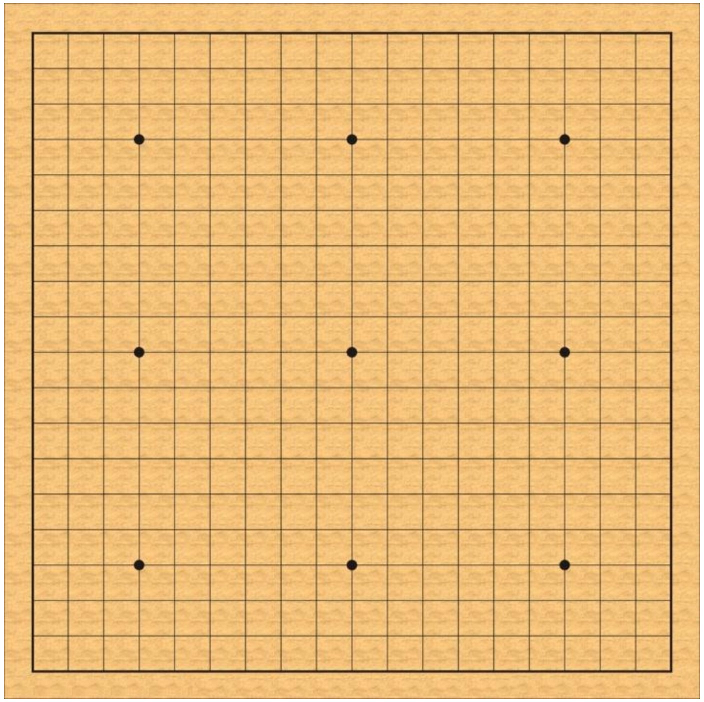
- 从下面的图示能够清楚的反映这一问题：




- 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)。

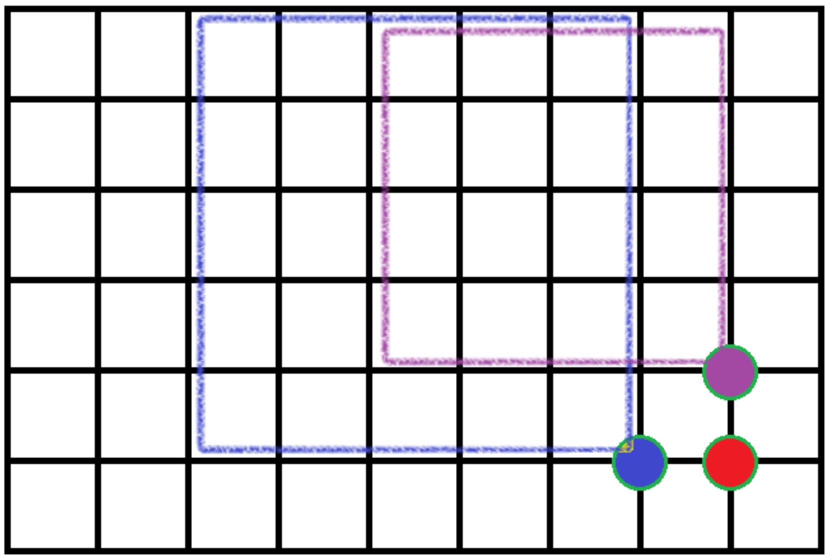
# 围棋中的正方形

- 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。

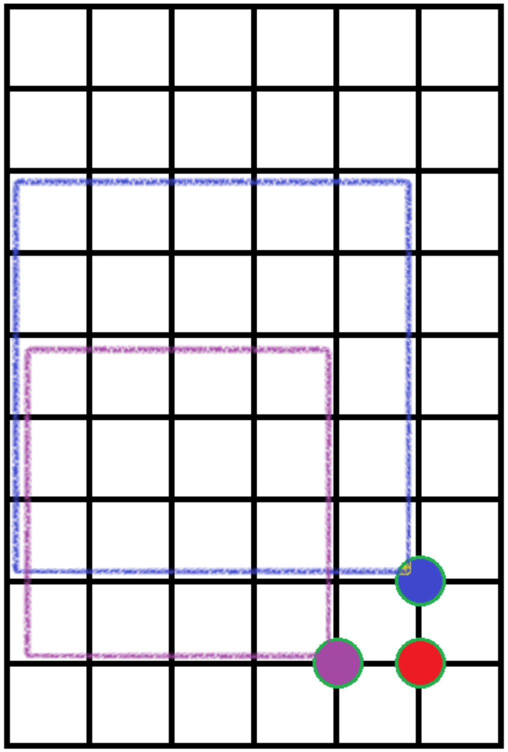


# 算法分析 $f(i,j)=\max(f(i-1,j),f(i,j-1)),\ i\neq j$

- 以(i,j)为右下角的正方形数目f(i,j)

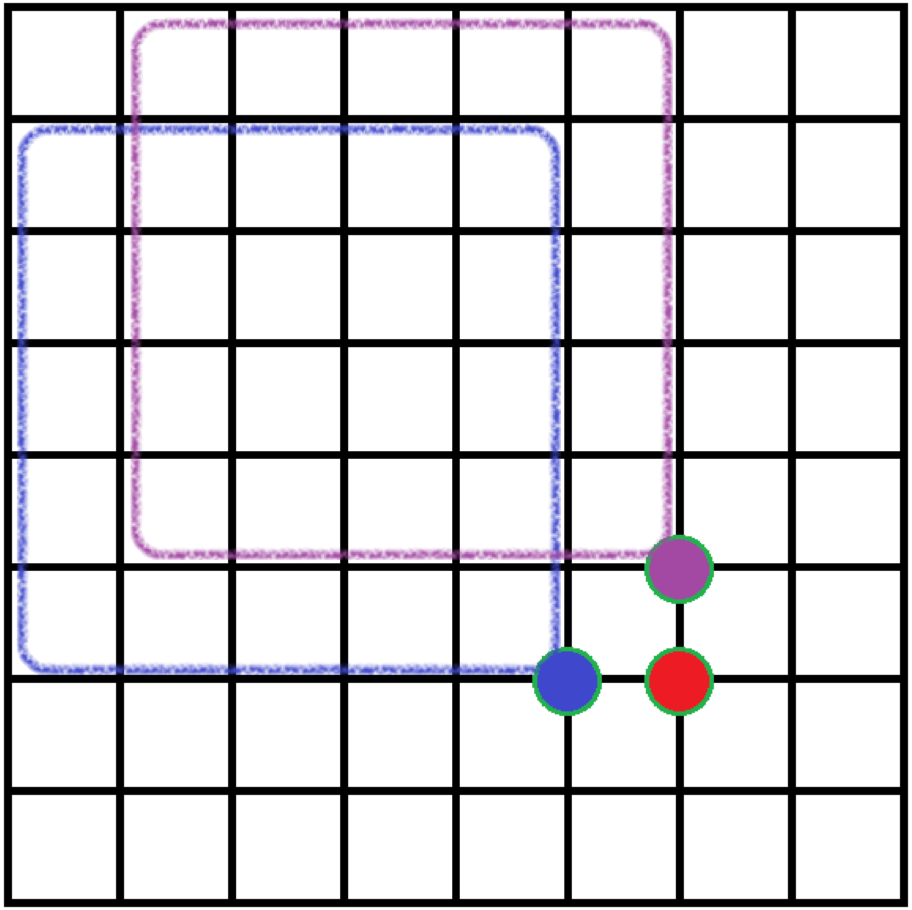


$$f(i,j)=f(i,j-1),\quad i<j$$



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

# 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

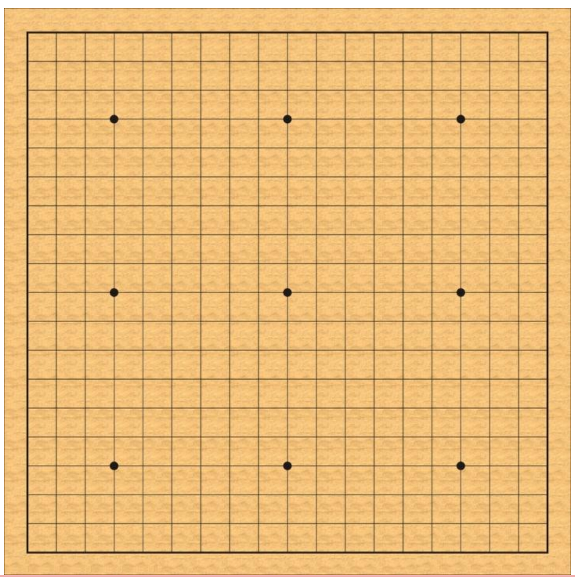
| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 0 | 1 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 0 | 1 | 2 | 3 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| 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：$x = x_0 - \dfrac{f(x_0)}{f'(x_0)}$

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

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

# 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

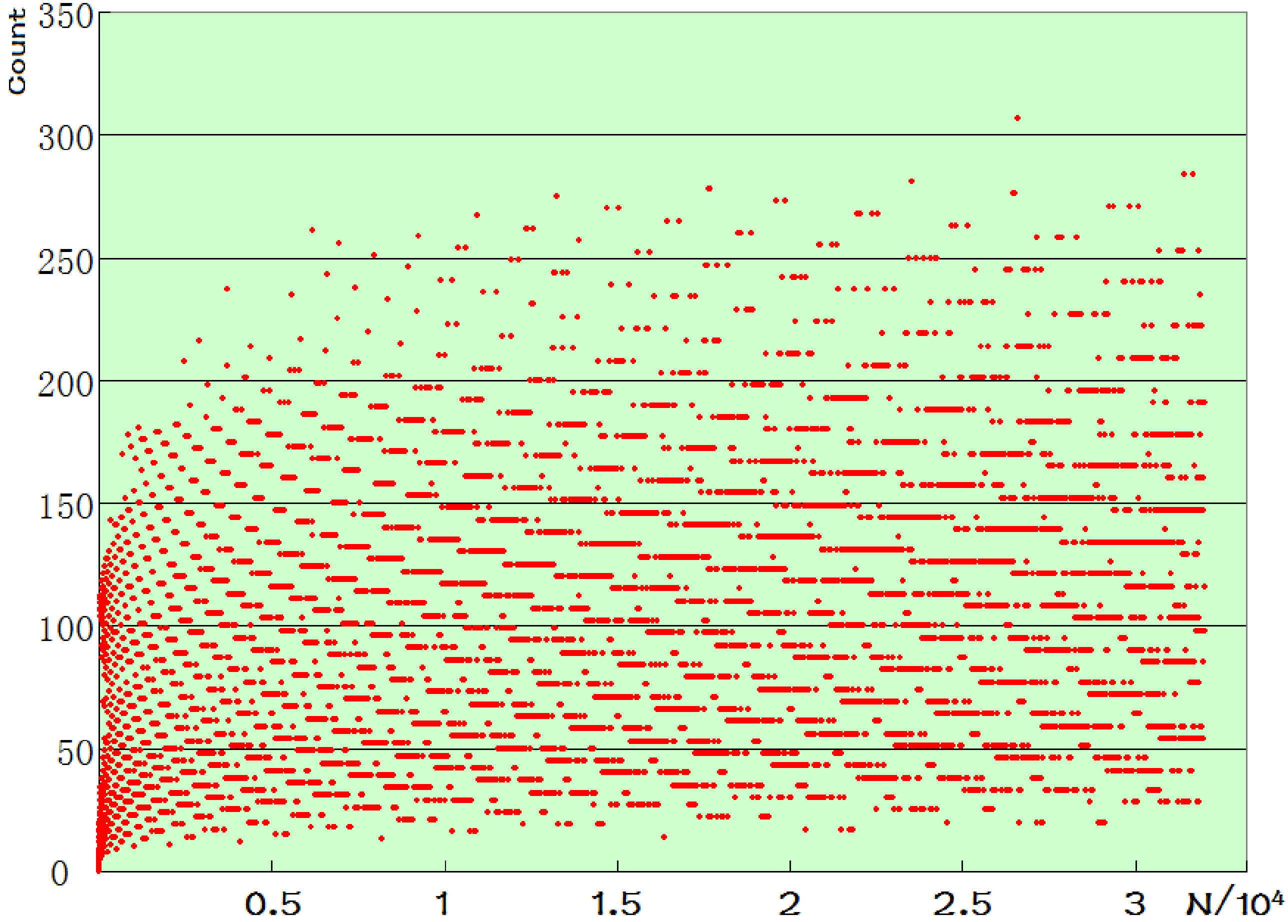
- 该问题又称3n+1猜想、角谷猜想、哈塞猜想、乌拉姆猜想、叙拉古猜想等。
- 过程非常简单：给定某正整数N，若为偶数，则N被更新为N/2；否则，N被更新为3*N+1；问：(1)经过多少步N变成1？(2)是否存在某整数X无法变成1？
- 思考：
  - 如果已经计算得到1~N-1的变换次数，如何计算N的变换次数？

# Code

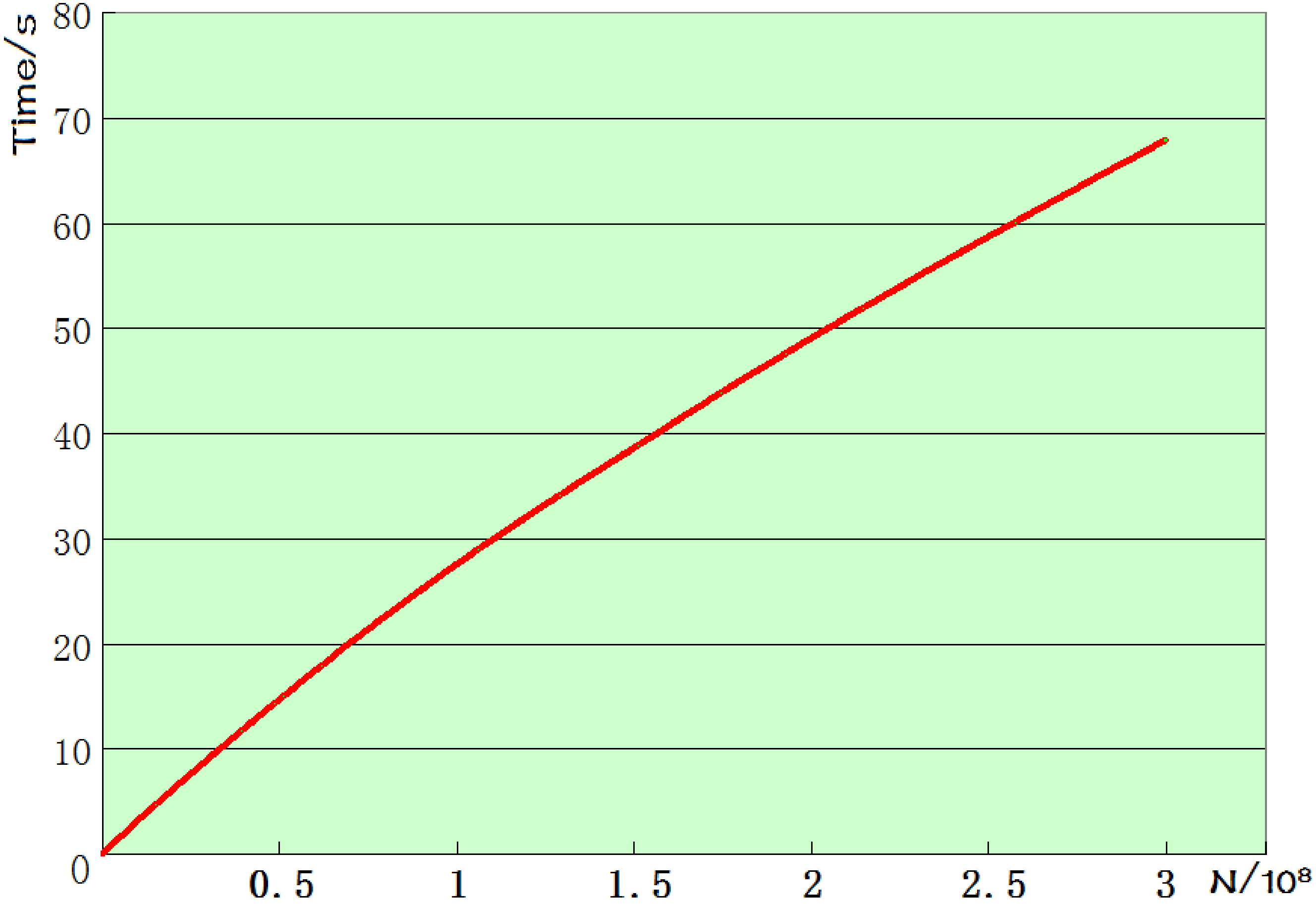
```
void Calc(long long i, int* p, int N, int timeStart)
{
    int cur = (int)i;
    int t = 0;
    while(true)
    {
        if(i % 2 == 0)
        {
            i /= 2;
            t++;
        }
        else
        {
            i = i * 3 + 1;
            t++;
        }
        if((i < N) && (p[(int)i]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



# 求1的个数

- 给定一个32位无符号整数N，求整数N的二进制数中1的个数。
  - 显然：可以通过不断的将整数N右移，判断当前数字的最低位是否为1，直到整数N为0为止。
    - 平均情况下，大约需要16次移位和16次加法。
  - 有其他更精巧的办法吗？

# 两种常规Code

- 思路1：
  - 每次右移一位
  - 奇数则累加1
- 思路2：
  - 每次最低位清0
  - 只需要n&=(n-1)即可

```
int OneNumber(int n)
{
    int c = 0;
    while(n != 0)
    {
        c += (n&1); //奇数则累加1
        n >>= 1;
    }
    return c;
}

int OneNumber2(int n)
{
    int c = 0;
    while(n != 0)
    {
        n &= (n-1); //最低为1的位清0
        c++;
    }
    return c;
}
```

# 分治

- 假定能够求出N的高16位中1的个数a和低16位中1的个数b，则a+b即为所求。
- 为了节省空间，用一个32位整数保存a和b：
  - 高16位记录a，低16位记录b，
  - (0xFFFF0000&N)筛选得到a；
  - (0x0000FFFF&N)筛选得到b；
  - (0xFFFF0000&N) + (0x0000FFFF&N)>>16
- 如何得到高16位中1的个数a呢？
  - 如何得到低16位中1的个数b呢？
  - 递归

# 递归过程

- 如果二进制数N是16位，则统计N的高8位和低8位各自1的数目a和b，而a、b用某一个16位数X存储，则使用0xFF00、0x00FF分别于X做与操作，筛选出a和b；
  - 原问题中的数据是32位，因此需要2个0xFF00/0x00FF，即0xFF00FF00/0x00FF00FF
- 如果二进制数是8位，则统计高4位和低4位各自1的数目，使用0xF0/0x0F分别做与操作，筛选出高4位和低4位；
  - 需要4个0xF0/0x0F，即0xF0F0F0F0/0x0F0F0F0F
- 如果是4位则统计高2位和低2位各自1的数目，用0xC/0x3筛选；
  - 需要8个0xC/0x3，即0xCCCCCCCC/0x33333333
- 如果是2位则统计高1位和低1位各自1的数目，用0x2/0x1筛选；
  - 需要16个0x2/0x1，即 0xAAAAAAAA/0x55555555

# Code

```c
int HammingWeight(unsigned int n)
{
    n = (n & 0x55555555) + ((n & 0xaaaaaaaa)>>1);
    n = (n & 0x33333333) + ((n & 0xcccccccc)>>2);
    n = (n & 0x0f0f0f0f) + ((n & 0xf0f0f0f0)>>4);
    n = (n & 0x00ff00ff) + ((n & 0xff00ff00)>>8);
    n = (n & 0x0000ffff) + ((n & 0xffff0000)>>16);
    return n;
}
```

# 总结与应用

- HammingWeight使用了分治/递归的思想，将问题巧妙解决，降低了运算次数。
  - 还可以使用其他分组方案，如3位一组等。
- 如果定义两个长度相等的0/1串中对应位不相同的个数为海明距离(即码距)，则某0/1串和全0串的海明距离即为这个0/1串中1的个数。
- 两个0/1串的海明距离，即两个串异或值的1的数目，因此，该问题在信息编码等诸多领域有广泛应用。

# 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*

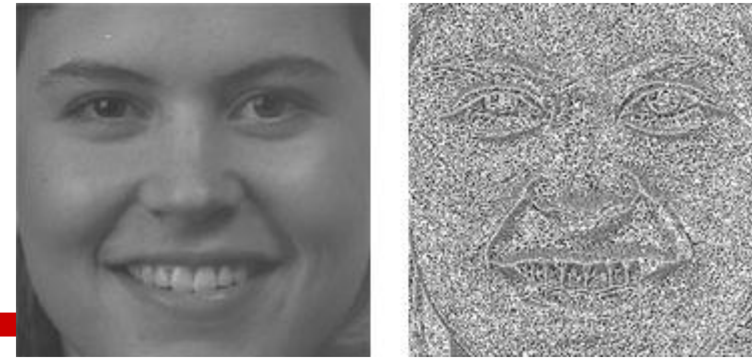
```
//循环左移一位
int RotateShiftLeft(int x, int N)
{
    int high = (x >> (N-1));
    x &= ((1<<(N-1)) -1);
    x <<= 1;
    x |= high;
    return x;
}
```

```
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;
            }
        }
    }
```

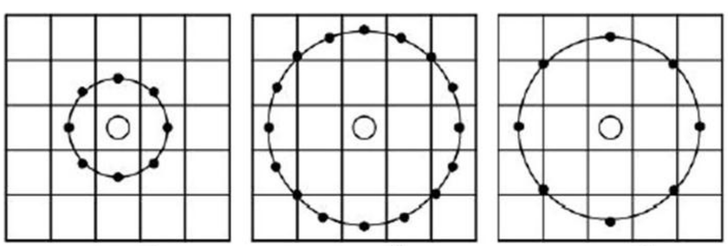
# 实验结果

- 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



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

# 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

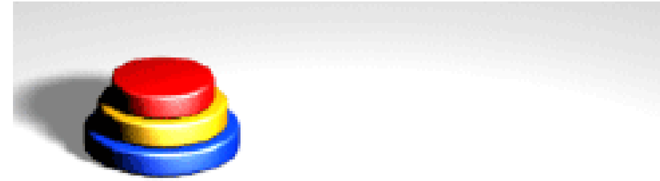
- N个盘子的Hanoi塔，至少需要几次移动？

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

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

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

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



# 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：由定义出发直接得出的算法未必是最好的。
- 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

```
void Calc(const int* a, int size)
{
    list<pair<int, bool> > op;
    int m = (int)pow(3.0, size-1);
    int i,j,t;
    int ans = 0;    //第几个解
    for(i = 0; i < m; i++)
    {
        //准备表达式
        op.clear();
        t = i;
        for(j = 0; t != 0; j++)
        {
            if(t%3 != 0)    //' ':0, '+':1, '-':2
                op.push_back(make_pair(j, (t%3 == 1)));
            t /= 3;
        }

        //计算表达式
        if(CalcExpress(a, size, op) == 100)
            Print(++ans, a, size, op);
    }
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    int a[] = {1,2,3,4,5,6,7,8,9};
    int size = sizeof(a)/sizeof(int);

    //方法1：直接计算
    Calc(a, size);

    //方法2：递归
    list<pair<int, bool> > op;
    int count = 0;
    Calc(a, size, 0, 0, 0, op, 100, count);
    return 0;
}
```

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

# Aux Code

```
int CalcExpress(const int* a, int size, const list<pair<int, bool> >& op)
{
    int cur = 0;
    int i = 0;
    bool bAdd = true;
    int t;
    for(auto o = op.begin(); o != op.end(); o++)
    {
        t = GetNumber(a, i, o->first);
        if(bAdd)
            cur += t;
        else
            cur -= t;
        bAdd = o->second;
        i = o->first+1;
    }
    t = GetNumber(a, i, size-1);
    if(bAdd)
        cur += t;
    else
        cur -= t;
    return cur;
}
```

```
//[from, to]
int GetNumber(const int* a, int from, int to)
{
    int n = 0;
    for(int i = from; i <= to; i++)
        n = 10 * n + a[i];
    return n;
}
```

# Code2：递归

```
//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,
    list<pair<int, bool> >& 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
        {
            Print(++count, a, size, op);
            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, true));      //'+'
    Calc(a, size, cur+1, bAdd ? n+last : n-last, 0, op, sum, count);
    op.back().second = false;                //'-'
    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
  - 小象学院
  - 大数据分析挖掘

感谢大家！

恳请大家批评指正！