Edit by James Wu 2011/1/10

1. 概率算法部分

1.0 几个基本概念

1.0.1 期望时间和平均时间的区别

确定算法的平均执行时间：输入规模一定的所有输入实例是等概率出现时，算法的平均执行时间

概率算法的期望执行时间：反复解同一个输入实例所花的平均执行时间

概率算法的平均期望时间：所有输入实例上平均的期望执行时间

概率算法的最坏期望时间：最坏的输入实例上的期望执行时间

1.0.2 Uniform 函数

在 X 中随机，均匀和独立地取一个元素的算法：

```
ModularExponent(a, j, p){
   //求方幂模 s=a^j mod p, 注意先求 a^j 可能会溢出
   s ← 1;
   while j>0 do {
        if (j is odd) s ← s·a mod p;
        a ← a^2 mod p;
        j ← j div 2;
   }
   return s;
}
Draw (a, p) {
   // 在 X 中随机取一元素
   j ← uniform(1..p-1);
   return ModularExponent(a, j, p); // 在 X 中随机取一元素
}
```

1.1 概率算法的分类

1.1.1 数字算法

主要用于找到一个数字问题的近似解

使用的理由

现实世界中的问题在原理上可能就不存在精确解，例如，实验数据本身就是近似的，一个无理数在计算机中只能近似地表示

精确解存在但无法在可行的时间内求得，有时答案是以置信区间的形式给出的

1.1.2 Monte Carlo 算法

特点：MC 算法总是给出一个答案，但该答案未必正确，成功(即答案是正确的)的概率正比于算法执行的时间

缺点：一般不能有效地确定算法的答案是否正确

1.1.3 Las Vagas 算法

LV 算法绝不返回错误的答案。

特点：获得的答案必定正确，但有时它仍根本就找不到答案。

和 MC 算法一样，成功的概率亦随算法执行时间增加而增加。无论输入何种实例，只要算法在该实例上运行足够的次数，则算法失败的概率就任意小。

1.1.4 Sherwood 算法

Sherwood 算法总是给出正确的答案。

当某些确定算法解决一个特殊问题平均的时间比最坏时间快得多时，我们可以使用 Sherwood 算法来减少，甚至是消除好的和坏的实例之间的差别。

1.2 算法的实现方式

1.1.1 数字算法

1.1.1.1 HitorMiss 算法计算积分（面积法）

```
HitorMiss (f, n) {
    k ← 0;
    for i ← 1 to n do {
        x ← uniform(0, 1);
        y ← uniform(0, 1);
        if f(x,y)在满足的面积内  then k++;
    }
    return k/n;
}
```

1.1.1.2 Crude 算法计算积分（积分化求平均值法）

```
Crude (f, n, a, b) {
    sum ← 0;
    for i ← 1 to n do {
        x ← uniform(a, b);
        sum ← sum + f(x);
    }
    return (b-a)sum/n;
}
```

1.1.1.3 两种方法的比较

对于给定的迭代次数 n，Crude 算法的方差不会大于 HitorMiss 的方差。但不能说，Crude 算法总是优于 HitorMiss。因为后者在给定的时间内能迭代的次数更多。例如，计算 π 值时，Crude 需计算平方根，而用投镖算法 darts 时，即 HitorMiss 无需计算平方根。

1.1.1.4 确定的算法——梯形算法（上底加下底乘以高除以 2）

```
Trapezoid (f, n, a, b) {
    // 假设  n ≥ 2
    delta ← (b-a)/(n-1);
    sum ← (f(a) + f(b))/2;
    for x ← a+delta step delta to b – delta do
        sum ← sum + f(x)
    return sum × delta;
}
```

一般地，在同样的精度下，梯形算法的迭代次数少于 MC 积分，但是有时确定型积分算法求不出解，若用 MC 积分则不会发生该类问题，或虽然发生，但概率小得多。

在确定算法中，为了达到一定的精度，采样点的数目随着积分维数成指数增长，例如，一维积分若有 100 个点可达到一定的精度，则二维积分可能要计算 100^2 个点才能达到同样的精度，三维积分则需计算 100^3 个点。(系统的方法)

但概率算法对维数的敏感度不大，仅是每次迭代中计算的量稍增一点，实际上，MC 积分特别适合用于计算 4 或更高维数的定积分。

若要提高精度，则可用混合技术：部分采用系统的方法，部分采用概率的方法

1.1.1.5 例 1：求集合的势

问题描述：估算一个集合中元素的个数

解决思路：设 X 是具有 n 个元素的集合，我们有回放地随机，均匀和独立地从 X 中选取元素，设 k 是出现第 1 次重复之前所选出的元素数目，则当 n 足够大时，k 的期望值趋近为$\beta\sqrt{n}$，这里

$$\beta = \sqrt{\frac{\pi}{2}} \approx 1.253$$

利用此结论可以得出估计|X|的概率算法：

$$\beta\sqrt{n} = \sqrt{\frac{n\pi}{2}} = k \implies n = \frac{2k^2}{\pi}$$

算法：

```
SetCount (X) {
    k ← 0; S ← Φ;
    a ← uniform(X);
    do {
        k++;
        S ← S∪{a}; a ← uniform(X);
    } while (a∉S)
    return 2k²/π
}
```

复杂度：注意：∵k 的期望值是$\sqrt{\frac{n\pi}{2}}$，∴上述算法 n 需足够大，且运行多次后才能确定 n=|X|，即取多次运行后的平均值才能是 n。

该算法的时间和空间均为$\theta(\sqrt{n})$，因为$k = \theta(\sqrt{n})$

1.1.1.6 例 2：多重集合中不同数目的估计

用散列表$\pi(m+1), m = 5 + \lg M$，若以元素 e 的 hash(e)以 00…01 开头（前面 k-1 个 0），则$\pi(k) = 1$

最后返回π中第一个出现 0 的位置z，则集合中不同元素的下界和上界分别是 $[2^{z-2}, 2^z]$

复杂度：时间 O(N)，空间：O(lgN)

1.1.2 Sherwood 算法

1.1.2.1 Sherwood 算法的基本思想

对于一个确定性算法，它的平均时间很容易计算：

T（确定性算法平均时间）= （每次执行的时间之和）/ （执行总次数）

但是会存在这样一个不好的情况：**某次执行的时间远远大于平均执行时间，比如最坏情况。**

Sherwood 算法的目的就是为了解决这个问题，**利用概率的方法（或者说随机的方法）避免了最坏情况的发生**，但这是要付出其他的时间代价（比如在取随机的时候需要花费一点时间），所以 Sherwood 算法的平均时间稍稍大于确定性算法的平均时间。

Sherwood 算法的应用范围：在确定性算法中，它的平均执行时间较优，最坏性能较差，这样的算法就用 Sherwood 算法去改进

Sherwood 一般方法是：

① 将被解的实例变换到一个随机实例。// 预处理

② 用确定算法解此随机实例，得到一个解。

③ 将此解变换为对原实例的解。 // 后处理

1.1.2.2 Sherwood 算法预处理的数学模型

1. 确定性算法：f: X -> Y
2. 确定性算法的实例集合：X, size 为 n 时写作 Xn
3. Sherwood 算法用于均匀随机抽样的集合：A，size 为 n 时写作 An，|An|=|Xn|
4. 随机抽样的预处理及后处理时用到的一对函数，对应上面的①③，（PPT 在这里有误）

   u: X × A -> Y

   v: A × Y -> X

   u,v 满足三个性质：

   - $(\forall n \in N)(\forall x, y \in Xn)(\exists! r \in An)$，使得 $u(x, r) = y$

     这条对应①，其中∃!表示有且仅有一个
   - $(\forall n \in N)(\forall x \in Xn)(\forall r \in An)$，使得 $f(x) = v(r, f(u(x, r)))$

     这条对应③
   - 函数 u,v 在最坏情况下能够有效计算

1.1.2.3 Sherwood 算法的过程

确定算法 f(x)可改造为 Sherwood 算法：

```
RH(x) {
    // 用 Sherwood 算法计算 f(x)
    n ← length[x]; // x 的 size 为 n
    r ← uniform(An); // 随机取一元素
    y ← u(x, r);   //将原实例 x 转化为随机实例 y
    s ← f(y); // 用确定算法求 y 的解 s
    return v(r, s); // 将 s 的解变换为 x 的解
}
```

1.1.2.4 例 1：选择和排序的 Sherwood 算法

只需进行简单的打乱顺序即可，u 即表示打乱顺序函数 shuffle

```
Shuffle (T) {
    n ← length[ T ];
    for i ← 1 to n-1 do {
        // 在 T[ i..n ]中随机选 1 元素放在 T[ i ]上
        j ← uniform(i..n);
        T[ i ] <-> T[ j ];
    }
}
```

1.1.2.5 例 2：离散对数计算

- **问题描述**：设 $a=g^x \bmod p$，记 $\log_{g,p}a=x$，称 x 为 a 的(以 g 为底模除 p)对数。从 p，g，a 计算 x 称为离散对数问题。
  问题在于：给出 p,g,a，怎么求 x

- **简单算法**：
  ① ∀x，计算 g^x
  最多计算 0≤x≤ p-1 或 1≤x≤p，因为实际上离散对数<g>是循环群；
  ② 验证 $a=g^x \bmod p$ 是否成立。

```
dlog(g, a, p) { // 当这样的对数不存在时，算法返回 p
    x ← 0; y ← 1;
    do {    x++;
       y ← y*g;    // 计算 y=g^x
    } while ( a ≠ y mod p) and (x ≠ p);
        return x
}
```

问题：最坏 O(p)，若 P 很大怎么办？所以简单算法不行
而且 x 的算出来的快慢取决于 a 的取值，a 的取值能够让算法较早找到正确的 x，则算法很快就完了，否则很慢，直到 p。

- **Sherwood 算法解决方法**
  根据上面的分析，Sherwood 算法应该使得这个算法不会根据 a,p 的取值影响算法的快慢
  定理：
  1. $\text{Log}_{g,p}(st \bmod p) = (\log_{g,p} s + \log_{g,p} t) \bmod (p - 1)$
  2. $\text{Log}_{g,p}(g^r \bmod p) = r, \quad 0 <= r <= p - 2$

```
dlogRH(g, a p) { // 求 log_{g,p}a, a = g^x mod p，求 x
    // Sherwood 算法
    r ← uniform(0..p-2);
    b ← ModularExponent(g, r, p); //求幂模 b=g^r mod p，定理 1 真数的一部分
    c ← ba mod p; //((g^r modp)(g^x modp))modp=g^{r+x}modp=c, 定理 2 中的真数
    y ← log_{g,p}c; // 使用确定性算法求 log_{p,g}c, y=r+x，定理 2
```

```
    return (y-r) mod (p-1); // 求 x，定理 1
}
```

在这里，唯一耗费时间的是 **b ← ModularExponent(g, r, p)**，它的执行时间与 a,p 的取值无关，只与随机取出的 r 有关

1.1.2.6 例 3：搜索有序表

问题描述：在有序表中搜索 x，如果存在返回其 index

基本搜索函数：

```
Search(x, i) {      //从 index = i 开始搜索 x，这是一个顺序查找过程
    while x > val[i] do
        i ← ptr[i];
    return i;
}
```

4 种算法：

- **A(x)，时间复杂度 O(n)**

```
A(x) {
    return Search(x, head);
}
```

- **D(x)，时间复杂度 O(n/3)**

```
D(x) {
    i ← uniform(1..n);
    y ← val[i];
    case {
        x < y: return Search(x, head); // case1
        x > y: return Search(x, ptr[i]); // case2
        otherwise: return i; // case3，  x = y
    }
}
```

- **B(x)，时间复杂度 O($\sqrt{n}$)**

算法基本思想：

对于一个有序表 S，它上面的元素分别是 $a_1,a_2,\ldots,a_n$，它们之间可以是乱序的，要查找的 x 是其中一员。

若把 $a_1,a_2,\ldots,a_n$ 有序排列成为 $a_{o1}<a_{o2}<\ldots<a_{on}$，x 仍然是其中一员。

把 $a_{o1}<a_{o2}<\ldots<a_{on}$ 划分成 L 个区间，x 也必然是某个区间中的一员。

那么根据 Search(x, i)是一个有序查找过程，只需要找到 x 所在区间中在 x 之前的元素的 index，或者 x 所在区间的前面任何一个区间的元素的 index，在调用 Search(x, index)，其时间肯定不超过 2n/L，而且期望时间是 n/L。

而根据 S 中元素分布的均匀性，$a_1,a_2,\ldots,a_n$ 排列的前 L 个元素在概率上，是由 $a_{o1}<a_{o2}<\ldots<a_{on}$ 的 L 划分中，每个区间各取一个元素组成的，所以会出现：$a_{o1}<a_{o2}<\ldots<a_{on}$ 的 L 划分中 x 所在区间中在 x 之前的元素，或者是，x 所在区间的前面任何一个区间的元素，满足这两点中的任意一点即可，数越大越靠近 X。

那么算法可以这么描述：找 S 的 $a_1,a_2,\ldots,a_n$ 序列的前 L 个元素中最大的数，得到它的 index，然后用 Search(x, index)，经过期望时间

n/L，最终找到 x，则时间复杂度是 O(L + n/L)，当 L = $\sqrt{n}$时取到最小值 O(2$\sqrt{n}$)

```
B(x) { //设 x 在 val[1..n]中
    i ← head;
    max ← val[i]; // max 初值是表 val 中最小值
    for j ← 1 to √n do
    { // 在 val 的前个√n数中找不大于 x
        y ← val[ j ]; // 的最大整数 y 相应的下标 i
        if max < y ≤x then {
            i ← j;
            max ← y;
        } //endif
    } // endfor
    return Search(x, i); // 从 y 开始继续搜索
}
```

- C(x)，时间复杂度 O($\sqrt{n}$)，平滑 B(x)在不同实例上的执行时间

```
C(x) { //设 x 在 val[1..n]中
    i ← head;
    max ← val[i]; // max 初值是表 val 中最小值
    for k ← 1 to √n do
    { // 在 val 中进行√n次寻找，找到最大的一个整数及其下标
        j ← uniform(1…n);
        y ← val[ j ]; // 的最大整数 y 相应的下标 i
        if max < y ≤x then {
            i ← j;
            max ← y;
        } //endif
    } // endfor
    return Search(x, i); // 从 y 开始继续搜索
}
```

1.1.3 Las Vegas 算法

1.1.3.1 与 Sherwood 算法比较

Sherwood 算法不算很优，因为它只改进确定性算法的最坏情况，所以平均执行时间与确定性算法相差无几。

为了提高平均执行时间，就采用 Las Vegas 算法。

Sherwood 算法能够计算出一个给定实例的执行时间上界，因为它总是能够正确运行，所以每次都有一定的执行时间，取最大值就是上界。

Las Vegas 的时间上界可能不存在，因为它可能找不到解陷入死循环。

1.1.3.2 Las Vegas 算法的特点

可能不时地要冒着找不到解的风险，算法要么返回正确的解，要么随机决策导致一个僵局。

若算法陷入僵局，则使用同一实例运行同一算法，有独立的机会求出解。

成功的概率随着执行时间的增加而增加。

1.1.3.3 算法的一般形式

LV(x, y, success) —— x 是输入的实例，y 是返回的参数，success 是布尔值，true 表示成功，false 表示失败

p(x) —— 对于实例 x，算法成功的概率

s(x) —— 算法成功时的期望时间

e(x) —— 算法失败时的期望时间

```
Obstinate(x) {
    repeat
        LV(x, y, success);
    until success;
    return y;
}
```

设 t(x)是算法 obstinate 找到一个正确解的期望时间，则

$t(x) = p(x)s(x) + (1 - p(x))(e(x) + t(x))$

注：之所以是$e(x) + t(x)$，是因为t(x)是指第一次成功的期望时间，第一次失败，后面再成功就需要花费$e(x) + t(x)$的时间

即

$$t(x) = s(x) + \frac{1 - p(x)}{p(x)} e(x)$$

若要最小化 t(x)，则需在 p(x), s(x)和 e(x)之间进行某种折衷，例如，若要减少失败的时间，则可降低成功的概率 p(x)。

1.1.3.4 例 1：8 皇后问题

问题描述：棋盘上放 8 个皇后，行、列、135°、45°不冲突

确定性算法：回溯法

Las Vegas 算法的要求：

1. 无行冲突：显然
2. 无列冲突：保存已经选择的列，不选它们即可
3. 无斜线冲突：保存已经占用的斜线，不选它们即可，每选一个 $a_{i,j}$，把 i-j 加入已经选好的 135°集合，把 i+j 加入已经选好的 45°集合

LasVegas 算法：

```
QueensLv (success){ //贪心的 LV 算法，所有皇后都是随机放置
    //若 Success=true，则 try[1..8]包含 8 后问题的一个解。
  col,diag45,diag135←Φ；//列及两对角线集合初值为空
  k ←0；//行号
  repeat      //try[1..k]是 k-promising，考虑放第 k+1 个皇后
    nb ←0；//计数器，nb 值为（k+1)th 皇后的 open 位置总数
    for i ←1 to 8 do { //i 是列号
      if (i     col) and (i-k-1      diag45) and (i+k+1      diag135) then{
          //列 i 对（k+1）th 皇后可用，但不一定马上将其放在第 i 列
          nb ←nb+1；
```

```
                    if uniform(1..nb)=1 then //或许放在第 i 列
                        j ←i；//注意第一次 uniform 一定返回 1，即 j 一定有值 1
                }//endif
            }//endfor，在 nb 个安全的位置上随机选择 1 个位置 j 放置之
        if(nb > 0) then{ //nb=0 时无安全位置，第 k+1 个皇后尚未放好
        //在所有 nb 个安全位置上，(k+1)th 皇后选择位置 j 的概率为 1/nb
                k←k+1；//try[1..k+1]是(k+1)-promising
                try[k] ←j；//放置(k+1)th 个皇后
            col ←col∪{ j }；
            diag45 ←diag45 ∪{ j-k }；
            diag135 ←diag135 ∪{ j+k }；
        } //endif
    until （nb=0）or（k=8）；//当前皇后找不到合适的位置或 try 是
                            // 8-promising 时结束。
    success ← （nb>0）；
}
```

4. 问题的改进

上述完全随机的方法的期望时间还是有点多，就采用折中的方法：先用 Las Vegas 排头几行，再用回溯法去做，只需将 QueensLV 的最后两行改为：

```
until nb = 0 or k = stepVegas；
if （nb>0)then //已随机放好 stopVegas 个皇后
        backtrace (k, col, diag45, diag135,success);
else
        success ←false；
```

结论：一半略少的皇后随机放置较好。

1.1.3.5 例 2：模 P 平方根

问题描述：设 p 是一个奇素数，若整数 x∈[1,p-1]且存在一个整数 y，使得 $x \equiv y^2 \ mod \ p$，

则称 x 为模 p 的二次剩余，若 y ∈[1,p-1]，则 y 称为 x 模 p 的平方根。

问题所求：当给定 x 和奇素数 p 时，求 y。

重要结论：

结论 PPT 上有很多，这里只列举一个，若 y 是 x 模 p 的平方根，则 p-y 也是。

Las Vegas 解决算法：

```
rootLV(x, p, y, success)
{//计算 y
  a ←uniform(1..p-1);//我们并不知道 a 应取多少
  if a^2 ≡ x mod p then {   //可能性很小
      success ← true；
      y ← a；
  }
```

```
  Else
  {
      计算 c，d 使得0 ≤ c, d ≤ p − 1, (a + √x)^((p−1)/2) ≡ c + d√x (mod p)
      if   d=0   then
           success ← false；//无法求出
      else
      {//c=0
           success ← true；
           y ← Euclid_Modify(d, p);// 计算y, 使得d * y ≡ 1 mod p
      }
  }
}

Euclid_Modify(a, b)
{//  计算x, 1 ≤ x ≤ b − 1, 使得ax ≡ 1 mod b，即ax + by ≡ 1 mod b
      stack stackAB(int, int)          //定义一个(int, int)类型的栈

      While(b > 0)
      {
           stackAB.Push(a,b);       //将(a,b)入栈
           t←a;
           a←b;
           b←t – t % b;
      }

      x = 1; y = 0;

      while(stackAB != empty)
      {
           (a,b) ←stackAB.Pop();   //出栈
           t←x;
           x←y;
           y←t – (a/b)y;
      }

      If(x>0)
           While(x – b > 0)
                x←x – b;
      else if(x < 0)
           while(x + b < 0)
                x←x + b;
      return x;
}
```

1.1.3.6 例 3：整数的因数分解

问题描述：寻找合数 n 的非平凡因数（不是 1 和 n）

算法步骤：

- Step1：在 1~n-1 之间随机选择 x

i)若 x 碰巧不与 n 互素，则已找到 n 的一个非平凡因子(即为 x)

ii)否则设$y \equiv x^2 \ mod \ n$，若 y 是 k-平滑，则将 x 和 y 的因数分解保存在表里。

此过程重复直至选择了 k+1 个互不相同的整数，并且这些整数的平方模 n 的因数已分解(当 k 较小时，用 split(n)分解）

- Step2：在 k+1 个等式之中找一个非空子集，使相应的因数分解的积中前 k 个素数的指数均为偶数(包含 0)，用 0-1 矩阵去实现

使用 Gause-Jordan 消去法可得到线性相关的行

- Step3：在 step2 中找到线性相关的行后：

1. 令 a 为相应 x_i 的乘积
2. 令 b 是 y_i 的乘积开平方

若$a \neq \pm b \bmod n$，则只需求 a+b 和 n 的最大公因子即可获得 n 的非平凡因子。

时间分析：如何选择 k.

1)k 越大，$x^2 \ mod \ n$是 k-平滑的可能性越大(x 是随机选取的)

2)k 越小，测试 k-平滑及因数分解 y_i 的时间越小，确定 y_i 是否线性相关的时间也越少，但$x^2 \ mod \ n$不是 k-平滑的概率也就较大。

设$\mathrm{L} = e^{\sqrt{\ln n \ln\ln n}}, b \in R^+$

通常取$\mathrm{k} \approx \sqrt{\mathrm{L}}$时较好，此时 Dixon 算法分裂 n 的期望时间为$\mathrm{O}(L^2) = \mathrm{O}(e^{2\sqrt{\ln n \ln\ln n}})$，成功的概率至少为 1/2.

1.1.4 Monte Carlo 算法

1.1.4.1 基本概念

Monte Carlo 算法偶然会犯错，但它无论对何实例均能以高概率找到正确解。当算法出错时，没有警告信息。

偏真偏假的概念只在 Monte Carlo 算法里出现

- Def1：设 p 是一个实数，且 1/2<p<1，若一个 MC 算法以不小于 p 的概率返回一个正确的解，则该 MC 算法称为 p-正确，算法的优势（advantage）是 p-1/2.
- Def2：若一个 MC 算法对同一实例决不给出两个不同的正确解，则该算法称是相容的（consistent）或一致的。
- 基本思想：为了增加一个一致的、p-正确算法成功的概率，只需多次调用同一算法，然后选择出现次数最多的解。
- Def：(偏真算法)为简单起见，设 MC(x)是解某个判定问题，对任何 x，若当 MC(x)返回 true 时解总是正确的，仅当它返回 false 时才有可能产生错误的解，则称此算法为偏真的(true-biased)。
- Def：(偏 y_0 算法)更一般的情况不再限定是判定问题，一个 MC 是偏 y_0 的(y_0 是某个特定解)，如果存在问题实例的子集 X 使得：

若被解实例x ∉ X，则算法 MC(x)返回的解总是正确的(无论返回 y_0 还是非 y_0)
若∀x ∈ X，正确解是 y_0 ，但 MC 并非对所有这样的实例 x 都返回正确解。

1.1.4.2 两个定理

1. 设 2 个正实数之和 ε+δ<0.5，MC(x)是一个一致的、(0.5+ε)-correct 的蒙特卡洛算法，设$C_\varepsilon = -2/lg(1-4\varepsilon^2)$，x 是某一被解实例，若调用 MC(x)至少$\left\lceil C_\varepsilon lg\left(\frac{1}{\delta}\right)\right\rceil$次，并返回出现频数最高的解，则可得到一个解同样实例的一致的(1−δ)-correct 的新 MC 算法
2. 若将一个偏 y，一致的、(0.5+ε)-correct 的蒙特卡洛算法改进到一致的(1−δ)-correct 的新 MC 算法，则至少需要调用 MC(x) $\left\lceil \frac{\lg\delta}{\lg(0.5-\varepsilon)}\right\rceil$ 次

这两个定理的不同之处在于第一个针对无偏算法，第二个针对有偏算法，第一个返回频数最高的解，第二个根据最后一次运行确定解

3. 将定理 2 换一种表述方式：若将一个偏 y，一致的、(0.5+ε)-correct 的蒙特卡洛算法（即出错概率为$0.5-\varepsilon$）改进到出错的概率小于 δ 的新 MC 算法，则至少需要调用 MC(x) $\left\lceil \frac{\lg\delta}{\lg(0.5-\varepsilon)}\right\rceil$ 次

1.1.4.3 例 1：主元素问题

问题描述：求一个数组中是否有一个元素出现的次数超过了一半

偏性特点：偏真，一次检验 1/2correct，只要有一次运行认定有主元素则一定有主元素

算法：一次出错概率为 1/2，将出错概率降到δ需要$\left\lceil \lg\left(\frac{1}{\delta}\right)\right\rceil$次

```
maj(T) {     //测试随机元素是否为 T 的主元素
        i←uniform(1..n);
        x←T [ i ] ;
        k←0;
        for j←1 to n do
            if T [ j ]=x then
                k←k+1;
        return (k>n/2);
}

majMC (T, ε) {
        k←
        for i ←1    to    k    do
            if    maj (T) then return true;     //成功
        return false;     //可能失败
}
```

时间复杂度：***O(nlg(1/ε))***

1.1.4.4 例 2：素数测定

问题描述：判断一个大于 4 的奇数是否是素数

基本概念：

- 伪素数和伪证据：设 2≤a≤n-2，一个满足$a^{n-1} \equiv 1\ mod\ n$（即 n 可整除 $a^{n-1}-1$）的合数 n 称为以 a 为底的伪素数，a 称为 n 的素性伪证据。
- 强伪素数和强伪证据：设 n 是一个大于 4 的奇整数，s 和 t 是使得 $n-1=2^s t$ 的正整数，其中 t 为奇数，设 B(n)是如下定义的整数集合：a ∈ B(n)当且仅当 2≤a≤n-2 且满足下述 2 个条件之一：

  ① $a^t \equiv 1\ mod\ n$

  ② ∀i ∈ [0, s) 且 i 为整数，使得$a^{2^i t} \equiv -1\ mod\ n$

  当 n 为当 n 为素数时，∀a ∈ [2, n − 2]，使得 n 满足条件 1 或者条件 2，所以a ∈ B(n)，这一点可以由 Fermat 定理很容易得证

  当 n 为合数时，若a ∈ B(n)，则称 n 为一个以 a 为底的**强伪素数**，称 a 为 n 素性的**强伪证据**。

强伪证据的性质：

- 强伪证据数目比伪证据数目少很多
- 若 n 是素数，则强伪证据集合$B(n) = \{2 \le a \le n-2\}$
- 若 n 是合数，则强伪证据的个数$|B(n)| \le (n-a)/4$，这一点说明当 n 为合数时，强伪证据数目<1/4。因此，当随机选 a 时，Btest 返回 false 的概率>3/4，正确的概率>75%’

偏性特点：偏假，一次检验 3/4correct，只要坚持出 n 不是素数，返回 false，则一定正确

算法：

```
Btest(a，n){//n 为奇数，a ∈ [2, n − 2]，返回true ⇔ a ∈ B(n)
            //即返回真说明 n 是强伪素数或素数
      s←0；  t ←n-1；  // t 开始为偶数
      repeat
            s++；t ← t÷2；
      until t mod 2 = 1；  //n-1=2^s t ，  t 为奇数
            x ←a^t mod n；
      if x=1 or x=n-1 then   return true；//满足①or②，
      for i ←1 to s-1 do{ //验证
            x ← x^2 mod n；
            if x=n-1 then return true；  //满足②，
      }
      return false；
}
```

一次测试：

```
MillRab(n) {   //奇 n>4，返回真时表示素数，假表示合数
 a←uniform(2..n-2);
 return Btest(a,n);    //测试 n 是否为强伪素数
}
```

多次测试：$k = \left\lceil \frac{1}{2} \lg\left(\frac{1}{\delta}\right) \right\rceil$

```
RepeatMillRob(n,k){
        for i ←1 to k do
```

```
        if MillRob(n) =false then
            return false;  //一定是合数
        return true;
    }
```

时间复杂度：$O(lg^3 n \lg\frac{1}{\delta})$

1.1.4.5 矩阵乘法验证

问题描述：设 A，B，C 为 n×n 矩阵，判定 AB=C?

MC 方法：设 X 是一个长度为 n 的二值向量(0/1 行向量)，将判断 AB=C 改为判断 XAB=XC?

偏性：偏假，一次检验 1/2correct，

算法：

一次检验：

```
goodproduct( A, B, C, n){
        for i←1 to do
            x [ i ] ←uniform(0..1);
        if (XA)B=XC then
            return true;
        else    return false;
}
```

多次检验，将出错概率降低到δ，则至少要$k = \left\lceil \lg\left(\frac{1}{\delta}\right) \right\rceil$次

```
RepeatGoodProduct( A, B, C, n) {
    for i←1 to ⌈lg(1/δ)⌉ do   //重复⌈lg(1/δ)⌉次
        if   GoodProduct(A, B, C, n) = false then
            return false;//偏假的，有一次假即可返回
    return true;
}
```

时间复杂度：$O(3n^2)$

2. 消息传递系统中的基本算法

2.1 基本概念

2.1.1 分布式模型

① **异步共享存储模型**：用于紧耦合机器，通常情况下各处理机的时钟信号不是来源于同一信号源

② **异步 msg 传递模型**：用于松散耦合机器及广域网

③ **同步 msg 传递模型**：这是一个理想的 msg 传递系统

2.1.2 错误的种类

① **初始死进程**：指在局部算法中没有执行过一步

② **崩溃错误(损毁模型)**：指处理机没有任何警告而在某点上停止操作

③ **拜占庭错误**：一个出错可引起任意的动作，即执行了与局部算法不一致的任意步。拜占庭错误的进程发送的消息可能包含任意内容

2.1.3 消息传递系统概念

状态：p_i 的每个状态由 2r 个 msg 集构成，$outbuf_i[l](1\le l\le r)$和 $inbuf_i[l](1\le l\le r)$，其中 r 是信道个数

初始状态：Q_i 包含一个特殊的初始状态子集：每个 $inbuf_i[l]$必须为空，但 $outbuf_i[l]$ 未必为空

转换函数：输入：p_i 可访问的状态，输出：对每个信道 l ，至多产生一个 msg 输出，转换函数使输入缓冲区$(1\le l\le r)$清空

配置：配置是分布式系统在某点上整个算法的全局状态，向量=$(q_0, q_1,\dots q_{n-1})$, q_i 是 p_i 的一个状态

事件：系统里所发生的事情均被模型化为事件，对于 msg 传递系统，有两种，comp(i)和 del(i,j,m)

执行：系统在时间上的行为被模型化为一个执行，它是一个由配置和事件交错的序列。该序列须满足各种条件，主要分为两类

- **安全性条件**：表示某个性质在每次执行中每个可到达的配置里都必须成立
- **活跃性条件**：表示某个性质在每次执行中的某些可达配置里必须成立。若一个执行也满足所有要求的活跃性条件，则称为**容许执行**

调度：一个调度总是和执行联系在一起的，它是执行中的事件序列：Φ_1, Φ_2, …。并非每个事件序列都是调度。例如，del(1,2,m)不是调度，因为此事件之前，p_1 没有步骤发送(send)m

容许的调度：若它是一个容许执行的调度

容许的执行：指无限的执行

Msg 复杂度：算法在所有**容许**的执行上发送 msg 总数的最大值(同步和异步系统)

时间复杂度：

① 同步系统：最大轮数，即算法的任何容许执行直到终止的最大轮数。

② 异步系统：假定任何执行里的 msg 延迟至多是 1 个单位的时间，然后计算直到终止的运行时间

消息复杂度：消息总数/消息中总的位数长度。请注意和 Msg 复杂度的不同。

2.2 生成树上的广播和汇集

2.2.1 广播

基本思想：根节点发送 Msg 给孩子。收到 Msg 的节点转发 Msg 给孩子。

Msg 复杂度：O(n-1)，因为生成树每条边有一个 Msg

时间复杂度：O(D)，D 为生成树直径

2.2.2 汇集/敛播

基本思想：汇集是从所有节点收集信息至根

Msg 复杂度：O(n-1)，因为生成树每条边有一个 Msg

时间复杂度：O(D)，D 为生成树直径

2.3 指定根构造生成树 DFS 或 BFS

2.3.1 洪泛

基本思想：设 p_r 是特殊处理器。从 p_r 开始，发送 M 到其所有邻居。当 p_i 第 1 次收到消息 M（不妨设此 msg 来自于邻居 p_j）时，p_i 发送 M 到除 p_j 外的所有邻居。

Msg 复杂度：O(2e-E(P_r))

时间复杂度：O(D)

2.3.2 构造生成树

基本思想：回应<parent>和<reject>，具体算法略

证明注意要点：在证明构造 DFS 或者 BFS 时，要按以下步骤证明：

① **先证连通性**。可用反证或归纳法证明。

② **再证无环性**。可用反证法证明假设存在环 $p_{i1},...p_{ik}p_{i1}$

③ **最后证明生成的是 DFS 或者 BFS**。可用归纳法证明

构造 BFS 树：Pj 收到 Pi 的 Msg 立马转发给所有除 Pi 以外的邻居

Msg 复杂度：O(m)，m 为边数

时间复杂度：O(D)，D 为直径

构造 DFS 树：只有收到一条 Msg 或<parent>或<reject>才转发 Msg

Msg 复杂度：O(m)

时间复杂度：O(m)

2.4 不指定根时构造生成树

基本思想：每个结点均可自发唤醒，试图构造一棵以自己为根的 DFS 生成树。若两棵 DFS 树试图链接同一节点(未必同时)时，该节点将加入根的 id 较大的 DFS 树。

已知条件：个具有 m 条边和 n 个节点的网络，自发启动的节点共有 p 个，其中 ID 值最大者的启动时间为 t

Msg 复杂度：O(pn^2)。最坏情况下，每个处理器均试图以自己为根构造一棵 DFS 树。因此，Alg2.4 的 msg 复杂性至多是 Alg2.3 的 n 倍：O(m*n)

时间复杂度：时间复杂度为 O(t+m)

3. 环上选举算法

3.1 无聊

3.2 Leader 选举问题

3.2.1 问题

在一组处理器中选出一个特殊结点作为 leader

3.2.2 用途

①简化处理器之间的协作；

有助于达到容错和节省资源。

例如，有了一个 leader，就易于实现广播算法

① 代表了一类破对称问题。

例如，当死锁是由于处理器相互环形等待形成时，可使用选举算法，找到一个 leader 并使之从环上删去，即可打破死锁。

3.2.3 问题描述

问题从具有同一状态的进程配置开始，最终达到一种配置状态。每个处理器最终确定自己是否是一个 leader，但只有一个处理器确定自己是 leader，而其他处理器确定自己是 non-leader。

3.2.4 选举算法定义

（1）每个处理器具有相同的局部算法；

（2）算法是分布式的，处理器的任意非空子集都能开始一次计算；

（3）每次计算中，算法达到终止配置。在每一可达的终止配置中，只有一个处理器处于领导人状态，其余均处于失败状态

3.3 匿名环

3.3.1 几个定义

匿名算法：若环中处理器没有唯一的标识符，则环选举算法是匿名的

一致性的算法：若算法不知道处理器数目，则算法称之为 uniform，否则若知道处理器数目 n，则称之为非一致性算法。

上面两个的形式化描述：在一个匿名、一致性的算法中，所有处理器只有一个状态机；在一个匿名、非一致性的算法中，对每个 n 值（处理器数目）都有单个状态机，但对不同规模有不同状态机，也就是说 n 可以在代码中显式表达。

3.3.2 几个定理

引理 3.1： 在环 R 上算法 A 的容许执行里，对于每一轮 k，所有处理器的状态在第 k 轮结束时是相同的。（用归纳法证明）

定理 1： 同步环系统中不存在匿名的、一致性的领导者选举算法（用引理 3.1 证明，note:每个处理器同时宣布自己是 Leader）

定理 2：异步环系统中不存在匿名的领导者选举算法

证明：每个处理器的初始状态相同，状态机相同，接收的消息序列也相同（只有接收消息的时间可能不同），故最终处理器的状态一致。由于处理一条消息的至多需要 1 时间单位，若某时刻某个处理器宣布自己是 Leader（接收到 m 条消息），则在有限时间内（m 时间单位）其他处理器也会宣布自己是 Leader。

所以，每个处理器会陆续宣布自己是 Leader。矛盾，证毕。

3.3 异步环

3.3.0 均匀和非均匀之分

由于没有匿名的异步环选举算法，这里默认所有算法非匿名，虽然说均匀对应一致，非均匀对应非一致，但实际上它们还是有差别的

①**均匀算法**：每个标识符 id，均有一个唯一的状态机，但与环大小 n 无关。而在匿名算法中，均匀则指所有处理器只有同一个状态。（不管环的规模如何，只要处理器分配了对应其标识符的唯一状态机，算法就是正确的。）

②**非均匀算法**：每个 n 和每个 id 均对应一个状态机，而在匿名非均匀算法中，每个 n 值对应一个状态机。（对每一个 n 和给定规模 n 的任意一个环，当算法中每个处理器具有对应其标识符的环规模的状态机时，算法是正确的。）

3.3.1 LCR：一个简单的 $O(n^2)$算法

基本思想：选择最大的 id

①每个处理器 P_i 发送一个 msg(自己的标识符)到左邻居，然后等其右邻居的 msg

②当它接收一个 msg 时，检验收到的 id_j，若 $id_j>id_i$，则 P_i 转发 id_j 给左邻，否则没收 id_j(不转发)。

③若某处理器收到一个含有自己标识符的 msg，则它宣布自己是 leader，并发送一个终止 msg 给左邻，然后终止。

④当一处理器收到一个终止 msg 时，向左邻转发此消息，然后作为 non-leader 终止。

因为算法不依赖于 n，故它是均匀的。

正确性：只要分析出最终只有一个 Leader 被选中就行了

Msg 复杂性：若处理器按照 n-1,n-2,…,0,顺时针排列，顺时针转发 msg，则标号为 i

的处理器的 msg 会被转发 i+1 次，最终的 Leader 又会向每个处理器转发通知（共 n 次转发），则

Msg 复杂性=n + $\sum_{i=0}^{n-1}(i+1) = \theta(n^2)$

时间复杂性：显然是 O(n)量级

3.3.2 K 邻居法：一个O(nlgn)算法

基本思想：

每个局部 Leader 慢慢扩大自己的邻居范围，每经过一个阶段增长一倍的邻居。

算法按阶段执行，在第 *l* 阶段一个处理器试图成为其 2^l-邻接的临时 leader。只有那些在 *l*-th 阶段成为临时领袖的处理器才能继续进行到(*l*+1)th 阶段。因此，*l* 越大，剩下的处理器越少。直至最后一个阶段，整个环上只有一个处理器被选为 leader。

在第 *i* 阶段的一开始，局部 Leader 向它的两边发送一个<prob, id, *i*, hop>的 msg，id 是局部 Leader 的 id，i 表示第 i 阶段，hop 表示经过的邻居数，也就是跳数。邻居分别沿路径转发 msg，每到达一个邻居，hop++。当到达 hop=2^i时，就不再转发，邻居回答 reply，转发向局部 Leader，这样一个局部 Leader 在第 i 阶段最多会产生 $4*2^i$ 个 msg，之所以说最多，是因为有些局部 Leader 会被别的局部 Leader 吞并，这样它的 msg 就不会再被邻居转发或者 reply

正确性：因为具有最大 id 的处理器的 probe 消息是不会被任何结点没收的，所以该处理器将作为 leader 终止算法；另一方面，没有其他 probe 消息能够周游整个环而不被吞没。因此，最大 id 的处理器是算法选中的唯一的 leader。

Msg 复杂度（最坏情况下）：

在 phase *i* 里：

- 一个处理器启动的 msg 数目至多为：$4*2^i$
- 有多少个处理器是启动者呢？

i =0，有 n 个启动着（最多），则会产生 4n 个 msg

i ≥1，在 *i*-1 阶段结束时成为临时 leader 的节点均是启动者。对每个 i ≥1，

在 phase i 结束时，临时 leader 数至多为 $n/(2^i+1)$.

在结束阶段，产生的总 Leader 会转发通知，产生 n 个 msg

则 msg 复杂度为：

$$4n + \sum_{i=1}^{\lg(n-1)} 4 \cdot 2^i \frac{n}{2^{i-1}+1} + n \le 8nlg(n-1) + 5n = O(nlgn)$$

时间复杂度：只盯着最终 Leader 看，因为它最终成为 Leader 了算法才能结束。它从局部 Leader 走向最终 Leader，需要经过$\lg(n-1)$个阶段，在第 i 个阶段，它至少需要经过 $2*2^i$ 个时刻才能确认自己还保持着局部 Leader 的身份，最终再发送自己是终极 Leader 的通知，所以时间复杂度是

$$\sum_{i=0}^{\lg(n-1)} 2 \cdot 2^i + n = 3n - 4 = O(n)$$

3.3.3 下界Ω(nlogn)

基本思想：对于大小为 n 的环采用构造法，将两个大小为 n/2 的不同环粘贴在一起形成一个大小为 n 的环，将两个较小环上的耗费执行组合在一起，并迫使 θ(n)个附加 msg 被接收，这样总的耗费就是 M(n) = 2M(n/2) +θ(n)，这样的递归方程的解是 M(n) =θ(nlgn)；

开调度定义：设 σ 是一个特定环上算法 A 的一个调度，若该环中存在一条边 e 使得在 σ 中，边 e 的任意方向上均无 msg 传递，则 σ 称为是开调度，e 是 σ 的一条开

边。

具体方法见 PPT，过于繁琐，就是利用开调度构造出 θ(n)=(n/2-1)/2。

总体过程：

1) 在 R1 和 R2 上构造 2 个独立的调度，每个接收 2M(n/2)各 msg: σ_1 σ_2

2) 强迫环进入一个静止配置： σ_1 σ_2 σ_3 （主要由调度片断 σ_3 ）

3) 强迫(n/2-1)/2 个附加 msg 被接收，并保持 ep 或 eq 是开的: σ_1 σ_2 σ_3 σ_4。

因此我们已构造了一个开调度，其中至少有 2M(n/2)+ (n/2-1)/2 个 msg 被接收。

Msg 复杂度：Ω(nlogn)

时间复杂度：此处不方便讨论

3.4 同步环

在这一节中研究了同步环中的上界 O(n)和下界 O(nlgn)

3.4.1 上界 O(n)

3.4.1.1 非均匀算法

基本思想：选择环中最小 id（各 id 互不相同）的结点作为 leader，按 Phase 运行，每个阶段由 n 个轮组成。在 Phase i (i ≥ 0)，若存在一个 id 为 i 的结点，则该结点为 leader，并终止算法，因此，最小 id 的结点被选为 leader。

算法思想其实很简单：

① 就是从 i=0 开始，所有的处理器依次转发 i

② 若有一个处理器的 id = i，则它站出来说自己是 Leader，转发通知，通知转发完成后算法终止。

③ 若②的情况没出现，即 i 转发一圈后没有处理器说自己是 Leader，则 i++，转到①继续下一圈。

那么怎么知道一圈已经完了呢？这就需要知道 i 已经被转发了 n 轮，所以就需要知道处理器的总个数，这也就是这个算法是非均匀算法的原因。

Msg 复杂度：n·(i+1)，i 为最终 Leader 的 id

时间复杂度：n·(i+1)

缺点：必须知道环大小 n 和同步开始。

① 为什么 id 为 i 的结点要在 phase i 发 msg？

答：因为各节点不知道彼此的 id，所以只有在第 i 阶段，id=i 的节点才知道自己是 Leader，再发送 msg

② 为什么每个 phase 要 n 轮？

答：因为少于 n 轮的话可能最后的几个处理器存在最小 id，那么它就在 phase i 转发自己是 leader 的 msg

3.4.1.2 均匀算法

特点：①无须知道环大小，②弱同步模型

一个处理器可以在任意轮里自发地唤醒自己，也可以是收到另一个处理器的 msg 后被唤醒

基本思想：

① 源于不同节点的 msg 以不同的速度转发

源于 id 为 i 的节点的 msg，在每一个接收该 msg 的节点沿顺时针转发到下一个处理器之前，被延迟 2^i-1 轮

② 为克服非同时启动，须加一个基本的唤醒阶段，其中每个自发唤

醒的结点绕环发送一个唤醒 msg，该 msg 转发时无延迟

③ 若一个结点在算法启动前收到一个唤醒 msg，则该结点不参与算法，只是扮演一个 relay(转发)角色：即转发或没收 msg

要点：在基本阶段之后，选举 leader 是在参与结点集中进行的，即只有自发唤醒的结点才有可能当选为 leader

具体实现：

① 唤醒：由一个结点发出的唤醒 msg 包含该结点的 id，该 msg 以每轮一边的正常速率周游，那些接收到唤醒 msg 之前未启动的结点均被删除(不参与选举)

② 延迟：当来自一个 id 为 i 的节点的 msg 到达一个醒着的节点时，该 msg 以 2^i 速率周游，即每个收到该 msg 的节点将其延迟 2^i-1 轮后再转发。

Note： 一个 msg 到达一个醒着的节点之后，它要到达的所有节点均是醒着的。一个 msg 在被一个醒着的节点接收之前是处在 1st 阶段 (唤醒 msg，非延迟) ，在到达一个醒着的节点之后，它就处于 2nd 阶段，并以 2^i 速率转发(非唤醒 msg，延迟)

③ 没收规则

a) 一个参与的节点收到一个 msg 时，若该 msg 里的 id 大于当前已看到的最小(包括自己)的 id，则没收该 msg；

b) 一个转发的节点收到一个 msg 时，若该 msg 里的 id 大于当前已看到的最小(不包括自己)的 id，则没收该 msg。

Msg 复杂性：

第一类：第一阶段的 msg(唤醒 msg)，每条边上一个，则至多为 n

第二类：最终 leader 的 msg 进入自己的第二阶段之前发送的第二阶段 msg(其它结点发出的)。因为要延迟 2^i-1 轮，则最多能转发$\frac{n}{2^i}$次，$\sum_{i=1}^{n-1}\frac{n}{2^i} \le n$。

第三类：最终 leader 的 msg 进入自己的第二阶段之后发送的第二阶段 msg(包括 leader 发出的)。Leader 的 id 要经过 $n\cdot 2^{id_i}$ 轮才能返回，所以其它的 id 能够存活的并转发的次数最多是$\sum_{i=1}^{n-1}\frac{n\cdot 2^{id_i}}{2^{id_j}} \le \sum_{k=1}^{n-1}\frac{n}{2^k} \le 2n$

总共加起来 4n

时间复杂度：当 leader 接收到自己的 id 时，计算终止。这发生在第一个启动算法的节点之后的 $O(n\cdot 2^i)$轮，其中 i 是 leader 的标识符

为何非唤醒 msg 要延迟 2^i -1 轮？

答：降低 Msg 复杂度。Id 最小的节点被选举为 Leader，Leader 节点消息的转发速度最快，这样就会使得某些非 Leader 的 msg 在延迟的时候就因为 id 没有 Leader 的 id 小而被没收了，就减小了 Msg 复杂度。

如何修改算法 3.2 来改善时间复杂性？

答：方案 1：添加 Relay 变量，保证消息在转发节点不延迟，时间复杂度由 O(n*2^i)降为 O(N*2^i+n-N)，N 为自发唤醒的节点数。

方案 2：原算法延迟函数为 f(id)=2^id，时间复杂度为 O（n*2^i）。通过重新定义延迟函数来降低时间复杂度，如 f(id)=c*id 等。但是提高了时间复杂度。

方案 2 中 Msg 复杂度与时间复杂度的关系是：

Msg 复杂度=n + $\sum_{i=1}^{n-1}\frac{n}{f(i)}+\sum_{i=0}^{n-1}\frac{n}{f(i)}$

时间复杂度=O(n*f(id))

3.4.2 有限制算法的下界Ω(nlogn)

可以看出，上面的上界的时间复杂度很烂，存在以下两个缺陷：

① 它们用一种非同寻常的方式使用 id，即 id 决定 msg 延迟多长；

② 在每个容许的执行中，执行轮数依赖于 id，而 id 相对于 n 而言可能是巨大的。(更主要的)

这一节分两步给出了一个独立于 id（即复杂度与 id 无关），且 msg 复杂度是Ω(nlogn)的算法：

第一步：分析序等价环R_n^{rev}的 Msg 复杂度是Ω(nlogn)

第二步：用 Ramsey 定理证明任意一个环都可以找到一个与之序等价的R_n^{rev}环，从而证明了任意一个环的 Msg 复杂度下界是Ω(nlogn)

3.4.2.1 基于比较的算法

基本概念：

序等价：两个环 $x_0, x_1,\ldots, x_{n-1}$ 和 $y_0, y_1,\ldots,y_{n-1}$ 是(次)序等价的，若对每个 i 和 j，$x_i<x_j$，当且仅当 $y_i<y_j$。

行为相似：考虑两个执行 α_1，α_2 和两个结点 p_i，p_j，我们说 p_i 在 α_1 的第 k 轮里的行为相似于 p_j 在 α_2 的第 k 轮里的行为，若下述条件成立：

① p_i 在 α_1 的第 k 轮里发送一个 msg 到其左(右)邻居当且仅当 p_j 在 α_2 的第 k 轮里发送一个 msg 到其左(右)邻居；

② p_i 在 α_1 的第 k 轮里作为一个 leader 终止当且仅当 p_j 在 α_2 的第 k 轮里作为一个 leader 终止。

主动轮：若某一轮在任何次序等价的环上均无 msg 发送，则该轮是无用的，而有用的轮被称为是主动的(active)。则对于序等价的两个环 Pi 和 Pj，经过相同的 K 个主动轮之后，Pi 和 Pj 的状态仍然相同。

几个定理：

定理 1：R_n^{rev}划分为长度为 j(j 是 2 的方幂)的连续片断，则所有这些片断是序等价的。

证明见作业

定理 2： 对所有 k<n/8 以及每个 S_n 的 k-邻居集 N，在 S_n 中与 N 序等价的 k-邻居集的个数(包括 N 本身)大于$\frac{n}{2(2k+1)}$

证明：N 由 2k+1 个 id 构成，设 j 是大于 2k+1 的 2 的最小方幂。将 S_n 划分为 n/j 个连续片断，使某一片段包含 N。

由 S_n 的构造可知，上述划分所得的所有片段均是序等价的。因此，至少有 n/j 个邻居集和 N 是序等价的。

设 $j=2^i$，∵$2^{i-1}<2k+1<2^i$，∴$j<2(2k+1)$

故与 N 序等价的邻居集数目=$\frac{n}{j}>\frac{n}{2(2k+1)}$

定理 3：在 exec(S_n)里，主动轮的数目至少为 n/8（反证法，T=k）

定理 4：对于∀k∈[1，n/8]，在 exec(S_n)的第 k 个主动轮里，至少有$\frac{n}{2(2k+1)}$个 msg 被发送

根据定理 4：在 exec(S_n) 里发送 msg 的总数至少为

$$\sum_{k=1}^{n/8}\frac{n}{2(2k+1)} \geq \frac{n}{6}\sum_{k=1}^{n/8}\frac{1}{k} > \frac{n}{6}\ln\frac{n}{8}$$

即Ω(nlogn)

注意：为了使上述定理成立，要求标识符是取自集合{0,1,…,n^2+2n-1}。//该集合的势为 n^2+2n。

原因是S_n中最小标识符为n，最大标识符为n^2+n-1 =(n+1)*rev(n-1)+n。

定理 5：设 A 是一个运行时间为 r(n)的时间受限的同步算法，则对于每个 n，存在一个具有 n^2+2n 个 id 的集合 C_n，使得 A 是 C_n 上的一个基于 r(n)-比较的算法，这里 n 是环大小。（Ramsey 定理证明）

定理 6：对每个同步的时间有界的 leader 选举算法 A，以及每个 n>8，n 为 2 的方幂，存在一个大小为 n 的环 R，使得 A 在 R 上的容许执行里发送 Ω(nlgn)个 msgs。

4. 分布式系统中的计算模型

4.1 基本知识

TCP 与 UDP 的区别：

① 于连接与无连接
② 对系统资源的要求（TCP 较多，UDP 少）
③ UDP 程序结构较简单
④ 流模式与数据报式
⑤ TCP 保证数据正确性，UDP 可能丢包
⑥ TCP 保证数据顺序，UDP 不保证

4.2 因果关系

4.2.1 分布式系统为何缺乏全局的系统状态？

答：1. **非即时通信**。系统的全局状态依赖于观察点，因为：传播延时、网络资源的竞争、丢失 msg 重发

2. **相对性影响**。因为大多数计算机的实际时钟均存在漂移，故相对速度不同，时钟同步仍然是一个问题。所以使用时间来同步不是一个可靠机制。

3. **中断**。即使可忽略其他影响，也不可能指望不同的机器会同时做出某些反应。

所以，不可能在同一时刻观察一个分布式系统的全局状态，必须找到某种可以依赖的性质，如时间回溯、因果相关。

4.2.2 Happens-before 关系（$<_H$）

该关系是节点次序和消息传递次序的传递闭包：

- 规则 1：若 $e_1<_p e_2$，则 $e_1<_H e_2$
- 规则 2：若 $e_1<_m e_2$，则 $e_1<_H e_2$
- 规则 3：若 $e_1<_H e_2$，且 $e_2<_H e_3$，则 $e_1<_H e_3$
- 并发事件：若两事件不能由$<_H$定序

4.2.3 Lamport 时间戳

基本思想：

- 每个事件 e 有一个附加的时戳：e.TS
- 每个节点有一个局部时戳：my_TS
- 每个 msg 有一个附加时间戳：m.TS
- 节点执行一个事件时，将自己的时戳赋给该事件；
- 节点发送 msg 时，将自己的时戳赋给所有发送的 msg。

算法实现：

```
Initially：my_TS=0;
On event e：
    if  ( e 是接收消息 m ) then
            my_TS = max ( m.TS, my_TS );
            //取 msg 时戳和节点时戳的较大者作为新时戳
    my_TS++;
    e.TS＝my_TS; //给事件 e 打时戳
    if ( e 是发送消息 m ) then
            m.TS＝my_TS; //给消息 m 打时戳
```

4.2.2 向量时戳

向量时戳意义：

在因果关系上，$e1.VT \leqslant_V e2.VT$ 表示 e2 发生在 e1 及 e1 前所有的事件之后。更精确的说，向量时钟的次序为：

$e1.VT \leqslant_V e2.VT$ iff $e1.VT[i] \leqslant e2.VT[i], i=1,2,\cdots,M$

$e1.VT <_V e2.VT$ iff $e1.VT \leqslant_{VT} e2.VT$ 且 $e1.VT \neq e2.VT$

算法实现：

```
Initially：my_VT=[0,…,0];
    On event e：
        if  ( e 是消息 m 的接收者 ) then
            for i=1 to M do //向量时戳的每个分量只增不减
                my_VT[ i ] = max( m.VT[i], my_VT[i] );
        my_VT[ self ]++; //设变量 self 是本节点的名字
        e.VT＝my_VT; //给事件 e 打时戳
        if ( e 是消息 m 的发送者 ) then
            m.VT＝my_VT; //给消息 m 打时戳
```

算法性质：

1）若 $e <_H e'$，则 $e.VT <_{VT} e'.VT$

∵算法确保对于每个事件满足：

若 $e <_P e'$ 或 $e <_m e'$ ，则 $e.VT <_{VT} e'.VT$

2）若 $e \nless_H e'$，则 $e.VT \nless_{VT} e'.VT$

pf：若 e 和 e' 因果相关，则有 $e' <_H e$，即 $e'.VT <_{VT} e.VT$

若 e 和 e‘ 是并发的，则在 H-DAG 上，从 e 到 e’和从 e’到 e 均无有向路径，即得：

$e.VT \nless_{VT} e'.VT$ 且 $e'.VT \nless_{VT} e.VT$

当且仅当 e.VT 和 e’.VT 是不可比时，称向量时戳是捕获并发的！

4.2.3 因果通信

基本思想：

- 抑制从 P 发送的消息 m，直至可断定没有来自于其它处理器上的消息 m′，使 $m' <_V m$.
- 在每个节点 P 上：

earliest[1..M]：存储不同节点当前能够传递的消息时戳的下界

earliest[k]表示在 P 上，对节点 k 能够传递的 msg 的时戳的下界

blocked[1..M]：阻塞队列数组，每个分量是一个队列

算法实现：

定义时戳 1_k： 若使用 Lamport 时戳，则 1_k＝1；

若用向量时戳，则 1_k＝(0,…,1,0,…,0)，k^{th} 位为 1

初始化

1: earliest[k] =1_k， k=1,…,M

2: blocked[k] ={ } ， k=1,…,M//每个阻塞队列置空

On the receipt of msg m from node p:

delivery_list={};

if （blocked[p]为空）then

earliest[p]=m.timestamp;

将 m 加到 blocked[p]队尾；//处理收到的消息

while（∃k 使 blocked[k]非空 and 对每个 i=1,…,M(除 k 和 self 外),

not_earliest(earliest[i], earliest[k], i)）{//处理阻塞队列

//对非空队列 k，若其他节点 i 上无比节点 k 更早的 msg 要达到本

//地，则队列 k 的队首可解除阻塞

将 blocked[k]队头元素 m′出队，且加入到 delivery_list;

if （blocked[k]非空）then

将 earliest[k]置为 m′.timestamp;

else increment earliest[k] by 1_k }//end while

deliver the msgs in delivery_list; //按因果序

```
not_earliest( proc_i_vts, msg_vts, i ) {//前者大于后者时为真
    if (proc_i_vts[i] > msg_vts[i] )
        return true;
    else return false;
}
```

问题：

上述算法可能会发生死锁：若一节点长时间不发送你要的 msg，会发生死锁。因此，上述因果通信算法通常被用于**组播**的一部分。