算法设计与分析

姓名：方国庆

学号：SA13011096

Ex.1 若将 y ← uniform (0, 1) 改为 y ← x, 则上述的算法估计的值是什么？

$$\text{因为}\frac{k}{n}=\frac{1}{\sqrt{2}}，\text{所以}\frac{4k}{n}=\frac{4}{\sqrt{2}}=2\sqrt{2}$$

Ex2. 在机器上用 $\mathbf{4}\int_{\mathbf{0}}^{\mathbf{1}}\sqrt{\mathbf{1}-\boldsymbol{x}^{\mathbf{2}}}\,\mathrm{dx}$ 估计 π 值，给出不同的 n 值及精度。

Source Code:

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

int main()
{
    srand((unsigned)time(NULL));
    int count, I; long n;
    for(n = 100; n <= 10e8; n *= 10)
    {
        for(i = 1, count = 0; i <= n; i++)
        {
            float x = (float)rand()/RAND_MAX;
            float y = (float)rand()/RAND_MAX;
            if((x*x+y*y) < 1)   count++;
        }
        printf("n = %d\nPi
= %f\n",n,4*count/(float)n);
    }
    return 0;
}
```

```
fanggq@DHMP:~/Algorithm-Design-and-Analyse$ ls
a.out  ex2.c  ex3.c  MyQueen.cpp  prime.cpp  QueensLV  SetCount.cpp
fanggq@DHMP:~/Algorithm-Design-and-Analyse$ gcc ex2.c
fanggq@DHMP:~/Algorithm-Design-and-Analyse$ ./a.out
n = 100
Pi = 3.160000
n = 1000
Pi = 3.272000
n = 10000
Pi = 3.126800
n = 100000
Pi = 3.147120
n = 1000000
Pi = 3.142744
n = 10000000
Pi = 3.142230
n = 100000000
Pi = 3.141282
```

由上图可知，不同 n 值与 Pi 精度的关系如下表：

n	10e2	10e3	10e4	10e5	10e6	10e7	10e8
Pi 精度	1	0.1	0.1	0.01	0.001	0.001	0.0001

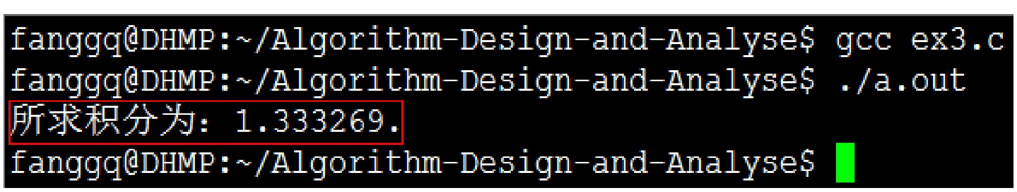
Ex3. 设 a, b, c 和 d 是实数，且 a ≤ b, c ≤ d, f:[a, b] → [c, d]是一个连续函数，写一概率算法计算积分：$\int_a^b f(x)dx$

注意，函数的参数是 a, b, c, d, n 和 f, 其中 f 用函数指针实现，请选一连续函数做实验，并给出实验结果。

Source Code:

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define N 10000000
double function(double x){return (x-2)*(x-2) - 1;}
double GenRandom(double x, double y){return ((double)rand()/RAND_MAX)*(y-x)+x;}

double caller(double(*f)(double), double a, double b, double c, double d)
{
    int i, count;
    for(i = count = 0; i < N; ++i)
    {
        double x = GenRandom(a, b);
        double y = GenRandom(c, d);
        double y0 = function(x);
        if((0 <= y) && (y <= y0))  count++;
        if((y0 <= y) && (y <= 0))  count--;
    }
    c = (c>0)?0:c;
    d = (d<0)?0:d;
    return count*(d-c)*(b-a)/N;
}

int main()
{
    srand((unsigned)time(NULL));
    double a,b,c,d;
    a = 0;  b = 4;  c = -1;  d = 3;
    double S = caller(function,a,b,c,d);
    printf("所求积分为：%lf.\n",S);
    return 0;
}
```

设计算法时，考虑到由 c,d 取值与 0 的关系导致的四种可能状况。最后通过 c = (c>0)?0:c; d = (d<0)?0:d; 进行统一化处理，使得最后的代码准确而且简洁。本例中，使用函数 $f(x) = (x-2)^2 - 1$ 运行结果如下：



所得结果在 N=10e7 时为 1.333269，与实际值 4/3 误差不足为 0.001%。

*Ex4. 设 ε,δ 是(0,1)之间的常数，证明：

若 I 是$\int_0^1 f(x)dx$的正确值，h 是由 HitorMiss 算法返回的值，则当 $n \geq I(1-I)/\varepsilon^2\delta$ 时有：

$$Prob[|h-I| < \varepsilon] \geq 1 - \delta$$

上述的意义告诉我们：$Prob[|h-I| \geq \varepsilon] \leq \delta$, 即：当 $n \geq I(1-I)/\varepsilon^2\delta$ 时，算法的计算结果的绝对误差超过 ε 的概率不超过 δ，因此我们根据给定 ε 和 δ 可以确定算法迭代的次数。解此问题时可用切比雪夫不等式，将 I 看作是数学期望。

证明：$I = \int_0^1 f(x)dx$ 为点落在 1/4 圆内的概率

记随机变量 X 为 n 个点中落在 1/4 圆内的点数量，则 $X \sim B(n, I)$, 所以有:

$$EX = n*I \qquad DX = n*I(1-I)$$

根据切比雪夫不等式：$P\{|X-E(X)|<\varepsilon\}>=1-\frac{D(X)}{\varepsilon^2}$;

则 $P\{|n*h-n*I|<x\}>=1-\frac{D(X)}{x^2}$;

则 $P\{|h-I|<\frac{x}{n}\}>=1-\frac{D(X)}{x^2}$;

令 $\varepsilon=\frac{x}{n}$，则 $x=\varepsilon*n$;

则 $P\{|h-I|<\frac{x}{n}\}= P\{|h-I|<\varepsilon\}>=1-\frac{D(X)}{\varepsilon^2*n^2}=1-\frac{n*I(1-I)}{\varepsilon^2*n^2}=1-\frac{I(1-I)}{\varepsilon^2*n}$;

又因为 $n \geqslant I(1-I)/\varepsilon^2\delta$

所以：$P\{|h-I|<\varepsilon\}>=1-\frac{I(1-I)}{\varepsilon^2*n}>=1-\frac{I(1-I)}{\varepsilon^2}*\frac{\varepsilon^2*\delta}{I(1-I)}=1-\delta$（得证）

EX. (ch2.3)用上述算法，估计整数子集 1~n 的大小，并分析 n 对估计值的影响。

Source Code:

```
/* Randomly get an element from set<int> S */
int uniform(set<int> X)
{
    int m = rand()%X.size();
    set<int>::iterator iter = X.begin();
    for(int i = 0; i < m; iter++,i++);
    return *iter;
}

/* Estimate the size of set<int> X */
int SetCount(set<int> X)
{
    set<int> S;
    int a ;
    while(S.find(a = uniform(X)) == S.end())
        S.insert(a);
    return (int)(2*S.size()*S.size()/Pi);
}

int main(int argc, char* argv[])
{
    set<int> X;
    srand((unsigned)time(NULL));

    /* Initialize the set<int> S */
    for(int i = 0; i < SizeX; ++i)
        X.insert(i);
    int N;
    for(N = 10; N <= 1000; N *= 10)
    {
        /* Call SetCount(X) N times and use its
              average */
        int Sum = 0;
        for(int i = 0; i < N; ++i)
            Sum += SetCount(X);
        printf("N = %d Xsize
= %d\n",N,(Sum/N));
    }
    return 0;
}
```

运行结果如下：

```
fanggq@DHMP:~/Algorithm-Design-and-Analyse$ !g++
g++ SetCount.cpp
fanggq@DHMP:~/Algorithm-Design-and-Analyse$ ./a.out
N = 10 Xsize = 1507
N = 100 Xsize = 1182
N = 1000 Xsize = 1157
```

由可见当 n 取值增大时，估计的集合大小与真实值越来越接近。

Ex. (Ch3.2 随机的预处理)分析 dlogRH 的工作原理,指出该算法相应的 u 和 v

该算法中的 u 是引入 r 将任意输入实例 p 随机化成实例 c 的部分, 即程序中的 b = ModularExponent(g,r,p) 和 c = ba mod p;

该算法重的 v 是由随机化后的输入实例 c 求出 y 后,利用 y 计算得到 x 的部分,即 (y-r)mod(p-1)

Ex. (Ch3.3 搜索有序表)写一 Sherwood 算法 C，与算法 A, B, D 比较，给出实验结果。

Source Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<time.h>
#include<sys/time.h>
#include<math.h>
#define REAPT_TIMES 128
#define n 120000

int val[n+1], ptr[n+1];
int count, head = 4;
int x = n/2;

int Search(int x, int i)
{
    count = 1;
    while(x > val[i])
    {
        i = ptr[i];
        count++;
    }
    return i;
}
int A(int x)
{
    return Search(x,head);
}

int D(int x)
{
    int i = rand()%n+1;
    int y = val[i];
    if(x < y) return Search(x, head);
    if(x > y) return Search(x, ptr[i]);
    return i;
}

int C(int x)
{
    int y, i = head;
    int max = val[i];
    for(int j = 1; j <= n/6; j++)
    {
        y = val[j];
        if(max < y && y <= x)
        {
            i = j; max = y;
        }
    }
    return Search(x,i);

}
int B(int x)
{
    int y, i = head;
    int max = val[i];
    for(int j = 1; j < sqrt(n); j++)
    {
        y = val[j];
        if(max < y && y <= x)
        {
            i = j; max = y;
        }
    }
    return Search(x, i);
}

void Gen_Data()
{
    int index, pre;
    head = (index = rand()%n + 1);
    val[pre = head] = 1;
    for(int i = 2; i <= n;)
    {
        index = rand()%n + 1;
        if(0==val[index])
        {           val[index] = i;
            ptr[pre] = index;
            pre = index;
            i++;
        }
    }
    ptr[index] = 0;
}
int main()
{
    srand((unsigned)time(NULL));
    memset(val,0,sizeof(val));

    Gen_Data();

    long long countA, countB, countC, countD;
```

```
countA = countB = countC = countD = 0;
for(int i = 1; i <= REAPT_TIMES; i++)
{
    A(x);
    countA += count;
}
for(int i = 1; i <= REAPT_TIMES; i++)
{
    D(x);
    countD += count;
}
for(int i = 1; i <= REAPT_TIMES; i++)
{
    B(x);
    countB += count;
}
for(int i = 1; i <= REAPT_TIMES; i++)
{
    C(x);
    countC += count;
}

    printf("countA = %lld\n",
countA/REAPT_TIMES);
    printf("countD = %lld\n",
countD/REAPT_TIMES);
    printf("countB = %lld\n",
countB/REAPT_TIMES + (long long)sqrt(n));
    printf("countC = %lld\n",
countC/REAPT_TIMES + n/6);

    return 0;
}
```

实验结果如下所示：

```
lutouch@acsa-gpu:~/Code/Algorithm-Design-And-Analyse$ ls
a.out  sherwood.cpp
lutouch@acsa-gpu:~/Code/Algorithm-Design-And-Analyse$ g++ sherwood.cpp ;./a.out
countA = 60000
countD = 42736
countB = 470
countC = 20007
lutouch@acsa-gpu:~/Code/Algorithm-Design-And-Analyse$
```

实验中同样采用的多次计算取平均值的方法，以尽量减少偶然误差。从实验结果来看，使用概率算法所需的比较次数要明显少于其他各种算法。我使用的算法 C 是将原来 B 算法中的$\sqrt{n}$改为 n/6,所得比较次数要低于只比较一次的算法 D，但不如比较$\sqrt{n}$次的算法 B。由此可见选取一个合理的比较次数对于提升算法性能十分重要。实验中的$\sqrt{n}$被证明是能使算法性能最优的比较次数。

Ex.(4.1 8 后问题)证明：当放置（k+1)th 皇后时，若有多个位置是开放的,则算法 QueensLV 选中其中任一位置的概率相等。

证明：设第 i 个位置被选中的概率为 P(i).由于第 i 个位置被选中表示在第 i 次判断 if uniform(1..nb)=1 结果为真，并且对于∀m 次(m>i),都有该判断为假。

$$\text{则P(i)} = \frac{1}{i} \times \left(1 - \frac{1}{i+1}\right) \times \left(1 - \frac{1}{i+2}\right) \times \ldots \times \left(1 - \frac{1}{nb}\right)$$

$$= \frac{1}{i} \times \frac{i}{i+1} \times \frac{i+1}{i+2} \times \frac{i+2}{i+3} \times \ldots \times \frac{nb-2}{nb-1} \times \frac{nb-1}{nb} = \frac{1}{nb}$$

因此，则算法 QueensLV 选中其中任一位置的概率相等。

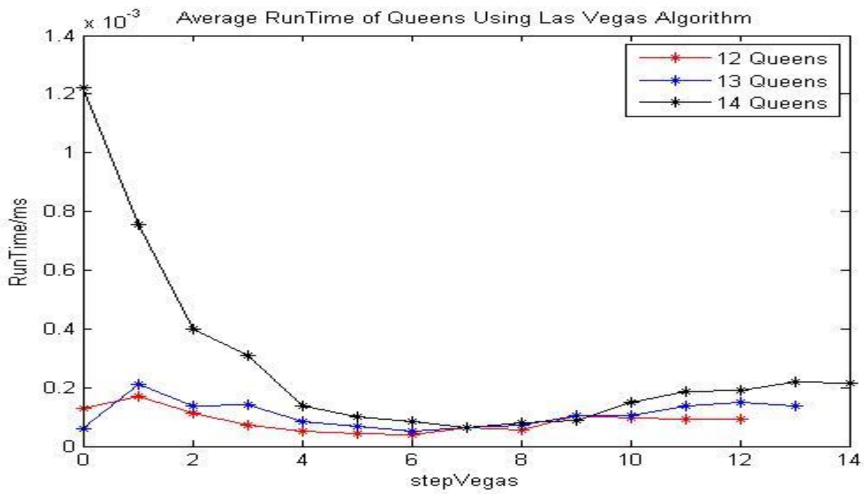
Ex. (4.1 8 后问题)写一算法, 求 n=12~20 时最优的 StepVegas 值。

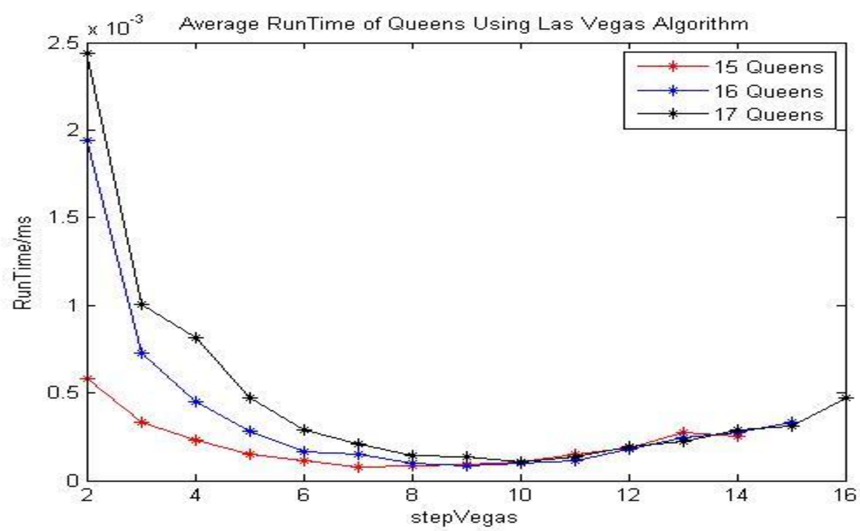
Source Code:

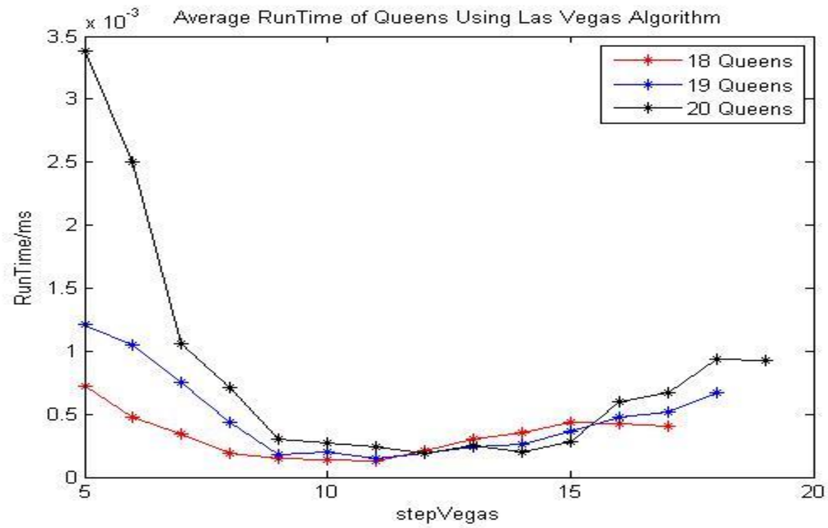
```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<sys/time.h>
#include<stack>
#define CHESS_SIZE  18
#define REAPT_TIMES 64
using namespace std;

int chess[CHESS_SIZE+1];
int stepVegas;
stack<int> st;

bool is_legal(int row, int col);
bool backtrace(int k);
bool QueenLV();
void Print_ChessBoard(int, int);

/* Check if the current postion is legal */
bool is_legal(int row, int col)
{
    if(row >= 2)
        for(int m = 1; m < row; m++)
            if( (col+row)==(chess[m]+m) || (col-
row)==(chess[m]-m) || (col == chess[m]) )
                return false;
    return true;
}


/* tranditional way to solve CHESS_SIZE
queens problem */
bool backtrace(int k)
{
    int i = k + 1; int j = 1;
    while( i <= CHESS_SIZE && i >= k + 1 )
    {
        for(; j <= CHESS_SIZE; j++)
            if(is_legal(i, j))
            {
                chess[i] = j;   st.push(j);
                i = i + 1;      j = 1;
                break;
            }
        if( j == CHESS_SIZE + 1 ) { i = i - 1;  if(i
<= k) return false; j = st.top() + 1; st.pop(); }
    }
    if( i <= k) return false;
    return true;
}

/* Use Las Vegas algorithm to determine first
stepVegas queens before calling the
tranditional algorithm */
bool QueenLV()
{
    int i, j, nb, k = 0;
    if(stepVegas == k) return backtrace(k);
    while(true)
    {
        nb = 0; /* number of open positions for
the (k+1)th queen */
        for(i = 1; i <= CHESS_SIZE; i++)
            if(is_legal(k+1, i)){nb += 1;
if( (rand()%nb + 1) == 1 ) j = i;}
        if(nb > 0){k = k + 1; chess[k] = j;}
        if( nb == 0 || k == stepVegas ) break;
    }
    if(nb > 0)  return backtrace(k);
    return false;
}

/* Print the first num_of_row rows of the
chess board */
void Print_ChessBoard(int num_of_row, int
num_of_column)
{
    for(int i = 1; i <= num_of_row; i++){
        for(int j = 1; j <= num_of_column; j++)
            if(chess[i]==j) printf("@ ");
            else printf("* ");
        printf("\n");
    }
}


/* get program run time */
double get_time() {
    struct timeval tv_start, tv_end;
    gettimeofday(&tv_start, NULL);
    for(int i = 1; i <= REAPT_TIMES; i++)
        while(!QueenLV());
    gettimeofday(&tv_end, NULL);
```

```
    return ( (tv_end.tv_sec - tv_start.tv_sec)
+ 1.0e-6*(tv_end.tv_usec - tv_start.tv_usec) )
/ REAPT_TIMES;
}

int main(int argc, char* argv[])
{
    srand((unsigned)time(NULL));
    printf("\nCHESS_SIZE
= %d\n",CHESS_SIZE);
    for(stepVegas = 0; stepVegas <=
CHESS_SIZE; stepVegas++)
        printf("%lf  ", get_time());
    while(!st.empty()) st.pop();
    return 0;
}
```

根据程序对 12~20 个皇后在不同 stepVegas 取值情况下的运行结果，测试运行时间，然后用 matlab 画出对应时间/取值图如下：







由这三张曲线图很容易观察到当皇后数与其最优 stepVegas 数的对应值如下：

Queens	12	13	14	15	16	17	18	19	20
stepVegas	6	6	7	7	9	10	10	11	12

Ex.(Ch5.25.2 素数测定)

```
PrintPrimes //打印 1 万以内的素数
{
        print 2，3；
        n ← 5；
    repeat
        if RepeatMillRab(n, lgn)  then print n;
        n ←n+2;
          until  n=10000;
}
```

与确定性算法相比较，并给出 100~10000 以内错误的比例。

Source Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

#ifdef WIN32
typedef _int64 i64;
#else
typedef long long i64;
#endif

/* get a^b mod n */
int modular_exponent(int a, int b, int n) {
    int ret = 1;
    for( ;b; b>>=1,a=(int) ((i64)a)*a%n )
        if(b&1)
        ret = (int)((i64)ret)*a%n;
    return ret;
}

int log(int n, int a)
{
    int count = 0;
    while( (n>>=1) >= 1) count ++ ;
    return count;
```

```
}

/* returns true means n is a prime or a strong psudo prime. note: n is odd, and a is a number between 2 to n - 2 */
bool Btest(int a, int n)
{
    int s = 0; int t = n - 1;
    do{
        s++; t >>= 1;
    }while(t&1 != 1);

    int x = modular_exponent(a, t, n);
    if(x == 1 || x == n - 1) return true;
    for(int i = 1; i <= s - 1; i++)
    {
        x = modular_exponent(x, 2, n);
        if(x = n - 1) return true;
    }
    return false;
}

bool MillRab(int n)
{
    int a = rand()%(n-3) + 2;
    return Btest(a, n);
}

bool RepeatMillRab(int n, int k)
{
    while(k--)
        if(!MillRab(n)) return false;
    return true;
}

int PrintPrimes()
{
    int count = 2;
    for(int n = 5; n < 10000; n += 2)
        if(RepeatMillRab(n, log(n,2))){
            count++;
        }
    return count;
}

/* search prime numbers using deterministic algorithm */
int plist[1300], pcount = 0;
int prime(int n)
{
    int i;
    if((n!=2&&!(n%2))||(n!=3&&!(n%3))||(n!=5&&!(n%5))||(n!=7&&!(n%7)))
        return 0;
    for(i = 0; plist[i]*plist[i] <= n; i++)
        if(!(n%plist[i]))
            return 0;
    return n > 1;
}

void initprime()
{
    int i;
    for(plist[pcount++]=2,i=3;i<10000;i++)
        if(prime(i))
            plist[pcount++]=i;
}
/* end of the deterministic algorithm*/

int main()
{
    srand((unsigned)time(NULL));
    initprime();
    int count = 0; int times = 4096;
    for(int i = 1; i <= times; i++)
        count += PrintPrimes();
    printf("error rate = %f\n",(float)(count-times*pcount)/(times*pcount));
    return 0;
}
```

为尽量减少偶然误差，我在程序中设置调用 PrintPrime 4096 次，最后计算时取这 4096 次的平均值。如此，多次调用结果的误差控制在了 0.0005%以内。

```
fanggq@DHMP:~/Algorithm-Design-and-Analyse$ ls
ex2.c   ex3.c   MyQueen.cpp   prime.cpp   QueensLV   SetCount.cpp
fanggq@DHMP:~/Algorithm-Design-and-Analyse$ g++ prime.cpp
fanggq@DHMP:~/Algorithm-Design-and-Analyse$ ./a.out
error rate = 0.000072
fanggq@DHMP:~/Algorithm-Design-and-Analyse$
```

Ex 证明：G 中最大团的 size 为 α 当且仅当 Gm 里最大团的 size 是 $m\alpha$

充分性：若 G 中最大团的 size 为 α，根据 Gm 的构造过程可得 Gm 中的至少存在一个 size 为 $m\alpha$ 的团。假设 Gm 中存在 size 大于 $m\alpha$ 的团，则说明构成 Gm 的各个 G 贡献的结点数大于 α，记为 β；对于每个副本 G，这 β 个结点同样可以构成完全图，这与“G 中最大团的 size 为 α”矛盾，因此 Gm 里最大团的 size 是 $m\alpha$。

必要性：若 Gm 里最大团的 size 是 $m\alpha$，根据 Gm 的构造过程可知每个 G 的副本贡献了 α 个两两相连的结点，即 G 中存在一个 size 至少为 α 的团。如果 G 的最大团的 size 为 $\beta > \alpha$，则可以构造出 Gm 中的 size 为 $m\beta$ 的完全子图，与“Gm 的最大团的 size 是 $m\alpha$”矛盾，因此 G 的最大团的 size 为 α。