# 算法分析与设计第一次作业

## 姓名：郭昊　　　学号：SA14011008

# 概率算法：

Ex1. 若将 y ← uniform(0, 1) 改为 y ← x, 则上述的算法估计的值是什么？

算法伪代码：

Darts(n){

　　$k \leftarrow 0$;

　　for $i \leftarrow 1$ to $n$ do{

　　　　$x \leftarrow uniform(0,1)$;

　　　　$y \leftarrow x$;

　　　　if $x^2 + y^2 \le 1$ then $k++$;

　　}

　　return $4k/n$;

}

实验结果：

$n = 1000$万: 2.827866, 2.82869

$n = 1$亿: 2.828503, 2.828453

$n = 10$亿: 2.828449, 2.828312

Ex2.在机器上用 $4\int_0^1 \sqrt{1-x^2}dx$ 估计 $\pi$ 值，给出不同的 $n$ 值及精度

算法伪代码：

HitorMiss(f,n){

　　$k \leftarrow 0$;

　　for $i \leftarrow 1$ to $n$ do{

　　　　$x \leftarrow uniform(0,1)$;

　　　　$y \leftarrow uniform(0,1)$;

　　　　if $y \le f(x)$ then $k++$;

　　}

```
    return 4k / n;
}
```

实验结果：

$n = 1000$万 : 3.141878, 3.141426

$n = 1$亿 : 3.141548, 3.141552

$n = 10$亿 : 3.141543, 3.141540

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

算法代码：

```
double integral(double(*func)(double &x), double a, double b, double c, double d, int n)
{
    int k1 = 0;//正区域的点
    int k2 = 0;//负区域的点
    srand(time(0));
    //先求[a,b],[0,c]之间正方形的面积，在[a,b]和[c,d]之间进行模拟,n 代表点的个数
    //这里需要考虑三种情况，即 c>0,d<0,(c<0 并且 d>0)
    if (c > 0){
        //点全部在 x 轴上方
        for (int i = 0; i < n; i++){
            double x = a + ((double)rand()) / RAND_MAX*(b - a);
            double y = d*((double)rand()) / RAND_MAX;
            if (func(x) >= y){
                k1++;
            }
        }
        return d*(b - a)*1.0*k1 / n;
    }
    else if(d<0){
        //点全部在 x 轴下方
        for (int i = 0; i < n; i++){
            double x = a + ((double)rand()) / RAND_MAX*(b - a);
            double y = c*((double)rand()) / RAND_MAX;
            if (func(x) <= y){
                k2++;
            }
        }
        return c*(b - a)*1.0*k2 / n;
    }
```

```
else if(c < 0 && d > 0){
    //部分点在 x 轴上方，部分点在 x 轴下方,分两段来求，先求出交点
    //首先二分法求零点，必然存在
    double limit = 0.00001;
    double first = a;
    double last = b;
    while (last - first > limit){
        double middle = (last + first) / 2;
        if (func(middle)*func1(last) < 0){
            first = middle;
        }
        else
        {
            last = middle;
        }
    }
    //得到零点后分段计算[a-first][first-b]
    if (func1(a) > 0){
        //[a - first]为正半段[first-b]为负半段
        for (int i = 0; i < n; i++){
            double x = a + ((double)rand()) / RAND_MAX*(first - a);
            double y = d*((double)rand()) / RAND_MAX;
            if (func(x) >= y){
                k1++;
            }
        }
        for (int i = 0; i < n; i++){
            double x = first + ((double)rand()) / RAND_MAX*(b - first);
            double y = c*((double)rand()) / RAND_MAX;
            if (func(x) <= y){
                k2++;
            }
        }
        return d*(first - a)*1.0*k1 / n + c*(b - first)*1.0*k2 / n;
    }
    else{
        //[first-b]为正半段[a - first]为负半段
        for (int i = 0; i < n; i++){
            double x = first + ((double)rand()) / RAND_MAX*(b - first);
            double y = d*((double)rand()) / RAND_MAX;
            if (func(x) >= y){
                k1++;
            }
        }
```

```
            for (int i = 0; i < n; i++){
                double x = a + ((double)rand()) / RAND_MAX*(first - a);
                double y = c*((double)rand()) / RAND_MAX;
                if (func(x) <= y){
                    k2++;
                }
            }
            return d*(b - first)*1.0*k1 / n + c*(first - a)*1.0*k2 / n;
        }

    }
}
double func1(double &x)
{
    return 1-x*x;
}
```

其中， $f(x)=1-x^2, a=0, b=1, c=0, d=1$

实验结果：

$n=1000$万: 0.6666521, 0.6665794

$n=1$亿 : 0.6666685, 0.6666497

$n=10$亿 :0.6666575，0.6666535

Ex4 设 ε , δ 是(0,1)之间的常数，证明：
若 I 是 ∫ $f(x)dx$1 0 的正确值，h 是由 HitorMiss 算法返回的值，则当 n ≥ I(1-I)/ ε 2 δ 时有：Prob[|h-I| < ε ] ≥ 1 − δ 上述的意义告诉我们：Prob[|h-I| ≥ ε ] ≤ δ，即：当 n ≥ I(1-I)/ ε 2 δ 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 $1/4$ 圆内的点数量，则 $X \sim B(n,I)$，所以有：

$EX = n*I$

$DX = n*I*(1-I)$

根据切比雪夫不等式：

$$P\{|n*h-n*I|<x\} \ge 1-\frac{D(x)}{x^2}$$

那么 $P\{|h-I|<\frac{x}{n}\} \ge 1-\frac{D(x)}{x^2}$

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

那么

$$P\{|h-I|<\frac{x}{n}\} = P\{|h-I|<\varepsilon\} \geq I - \frac{D(x)}{\varepsilon^2+n^2} = I - \frac{n*I(1-\mathrm{I})}{\varepsilon^2+n^2} = I - \frac{I(1-\mathrm{I})}{\varepsilon^2/n+n}$$

又因为

$$n \geq I(1-\mathrm{I})/\varepsilon^2\delta$$

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

原命题得证

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。

算法代码：

```
#include<iostream>
#include<set>
#include<cmath>
#include<ctime>
using namespace std;
#define PI 3.1415926
#define SizeX 1000
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;
}
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()
{
```

```
    set<int> X;
    srand((unsigned)time(NULL));
    for (int i = 0; i < SizeX; i++){
        X.insert(i);
    }
    int N;
    for (N = 10; N <= 1000; N *= 10){
        int Sum = 0;
        for (int i = 0; i < N; i++){
            Sum += SetCount(X);
        }
        printf("N=%d Xsize = %d\n",N,(Sum/N));
    }
    return 0;
}
```

算法运行结果：

```
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

dlogRH 算法就是 Sherwood 算法的一个实例，通过随机预处理，将输入实例 p 随机变换为 c，一定概率上可以降低计算 dlog 的时间复杂度，这里采用的函数 u 就是 ModuleExponent 和模乘法将 p 转换为 c， 然后利用确定性算法 dlog 算法计算 c 的离散对数 y，最后根据数论知识，通过变换(y-r)mod(p-1)也就是函数 v 将 y 还原出原问题的解 x。

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。

算法代码：

```
import java.util.Random;
public class Ex7 {
    int n=20;
    int head=3;
    int val[]={2,17,11,1,7,15,20,3,19,14,8,5,18,10,13,6,4,12,16,9};
    int ptr[]={7,12,17,0,10,18,100,16,6,5,19,15,8,2,9,4,11,14,1,13};
    public int search(int x,int i){
        int k=0;
        while(x>val[i]){
            i=ptr[i];
            k++;
        }
        System.out.print(",共比较了"+k+"次,");
        return i;
```

```
}
public int A(int x){//  时间为 O(n)的确定性算法
        return search(x,head);
}
public int B(int x){//时间为 O( √n)的确定性算法
        int j ,y, i=head;
        int max=val[i];
        for(j=0;j<Math.sqrt(1.0*n);j++){
                y=val[j];
                if(max<y&&y<=x){
                        i=j;
                        max=y;
                }
        }
        return search(x,i);
}
public int C(int x){//在算法那 B 上改进的 sherwood 算法
        Random r=new Random();
        int j,k,i=head;
        int max=val[i];
        for(j=0;j<Math.sqrt(1.0*n);j++){
                k=r.nextInt(n);
                if(max<val[k]&&val[k]<=x){
                        i=k;
                        max=val[k];
                }
        }
        return search(x,i);
}
public int D(int x){ //时间为 O(n)的概率算法
        Random r = new Random();
        int i=r.nextInt(n);
        int y=val[i];
        if(x<y)
                return search(x,head);
        else if(x>y)
                return search(x,ptr[i]);
        else
                return i;
}
public static void main(String args[]){
        Ex7 ex=new Ex7();
        int pos=0;
        int target=19;
```

```
            System.out.println("有序静态链表:");
            System.out.println("val[]={2,17,11,1,7,15,20,3,19,14,8,5,18,10,13,6,4,12,16,9}");
            System.out.println("ptr[]={7,12,17,0,10,18,100,16,6,5,19,15,8,2,9,4,11,14,1,13}");
            System.out.print("算法 A 查找"+target);
            pos=ex.A(target);
            System.out.println(",位置是:"+pos);
            System.out.print("算法 B 查找"+target);
            pos=ex.B(target);
            System.out.println(",位置是:"+pos);
            System.out.print("算法 C 查找"+target);
            pos=ex.C(target);
            System.out.println(",位置是:"+pos);
            System.out.print("算法 D 查找"+target);
            pos=ex.D(target);
            System.out.println(",位置是:"+pos);
      }
}
```

实验结果：

| 算法 | 比较次数 |
|---|---|
| A | 18 |
| B | 2 |
| C | 3 |
| D | 6 |

Ex. 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： 对于任意 $m \in Z$ 满足$1 \le m \le n_b$，第$m$个位置被选中的概率等于

$$\frac{1}{m} \times \frac{m}{m+1} \times \frac{m+1}{m+2} \times \cdots \times \frac{n_b - 1}{n_b} = \frac{1}{n_b}$$

故对于$(k+1)th$皇后，若有个开放位置，则每个位置被选中的概率都是$\frac{1}{n_b}$

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

算法关键代码：

```
/* 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]))
```

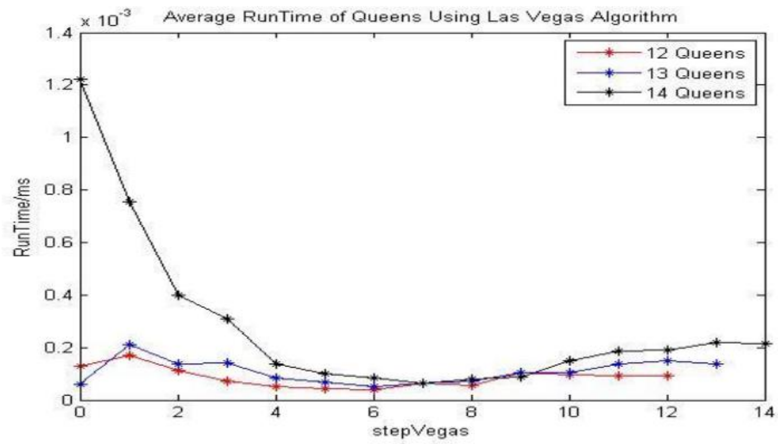
```
                              return false;
        return true;
}
/* tranditional way to solve CHESS_SIZE
queens problem */
bool backtrace(int k)
{
        int i = k + 1; int j = 1;
        while (i <= CHESS_SIZE && i >= k + 1)
        {
                for (; j <= CHESS_SIZE; j++)
                        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 firs
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;
```

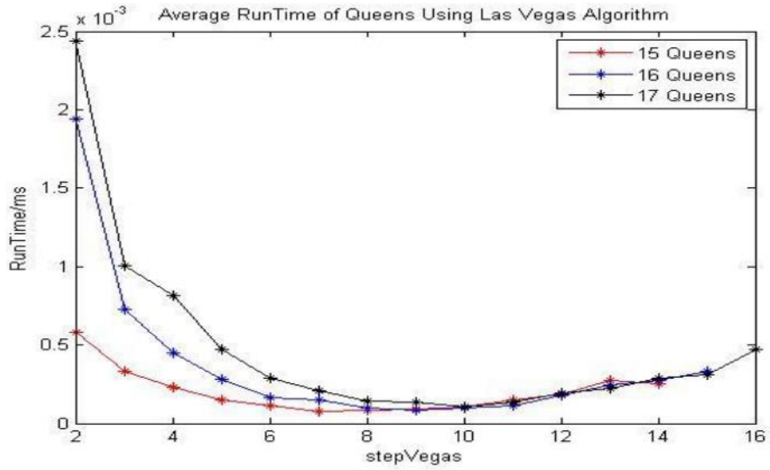
```
        }
        if (nb > 0)    return backtrace(k);
        return false;
}
/* Print the first num_of_row rows of the
chess board */
void Print_ChessBoard(int num_of_row, int
        num_of_column)
{
        for (int i = 1; i <= num_of_row; i++){
                for (int j = 1; j <= num_of_column; 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;
}
```

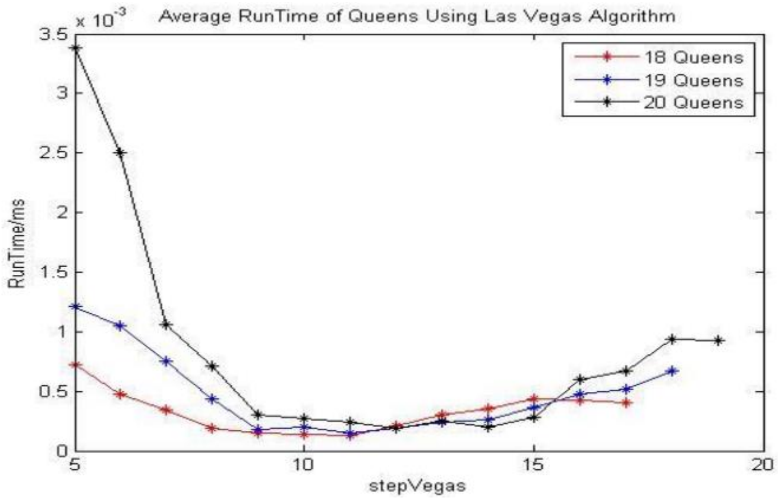
Main 函数：

```
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();
```

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 100~1000 以内错误的比例。

算法关键代码：

```
/* 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;
}
main 函数部分
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));
```

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

```
error rate = 0.000072
```

# 近似算法：

Ex　证明：G 中最大团的 size 为α当且仅当 Gm 里最大团的 size 是 mα

(→)记G中最大团为G'，G的边集合为E，则$|G'|$的size为α，根据最大团的定义，对于$\forall u \in G - G'$，总有$v \in G'$使得$uv \notin E$,故G的m次拷贝的最大团G'的m次拷贝，即$G^m$里最大团的size是mα。

(←)根据$G^m$的定义，任何一个点与其他拷贝中的所有点都是相连的，可以知道对于$G^m$，若最大团$G^{m}{}'$的size为mα，则对应点集合可以写成$\{v_{11}, v_{12}, \cdots, v_{1m}, \cdots, v_{i1}, v_{i2}, \cdots, v_{im}, \cdots, v_{\alpha 1}, v_{\alpha 2}, \cdots, v_{\alpha m}\}$，其中$v_{i1}, v_{i2}, \cdots, v_{im}$刚好对应于 G 中的点 vi，即$G^m$的最大团刚好对应于 G 的最大团，其点集合为$\{v_1, v_1, \cdots, v_{\alpha},\}$，假设没有这种对应关系，则必然存在一点 u 属于 G 的最大团，然而并不属于$G^m$的最大团，可以发现 u 与$G^m$的最大团中所有点都相连，则可以把 u 加入$G^m$的最大团，故这种对应关系存在，$|G^{m}{}'| = m\alpha \ \rightarrow |G'| = \alpha$。