**概率算法**

**P20.EX1**

因为y=x,所以算法统计的是第一象限内落在y=x这条直线上的点，可得：

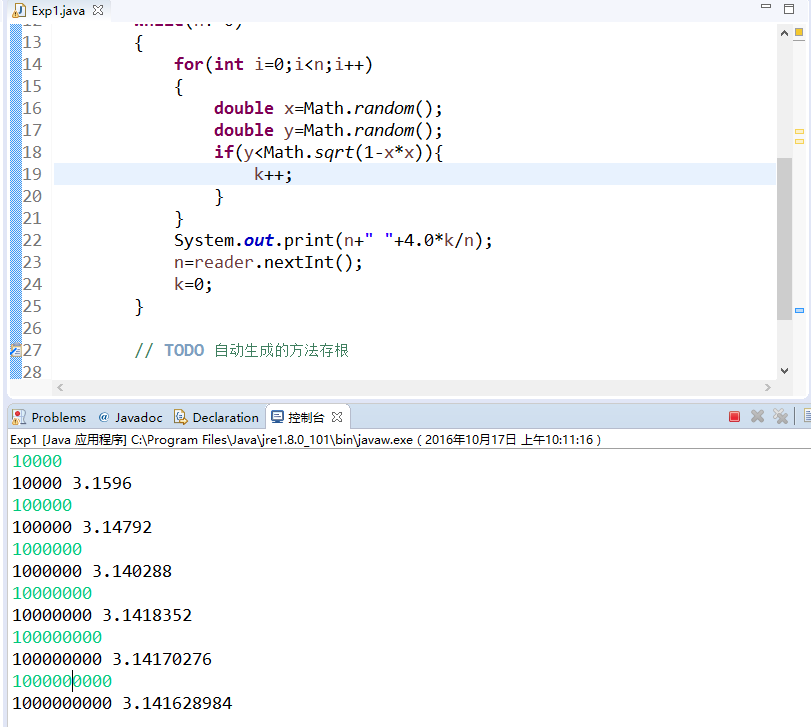


代表y=x这条直线在圆弧内的长度和正方形对角线的长度之比

所以算法估计4\*1/=的值。

**P23.EX2**

结果如下：



实验程序如下：

**package** com.wjs.designOfalgorithms;

**import** java.util.Scanner;

**public** **class** Exp1 {

**public** **static** **void** main(String[] args) {

**long** n=0,k=0;

**int** iter=1;

Scanner reader=**new** Scanner(System.***in***);

n=reader.nextInt();

**while**(n!=0)

{

**for**(**int** i=0;i<n;i++)

{

**double** x=Math.*random*();

**double** y=Math.*random*();

**if**(y<Math.*sqrt*(1-x\*x)){

k++;

}

}

System.***out***.print(n+" "+4.0\*k/n);

n=reader.nextInt();

k=0;

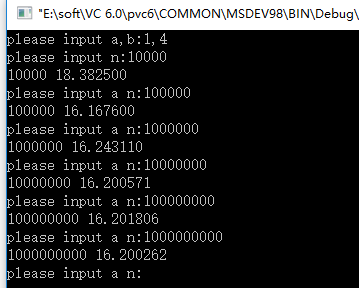
}

}

}

**P23.EX3**

取,设a=1,b=4,c=1,d=16,n=10000,100000,1000000,10000000,100000000,1000000000的情况如下：



代码如下：

#include<iostream>

#include<time.h>

#include<stdlib.h>

using namespace std;

int main()

{

float myFunction(float x);

void intergral(float a,float b,float c,float d,float (\*p)(float x));

float a,b,c,d;

float temp;

printf("please input a,b:");

scanf("%f,%f",&a,&b);

c=myFunction(a);

d=myFunction(b);

if(c>d)

{

temp=c;

c=d;

d=temp;

}

intergral(a,b,c,d,&myFunction);

return 0;

}

float myFunction(float x)

{

return x\*x;

}

void intergral(float a,float b,float c,float d,float (\*p)(float x))

{

int k=0,i=0,n=0;

float x,y;

printf("please input n:");

srand((unsigned)time(NULL));

scanf("%d",&n);

while(n!=0)

{

for(;i<n;i++)

{

x=(a+(b-a)\*rand()/RAND\_MAX);

y=(c+(d-c)\*rand()/RAND\_MAX);

if(y<(\*p)(x))

k++;

}

printf("%d %f\n",n,(d-c)\*(b-a)\*k/n);

printf("please input a n:");

scanf("%d",&n);

k=0;

}

}

**P24.EX4**

解：由题意知：h是一个随机变量，设其期望和方差分别为E(h)和Var(h),由切比雪夫不等式知：

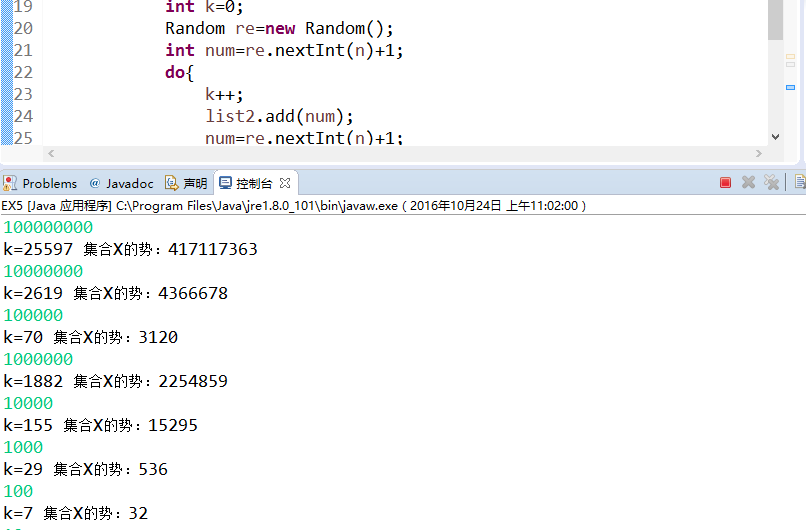


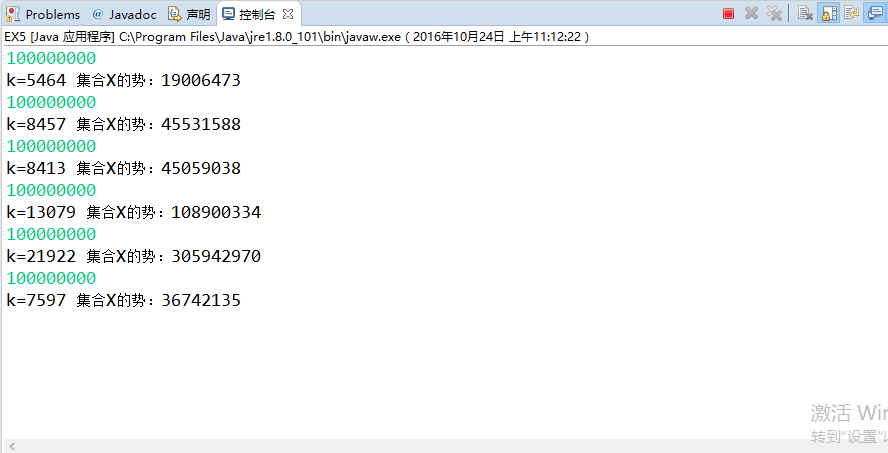
由题意可知I=E(h),从算法中我们知道随机取n个点，若k个点落在积分范围内，则h=k/n,k是服从二项分布K~B(n,I),所以Var(K)=n\*I\*(I-1),由于k=n\*h,则Var(h)=I\*(I-1)/n,将带入

可得

 得证。

P36 EX5程序的运行结果如图所示：





程序如下：

**package** exercise;

**import** java.util.ArrayList;

**import** java.util.Random;

**import** java.util.Scanner;

**public** **class** EX5 {

**public** **static** **void** main(String[] args) {

Scanner reader=**new** Scanner(System.***in***);

**int** n=reader.nextInt();

**while**(n>0)

{

ArrayList<Integer> list1=**new** ArrayList<>(n);

ArrayList<Integer> list2=**new** ArrayList<>(n);

**for**(**int** i=0;i<n;i++){

list1.add(i);

}

**int** k=0;

Random re=**new** Random();

**int** num=re.nextInt(n)+1;

**do**{

k++;

list2.add(num);

num=re.nextInt(n)+1;

} **while**(list2.indexOf(num)==-1);

**int** size=(**int**)Math.*ceil*(2\*k\*k/3.1415926);

System.***out***.println("k="+k+" 集合X的势："+size);

n=reader.nextInt();

}

}

}

结果分析：n越大，且多次运行，对结果的估计值越准确。

**P54 EX6**

dlogRH的工作原理如下：

算法在(0,...,p-2)中随机选取一个r,计算r的离散对数b，因为，所以程序接下来计算b\*a mod p相当于计算，所以引入了随机的思想，讲输入实例a变为变为一个随机的输入实例c，所以程序接下求,y的值是随机实例的值，其实也是x+r的值，所以程序最后返回就是原问题所求的x。

算法中对应的

**P67 EX7**

对应的各个算法实现如下：

**package** com.wjs.designOfalgorithms;

**public** **class** MySearch {

**int** val[]={-1,2,3,13,1,5,21,8};

**int** ptr[]={-1,2,5,6,1,7,0,3};

**int** head=4;

**public** **int** Search(**int** x,**int** i) {

**while** (x>val[i]) {

i=ptr[i];

}

**return** i;

}

}

**package** com.wjs.designOfalgorithms;

**import** java.util.Scanner;

**public** **class** A {

**public** **static** **void** main(String[] args) {

// **TODO** 自动生成的方法存根

Scanner reader=**new** Scanner(System.***in***);

**int** n=reader.nextInt();

MySearch mySearch=**new** MySearch();

System.***out***.println(mySearch.Search(n, mySearch.head));

reader.close();

}

}

**package** com.wjs.designOfalgorithms;

**import** java.util.Scanner;

**public** **class** B {

**public** **static** **void** main(String[] args) {

// **TODO** 自动生成的方法存根

MySearch mySearch=**new** MySearch();

Scanner reader=**new** Scanner(System.***in***);

**int** n=reader.nextInt();

**int** i=mySearch.head;

**int** max=mySearch.val[i];

**int** ceil=(**int**) Math.*floor*(Math.*sqrt*(n));

**for**(**int** j=1;j<ceil;j++){

**int** y=mySearch.val[j];

**if** (max<y&&y<=n) {

i=j;

max=y;

}

}

System.***out***.println(mySearch.Search(n, i));

reader.close();

}

}

package com.wjs.designOfalgorithms;

import java.util.Random;

import java.util.Scanner;

public class C {

public static void main(String[] args) {

Scanner reader=new Scanner(System.in);

int x=reader.nextInt();

MySearch mySearch=new MySearch();

int n=7;

int i=mySearch.head;

int max=mySearch.val[i];

int ceil=(int) Math.ceil(n-Math.sqrt(n));

Random random=new Random();

int rd=random.nextInt(ceil);

int y=0;

for(int j=1+rd;j<rd+Math.sqrt(n);j++){

y=mySearch.val[j];

if (max<y&&y<=x) {

i=j;

max=y;

}

}

System.out.println(mySearch.Search(x, i));

reader.close();

}

}

**package** com.wjs.designOfalgorithms;

**import** java.util.Random;

**import** java.util.Scanner;

**public** **class** D {

**public** **static** **void** main(String[] args) {

// **TODO** 自动生成的方法存根

MySearch mySearch=**new** MySearch();

Random random=**new** Random();

Scanner reader=**new** Scanner(System.***in***);

**int** n=reader.nextInt();

**int** i=1+random.nextInt(mySearch.val.length-1);

**int** y=mySearch.val[i];

**if**(n<y){

System.***out***.println(mySearch.Search(n, mySearch.head));

}

**else** **if** (n>y) {

System.***out***.println(mySearch.Search(n, mySearch.ptr[i]));

}

**else** {

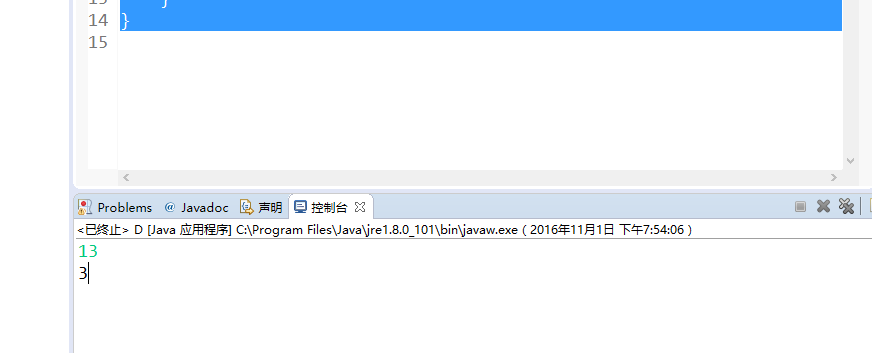
System.***out***.println(i);

}

}

}

实验结果：



**P77 EX8**

证明：

设个开放位置记为，要想当放置(K+1)个皇后，有n个开放位置，选择位置，有LV算法可知，必须满足以下条件：

1. 
2. 

易知：的概率为，的概率为。

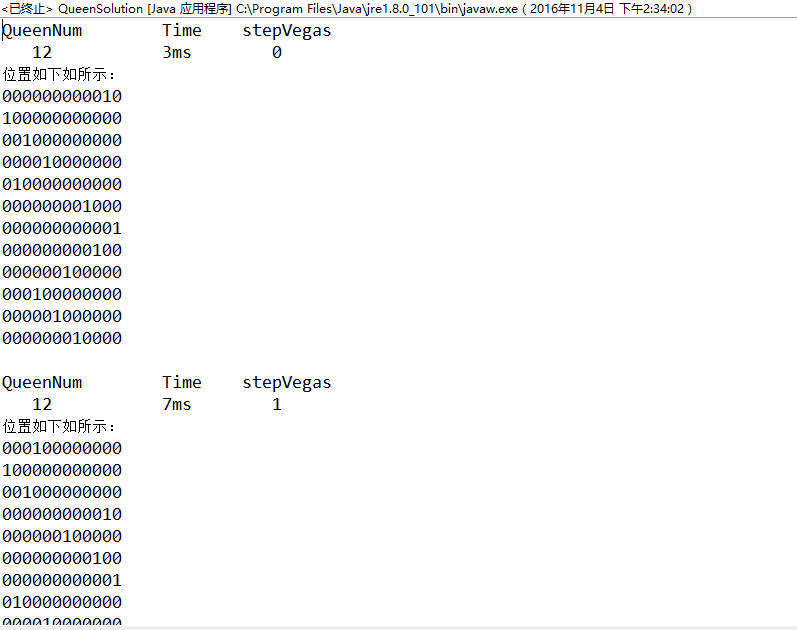
所以：

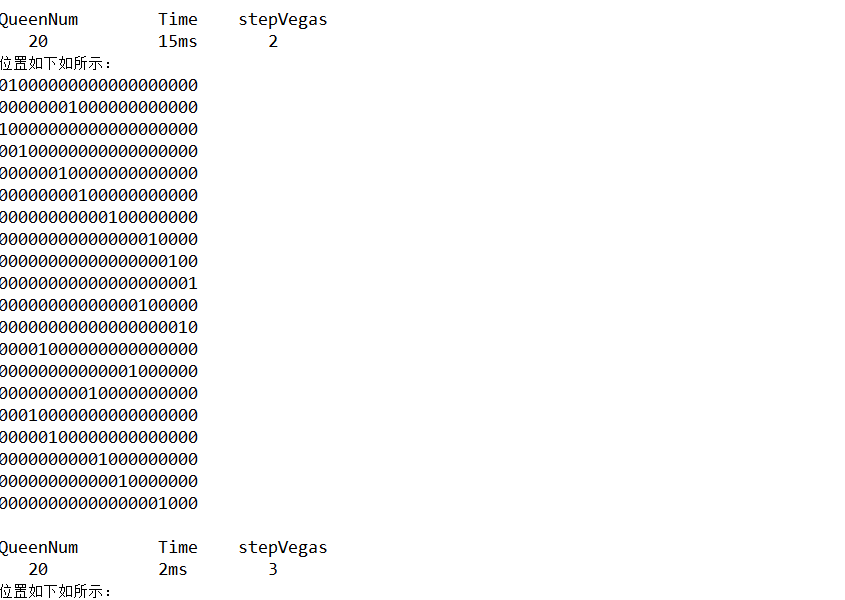


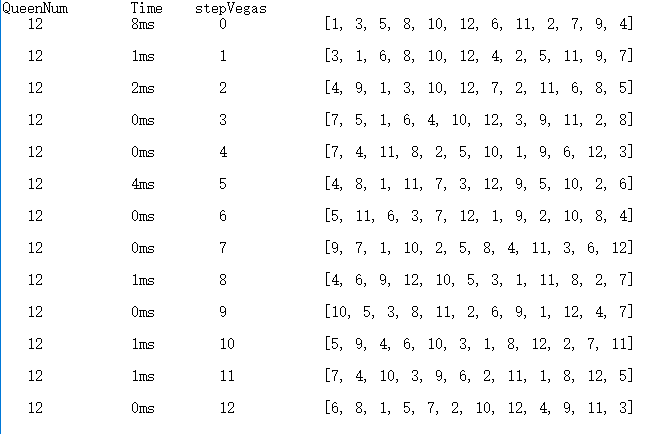
即 算法选择任何一个位置的概率都是相同的。

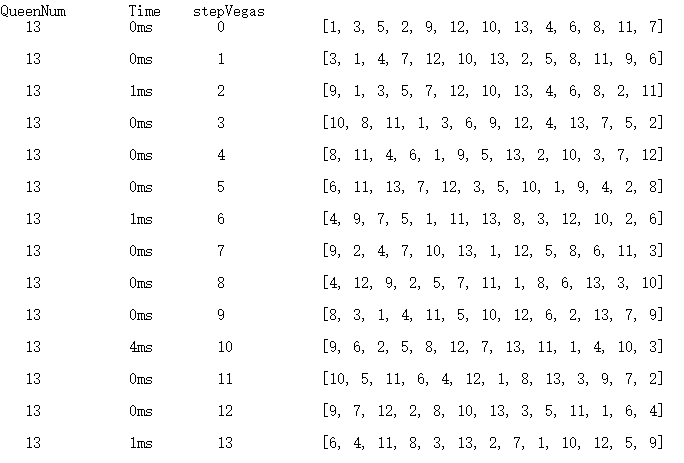
**P83 EX9**

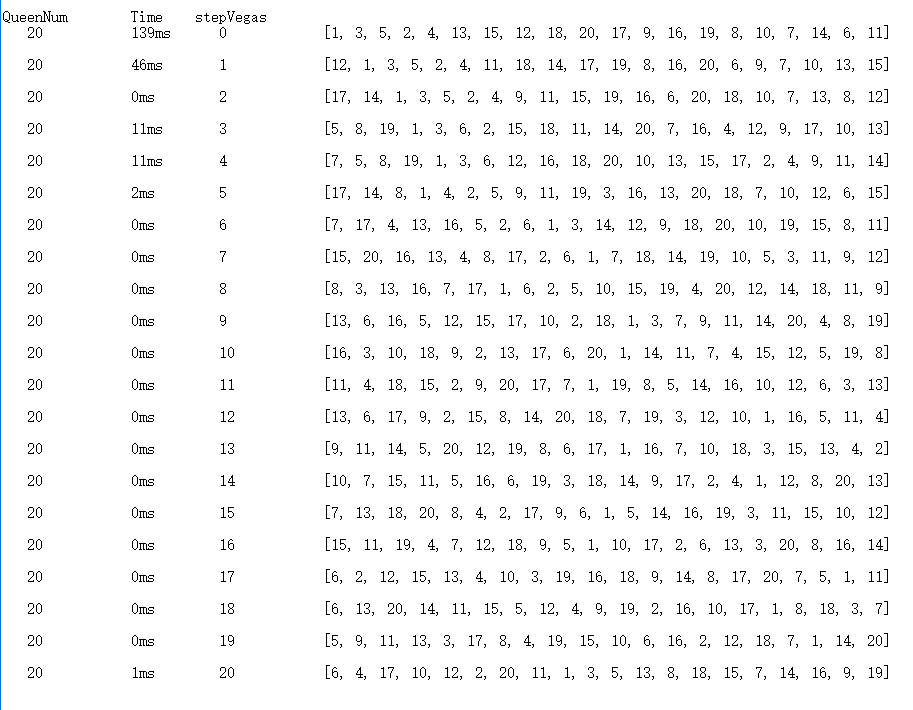
实验结果部分截图：











具体结果请看附件wjs.txt

实验的代码：

**package** algorithm\_design;

**import** java.io.BufferedWriter;

**import** java.io.File;

**import** java.io.FileWriter;

**public** **class** QueenSolution {

**public** **static** **void** main(String[] args) {

**try** {

File file=**new** File("wjs.txt");

FileWriter fileWriter=**new** FileWriter(file);

BufferedWriter bufferedWriter=**new** BufferedWriter(fileWriter);

**for**(**int** n=12;n<=20;n++){

bufferedWriter.write("QueenNum\tTime\tstepVegas");

bufferedWriter.newLine();

**for**(**int** stepVegas=0;stepVegas<=n;stepVegas++){

**boolean** flag=**false**;

**long** startTime=System.*currentTimeMillis*();

**for**(**int** i=0;i<100;i++){

flag=**false**;

**while**(!flag){

NQueen queen=**new** NQueen(n);

queen.NQueenLv(stepVegas);//确定算法与LV的折中算法

**if** (queen.success) {

flag=**true**;

**break**;

}

}

}

**long** endTime=System.*currentTimeMillis*();

System.***out***.println("QueenNum\tTime\tstepVegas");

System.***out***.println(" "+n+"\t\t"+(endTime-startTime)/100+"ms\t "+stepVegas+"\t\t");

StringBuilder stringBuilder=**new** StringBuilder(" "+n+"\t\t"+(endTime-startTime)/100+"ms\t "+stepVegas);

bufferedWriter.write(stringBuilder.toString());

bufferedWriter.newLine();

bufferedWriter.newLine();

}

}

bufferedWriter.close();

fileWriter.close();

} **catch** (Exception e) {

System.***out***.println("error"+e);// **TODO**: handle exception

}

}

}

package com.wjs.designOfalgorithms;

import java.util.ArrayList;

import java.util.Iterator;

import java.util.Random;

import java.util.Stack;

import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter.DEFAULT;

public class NQueen {

public ArrayList<Integer> coList;

public ArrayList<Integer> diag45;

public ArrayList<Integer> diag135;

int N;

boolean success;

public NQueen(int n){

coList=new ArrayList<>();

diag45=new ArrayList<>();

diag135=new ArrayList<>();

N=n;

success=false;

}

protected void removeElement(ArrayList<Integer> list,int element) {

Iterator<Integer> iterator=list.iterator();

while (iterator.hasNext()) {

if (iterator.next().equals(element)) {

iterator.remove();

}

}

}

protected void dfs(int k){

//递归的深度优先遍历

if (k>=N) {

success=true;

return;

}

for(int i=1;i<=N;i++){

if (coList.contains(i)||diag45.contains(i-k-1)||diag135.contains(i+k+1))

continue;

k=k+1;

coList.add(i);

diag45.add(i-k);

diag135.add(i+k);

dfs(k);

if (success) {

return;

}

removeElement(coList, i);

removeElement(diag45, i-k);

removeElement(diag135, i+k);

k--;

}

}

protected void backTrace(int k){

dfs(k);

}

public void NQueenLv(int stepVagas) {

int k=0;//行号

int j=0;

int spcount=1;

Random random=new Random();

//LV算法找前stepVegas行

if (stepVagas>0) {

do{

spcount=0;

j=0;

for(int i=1;i<=N;i++){

if (!coList.contains(i)&&!diag45.contains(i-k-1)&&!diag135.contains(i+k+1)) {

spcount++;

int selection=random.nextInt(spcount)+1;

if (selection==1) {

j=i;

}

}

}

if (spcount>0) {

k=k+1;

coList.add(j);

diag45.add(j-k);

diag135.add(j+k);

}

}while(spcount!=0&&k<stepVagas);

if (spcount>0) {

backTrace(k);//确定的回溯算法

}

else {

success=false;

}

}

else{

backTrace(k);

}

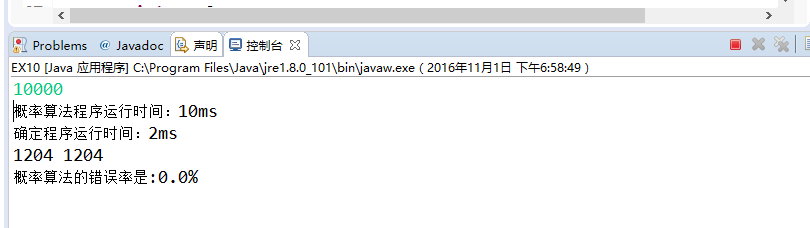
}

}

观察wjs.txt中的结果可以知道，stepVegas一般选在n一半附近比较好，验证的理论。

**P147 EX10**

实验结果：



分析：概率算法运行时间并不比确定性算法运行时间快。

实验代码如下：

**package** com.wjs.designOfalgorithms;

**import** java.util.HashMap;

**import** java.util.Scanner;

**public** **class** EX10 {

**public** **static** **void** main(String[] args) {

// **TODO** 自动生成的方法存根

Scanner reader=**new** Scanner(System.***in***);

**int** n=reader.nextInt();

**while**(n!=0)

{

**int** []cernum=**new** **int**[n];

**int** []pronum=**new** **int**[n];

**int** len1=0,len2=0;

Prime p=**new** Prime(n);

**long** startTime=System.*currentTimeMillis*();

len1=p.CaculatePrime(n);//概率算法

**long** endTime=System.*currentTimeMillis*();

System.***out***.println("概率算法程序运行时间："+(endTime-startTime)+"ms");

startTime=System.*currentTimeMillis*();

len2=*CertainAlgorithm*(cernum,n);//确定算法

endTime=System.*currentTimeMillis*();

System.***out***.println("确定程序运行时间："+(endTime-startTime)+"ms");

pronum=p.GetPrime();

/\*for(int i=0;i<len1;i++){

System.out.print(pronum[i]+"\t");

if ((i+1)%10==0) {

System.out.println();

}

}

System.out.println();

for(int i=0;i<len2;i++){

System.out.print(cernum[i]+"\t");

if ((i+1)%10==0) {

System.out.println();

}

}

System.out.println();\*/

HashMap<Integer, Integer> map=**new** HashMap<>();

**int** correctNum=0;

**int** total=0;

**for**(**int** i=0;i<len2;i++){

map.put(cernum[i], cernum[i]);

}

**for**(**int** i=0;i<len1;i++){

**if** (pronum[i]>=100) {

total++;

**if** (map.containsKey(pronum[i])&&map.get(pronum[i])==pronum[i]) {

correctNum++;

}

}

}

System.***out***.println(total+" "+correctNum);

System.***out***.println("概率算法的错误率是:"+(total-correctNum)\*1.0/total\*100+"%");

n=reader.nextInt();

reader.close();

}

}

**public** **static** **int** CertainAlgorithm(**int** []num, **int** n){

**int** up=0;

**int** k=0;

**int** flag=0;

**for**(**int** i=2;i<=n;i++){

up=(**int**)Math.*sqrt*(i);

**for**(**int** j=2;j<=up;j++){

**if** (i%j==0) {

flag=1;

**break**;

}

}

**if** (flag==0) {

num[k++]=i;

}

flag=0;

}

**return** k;

}

}

**package** com.wjs.designOfalgorithms;

**import** java.util.Random;

**public** **class** Prime {

**private** **int** primNum[];

**public** Prime(**int** n) {

primNum=**new** **int**[n];

// **TODO** 自动生成的构造函数存根

}

**public** **int**[] GetPrime() {

**return** primNum;

}

**protected** **int** ModularExponent(**int** a,**int** j,**int** p){

**int** s=1;

**while**(j>0){

**if**(j%2!=0){

s=(s\*a)%p;

}

a=a\*a%p;

j=j/2;

}

**return** s;

}

**protected** **boolean** Btest(**int** a,**int** n){

**int** s=0;

**int** t=n-1;

**do** {

s++;

t=t/2;

} **while** (t%2!=1);

**int** x=ModularExponent(a, t, n);

**if** (x==1||x==n-1) {

**return** **true**;

}

**for**(**int** i=1;i<s;i++){

x=x\*x%n;

**if** (x==n-1) {

**return** **true**;

}

}

**return** **false**;

}

**protected** **boolean** MillRab(**int** n) {

Random random=**new** Random();

**int** a=random.nextInt(n-3)+2;

**return** Btest(a, n);

}

**protected** **boolean** RepeatMillRob(**int** n,**int** k) {

**for**(**int** i=0;i<k;i++){

**if**(!MillRab(n)){

**return** **false**;

}

}

**return** **true**;

}

**public** **int** CaculatePrime(**int** n) {

primNum[0]=2;

primNum[1]=3;

**int** k=2;

**int** i=5;

**while**(i<=n){

**int** iter=(**int**)Math.*floor*(Math.*log*(i)/Math.*log*(2));

**if** (RepeatMillRob(i, iter)) {

primNum[k++]=i;

}

i+=2;

}

**return** k;

}

}

近似算法

**EX1 G中最大团的size为α当且仅当里最大团的size是mα.**

充分性：对图中的顶点的删除，每删除一个G最大团中的顶点，G最大团的size都减1，以此类推，显然有最大团的size为mα。

必要性：由定义可知，是不同G中任意两顶点连线以及G中原有边形成的图，不妨设G中的两个图分别标记为m,n。m中的每个顶点与n中的一个点都有边相连，所以假设G中的一个最大团size为α，以此类推，m,n组合形成的最大团的size为2α，同理最大团的size为mα。

**EX2 完善多机调度的LPT算法性能（近似）比的证明。**

PPT给出了近似比的上界，为了完善证明，我们需要证明这个上界在某些实例下成立。考虑输入实例，满足如下的条件：

，

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
| ... |  |  |
|  |  |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
| ... |  |  |
|  |  |  |
|  |  |  |

LPT运行结果 OPT运行结果

可见且=3m,近似比为。

分布式算法

**2.1 分析在同步和异步模型下，convergecast算法的时间复杂性。**

证明：

1. 同步模型：在同步模型中，在汇聚算法的每个可允许执行里，每个高为t的树的根节点在t轮里收到所有孩子的消息，使用归纳如下：

归纳基础：t=1时，每个叶子节点距离根节点为1，每个叶子节点发送消息给根节点，根节点在t=1时刻均收到所有叶子节点发来的消息。

归纳假设：假设树的高度为t-1>1，跟节点在t-1轮收到所有孩子的msg。

归纳步骤：当树的高度为t时，由归纳假设，t-1时根的孩子会收到所有叶子节点发来的msg，由同步算法的描述可知，在第t轮时，根会收到所有孩子的msg，所以在t轮，根收到所有叶子节点的msg。

1. 异步模型：在异步模型中，在汇聚算法的每个可允许执行里，每个高为t的树的根节点至多在t时刻里收到所有孩子的消息，使用归纳如下：

归纳基础：t=1时，每个叶子节点距离根节点为1，每个叶子节点发送消息给根节点，由异步模型算法时间复杂度可知，根节点至多在t=1时刻收到所有叶子节点发来的消息。

归纳假设：假设树的高度为t-1>1，根节点至多在t-1时刻收到所有孩子的msg。

归纳步骤：当树的高度为t时，由归纳假设，t-1时刻根的孩子会收到所有叶子节点发来的msg，在至多第t时刻时，根会收到所有孩子的msg，所以在t时刻，根收到所有叶子节点的msg。

**2.2证明在引理2.6中，一个处理器在图G中是从Pr可达的，当且仅当它的parent变量曾被赋过值**

证明：1.首先证明一个处理器在图G中是从Pr可达的，可以推出它的parent变量被赋过值，因为从pr可达，所以可达节点收到过M，则执行了算法中的第五行，即upon receiving M from neighbor pj，由于是容许执行，所以第七行也会被执行，即parent变量会被设置值。

1. 再证明一个处理器的parent变量被赋过值，它在图G是从Pr可达的。因为处理器被赋过值，则算法2.2第七行被执行过，由于是容许执行，所以第五行也会被执行，收到过M，而M是由根节点发出的，所有处理器是从Pr可达的。

**2.3 证明Alg2.3构造一棵以Pr为根的DFS树。**

证明：1.先证明连通性

反证：假设存在两个相邻节点Pi，Pj，Pj是从根Pr可达的，而Pi是不可达的。因为G里一个结点从Pr可达的，当且仅当设置过自己的parent变量，所以Pi在整个过程中parent变量为nil，而Pj设置过自己的parent变量，Pj会发送M给Pi，因该执行是容许的，Pi定会收到M，则执行parent：=j，矛盾。

1. 证明无环：

假设存在一个环，Pi1,....,Pik,若Pi是Pj的孩子，则Pi会在Pj第一次收到M之第一次收到M，因为每个处理器在该环上是下一个处理器的双亲，则Pi1会在Pi1第一收到M之后第一次收到M，则矛盾！

3证明是DFS树。

根据题意只要证明任意结点的子孙结点先于其兄弟结点加入书中，设有结点P1，P2，P3，P2，P3是P1的直接相邻结点，由算法的执行可知，P1先发送M给P2，当且仅当P2向其反回一个<parent>才会向P3发送M，而P2仅在所有子结点向其返回一个<parent>才会向P1发送<parnet>,由此可知，P2的子结点永远是先入P3加入树中。

综上1,2,3可知，Alg2.3构造一棵以Pr为根的DFS树。

**2.4 证明Alg2.3的时间复杂性为O(m)。**

1同步模型：每一轮，根据算法可知消息只发往一个处理器节点，每轮中有且仅有一个消息被传输，除根节点外，所有处理器都是收到消息才被激活，所以不存在多个处理器在同一轮发送消息的情况，所以时间复杂度和消息复杂度一样。

2 异步模型：在一个时刻内至多有一个消息在传输，因此时间复杂度与消息复杂度也是一样的。

3消息复杂度：对任意一条边，在其上传输的消息至多为4个，即2个msgs和2个应答消息（parent or reject）。有m条变，消息至多为4m，所以消息复杂度维O(m)，时间复杂度也为O(m).

**2.5 修改Alg2.3获得一新算法，使构造DFS树的时间复杂性为O(n)，并证明。**

修改方法：当一个几点确定自己的双亲后，向除自己双亲的所有邻居发送一个广播消息，告诉自己已经确定双亲，让其他邻居将这个节点从未访问过的节点集合中删去。这样便不会再向它发送消息，这样每个几点只需三个消息就处理完自己的事情，所以总得消息为3n,时间复杂度为O(n).

**3.1证明同步环上不存在匿名的、一致性的Leader选举算法**

假设A是同步环上的一个匿名算法，每个处理器在系统中具有相同的状态机，如果它选中某处理器为leader，因为环是同步的且只有一种初始配置，故在R上A只有唯一的合法执行。Lemma3.1:在环R上算法A的容许执行里，对于每轮k,所有处理器的状态在第k轮结束时是相同的。即在每轮里，各处理器均发出同样的message，所以在各轮里各个处理器接收到相同的message，则状态改变也相同。导致每个处理器同时宣布自己是Leader。所以同步环系统中匿名的、一致性的领导者选举算法的算法是不存在的。

**3.2 证明异步环系统中不存在匿名的领导者选举算法。**

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，可能只是接收的时间不相同，故最终处理器的状态也相同。由于处理器接收一条消息至多需要一个时间的单位，假设某时刻某个处理器宣布自己是leader(接收m个消息)，则在有限的时间内（m个时间单位）其他处理器也会宣布自己是leader。所以异步环系统不存在匿名的选举算法。

**EX3.9 若将环划分为长度为j(j是2的方幂)的连续片断，则所有这些片断是序等价的.**

对一个整数P(0<=P<=n-1),可以表示为：



其中m=lgn;

则有

设P,Q是在同一个片段，P1，Q1在同一个片段,且设这两个片段是相邻的，由模运算的加法可得:

P1=P+L;

Q1=Q+L;

式中L是片段的长度，。

又





且P,Q在同一片段上，有



所以存在r(0<=r<=k),满足，否则,这与P,Q在同一片段上矛盾。设，则根据rev(P),rev(Q)的表示方法可得：



而





显然，P与P1的前K位相同，Q与Q1的前K位相同，由0<=s<=k得：



这两个相邻片段是序等价的，根据等价的传递性，可得所有的片段都是序等价的。