概率算法作业

- Ex.1

  显然当$y = uniform(0,1)$改为$y = x$时，k++只会在$2x^2 \leq 1$即$x \leq \frac{\sqrt{2}}{2}$时执行，那么$\frac{k}{n} \backsim \frac{\sqrt{2}}{2}$即$\frac{4k}{n} \backsim \frac{4\sqrt{2}}{2}$.因此上述算法的估计值就是$2\sqrt{2}$.

- Ex.2

n值	计算结果	计算误差
10000	3.1416	0.000007
100000	3.14212	0.000527
1000000	3.140648	0.000945
10000000	3.141111	0.0.000482
100000000	3.141555	0.000038

```
#代码
from math import sqrt
import random

def FuncPi(x):
    return sqrt(1-x*x)

def HitorMiss(n, f = None):
    k = 0
    for i in range(n):
        x = random.random()
        y = random.random()
        if y <= f(x):
            k = k + 1
    return k/n

if __name__ == '__main__':
    pi = 3.141593
    lt = [10000, 100000, 1000000, 10000000, 100000000]
    for i in lt:
        result = 4*HitorMiss(i, FuncPi)
        print(result, 'wucha:', abs(pi-result))
```

- EX.3

  函数$f(x) = \sqrt{x}$,a=0,b=4,c=0,d=2

n值	计算结果
10000	5.3208
100000	5.34624
1000000	5.330712
10000000	5.3330888

```
#代码
import math
import random

def FuncSqrt(x):
    return math.sqrt(x)


def sum(a, b, c, d, n, funcptr):
    k = 0
    size = (b-a)*(d-c)
    rect = c*(b-a)
    for i in range(n):
        x = random.uniform(a, b)
        y = random.uniform(c, d)
        if y <= (funcptr(x)):
            k+=1
    return (k/n)*size+rect


if __name__ == '__main__':
    lt = [10000, 100000, 1000000, 10000000]
    for i in lt:
        print(i, ':', sum(0, 4.0, 0, 2.0, i, FuncSqrt))
```

- EX.5
  用概率计数估计整数集合1-n的大小，并分析n的大小对估计值的影响。
  结果如下

n值	计算结果	误差率
10	8	0.2
100	93	0.07
1000	968	0.032
10000	9923	0.0077
100000	101580	0.0158

n值	计算结果	误差率
1000000	996546	0.003454
10000000	10174525	0.0174525

大体上随着n值的增大，估计值与实际值的误差逐渐缩小。

```
#代码
import random
import math

def setcount(x):
    k = 0
    S = set()
    a = random.randint(1, x)
    while True:
        k+=1
        S.add(a)
        a = random.randint(1, x)
        if a in S:
            break
    return k

def SetCount(x):
    num = 10000
    sum = 0
    for i in range(num):
        sum += setcount(x)
    avg_k = sum/num
    return 2*avg_k**2/math.pi


if __name__ == '__main__':
    for n in range(1,8):
        x = math.pow(10, n)
        print(x, ':', int(SetCount(x)))
```

- EX.6

  分析dlogRH的工作原理，指出该算法相应的u和v.

  分析：通过在x的所有可能值中进行随机选择r，并且通过$g^{r+x} \pmod p$来对实例x进行随机预处理，从而使离散对数计算的简单算法的执行与a无关.再通过$(y-r) \pmod{p-1}$将随机实例y的解转变为原实例x的解。

  $u(x,r)=((g^r \pmod p)*a) \pmod p$

  $v(y,r)=(y-r) \pmod{p-1}$

- Ex.7

  取长为10000的元素为0-999的静态链表并对链表做随机化处理，重复执行算法1000次取平均值，四种算法的平均时间和最坏时间如下：

  algA: 4940.743 9983

algB: 196.435 513

algC: 200.807 869

algD: 3301.994 9513

算法	平均复杂度	最坏复杂度
A	$\frac{N}{2}$	N
B	$2\sqrt{N}$	X
C	$2\sqrt{N}$	X
D	$\frac{N}{3}$	N

```
#代码
import random
from math import sqrt

class Sherwood(object):
    # initialize random list
    def __init__(self):
        self.__N = 10000
        self.__cnt = 0
        self.__val = []
        self.__ptr = []
        self.__head = 0
        for i in range(self.__N):
            self.__val.append(i)
        random.shuffle(self.__val)
        for i in self.__val:
            if i == self.__N - 1:
                self.__ptr.append(0)
            else:
                self.__ptr.append(self.__val.index(i+1))
        self.__head = self.__val.index(0)

    # search function
    def search(self, x, i):
        while x > self.__val[i]:
            i = self.__ptr[i]
            self.__cnt+=1
        return i, self.__cnt

    # cnt = 0
    def cnt_zero(self):
        self.__cnt = 0

    # return x
    def val_x(self):
        return random.randint(0, self.__N)

    def alg_a(self,x):
        self.cnt_zero()
        return self.search(x, self.__head)

    def alg_b(self,x):
        self.cnt_zero()
        ind = self.__head
        maxn = ind
        for j in range(int(sqrt(self.__N))):
            self.__cnt+=1
            if self.__val[maxn] < self.__val[j] and self.__val[j] <= x:
                maxn = j
        return self.search(x, maxn)

    def alg_c(self,x):
        self.cnt_zero()
        ind = self.__head
```

```
            maxn = ind
            for j in range(int(sqrt(self.__N))):
                self.__cnt+=1
                randj = random.randint(0, self.__N-1)
                if self.__val[maxn] < self.__val[randj] and self.__val[randj] <= x:
                    maxn = randj
            return self.search(x, maxn)

        def alg_d(self,x):
            self.cnt_zero()
            ind = random.randint(0, self.__N)
            y = self.__val[ind]
            self.__cnt += 1
            if x < y:
                return self.search(x, self.__head)
            elif x > y:
                return self.search(x, self.__ptr[ind])
            else:
                return ind, self.__cnt


    if __name__ == '__main__':
        S = Sherwood()
        sa = []
        sb = []
        sc = []
        sd = []
        for i in range(1000):
            x = S.val_x()
            sa.append(S.alg_a(x)[1])
            sb.append(S.alg_b(x)[1])
            sc.append(S.alg_c(x)[1])
            sd.append(S.alg_d(x)[1])
        print('algA:',sum(sa) / len(sa), max(sa))
        print('algB:',sum(sb) / len(sb), max(sb))
        print('algC:',sum(sc) / len(sc), max(sc))
        print('algD:', sum(sd) / len(sd), max(sd))
```

- Ex.8

  证明：设有m个可有位置，对任意$k \in m$算法选中k的概率为$p = \frac{1}{k} * \frac{k}{k+1} * \frac{k+1}{k+2} * \cdots * \frac{m-1}{m} = \frac{1}{m}$.因此算法选中多个可选位置中的任意一个的概率都是相等的。
- Ex.9

  对每个n值重复执行100次算法，取搜索的平均节点数为判断标准。

  结果如下：

n值	stepVegas值	nodes
12	5	78.08
13	7	88.71

n值	stepVegas值	nodes
14	7	95.81
15	7	100.02
16	8	109.28
17	8	124.54
18	9	136.30
19	11	149.31
20	13	164.18

```
#code
from random import randint
class Queen(object):
    def __init__(self):
        self.__trycnt = 0

    def queensLV(self, stepVegas, x, n):
        col = set()
        diag45 = set()
        diag135 = set()
        k = 0
        while True:
            nb = 0
            j = 0
            for i in range(n):
                if (i not in col) and (i - k not in diag45) and (i + k not in diag135):
                    nb += 1
                    if randint(1, nb) == 1:
                        j = i
            if nb > 0:
                x[k] = j
                self.__trycnt += 1
                col.add(j)
                diag45.add(j - k)
                diag135.add(j + k)
                k += 1
            if nb == 0 or k == stepVegas:
                if nb > 0:
                    return self.backtrace(k, x, n)
                else:
                    return False


    def test(self, k, x):
        for i in range(0, k):
            if x[i] == x[k] or abs(x[k] - x[i]) == abs(k - i):
                return False
        return True


    def backtrace(self, start, x, n):
        if start >= n:
            return True
        for i in range(0, n):
            x[start] = i
            if self.test(start, x):
                self.__trycnt += 1
                if self.backtrace(start + 1, x, n):
                    return True
        return False

    def search(self, stepV, x, n):
        self.__trycnt = 0
        while 1:
```

```
            if self.queensLV(stepV, x, n):
                return self.__trycnt

if __name__ == '__main__':
    total = 100
    Qn = Queen()
    # queen numbers
    for n in range(12, 21):
        # stepvegas choice
        print('queen number:',n)
        beststep = 0
        min = 0
        for stepVegas in range(1, n+1):
            # avg
            sum = 0
            for _ in range(total):
                x = [-1 for _ in range(n)]
                sum += Qn.search(stepVegas, x, n)
            if beststep == 0 or sum < min:
                min = sum
                beststep = stepVegas
        print('n=',n,'beststep=',beststep,'nodes=',min/total)
```

- Ex.10
  正确率为99% 结果:
  total prime numbers: 1204
  Millrab result: 1205
  accuracy: 0.9991701244813278

```
# code
from math import sqrt, log10
from random import randint, random

class Millrab(object):
    def __init__(self):
        self.__mr_prime = set()
        self.__prime = set()

    def btest(self, a, n):
        s = 0
        t = n - 1
        while t % 2 == 0:
            s = s + 1
            t = t // 2
        x = a ** t % n
        if x == 1 or x == n - 1:
            return True
        for i in range(1, s):
            x = x ** 2 % n
            if x == n - 1:
                return True
        return False

    def millrab(self,n):
        a = randint(2, n-2)
        return self.btest(a, n)

    def repeatmillrab(self, n, k):
        for _ in range(1, k+1):
            if self.millrab(n) == False:
                return False
        return True

    def mrgetprime(self):
        self.__mr_prime.clear()
        n = 5
        while True:
            k = int(log10(n))
            if 10000 >= n >= 100 and self.repeatmillrab(n,  k):
                self.__mr_prime.add(n)
            n += 2
            if n > 10000:
                break

    def getprime(self):
        self.__prime.clear()
        for i in range(100, 10001):
            prime = True
            for j in range(2, int(sqrt(i)) + 1):
                if i % j == 0:
                    prime = False
                    break
            if prime:
```

```
                self.__prime.add(i)

    def printset(self):
        return self.__mr_prime, self.__prime


if __name__ == '__main__':
    Mr = Millrab()
    Mr.getprime()
    Mr.mrgetprime()
    s1 = Mr.printset()[0]
    s2 = Mr.printset()[1]
    print("total prime numbers:", len(s2))
    print("Millrab result:", len(s1))
    print("accuracy:", len(s1.intersection(s2))/len(s1))
```