Adam Schaible

10/22/19

CS 4050

Programming Assignment #3

1. Computing π probablistically. Pi.java was written to calculate the value of π. The radius of the circle was half of the length of the square. For the coordinates that were randomly generated for the dart within the square had a radius that was equal to or less then the radius of the circle then the dart was considered to be within the radius of the circle. Π was calculated to be 4\*(number of darts inside the circle/number of darts that were inside the boundaries of the square. Below is the results for the estimation of π based on the number of darts thrown.

|  |  |
| --- | --- |
| Number of Darts Thrown | Result for π |
| 1,000 | 3.132 |
| 10,000 | 3.1612 |
| 100,000 | 3.13368 |
| 1,000,000 | 3.140864 |
| 100,000,000 | 3.1415975 |

As the number of darts thrown at random coordinates within the square increased the estimated value of π approached the expected value of π, which is 3.14159… according to the TI-83 calculator, so the greater the number of darts(random coordinate samples) that were used the more accurate the estimated value of π was. So, as a result using a large number of darts is a very accurate way to calculate π.

2. Testing for prime numbers. In order to test for prime numbers I wrote and ran the TestingForPrimeNumbers.java. The testing for prime numbers checked to see if a number was actually a composite number and if it was discovered that it was a composite number then by extension it could not be a prime number. For the composite numbers that I ran in the program included even numbers, numbers whose final digit is a 5, numbers whose sum of the digits is divisible by 3, numbers constructed as the product of two positive integers, randomly generated numbers that were divisible by some value that was divisible by a number that was greater than 1 and 40 of each type of composite number was randomly generated. The total number of composite numbers that was tested was 200. A number is considered prime(P) if P%n ≠ 0 for all values of 1<n<P, and in the program numbers(n) were randomly generated to test if the composite%n gave a result that was 0 then a tested composite has been verified as a composite. In the results below k represents the number of random values that the composite was tested against. The random values generated are integers.

|  |  |
| --- | --- |
| K # of random values tested against the composite | Percent of composite numbers detected |
| 10 | 13.500001% |
| 100 | 56.5% |
| 1,000 | 98.5% |
| 10,000 | 100% |

The trend in the results shows that as the number of random values that were used to test the composite increased so did the detection of the composite numbers. At K = 10,000 a 100% of all composite numbers had been detected and are detectable, but also since they were all proven to be composite numbers none of the numbers could be prime.

3. For searching an array I created an array of 1000 integers. A value in a slot in the array was randomly chosen as the value to search for and once chosen the program would then randomly pick array slots and get the average number of comparisons that a program had to do in order to find the value. Each search was restricted to up to 5000 guesses. My program SearchingAnArray.java performed a 100 searches and averaged the number of comparisons that had to be made to find a value, which my program returned that the average number of random guesses to find the array slot with a known value is 871.94. If on average one would have to search 500 slots in an array that contained 1000 using a linear search then an average search of 871.94 would indicate that randomly looking for a value in an array will require one to do far more searches than a linear search in order to find a value.

4. With the dart throwing method of integration I constructed a rectangular around the function and randomly picked coordinates within the rectangle and found the percentage of coordinates that landed within the area under of the function and then multiplied that by the area of the rectangle to get the integral of the function. For the mean of values at random locations took random x-values and used them to calculate the y-values and then computed the average y-value and multiplied that by the interval’s width to get the net area under the function. Both of these methods for calculating the integration of a function were compared to the trapezoidal method of integration in terms of both run time and accuracy and on the next page is the results of the comparisons using various samples sizes. The code that I wrote to implement this is MonteCarloIntegration.java.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **All functions were integrated from x=-3 to x=3** | | |  |  |  |  |
| Sample size indicates the number of darts in **dart throwing**, the number of random x values used in the **Mean of values at random locations** and the number of y-values that were calculated in the **Trapezoid Method** | | | | | | |
| **Integration Method** | **Function** | **Run Time in milliseconds** | **approximate integral** | **actual integral according to TI-83** | **Sample Size** | **actual integral – approximate integral** |
| dart throwing | x3 | 3,782,676 | -3.564001 | 0 | 1,000 | 3.564001 |
| mean of values | x3 | 407,523 | -4.174524 | 0 | 1,000 | 4.174524 |
| trapezoidal | x3 | 380,284 | 0.000000 | 0 | 1,000 | 0.000000 |
|  |  |  |  |  |  |  |
| dart throwing | x3 | 8,923,372,252 | 0.003153 | 0 | 100,000,000 | -0.003153 |
| mean of values | x3 | 3,860,071,777 | 0.002437 | 0 | 100,000,000 | -0.002437 |
| trapezoidal | x3 | 1,518,610,743 | 0.000000 | 0 | 100,000,000 | 0.000000 |
|  |  |  |  |  |  |  |
| dart throwing | x2-x3 | 2,770,236 | 20.412003 | 18 | 1,000 | -2.412003 |
| mean of values | x2-x3 | 357,998 | 15.418985 | 18 | 1,000 | 2.581015 |
| trapezoidal | x2-x3 | 754,554 | 18.000036 | 18 | 1,000 | -0.000036 |
|  |  |  |  |  |  |  |
| dart throwing | x2-x3 | 7,832,404,438 | 18.001621 | 18 | 100,000,000 | -0.001621 |
| mean of values | x2-x3 | 3,520,080,585 | 17.996500 | 18 | 100,000,000 | 0.003500 |
| trapezoidal | x2-x3 | 1,952,000,382 | 18.000000 | 18 | 100,000,000 | 0.000000 |
|  |  |  |  |  |  |  |
| dart throwing | -absolute of x | 4,289,603 | -8.568000 | -9 | 1,000 | -0.432000 |
| mean of values | -absolute of x | 327,929 | -9.356737 | -9 | 1,000 | 0.356737 |
| trapezoidal | -absolute of x | 366,841 | -9.000009 | -9 | 1,000 | 0.000009 |
|  |  |  |  |  |  |  |
| dart throwing | -absolute of x | 7,691,829,201 | -9.001370 | -9 | 100,000,000 | 0.001370 |
| mean of values | -absolute of x | 3,816,808,551 | -9.000067 | -9 | 100,000,000 | 0.000067 |
| trapezoidal | -absolute of x | 1,964,268,520 | -9.000000 | -9 | 100,000,000 | 0.000000 |

For small samples that were of size 1,000 trapezoidal was always the most accurate for calculating integrals. For small samples of 1,000 the mean of values was the fastest for 2 out of 3 functions and only for one equation was fastest for trapezoidal. For the larger sample size of 100,000,000 the trapezoidal method was always both the most accurate in calculating the integral and the fastest for the equations tested.

5. 8 Queens Problem – I tested the running times of first placing 0 to 8 queens on the board randomly and followed by using backtracking to place the remaining queens. The java program that I wrote to test this is named QueensTest.java. The results of this are below and additional sources used for writing this code are on the next page.

|  |  |
| --- | --- |
| Number of Queens first glued to board before backtracking | Time in milliseconds needed to generate solutions |
| 0 | 380,212,145 |
| 1 | 1,916,631 |
| 2 | 53,417 |
| 3 | 21,933 |
| 4 | 22,994 |
| 5 | 40,328 |
| 6 | 334,650 |
| 7 | 1,881,963 |
| 8 | 35,549,652 |

The result says that the worst way to solve the 8 queens problem is by attempting to randomly place glue down all queens, which takes over 10 times longer than any other solution. The 2nd slowest method to solve the 8 queens problem is to backtrack all 8 queens. The method that solved the 8 queens problem the fastest is to first randomly glue down 3 queens and let backtracking place the remaining queens.

Sources used in solving 8 queens method:

Admin. “Print All Possible Solutions to N Queens Problem.” *Techie Delight*, 23 July 2018, https://www.techiedelight.com/print-possible-solutions-n-queens-problem/.

“N Queen Problem: Backtracking-3.” *GeeksforGeeks*, 16 Oct. 2019, https://www.geeksforgeeks.org/n-queen-problem-backtracking-3/.