OPERATING SYSTEMS

THREADS ASSIGNMENT DOCUMENTATION

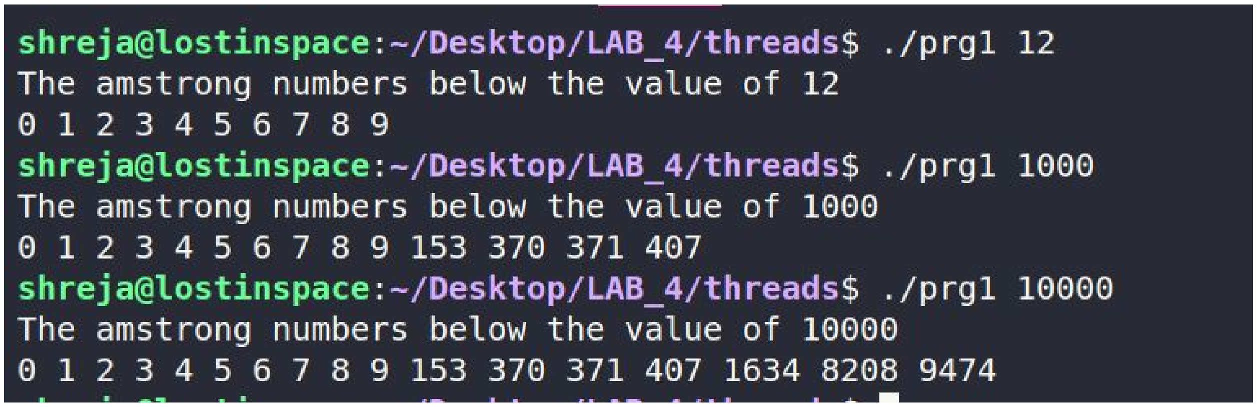
R SHREJA
COE18B043

NOTE: This document contains only the output of the codes. The codes with comments are attached in the classroom.

PROGRAM_1

Generate Armstrong number generation within a range.

OUTPUT:



```
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg1 12
The amstrong numbers below the value of 12
0 1 2 3 4 5 6 7 8 9
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg1 1000
The amstrong numbers below the value of 1000
0 1 2 3 4 5 6 7 8 9 153 370 371 407
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg1 10000
The amstrong numbers below the value of 10000
0 1 2 3 4 5 6 7 8 9 153 370 371 407 1634 8208 9474
```

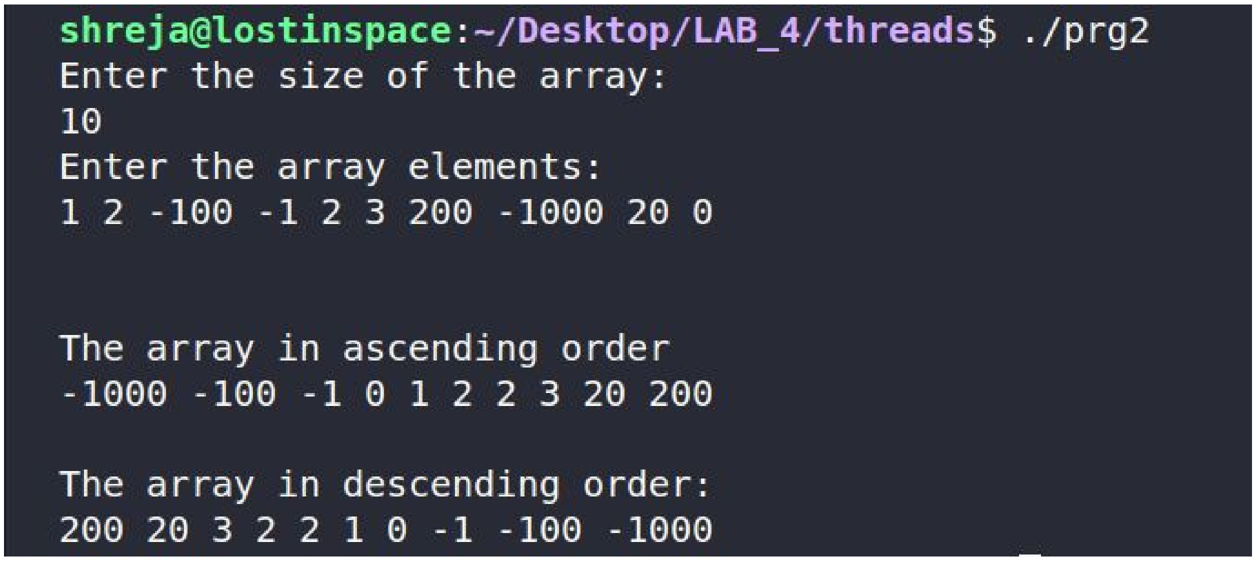
EXPLANATION:

The range is taken as the input in the executable itself then it range number of threads are created and each thread calculates if the number is an Armstrong number and outputs only if it an Armstrong number in the runner code itself.

PROGRAM_2

Ascending Order sort and Descending order sort.

OUTPUT:



```
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg2
Enter the size of the array:
10
Enter the array elements:
1 2 -100 -1 2 3 200 -1000 20 0


The array in ascending order
-1000 -100 -1 0 1 2 2 3 20 200

The array in descending order:
200 20 3 2 2 1 0 -1 -100 -1000
```

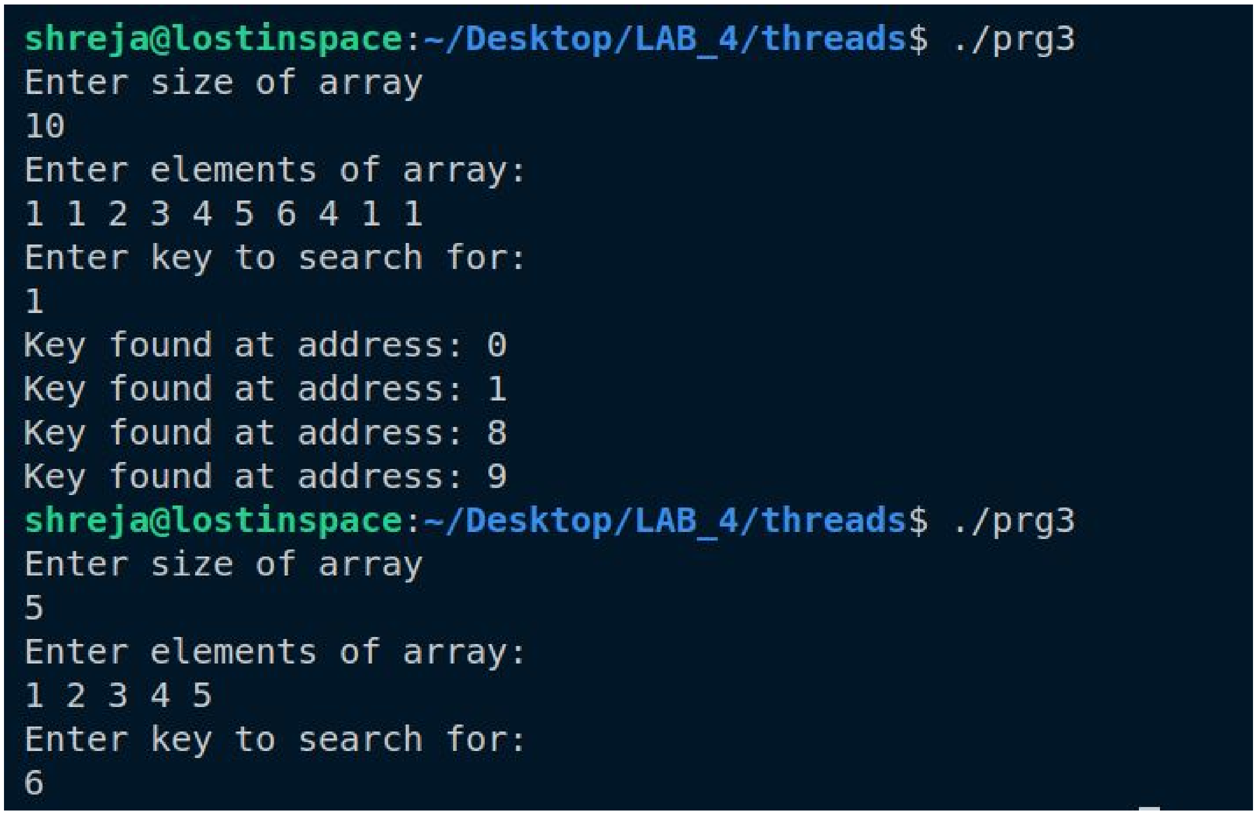
EXPLANATION:

The user prompts for the input size and consequently the input array and then this is copied into another array both these independent arrays are fed into 2 runner codes one for ascending sort and another for descending sort. Here both the arrays and printed out in the runner codes itself and the reason we created 2 arrays is that using one array is going give a wrong output as both ascd and dscd sort happen at the same time.

PROGRAM_3

Implement a multithreaded version of binary search. By default, you can implement a search for the first occurrence and later extend to support multiple occurrence (duplicated elements search as well)

OUTPUT:



```
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg3
Enter size of array
10
Enter elements of array:
1 1 2 3 4 5 6 4 1 1
Enter key to search for:
1
Key found at address: 0
Key found at address: 1
Key found at address: 8
Key found at address: 9
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg3
Enter size of array
5
Enter elements of array:
1 2 3 4 5
Enter key to search for:
6
```

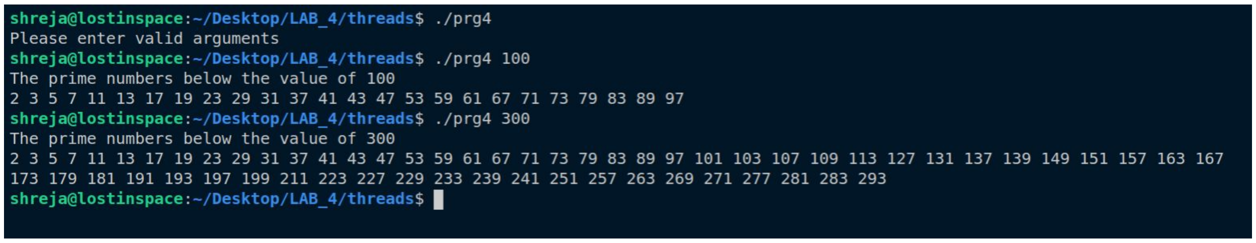
EXPLANATION:

Here the every time the array splits its assigned to a new thread the program is just like merge sort and instead of sorting we search for the key.

PROGRAM_4:

Generation of Prime Numbers up to a limit supplied as command Line Parameter.

OUTPUT:



```
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg4
Please enter valid arguments
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg4 100
The prime numbers below the value of 100
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg4 300
The prime numbers below the value of 300
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 163 167
173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293
shreja@lostinspace:~/Desktop/LAB_4/threads$ 
```

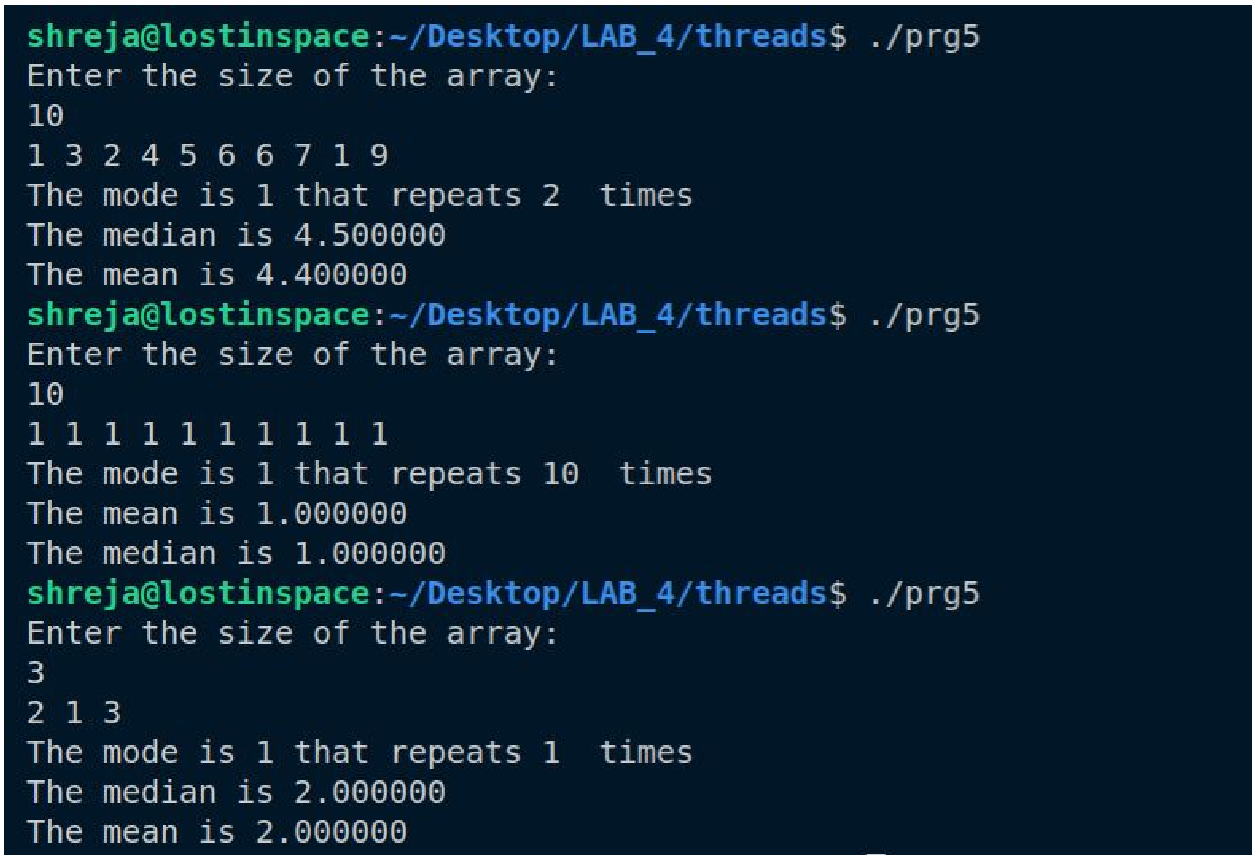
EXPLANATION:

The input of the limit is taken in the command line itself and limit number of threads are created and each thread finds if the nth number is prime and not and prints it out if its prime.

PROGRAM_5

Computation of Mean, Median, Mode for an array of integers.

OUTPUT:



```
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg5
Enter the size of the array:
10
1 3 2 4 5 6 6 7 1 9
The mode is 1 that repeats 2  times
The median is 4.500000
The mean is 4.400000
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg5
Enter the size of the array:
10
1 1 1 1 1 1 1 1 1 1
The mode is 1 that repeats 10  times
The mean is 1.000000
The median is 1.000000
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg5
Enter the size of the array:
3
2 1 3
The mode is 1 that repeats 1  times
The median is 2.000000
The mean is 2.000000
```

EXPLANATION:

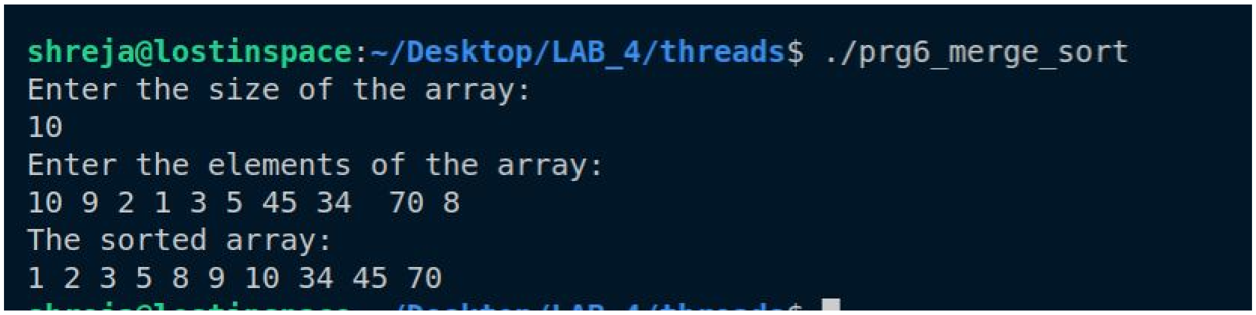
The mean, median and mode are calculated in three different threads using just one copy of the input sorted array.

PROGRAM_6

Implement Merge Sort and Quick Sort in a multithreaded fashion.

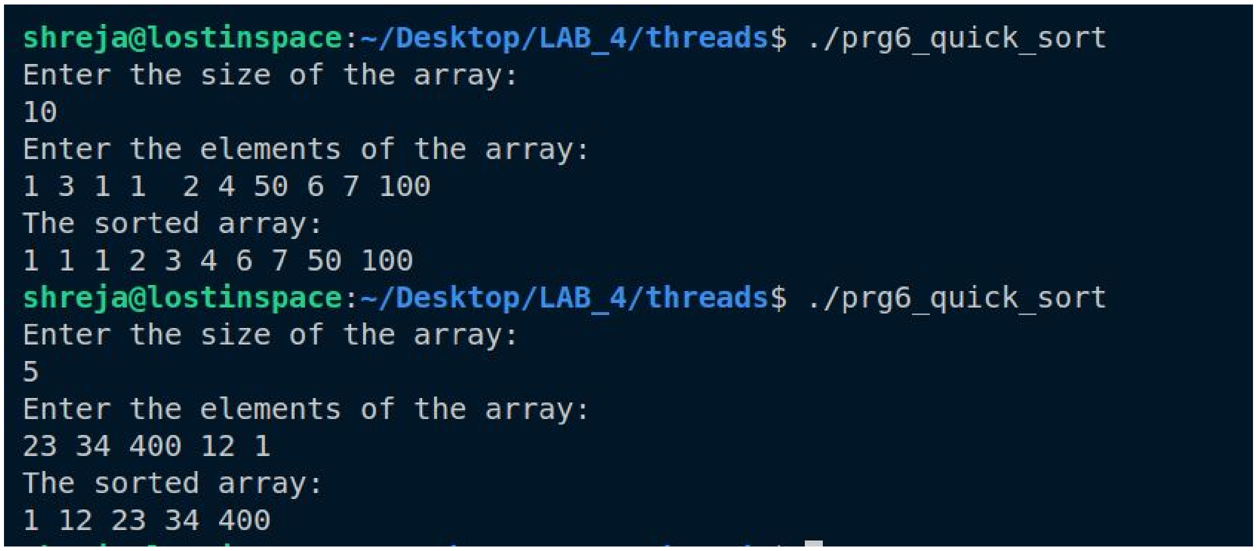
OUTPUT:

Merge sort



```
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg6_merge_sort
Enter the size of the array:
10
Enter the elements of the array:
10 9 2 1 3 5 45 34  70 8
The sorted array:
1 2 3 5 8 9 10 34 45 70
```

Quicksort



```
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg6_quick_sort
Enter the size of the array:
10
Enter the elements of the array:
1 3 1 1  2 4 50 6 7 100
The sorted array:
1 1 1 2 3 4 6 7 50 100
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg6_quick_sort
Enter the size of the array:
5
Enter the elements of the array:
23 34 400 12 1
The sorted array:
1 12 23 34 400
```

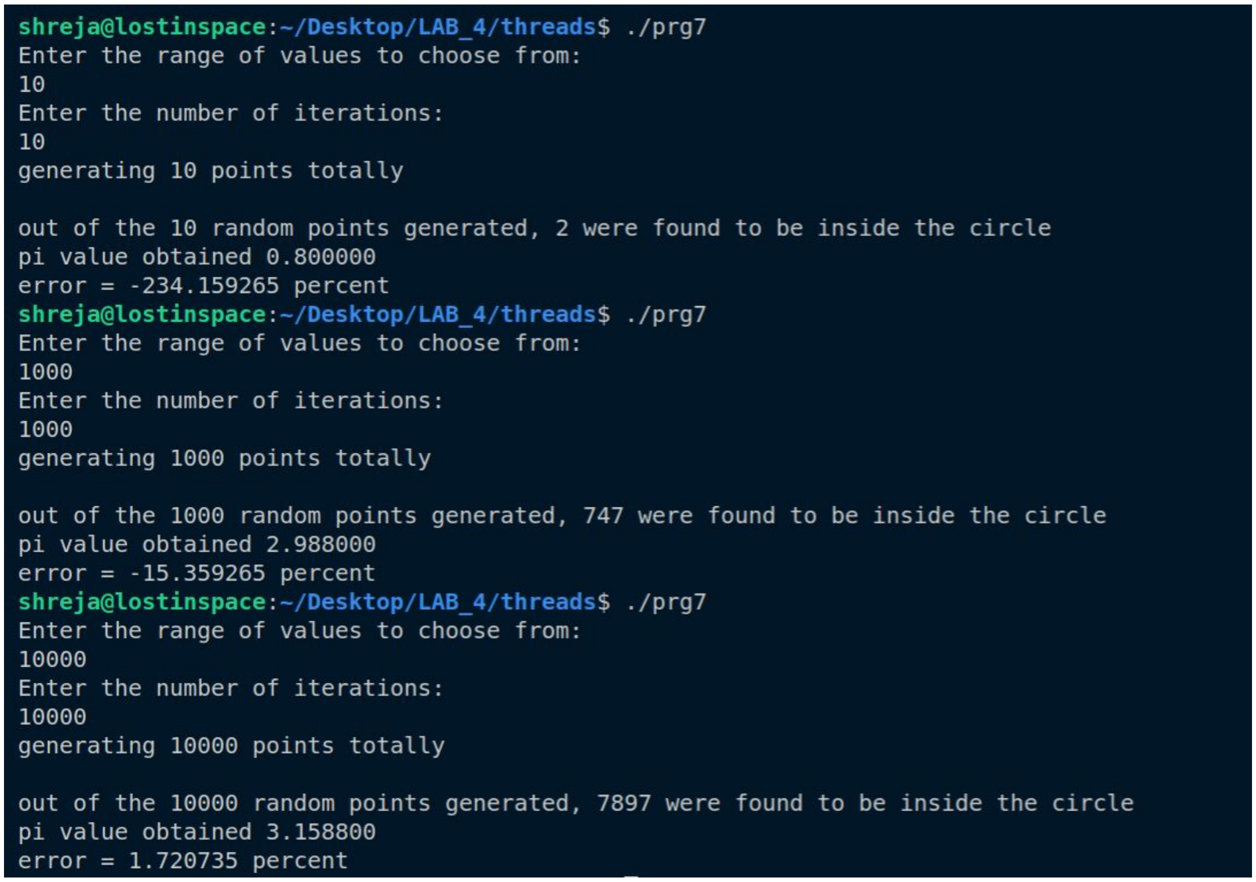
EXPLANATION:

In both the sorting algorithms when there’s a split in the array it is assigned to a new thread.

PROGRAM_7

Estimation of PI Value using Monte Carlo simulation technique (refer the internet for the method..) using threads.

OUTPUT:



```
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg7
Enter the range of values to choose from:
10
Enter the number of iterations:
10
generating 10 points totally

out of the 10 random points generated, 2 were found to be inside the circle
pi value obtained 0.800000
error = -234.159265 percent
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg7
Enter the range of values to choose from:
1000
Enter the number of iterations:
1000
generating 1000 points totally

out of the 1000 random points generated, 747 were found to be inside the circle
pi value obtained 2.988000
error = -15.359265 percent
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg7
Enter the range of values to choose from:
10000
Enter the number of iterations:
10000
generating 10000 points totally

out of the 10000 random points generated, 7897 were found to be inside the circle
pi value obtained 3.158800
error = 1.720735 percent
```

EXPLANATION:

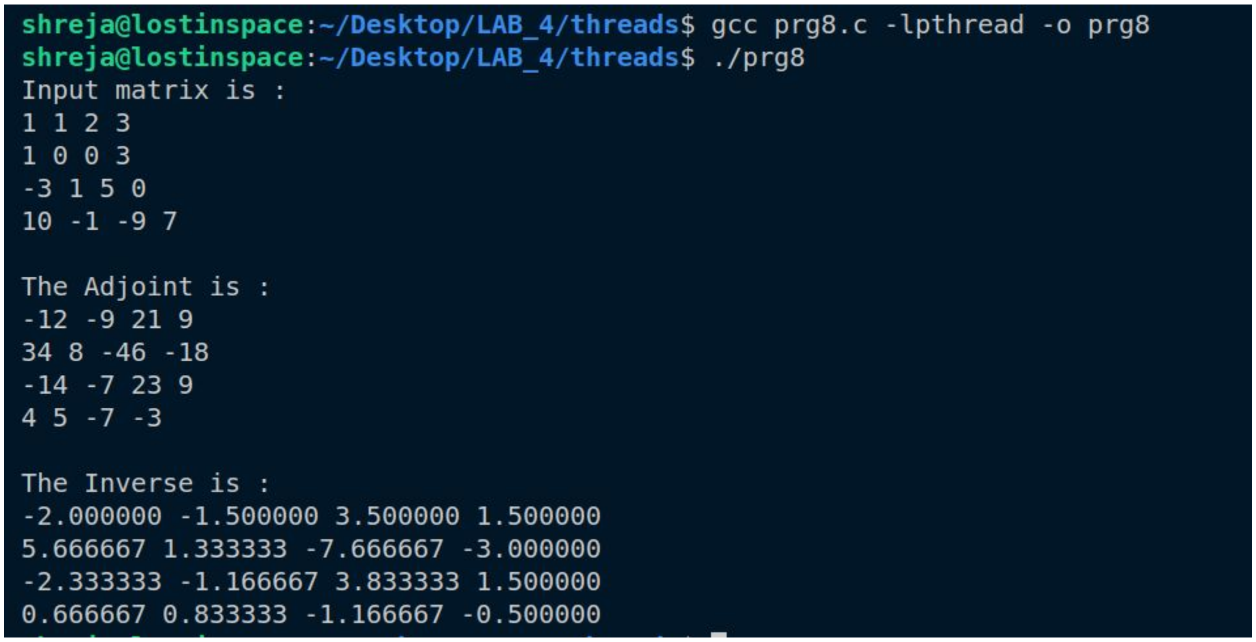
The idea is to simulate random (x, y) points in a 2-D plane with domain as a square of side 1 unit. Imagine a circle inside the same domain with the same diameter and inscribed into the square. We then calculate the ratio of number points that lied inside the circle and the total number of generated points. Here the multi-threading happens with the number of iterations.

OPTIONAL

PROGRAM_8

Computation of a Matrix Inverse using Determinant, Cofactor threads, etc.

OUTPUT:



```
shreja@lostinspace:~/Desktop/LAB_4/threads$ gcc prg8.c -lpthread -o prg8
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg8
Input matrix is :
1 1 2 3
1 0 0 3
-3 1 5 0
10 -1 -9 7

The Adjoint is :
-12 -9 21 9
34 8 -46 -18
-14 -7 23 9
4 5 -7 -3

The Inverse is :
-2.000000 -1.500000 3.500000 1.500000
5.666667 1.333333 -7.666667 -3.000000
-2.333333 -1.166667 3.833333 1.500000
0.666667 0.833333 -1.166667 -0.500000
```

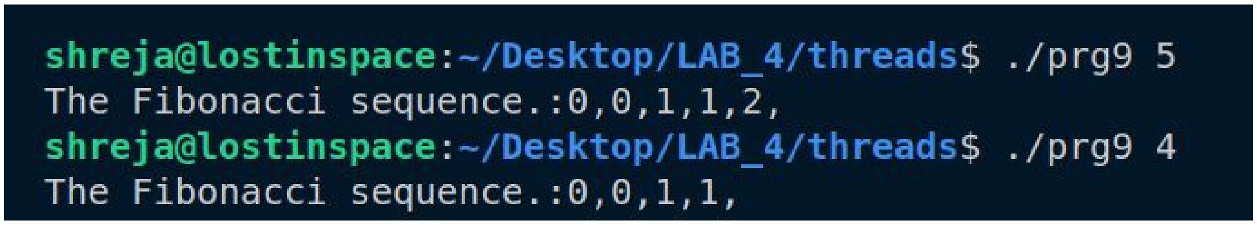
EXPLANATION:

To compute the inverse of a matrix we need to find adjoint of the cofactor matrix so the multithreading is done for finding the cofactor matrix.Ie for an n*n matrix n*n threads are created.

PROGRAM_9

Read upon efficient ways of parallelizing the generation of Fibonacci series and apply the logic in a multithreaded fashion to contribute a faster version of fib series generation.

OUTPUT:



```
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg9 5
The Fibonacci sequence.:0,0,1,1,2,
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg9 4
The Fibonacci sequence.:0,0,1,1,
```

EXPLANATION:

Fibonacci series depends on its previous 2 elements. Hence whatever algo is used, to calculate the nth element, its previous elements have to be calculated. So, the time complexity can never be below O(n).

For example, to calculate fib(7), we need to find fib(6), fib(5), ... fib(2), fib(1). To calculate fib(6), we need to find fib(5), fib(4), ... fib(2), fib(1). Hence finding all these in a linear sequential manner would lead to a time complexity of about O(2n).

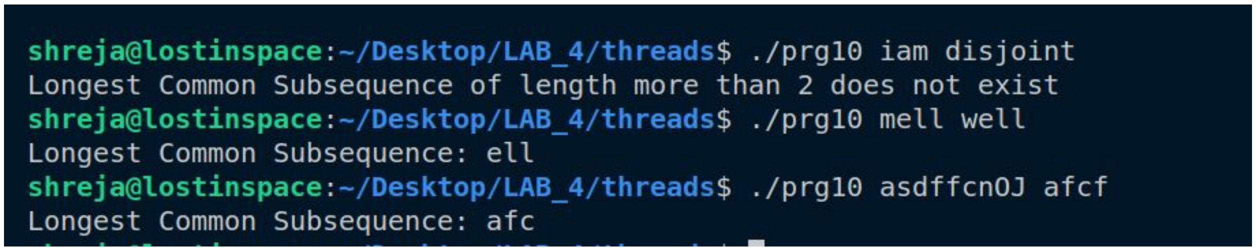
But with parallelization or by calculating the series till n from a bottom-up manner, we can get the time

complexity down to O(n). Using threading or forking will not improve the theoretical complexity of the algorithm. Having said all this I have implemented an algorithm where fibo(n) is calculated in n threads but the time of the creation of threads exceeds the fibo calculation that leads to the loss of data most of the times.

PROGRAM_10

Longest common subsequence generation problem using threads.

OUTPUT:



```
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg10 iam disjoint
Longest Common Subsequence of length more than 2 does not exist
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg10 mell well
Longest Common Subsequence: ell
shreja@lostinspace:~/Desktop/LAB_4/threads$ ./prg10 asdffcnOJ afcf
Longest Common Subsequence: afc
```

EXPLANATION:

The program can be viewed as two parts. The first part finds all the subsequences of the first string. The second part checks if any of those subsequences matches with any subsequence of the second string. In this program, I have multithreaded the second part.