# Multithreading Assignment

COM301-P Assignment - 6

G S Santhosh Raghul
COE18B045

I have put only the outputs in this document. Please check the .c files for code. There are comments in the .c files for more clarity. I have also included the gcc command in the screenshots to indicate which file(s) correspond to which question.

Thank you!

## 1. Generate Armstrong number generation within a range.

```
santi@edith:~/OS/L6$ gcc armstrong.c -pthread -o armstrong
santi@edith:~/OS/L6$ ./armstrong 1000
Sum is calculated in the thread and checking if sum == number is done in the main process.
Armstrong numbers from 1 to 1000 are :
1
2
3
4
5
6
7
8
9
153
370
371
407
santi@edith:~/OS/L6$
```

## 2. Ascending Order sort and Descending order sort.

```
santi@edith:~/OS/L6$ gcc asc_desc_sort.c -pthread -o sort
santi@edith:~/OS/L6$ ./sort
Enter no of elements : 20
Random array :13 68 76 71 96 55 60 43 9 96 52 99 90 44 39 19 45 73 67 78
Array after asc sorting : 9 13 19 39 43 44 45 52 55 60 67 68 71 73 76 78 90 96 96 99
Array after desc sorting : 99 96 96 90 78 76 73 71 68 67 60 55 52 45 44 43 39 19 13 9
santi@edith:~/OS/L6$
```

## 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)

```
santi@edith:~/OS/L6$ gcc binsearch.c -pthread -o binsearch
santi@edith:~/OS/L6$ ./binsearch
Implementation of binary search.
Enter no of elements : 30
Random array :6 6 0 7 1 8 2 4 3 9 4 3 9 1 3 5 2 7 1 3 6 0 5 2 5 9 9 5 9 8
Enter element to be searched : 3
3 found at index 14
3 found at index 8
3 found at index 11
3 found at index 19
santi@edith:~/OS/L6$
```

## 4.Generation of Prime Numbers upto a limit supplied as Command Line Parameter

```
santi@edith:~/OS/L6$ gcc prime_numbers.c -lm -pthread -o prime
santi@edith:~/OS/L6$ ./prime 500
Prime numbers from 1 to 500 are :
3 5 7 11 13 17 19 23 29 31 37 41 43 47 59 61 53 67 71 73 79 83 89
97 101 103 109 113 107 127 131 137 139 157 163 149 167 151 173 179
 181 191 193 197 199 223 227 211 233 229 239 241 251 257 263 269 2
71 277 281 283 293 307 311 313 317 331 337 347 349 353 359 373 367
 379 383 397 401 389 409 419 421 433 431 439 443 449 467 463 461 4
79 487 491 499 457
santi@edith:~/OS/L6$
```

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

```
santi@edith:~/OS/L6$ gcc mean_median_mode.c -pthread -o stats
santi@edith:~/OS/L6$ ./stats
Enter no of elements : 15
Random array :3 30 53 24 45 56 55 57 52 80 30 43 91 46 74
median = 52
mean = 49.266666
mode = 30 (displaying one of the modes if there are multiple modes)
santi@edith:~/OS/L6$
```

## 6. Implement Merge Sort and Quick Sort in a multithreaded fashion

```
santi@edith:~/OS/L6$ gcc merge_sort.c -pthread -o merge_sort
santi@edith:~/OS/L6$ ./merge_sort 20

Original array :
  69 7785 8373 2904 5433 4764 7781  972 2798  174 8866 7228 3320 4184 2532
  965 9328  717 5894 9404
Serial sorted :
  69  174  717  965  972 2532 2798 2904 3320 4184 4764 5433 5894 7228 7781
 7785 8373 8866 9328 9404
Parallel sorted :
  69  174  717  965  972 2532 2798 2904 3320 4184 4764 5433 5894 7228 7781
 7785 8373 8866 9328 9404

time taken for:
parallel: 0.009554s
serial:   0.000015s
santi@edith:~/OS/L6$
```

```
santi@edith:~/OS/L6$ gcc quick_sort.c -pthread -o quick_sort
santi@edith:~/OS/L6$ ./quick_sort 20

Original array :
6834 6967  252 1296 2141 8872 9631 8584 7583 3201 3965 9955 2921 5784 3375 1669
6411 1464 2496 5644
Serial sorted :
 252 1296 1464 1669 2141 2496 2921 3201 3375 3965 5644 5784 6411 6834 6967 7583
8584 8872 9631 9955
Parallel sorted :
 252 1296 1464 1669 2141 2496 2921 3201 3375 3965 5644 5784 6411 6834 6967 7583
8584 8872 9631 9955

time taken for:
parallel: 0.010177s
serial:   0.000009s
santi@edith:~/OS/L6$
```

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

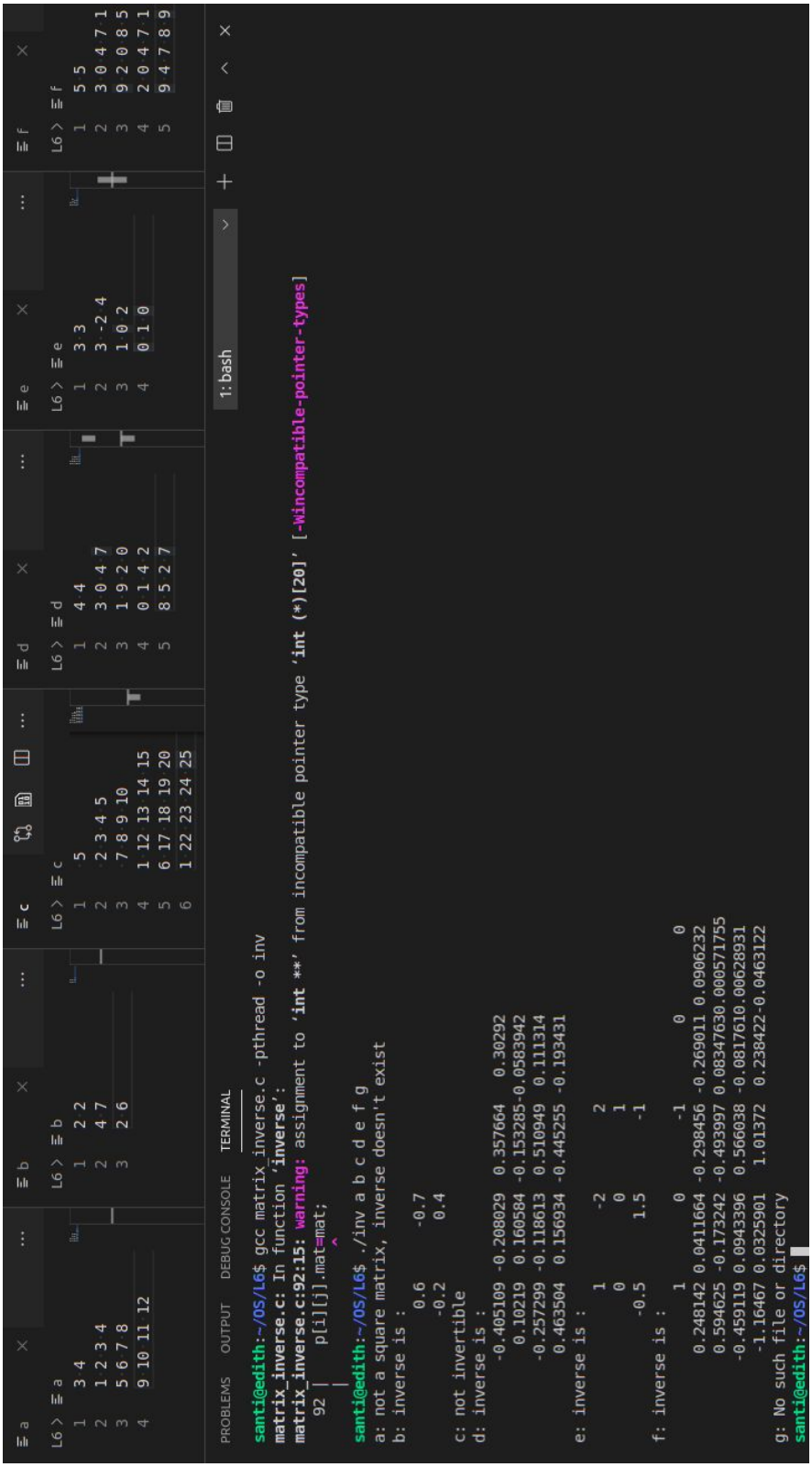
```
santi@edith:~/OS/L6$ gcc pi_monte_carlo.c -pthread -o pi
santi@edith:~/OS/L6$ ./pi 100 10000
parameters are: 100, 10000
generating 10000 random points in the range 0 to 100

number of CPU cores was found to be 4 so creating 4 threads
        thread 1 will generate 2500 random points
        thread 2 will generate 2500 random points
        thread 3 will generate 2500 random points
        thread 4 will generate 2500 random points
generating 10000 points totally

out of the 10000 random points generated, 7762 were found to be inside
the circle
pi value obtained = 4*7762/10000 = 3.104800
error = -3.679265 %
santi@edith:~/OS/L6$
```

```
santi@edith:~/OS/L6$ ./pi 10 10
parameters are: 10, 10
generating 10 random points in the range 0 to 10

number of CPU cores was found to be 4 so creating 4 threads
        thread 1 will generate 3 random points
        thread 2 will generate 3 random points
        thread 3 will generate 2 random points
        thread 4 will generate 2 random points
generating 10 points totally

out of the 10 random points generated, 10 were found to be inside the
circle
pi value obtained = 4*10/10 = 4.000000
error = 85.840735 %
santi@edith:~/OS/L6$
```

```
santi@edith:~/OS/L6$ ./pi 100 10000
parameters are: 100, 10000
generating 10000 random points in the range 0 to 100

number of CPU cores was found to be 4 so creating 4 threads
        thread 1 will generate 2500 random points
        thread 2 will generate 2500 random points
        thread 3 will generate 2500 random points
        thread 4 will generate 2500 random points
generating 10000 points totally

out of the 10000 random points generated, 7787 were found to be inside
 the circle
pi value obtained = 4*7787/10000 = 3.114800
error = -2.679265 %
santi@edith:~/OS/L6$
```

```
santi@edith:~/OS/L6$ ./pi 10000 10000000
parameters are: 10000, 10000000
generating 10000000 random points in the range 0 to 10000

number of CPU cores was found to be 4 so creating 4 threads
        thread 1 will generate 2500000 random points
        thread 2 will generate 2500000 random points
        thread 3 will generate 2500000 random points
        thread 4 will generate 2500000 random points
generating 10000000 points totally

out of the 10000000 random points generated, 7853227 were found to be
inside the circle
pi value obtained = 4*7853227/10000000 = 3.141291
error = -0.030185 %
santi@edith:~/OS/L6$
```

# 8.Computation of a Matrix Inverse using Determinant, Cofactor threads,



# 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.

Here, the question talks about a "faster version" of fibonacci series generation using parallel processing. **I claim that there is no "faster version" than the dynamic programming approach of generating fibonacci series which incurs a time complexity of O(n).** Here is my argument :

The formal definition of the fibonacci numbers is as follows:

**Fibonacci numbers form a sequence called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1.**

**(i.e) $F_0 = 0$ , $F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for $n > 1$.**

Here, it is very clear that we need the value of $F_{n-1}$ and $F_{n-2}$ in order to find the value of $F_n$ . So, for an arbitrary number k, to display Fibonacci numbers until k, we need to calculate $F_{k-1}$ , $F_{k-2}$ , $F_{k-3}$ , ... , $F_{k-(k-1)} = F_1$ , $F_{k-k} = F_0$ . So, we have to make k+1 operations in total and each operation is of O(1). So, the total time complexity is O(k).

One might ask what about parallel processing. To answer this question, we have to note that each fibonacci value except $F_0$ and $F_1$ depend entirely on the previous terms. It is impossible to calculate the $k^{th}$ term unless we have already calculated the $(k-1)^{th}$ term and the $(k-2)^{th}$ term. So, applying parallel processes to calculate different fibonacci numbers will only result in more number of processes which wait for other processes and there won't truly be any parallel processing.

There are other approaches for finding the $n^{th}$ fib term in O(logn) time but the question talks about fibonacci series generation and not $n^{th}$ term generation and more over, those algorithms don't use parallel processing to optimize the $n^{th}$ term generation so I have not mentioned about those in this assignment.

So, my conclusion is that there is no way of parallelizing the generation of fibonacci series to make it faster since every term depends on the previous terms.

Because of what I've mentioned above, I have not attached any program with this question.

## 10. Longest common subsequence generation problem using threads.