OS Assignment 6

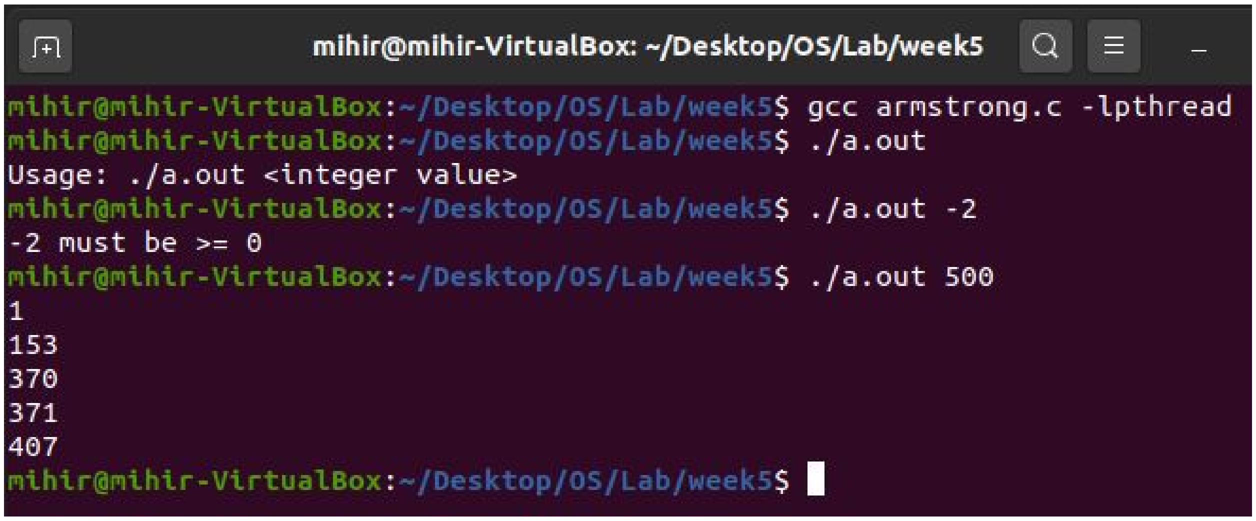
1. Generate Armstrong number generation within a range.

Logic: As mentioned in the question, digit extraction and cubing is done by the parent (main) process without using any threads. An array of size 2 has been maintained whose first element is the original number while the second element is the sum of the cube of its digits. A runner function has been created and called using the pthread_create() function call. The array created above is passed as an argument to our thread function (runner function). The thread checks whether the first and second element of the array are equal or not (i.e. whether the number is equal to the sum of cubes of its digits or not) and prints the number if they are equal.

Code:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *runner(void *param);

int main(int argc, char const *argv[])
{
    int n;
    pthread_t tid;
    pthread_attr_t attr;

    if (argc != 2)
    {
        fprintf(stderr, "Usage: ./a.out <integer value>\n");
        return -1;
    }

    n = atoi(argv[1]);

    if (n < 0)
    {
        fprintf(stderr, "%d must be >= 0\n", n);
        return -1;
    }
    for (int i = 1; i <= n; i++)
    {
        int *result = (int *)malloc(2*sizeof(int *));
        int num = i, d, cube = 0;

        while (num > 0)
        {
            d = num % 10;
            cube += d * d * d;
            num /= 10;
        }

        result[0] = i;
        result[1] = cube;

        pthread_attr_init(&attr);
        pthread_create(&tid, NULL, runner, (void *)result);
        pthread_join(tid, NULL);
        free(result);
    }

    return 0;
}

void *runner(void *param)
{
    int *result = (int *)param;

    if (result[0] == result[1])
        printf("%d\n", result[0]);

    pthread_exit(0);
}
```

Output ss:



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week5$ gcc armstrong.c -lpthread
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week5$ ./a.out
Usage: ./a.out <integer value>
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week5$ ./a.out -2
-2 must be >= 0
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week5$ ./a.out 500
1
153
370
371
407
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week5$
```

2. Ascending Order sort and Descending order sort.

Logic: The thread (runner function) is responsible for arranging the array elements in descending order while the parent process (main) arranges the elements in descending order.

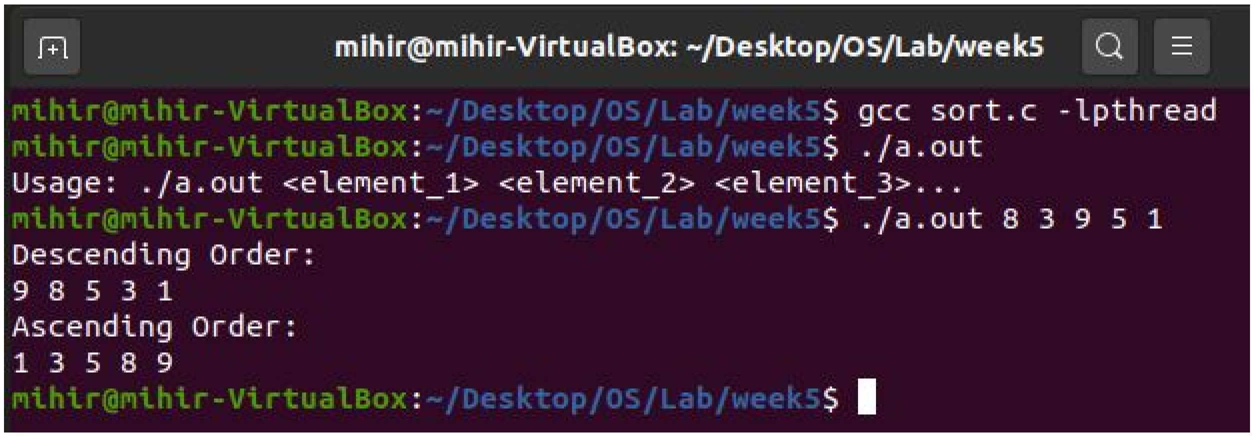
Code:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *runner(void *param);

int main(int argc, char const *argv[])
{
    int n;
    pthread_t tid;
    pthread_attr_t attr;

    if (argc < 2)
    {
          fprintf(stderr, "Usage: ./a.out <element_1> <element_2> <element_3>...\n");
```

```
            return -1;
    }

    n = argc;
    int *result = (int *)malloc((n+1) * sizeof(int *));
    result[0] = n;

    for(int i = 1; i < argc; i++)
            result[i] = atoi(argv[i]);

    pthread_attr_init(&attr);
    pthread_create(&tid, &attr, runner, (void *)result);
    pthread_join(tid, NULL);

    printf("Ascending Order: \n");

    for (int i = 1 ; i < n; i++)
    {
            for (int j = i + 1; j < n; ++j)
            {
                    if(result[j] < result[i])
                    {
                            int t = result[i];
                            result[i] = result[j];
                            result[j] = t;
                    }
            }
    }

    for(int i=1;i<n;i++)
            printf("%d ", result[i]);

    printf("\n");

    free(result);

    return 0;
}

void *runner(void *param)
```

```
{
    int *result = (int *)param;
    int n = result[0];

    printf("Descending Order: \n");

    for(int i = 1; i < n; i++)
    {
        for (int j = i + 1; j < n; ++j)
        {
            if(result[i] < result[j])
            {
                int t = result[i];
                result[i] = result[j];
                result[j] = t;
            }
        }
    }

    for(int i=1;i<n;i++)
        printf("%d ",result[i]);

    printf("\n");

    pthread_exit(0);

}
```

Output ss:



```
mihir@mihir-VirtualBox: ~/Desktop/OS/Lab/week5
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week5$ gcc sort.c -lpthread
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week5$ ./a.out
Usage: ./a.out <element_1> <element_2> <element_3>...
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week5$ ./a.out 8 3 9 5 1
Descending Order:
9 8 5 3 1
Ascending Order:
1 3 5 8 9
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week5$
```

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)

Logic: The thread (runner function0 is responsible for sorting and searching for the element in the second part of the array. While implementing binary search in the second part, beg = n/2, end = n. To tackle multiple occurrences of an element, the logic is, that since the array is sorted, all the duplicates of an element will be either towards the left of the element or towards the right of the element. The parent (main) function is responsible for doing the same (sorting and searching) in the second half of the array.

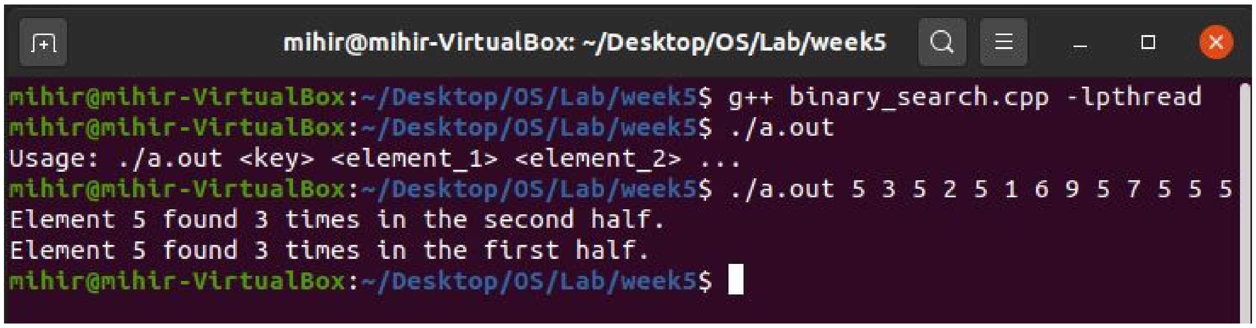
Code:
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>

using namespace std;

void *runner(void *param);

int main(int argc, char const *argv[])
{
   pthread_t tid;
   pthread_attr_t attr;

   if (argc < 3)
   {
         fprintf(stderr, "Usage: ./a.out <key> <element_1> <element_2> ...\n");
         return -1;
   }

   int key = atoi(argv[1]);
   int n = argc - 2;
   int *result = (int *)malloc(n * sizeof(int *));

   result[0] = n;
   result[1] = key;
```

```
for (int i = 2; i < argc; i++)
{
        result[i] = atoi(argv[i]);
}

pthread_attr_init(&attr);
pthread_create(&tid, &attr, runner, result);
pthread_join(tid, NULL);

for (int i = 0; i < n - 2; i++)
        result[i] = result[i+2];

sort(result, result+n/2);

int beg = 0;
int end = n/2-1;
int flag = 0;
int c = 0;

while (beg <= end)
{
        int mid = (beg+end) / 2;

        if (key == result[mid])
        {
                c++;

                for(int i = mid+1; i<= end; i++)
                {
                        if(result[i] == key)
                        c++;

                        else
                        break;
       }

                for(int i = mid-1; i >= beg; i--)
                {
                        if(result[i] == key)
                        c++;
```

```
                    else
                    break;
            }

                break;
            }

        else if (key < result[mid])
                end = mid - 1;

        else
                beg = mid + 1;
    }

    printf("Element %d found %d times in the first half.\n", key, c);

    return 0;
}

void *runner(void *param)
{
    int *result = (int *)param;
    int n = result[0], key = result[1];
    int c = 0;

    for (int i = 0; i < n - 2; i++)
        result[i] = result[i+2];

    sort(result+n/2, result+n);

    int beg = n/2;
    int end = n-1;
    int flag = 0;

    while (beg <= end)
    {
        int mid = (beg+end) / 2;

        if (key == result[mid])
```

```
         {
                c++;

                for(int i = mid+1; i<= end; i++)
                {
                        if(result[i] == key)
                        c++;

                        else
                        break;
        }

                for(int i = mid-1; i >= beg; i--)
                {
                        if(result[i] == key)
                        c++;

                        else
                        break;
        }

                break;
         }

         else if (key < result[mid])
                end = mid - 1;

         else
                beg = mid + 1;
    }

    printf("Element %d found %d times in the second half.\n", key, c);

    pthread_exit(0);
}
```

Output ss:



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week5$ g++ binary_search.cpp -lpthread
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week5$ ./a.out
Usage: ./a.out <key> <element_1> <element_2> ...
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week5$ ./a.out 5 3 5 2 5 1 6 9 5 7 5 5 5
Element 5 found 3 times in the second half.
Element 5 found 3 times in the first half.
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week5$
```

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

Logic: In the parent process, for each number between 2 to n, we create a new thread. All threads call to the same runner function which receives that number as a parameter and checks and prints the number if it is a prime number.

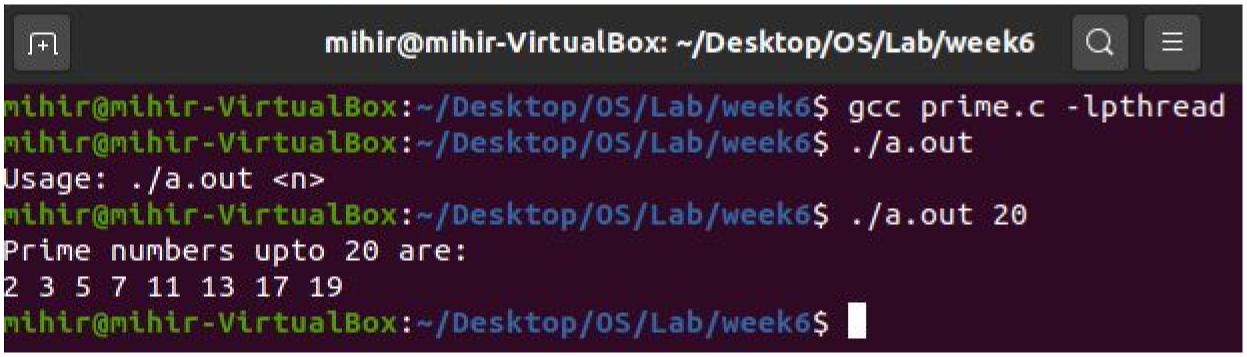
Code:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *runner(void *param);

int prime(int n)
{
   int c=0;
   for(int i=1;i<=n;i++)
   {
         if(n%i==0)
                c++;
   }

   if(c==2)
         return 1;
```

```
    else
            return 0;
}

int main(int argc, char const *argv[])
{
    pthread_t tid;
    pthread_attr_t attr;

    if (argc < 2)
    {
            fprintf(stderr, "Usage: ./a.out <n>\n");
            return -1;
    }

    int n = atoi(argv[1]);

    printf("Prime numbers upto %d are:\n", n);

    for (int i = 2; i <= n; i++)
    {
            int *nums = (int*)malloc(sizeof(int));
            nums[0] = i;
            pthread_attr_init(&attr);
            pthread_create(&tid, &attr, runner, nums);
            pthread_join(tid, NULL);
    }

    printf("\n");

    return 0;
}

void *runner(void *param)
{
    int *nums = (int *)param;
    int num = nums[0];

    if (prime(num))
```

```
            printf("%d ", num);

    pthread_exit(0);
}
```

Output ss:



```
mihir@mihir-VirtualBox: ~/Desktop/OS/Lab/week6
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week6$ gcc prime.c -lpthread
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week6$ ./a.out
Usage: ./a.out <n>
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week6$ ./a.out 20
Prime numbers upto 20 are:
2 3 5 7 11 13 17 19
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week6$ 
```

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

Logic: 3 separate runner functions are created, each given the task of calculating the mean, median and mode respectively. Inside the main function, an array is passed as a command line argument which is then passed on to the respective runner functions through the creation of 3 threads.

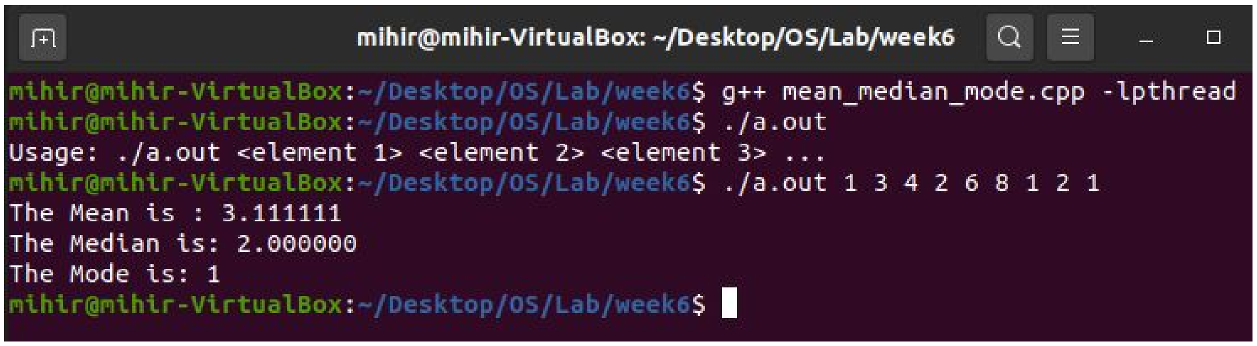
Code:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <bits/stdc++.h>

using namespace std;

void *runner1(void *param);
void *runner2(void *param);
void *runner3(void *param);
```

```
int main(int argc, char const *argv[])
{
    pthread_t tid1, tid2, tid3;
    pthread_attr_t attr1, attr2, attr3;

    if (argc < 2)
    {
        fprintf(stderr, "Usage: ./a.out <element 1> <element 2> <element 3> ...\n");
        return -1;
    }

    int n = argc;
    int *nums = (int *)malloc(n * sizeof(int *));
    nums[0] = n;

    for (int i = 1; i < argc; i++)
    {
        nums[i] = atoi(argv[i]);
    }

    pthread_attr_init(&attr1);
    pthread_create(&tid1, &attr1, runner1, nums);

    pthread_attr_init(&attr2);
    pthread_create(&tid2, &attr2, runner2, nums);

    pthread_attr_init(&attr3);
    pthread_create(&tid3, &attr3, runner3, nums);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    pthread_join(tid3, NULL);

    return 0;
}

void *runner1(void *param)
{
    int *nums = (int *)param;
    float n = nums[0];
    float sum = 0, mean;

    for (int i = 1; i < n; i++)
    {
            sum += nums[i];
    }

    mean = sum / (n-1);
    printf("The Mean is : %f\n", mean);

    pthread_exit(0);
}

void *runner2(void *param)
{
    int *nums = (int *)param;
    int n = nums[0];
    int *new_nums = (int *)malloc(n * sizeof(int *));
    float median;

    for (int i = 0; i < n-1; i++)
            new_nums[i] = nums[i+1];

    sort(new_nums, new_nums+n-1);

    if ((n-1) % 2 != 0)
            median = new_nums[(n-1)/2];

    else
            median = (new_nums[((n-1)/2) - 1] + new_nums[((n-1)/2)]) / 2.0;

    printf("The Median is: %f\n", median);

    pthread_exit(0);
}

void *runner3(void *param)
{
    int *a = (int *)param;
    int n = a[0];
    int *b = (int *)malloc(n * sizeof(int *));
    int mode, k = 0, c = 1, max = 0;

    for (int i = 0; i < n-1; i++)
            a[i] = a[i+1];

    n--;
    sort(a, a+n);

    for(int i = 0; i < n-1; i++)
      {
            mode = 0;

            for(int j = i+1; j < n; j++)
                    {
                            if(a[i]==a[j])
                    {
                            mode++;
                    }
                    }

            if((mode > max) && (mode != 0))
                    {
                            k = 0;
                            max = mode;
                            b[k] = a[i];
                            k++;
                    }

            else if(mode == max)
                    {
                            b[k] = a[i];
                            k++;
                    }
      }

            for(int i = 0; i < n; i++)
      {
            if(a[i] == b[i])
```

```
                c++;
        }

            if(c == n+1)
                printf("There is no mode\n");

            else
        {
            printf("The Mode is: ");

            for(int i = 0; i < k; i++)
                printf("%d ",b[i]);
        }

        printf("\n");

        pthread_exit(0);
}
```

Output ss:



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week6$ g++ mean_median_mode.cpp -lpthread
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week6$ ./a.out
Usage: ./a.out <element 1> <element 2> <element 3> ...
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week6$ ./a.out 1 3 4 2 6 8 1 2 1
The Mean is : 3.111111
The Median is: 2.000000
The Mode is: 1
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week6$
```

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

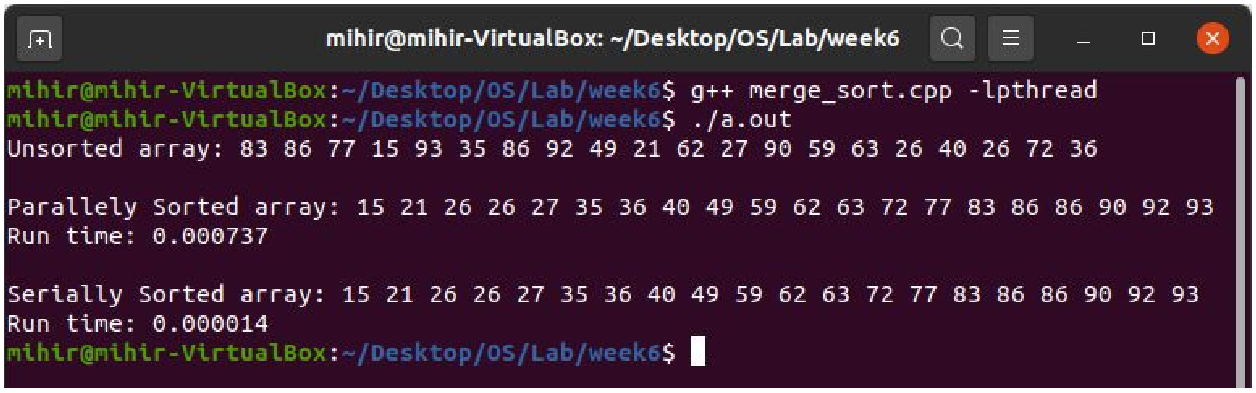
<u>MERGE SORT</u>

Logic: Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted subarrays into one. See the following C++ implementation for details.

Code:
```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#define MAX 20
#define THREAD_MAX 4

int a[MAX];
int part = 0;

void merge(int low, int mid, int high)
{
   int *left = new int[mid - low + 1];
   int *right = new int[high - mid];

   int n1 = mid - low + 1, n2 = high - mid, i, j;

   for (i = 0; i < n1; i++)
         left[i] = a[i + low];
   for (i = 0; i < n2; i++)
         right[i] = a[i + mid + 1];

   int k = low;
   i = j = 0;

   // merge left and right in ascending order
   while (i < n1 && j < n2)
   {
         if (left[i] <= right[j])
                  a[k++] = left[i++];
         else
                  a[k++] = right[j++];
   }

   while (i < n1)
         a[k++] = left[i++];
```

```
    while (j < n2)
            a[k++] = right[j++];
}

void merge_sort(int low, int high)
{
    int mid = low + (high - low) / 2;
    if (low < high)
    {
            merge_sort(low, mid);
            merge_sort(mid + 1, high);
            merge(low, mid, high);
    }
}

void *merge_sort(void *arg)
{
    int thread_part = part++;

    int low = thread_part * (MAX / 4);
    int high = (thread_part + 1) * (MAX / 4) - 1;

    int mid = low + (high - low) / 2;
    if (low < high)
    {
            merge_sort(low, mid);
            merge_sort(mid + 1, high);
            merge(low, mid, high);
    }
    pthread_exit(NULL);
}

int main()
{
    for (int i = 0; i < MAX; i++)
            a[i] = rand() % 100;

    printf("Unsorted array: ");
    for (int i = 0; i < MAX; i++)
            printf("%d ", a[i]);
    clock_t t1, t2;

    t1 = clock();
    pthread_t threads[THREAD_MAX];

    for (int i = 0; i < THREAD_MAX; i++)
        pthread_create(&threads[i], NULL, merge_sort, (void *)NULL);
    for (int i = 0; i < 4; i++)
        pthread_join(threads[i], NULL);

    merge(0, (MAX / 2 - 1) / 2, MAX / 2 - 1);
    merge(MAX / 2, MAX / 2 + (MAX - 1 - MAX / 2) / 2, MAX - 1);
    merge(0, (MAX - 1) / 2, MAX - 1);
    t2 = clock();
    printf("\n\nParallely Sorted array: ");
    for (int i = 0; i < MAX; i++)
        printf("%d ", a[i]);
    printf("\nRun time: %f\n", (t2 - t1) / (double)CLOCKS_PER_SEC);

    t1 = clock();
    merge_sort(0, MAX - 1);
    printf("\nSerially Sorted array: ");
    for (int i = 0; i < MAX; i++)
        printf("%d ", a[i]);
    t2 = clock();

    printf("\nRun time: %f\n", (t2 - t1) / (double)CLOCKS_PER_SEC);
    return 0;
}
```

Output ss:



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week6$ g++ merge_sort.cpp -lpthread
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week6$ ./a.out
Unsorted array: 83 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36

Parallely Sorted array: 15 21 26 26 27 35 36 40 49 59 62 63 72 77 83 86 86 90 92 93
Run time: 0.000737

Serially Sorted array: 15 21 26 26 27 35 36 40 49 59 62 63 72 77 83 86 86 90 92 93
Run time: 0.000014
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week6$
```

QUICK SORT

Logic: The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

```
low --> Starting index, high --> Ending index
quickSort(arr[], low, high)
{
        if (low < high)
        {
                /* pi is partitioning index, arr[pi] is now at right place */
                pi = partition(arr, low, high);

                quickSort(arr, low, pi - 1); // Before pi
                quickSort(arr, pi + 1, high); // After pi
        }
}
```

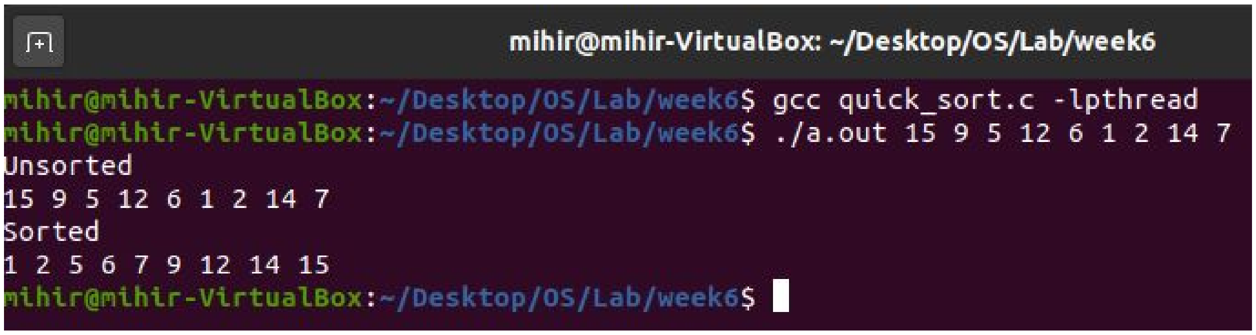
To implement in a multithreaded fashion a typedef struct is created for storing global variables. The recursive function is not thread implemented, but the pivot comparison function is implemented using threads. The depth of threads used is 4 in this case.

Code:

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <time.h>
#include <sys/time.h>

typedef struct qsort_starter
{
  int *arr;
  int low;
  int high;
  int depthOfThreadCreation;
} quickSort_parameters;
```

```
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}


int partition (int *arr, int low, int high, int pivot)
{
    int pivotValue = arr[pivot];
  swap(&arr[pivot],&arr[high]);
    int s = low;

    for (int i = low; i <high; i++)
    {
            if (arr[i] <= pivotValue)
            {
                        swap(&arr[i], &arr[s]);
           s++;
            }
    }
    swap(&arr[s], &arr[high]);
    return s;
}
void quickSort(int *arr, int low, int high)
{
    if (low < high)
    {
           int pivotPosition = low+ (high-low)/2;
           pivotPosition= partition(arr, low, high, pivotPosition);
            quickSort(arr, low, pivotPosition - 1);
            quickSort(arr, pivotPosition + 1, high);
    }
}
void concurrent_quickSort(int *arr, int low, int high, int depthOfThreadCreation);
void* worker_quickSort(void * initialValues){
  quickSort_parameters* parameters = initialValues;
  concurrent_quickSort(parameters->arr, parameters->low,
parameters->high,parameters->depthOfThreadCreation);
  return NULL;
}
void concurrent_quickSort(int *arr, int low, int high, int depthOfThreadCreation){
        if (low < high){

        int pivotPos = low + (high - low)/2;
        pivotPos = partition(arr, low, high, pivotPos);
        pthread_t thread;


        if (depthOfThreadCreation > 0){
        quickSort_parameters thread_param = {arr, low, pivotPos-1,
depthOfThreadCreation};
        int result;
        result = pthread_create(&thread, NULL, worker_quickSort, &thread_param);
        concurrent_quickSort(arr, pivotPos+1, high, depthOfThreadCreation);
        pthread_join(thread, NULL);

        } else
        {
        quickSort(arr, low, pivotPos-1);
        quickSort(arr, pivotPos+1, high);
        }
        }
}
int main(int argc, char **argv)
{
        int depthOfThreadCreation = 4;
        int *arrayElements = malloc((argc-1)*sizeof(int));
        int size=argc-1;
        for (int i=0 ; i<size ; i++){
        arrayElements[i] = atoi(argv[i+1]);
        }

        printf("Unsorted\n");
        for(int i=0; i<size; i++){
        printf("%d ", arrayElements[i]);
        }
        concurrent_quickSort(arrayElements, 0, size-1, depthOfThreadCreation);
        printf("\n");
        printf("Sorted\n");

        for(int i=0; i<size; i++)
        {
        printf("%d ", arrayElements[i]);
        }
        printf("\n");
        free(arrayElements);
        return 0;
}
```

Output ss:



```
mihir@mihir-VirtualBox: ~/Desktop/OS/Lab/week6
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week6$ gcc quick_sort.c -lpthread
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week6$ ./a.out 15 9 5 12 6 1 2 14 7
Unsorted
15 9 5 12 6 1 2 14 7
Sorted
1 2 5 6 7 9 12 14 15
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week6$
```

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

Logic: We have a circle of radius 0.5, enclosed by a 1 × 1 square. The area of the circle is π r 2 = π / 4 , the area of the square is 1. If we divide the area of the circle, by the area of the square we get. The code uses a thread to calculate the estimation part. There is a function for generating random numbers. The more the size of the dataset the better is the valid estimation procedure.

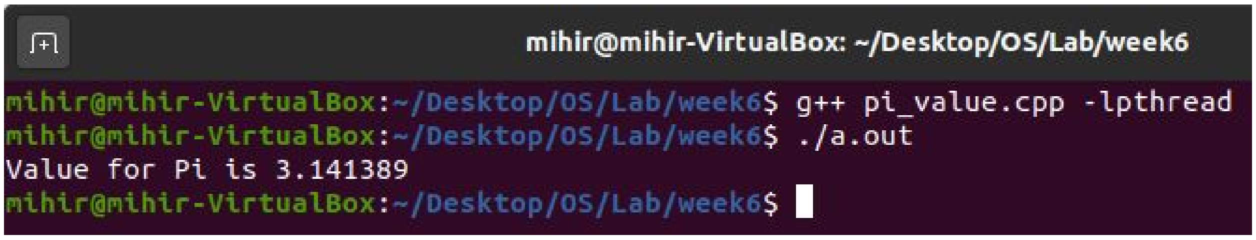
Code:

```
#include <pthread.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NUM_THREADS 8
#define TOT_COUNT 10000055
float randNumGen()
{
  int random_value = rand();
  float unit_random = random_value / (float) RAND_MAX;
  return unit_random;
}
void *doCalcs(void *threadid)
{
  long longTid;
  longTid = (long)threadid;

  int tid = (int)longTid;
  float *in_count = (float *)malloc(sizeof(float));
  *in_count=0;
  float tot_iterations= TOT_COUNT/NUM_THREADS;

  int counter=0;
  for(counter=0;counter<tot_iterations;counter++){
        float x = randNumGen();
        float y = randNumGen();

        float result = x*x+y*y;
        result= sqrt(result);

        if(result<1){
        *in_count+=1;}

  }
  if(tid==0){
        float remainder = TOT_COUNT%NUM_THREADS;

        for(counter=0;counter<remainder;counter++){
        float x = randNumGen();
        float y = randNumGen();
```

```
          float result = sqrt((x*x) + (y*y));

          if(result<1){
          *in_count+=1;                    }

   }
   }
   pthread_exit((void *)in_count);
}

int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   void *status;
   float tot_in=0;

   for(t=0;t<NUM_THREADS;t++){
          rc = pthread_create(&threads[t], NULL, doCalcs, (void *)t);
          if (rc){
          printf("ERROR; return code from pthread_create() is %d\n", rc);
          exit(-1);
          }
          }
   for(t=0;t<NUM_THREADS;t++){

          pthread_join(threads[t], &status);
          tot_in+=*(float*)status;
          }
          printf("Value for Pi is %f \n",4*(tot_in/TOT_COUNT));
          pthread_exit(NULL);
}
```

Output ss:



```
mihir@mihir-VirtualBox: ~/Desktop/OS/Lab/week6
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week6$ g++ pi_value.cpp -lpthread
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week6$ ./a.out
Value for Pi is 3.141389
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week6$
```

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

Logic: The inverse of A is A-1 only when:

A × A-1 = A-1 × A = I

1. For each element, calculate the determinant of the values not on the row or column, to make the Matrix of Minors.
2. Apply a checkerboard of minuses to make the Matrix of Cofactors.
3. Transpose to make the Adjugate.
4. Multiply by 1/Determinant to make the InverseDifferent threads are created for cofactor and transpose matrix

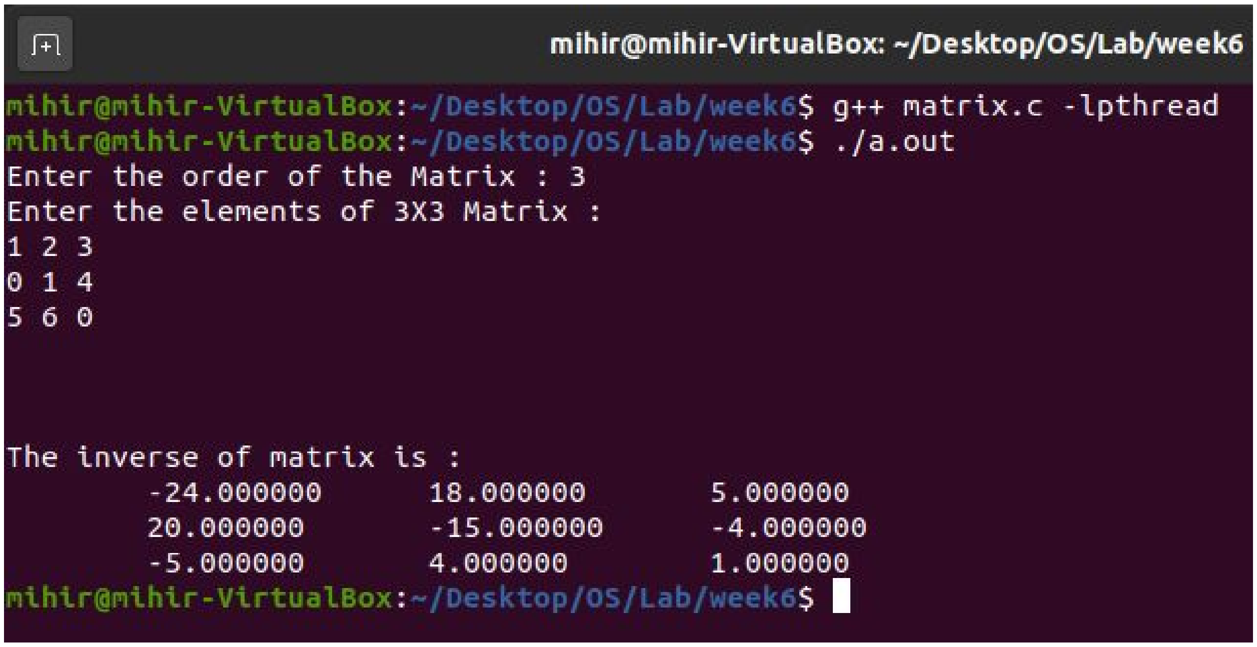
Code:

```
#include<stdio.h>
#include<math.h>
#include<pthread.h>
float a[25][25], k,fac[25][25];

/*For calculating Determinant of the Matrix */
float determinant(float a[25][25], float k)
{
  float s = 1, det = 0, b[25][25];
  int i, j, m, n, c;
  if (k == 1)
        {
        return (a[0][0]);
        }
  else
        {
        det = 0;
        for (c = 0; c < k; c++)
        {
        m = 0;
        n = 0;
        for (i = 0;i < k; i++)
        {
        for (j = 0 ;j < k; j++)
        {
                b[i][j] = 0;
```

```
                    if (i != 0 && j != c)
                    {
                    b[m][n] = a[i][j];
                    if (n < (k - 2))
                    n++;
                    else
                    {
                    n = 0;
                    m++;
                    }
                    }
                    }
          }
          det = det + s * (a[0][c] * determinant(b, k - 1));
          s = -1 * s;
          }
          }

          return (det);
}
/*Finding transpose of matrix*/
void *transpose(void *nums)
{
  int i, j;float r=k;
  float b[25][25], inverse[25][25], d;

  for (i = 0;i < r; i++)
          {
          for (j = 0;j < r; j++)
          {
          b[i][j] = fac[j][i];
          }
          }
  d = determinant(a, r);
  for (i = 0;i < r; i++)
          {
          for (j = 0;j < r; j++)
          {
          inverse[i][j] = b[i][j] / d;
          }
```

```
        }
  printf("\n\n\nThe inverse of matrix is : \n");

  for (i = 0;i < r; i++)
        {
        for (j = 0;j < r; j++)
        {
        printf("\t%f", inverse[i][j]);
        }
        printf("\n");
        }
 pthread_exit(0);
}

void *cofactor(void *nums)
{

 float b[25][25], f=k;
 int p, q, m, n, i, j;
 for (q = 0;q < f; q++)
 {
  for (p = 0;p < f; p++)
        {
        m = 0;
        n = 0;
        for (i = 0;i < f; i++)
        {
        for (j = 0;j < f; j++)
        {
        if (i != q && j != p)
        {
        b[m][n] = a[i][j];
        if (n < (f - 2))
        n++;
        else
        {
                n = 0;
                m++;
                }
        }
        }
        }
        fac[q][p] = pow(-1, q + p) * determinant(b, f - 1);
        }
  }
        pthread_exit(0);
}

int main()
{
  pthread_t tid1,tid2;
  pthread_attr_t attr;
  float d;
  int i, j;
  printf("Enter the order of the Matrix : ");
  scanf("%f", &k);
  printf("Enter the elements of %.0fX%.0f Matrix : \n", k, k);
  for (i = 0;i < k; i++)
        {
        for (j = 0;j < k; j++)
        {
        scanf("%f", &a[i][j]);
        }
        }
  d = determinant(a, k);
  if (d == 0)
   printf("\nInverse of Entered Matrix is not possible\n");
  else
   {
        pthread_attr_init(&attr);
        pthread_create(&tid1,NULL,cofactor,NULL);
        pthread_join(tid1,NULL);

        pthread_attr_init(&attr);
        pthread_create(&tid2,NULL,transpose,NULL);
        pthread_join(tid2,NULL);
   }
  return 0;
}
```

Output ss:



```
mihir@mihir-VirtualBox: ~/Desktop/OS/Lab/week6
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week6$ g++ matrix.c -lpthread
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week6$ ./a.out
Enter the order of the Matrix : 3
Enter the elements of 3X3 Matrix :
1 2 3
0 1 4
5 6 0



The inverse of matrix is :
        -24.000000      18.000000       5.000000
        20.000000       -15.000000      -4.000000
        -5.000000       4.000000        1.000000
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week6$
```

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.

Logic: The Fibonacci numbers are the numbers in the following integer sequence. 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …….. In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation Fn = Fn-1 + Fn-2 with seed values F0 = 0 and F1 = 1. Now for calculating the f(n-1) and f(n-2), we can use 2 threads separately and add the returned values to calculate the fib(n). The most essential aspect is to observe that parallelization can help in reducing the time taken to generate the series. This is an effective setup in the logical wireframe.

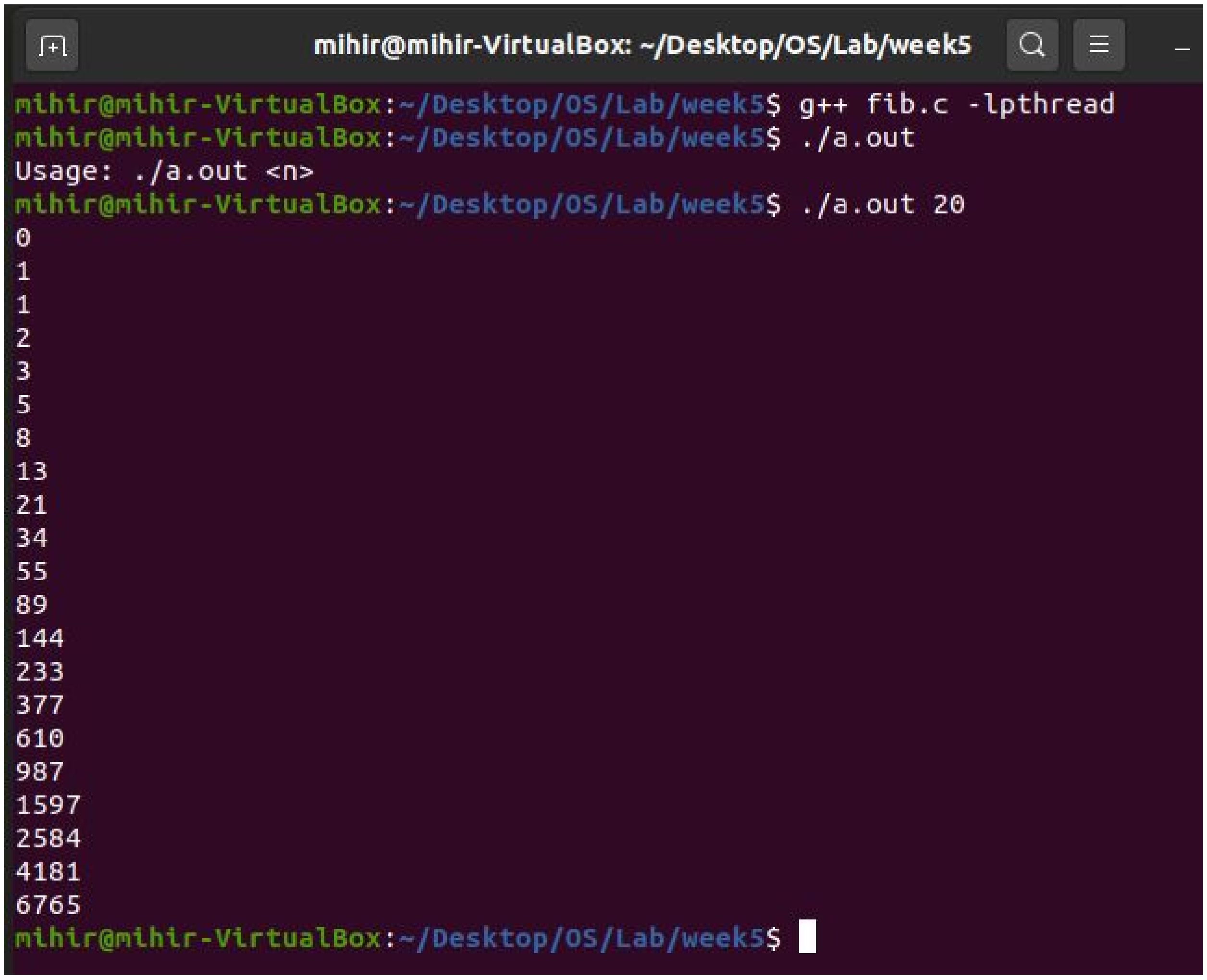
Code:

```
#include<pthread.h>
#include<stdio.h>
#include<stdlib.h>
int fib=1,fib1=1,fib2=1;
```

```
void *gen1(void *param);
void *gen2(void *param);
int main(int argc, char const *argv[])
{
    pthread_t tid1,tid2;
    pthread_attr_t attr1,attr2;

  if (argc < 2)
  {
        fprintf(stderr, "Usage: ./a.out <n>\n");
        return -1;
  }

    printf("0\n");
    printf("1\n");
    int n=atoi(argv[1]);
    for(int i=2;i<=n;i++)
    {
        int *nums1 = (int*)malloc(sizeof(int));
        int *nums2 = (int*)malloc(sizeof(int));
        nums1[0]=i-1;nums2[0]=i-2;
         pthread_attr_init(&attr1);
         pthread_create(&tid1,NULL,gen1,(void *)nums1);
         pthread_attr_init(&attr2);
         pthread_create(&tid2,NULL,gen2,(void *)nums2);
         pthread_join(tid1,NULL);
         pthread_join(tid2,NULL);
         fib=fib1+fib2;
         printf("%d\n",fib);
    }
    return 0;
}
void *gen1(void *param)
{
    int * ar=(int *)param; int n = ar[0];
    int a = 0, b = 1, c, i;
        if( n == 0)
        fib1=a;
        else{
        for(i = 2; i <= n; i++)
```

```
            {
            c = a + b;
            a = b;
            b = c;
            }
            fib1=b;
    }
    pthread_exit(0);
}
void *gen2(void *param)
{
  int * ar=(int *)param; int n = ar[0];
    int a = 0, b = 1, c, i;
            if( n == 0)
            fib2=a;
            else{
            for(i = 2; i <= n; i++)
            {
            c = a + b;
            a = b;
            b = c;
            }
            fib2=b;
    }
    pthread_exit(0);
}
```

Output ss:


mihir@mihir-VirtualBox: ~/Desktop/OS/Lab/week5

```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week5$ g++ fib.c -lpthread
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week5$ ./a.out
Usage: ./a.out <n>
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week5$ ./a.out 20
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week5$
```

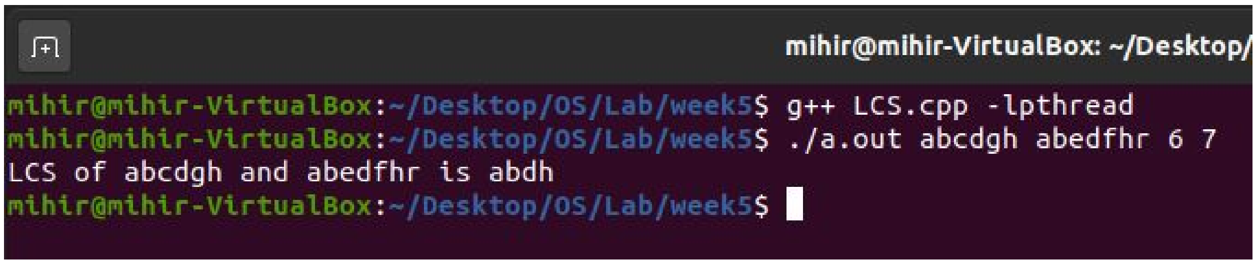
10. Longest common subsequence generation problem using threads.

Logic: The code requires 2 aspects for generation of the longest common subsequence. The first aspect being initialization. I have used a runner function gen1 for that purpose. It initializes the matrix for the base values of the substructure table. The next aspect is implementing the approach. The gen runner function is used to implement the approach as explained in the explanation above. The final string is stored in the global character array named lcs.

Code:

```
#include<pthread.h>
#include<stdio.h>
#include<stdlib.h>
#include<bits/stdc++.h>
using namespace std;
int L[100][100]; char lcs[100];
void *gen(void *param)
{
           char **argv= (char **)param;
           char *X = argv[1];
    char *Y = argv[2];
    int m=atoi(argv[3]); int n=atoi(argv[4]);
   // Following code is used to print LCS
   int index = L[m][n];
   //char lcs[index+1];
   lcs[index] = '\0';
   int i = m, j = n;
   while (i > 0 && j > 0)
   {
          if (X[i-1] == Y[j-1])
          {
          lcs[index-1] = X[i-1]; // Put current character in result
          i--; j--; index--;          // reduce values of i, j and index
          }
          else if (L[i-1][j] > L[i][j-1])
          i--;
          else
          j--;
   }

    pthread_exit(0);
}
int main(int argc, char  *argv[])
{
          memset(L,-1,sizeof(L));
          pthread_t tid;
    pthread_attr_t attr;
    int m= atoi(argv[argc-2]);
    int n= atoi(argv[argc-1]);
    char *X = argv[1];
    char *Y = argv[2];
    for (int i=0; i<=m; i++)
     {
          for (int j=0; j<=n; j++)
          {
          if (i == 0 || j == 0)
          L[i][j] = 0;
          else if (X[i-1] == Y[j-1])
          L[i][j] = L[i-1][j-1] + 1;
          else
          L[i][j] = max(L[i-1][j], L[i][j-1]);
          }
          }
          pthread_attr_init(&attr);
    pthread_create(&tid,NULL,gen,(void *)argv);
    pthread_join(tid,NULL);
    cout << "LCS of " << X << " and " << Y << " is " << lcs<<endl;
          return 0;
}
```

Output ss:



```
mihir@mihir-VirtualBox: ~/Desktop/
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week5$ g++ LCS.cpp -lpthread
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week5$ ./a.out abcdgh abedfhr 6 7
LCS of abcdgh and abedfhr is abdh
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week5$
```