Report For Question 1 (Quicksort)
-Risubh Jain(2018101104)

Insertion Sort

- Used when size of subarray is less than 5

```
void smolsort(int arr[], int l, int r)
{
    for (int i = l; i < r; i++)
    {
        int key = arr[i];
        int ind = i;
        for (int j = i; j < r; j++)
        {
            if (key > arr[j])
            {
                ind = j;
                key = arr[j];
            }
        }
        swap(&arr[i], &arr[ind]);
    }
}
```

issorted

- Function to check if array is sorted

```
void issorted(int arr[], int l, int r)
{
   for (int i = l; i < r - 1; i++)
       if (arr[i] > arr[i + 1])
       {
           printf("Not Sorted\n");
           return;
       }
   printf("Sorted\n");
}
```

getTime

- Function to return current time
- Used to calculate time by different versions of quicksort and compare them

```
long double getTime()
{
   struct timespec ts;
   clock_gettime(CLOCK_MONOTONIC_RAW, &ts);
   return ts.tv_nsec / (1e9) + ts.tv_sec;
}
```

partition

- Function to partition array based on randomly chosen pivot and return the index of the final position of pivot

```
int partition(int arr[], int l, int r)
{
   int random_pivot = rand() % (r - l) + l;
   swap(&arr[random_pivot], &arr[r - 1]);
   int pivot = arr[r - 1];
   int j = l - 1;
   for (int i = l; i < r - 1; i++)
   {
       if (arr[i] < pivot)
       {
           j++;
           swap(&arr[i], &arr[j]);
       }
   }
   swap(&arr[j + 1], &arr[r - 1]);
   return j + 1;
}
```

shareMem

- Function to allocate shared memory and return pointer to that memory segment

```
int *shareMem(size_t size)
{
   key_t mem_key = IPC_PRIVATE;
   int shm_id = shmget(mem_key, size, IPC_CREAT | 0666);
   return (int *)shmat(shm_id, NULL, 0);
}
```

Quicksort

- Choose pivot and partition the array based on that pivot
- Recursively repeat on left and right of pivot
- Sort array of size less than 5 using insertion sort

```
void Normal_quicksort(int arr[], int l, int r)
{
    if (r - l < 5)
    {
        smolsort(arr, l, r);
        return;
    }
    int par = partition(arr, l, r);
    Normal_quicksort(arr, l, par);
    Normal_quicksort(arr, par + 1, r);
}
```

Concurrent Quicksort

- Instead of waiting for one recursion call to finish we can simultaneously call partition function on both left and right subarray
- Partition the array using partition function
- Create a child process using fork and sort one half in child and one half in parent
- In parent, wait for child to finish
- Sort using insertion sort if size of array is less than 5

```
if (r - l < 5)
{
    smolsort(arr, l, r);
    return;
}
int par = partition(arr, l, r);
pid_t pid = fork();
int status = 0;
if (pid < 0)
{
    printf("Error while forking\n");
    return ;
}
else if (pid == 0)
{
    Process_quicksort(arr, l, par);
    _exit(0);
}
else
{
    int status2;
    Process_quicksort(arr, par + 1, r);
    waitpid(pid, &status2, 0);
}
```

Threaded Quicksort

- Following the same logic of parallelizing the recursive calls, we create different thread instead of process for each subarray

```
void *Thread_quicksort(void *thr)
{
    struct arg *a = (struct arg *)thr;
    int l = a->l;
    int r = a->r;
    int *arr = a->arr;
    if (l > r)
        return NULL;
    if (r - l < 5)
    {
        smolsort(arr, l, r);
        return NULL;
    }
    pthread_t tidll, tidrr;
    struct arg ll, rr;
    int par = partition(arr, l, r);
    ll.l = l, ll.r = par, ll.arr = arr;
    rr.l = par + 1, rr.r = r, rr.arr = arr;
    pthread_create(&tidll, NULL, (void *)Thread_quicksort, &ll);
    pthread_join(tidll, NULL);
    pthread_create(&tidrr, NULL, (void *)Thread_quicksort, &rr);
    pthread_join(tidrr, NULL);
    return NULL;
}
```

Performance Report

For n = 10

Time for Normal Quicksort is 0.000006s
Time for Concurrent Quicksort is 0.001455s
Time for Threaded Quicksort is 0.001439s
Threaded Quicksort is 254.625254 times slower than Normal Quicksort
Concurrent Quicksort is 257.487957 times slower than Normal Quicksort

For n = 100

Time for Normal Quicksort is 0.000040s
Time for Concurrent Quicksort is 0.004268s
Time for Threaded Quicksort is 0.014330s
Threaded Quicksort is 357.126740 times slower than Normal Quicksort
Concurrent Quicksort is 106.374534 times slower than Normal Quicksort

For n = 1000

Time for Normal Quicksort is 0.000447s
Time for Concurrent Quicksort is 0.008441s
Time for Threaded Quicksort is 0.015410s
Threaded Quicksort is 34.448377 times slower than Normal Quicksort
Concurrent Quicksort is 18.870175 times slower than Normal Quicksort

For n = 10000
Time for Normal Quicksort is 0.005434s
Time for Concurrent Quicksort is 0.069490s
Time for Threaded Quicksort is 0.266480s
Threaded Quicksort is 49.037592 times slower than Normal Quicksort
Concurrent Quicksort is 12.787486 times slower than Normal Quicksort