Quicksort in 3 ways
This code aims to simulate quick sort in 3 different ways

- Normal
- Concurrent
- Threaded

And compare their respective run times

Approach:-

Normal quicksort

Here in each iteration after we pick a pivot (random element in array) , we partition the array such that pivot is placed in its required spot in the sorted version of the array in such a way that elements smaller than the pivot are on the left and larger ones are on the right. Partition function is then called recursively on each half of the array, i.e the portion to the left of the pivot and the portion to the right of the pivot
This method is generally the quickest

Concurrent quicksort
Here we use the same algorithm of quicksort but after every partition the halves are operated on recursively using forked processes.This method is generally the slowest due to the time it takes to create the processes

Threaded quicksort
Here we use the same algorithm of quicksort but after every partition the halves are operated on in separate new threads created using pthreads library
Since thread creation is faster than process creation this is quicker than concurrent version and is similar in time of completion to normal version. Threaded version is always found to be a little slower than the normal version however, i.e it takes more time to complete.

In all above three versions of the quicksort, the algorithm changes into insertion sort when the sive of the remaining elements is lesser than or equal to 5

Implementation:-

Partition function:-

```
int partition(int *arr, int low, int high) {

  ///Setting random element as the last element which shall be used as pivot
  srand(time(NULL));
  int random = low + rand() % (high - low);
  swap(&arr[random], &arr[high]);

  int pivot = arr[high]; /// pivot
  int i = (low - one); /// Index of smaller element

  for (int j = low; j <= high- one; j+=one)
  {
      /// If current element is smaller than the pivot
      if (arr[j] < pivot)
      {
          i+=one; // increment index of smaller element
          swap(&arr[i], &arr[j]);
      }
  }
  swap(&arr[i + one], &arr[high]);
  return (i + one);
}
```

This is a common function used by all the types of quicksort. It takes the array provided in its argument and partitions is (like explained before) where low and high are the bounds of the array
In the 3rd line of code, we swap the last element with a random element of the array and then continue with the partitioning code where the last element in taken as pivot
This is done to ensure that the pivot is a random element

Normal quicksort:-

```
void quickSort(int *arr, int low, int high) {
  if(low>high) _exit(one);

  if(high-low+one<=5){
      for(int i=low;i<high;i+=one)
      {
          int j=i+one;
          for(;j<=high;j+=one)
              if(arr[j]<arr[i] && j<=high)
              {
                  int temp = arr[i];
                  arr[i] = arr[j];
                  arr[j] = temp;
```

```
            }
        }
        return;
    }

    int pi = partition(arr, low, high);

    quickSort(arr, low, pi - one);
    quickSort(arr, pi + one, high);

}
```

Here the code works as stated before

Concurrent quicksort:-
For this version of quicksort , the array of values must exist in a shared memory space so that all forked processes that access it can work on the same memory. This is ensured by the function shareMem

```
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, zero);
}
```

The implementation of concurrent quicksort is as follows and works the same way as explained before

```
void quicksort_C(int *arr, int l, int r) {
    if(l>r) _exit(one);

    //insertion sort
    if(r-l+one<=5){
        for(int i=l;i<r;i+=one)
        {
            int j=i+one;
            for(;j<=r;j+=one)
                if(arr[j]<arr[i] && j<=r)
                {
                    int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        return;
    }

    int pid1 = fork();
    int pid2;

    int pi = partition(arr, l, r);

    if(pid1==zero){
        quicksort_C(arr, l, pi - one);
        _exit(one);
    }
    else{
        pid2 = fork();
        if(pid2==zero){
            //sort right half array
            quicksort_C(arr, pi + one, r);
            _exit(one);
        }
        else{
            //wait for the right and the left half to get sorted
            waitpid(pid1, NULL, zero);
            waitpid(pid2, NULL, zero);
        }
    }
}
```

Threaded quicksort:-

Here the sort is called as separate threads
Pthread_create make the execution of the function as separate threads and pthread_join waits for the completion of the thread

The value to be passed into the function(3rd parameter) is given by the struct value present in the 4th parameter

Code:-

```
        pthread_create(&tid, NULL, threaded_quicksort, &a);
        pthread_join(tid, NULL);
```

The implementation of this version is as follows and works just like it was explained before

```
void *threaded_quicksort(void *a) {
  struct arg *args = (struct arg*) a;

  int l = args->l;
  int r = args->r;
  int *arr = args->arr;
  if(l>r) return NULL;

  if(r-l+one<=5){
     for(int i=l;i<r;i+=one)
     {
        int j=i+one;
        for(;j<=r;j+=one)
           if(arr[j]<arr[i] && j<=r)
           {
              int temp = arr[i];
              arr[i] = arr[j];
              arr[j] = temp;
           }
     }
     return NULL;
  }

  int pi=partition(arr,l,r);

  struct arg a1;
  a1.l = l;
  a1.r = pi-1;
  a1.arr = arr;
  pthread_t tid1;
  pthread_create(&tid1, NULL, threaded_quicksort, &a1);

  struct arg a2;
  a2.l = pi+1;
  a2.r = r;
  a2.arr = arr;
  pthread_t tid2;
  pthread_create(&tid2, NULL, threaded_quicksort, &a2);

  //wait for the two halves to get sorted
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);

  return NULL;
}
```

Results

1) Normal quicksort is almost always faster than concurrent and threaded.
2) Concurrent quicksort is generally the slowest
3) Threaded quicksort is comparable to normal in speed

For array length = 10

Time for normal quicksort = 0.000008s
Time for concurrent quicksort = 0.002828s
Time for threaded quicksort=0.001439s

For array length =100
Time for normal quicksort =0.007629s
Time for concurrent quicksort=15.341498s
Time for threaded quicksort = 0.110237s