Multithreaded Sorting Application

In this project, we use Bubble Sort to sort two arrays and design sorting threads for them

```
void sort(int a, int b)
{
    for(int i = a; i < b; i++)
    {
        for(int j = a+1; j < b + a - i + 1;j++)
        {
            if(array[j] < array[j-1])
            {
                int tmp = array[j];
                array[j] = array[j-1];
                array[j-1] = tmp;
            }
        }
    }
}
void* sortThread(void *arg)
{
    struct size_* s = (struct size_ *) arg;
    sort(s->low,s->high);

}
```

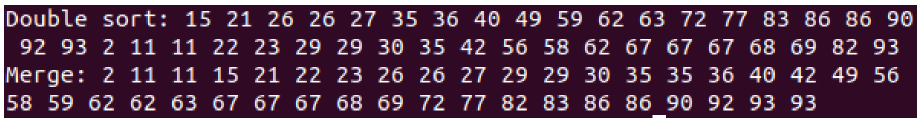
Then merge them into one array.

```
void* merge(void *arg)
{
    int idx1 = 0;
    int idx2 = 20;
    int cnt = 0;
    while(idx1 <= 19 || idx2 <= 39)
    {
        if(idx1 <= 19 && idx2 <= 39)
        {
            if(array[idx1] > array[idx2])
            {
                res[cnt++] = array[idx2++];
            }
            else
            {
                res[cnt++] = array[idx1++];
            }
        }
        else
        {
            if(idx1 <= 19)
            {
                while(cnt <= 39)
                {
                    res[cnt++] = array[idx1++];
                }
            }
            else
            {
                while(cnt <= 39)
                {
                    res[cnt++] = array[idx2++];
                }
            }
        }

    }
}
```

use `<pthread.h>` to realize the application of multithread.

```
int main()
{
    array = malloc(40* sizeof(int));
    res = malloc(40* sizeof(int));

    for(int i = 0; i < 40; i++)
    {
        array[i] = rand()%100;
        //printf("%d ", array[i]);
    }

    pthread_t sortingThreads[2];

    struct size_ sizes[2];
    sizes[0].low = 0;
    sizes[0].high = 19;
    sizes[1].low =20;
    sizes[1].high = 39;

    for (int i=0; i<2; i++)
    {
        pthread_create(&sortingThreads[i], NULL,
sortThread, (void *)&sizes[i]);

    }
    for (int i = 0; i < 2; i++)
        pthread_join(sortingThreads[i], NULL);

    printf("Double sort: ");
    for (int i=0; i<40; i++) printf("%d ", array[i]);
    printf("\n");

    pthread_t mergingThread;
    pthread_create(&mergingThread, NULL, merge, NULL);
    pthread_join(mergingThread, NULL);

    printf("Merge: ");
    for (int i=0; i<40; i++) printf("%d ", res[i]);
    printf("\n");


    return 0;
}
```

Result



```
Double sort: 15 21 26 26 27 35 36 40 49 59 62 63 72 77 83 86 86 90
 92 93 2 11 11 22 23 29 29 30 35 42 56 58 62 67 67 67 68 69 82 93
Merge: 2 11 11 15 21 22 23 26 26 27 29 29 30 35 35 36 40 42 49 56
58 59 62 62 63 67 67 67 68 69 72 77 82 83 86 86 90 92 93 93
```

Fork-Join Sorting Application

In this project, we will implement two sorting algorithm based on divide-and-conquer using fork-join API in java.

- Quick sort
- Merge sort

Fork-Join API

In main class, we initialize Fork-join pool and use `pool(task)` to execute the running.

```
public class ForkJoinArraySort {
    // From Java 7 '_' can be used to separate digits.
    public static final int ARRAY_SIZE = 20;
    public static void main(String[] args) {
        // Create a pool of threads
        ForkJoinPool pool = new ForkJoinPool();
        int[] array1 = createArray(ARRAY_SIZE);
        int[] array2 = array1.clone();
        System.out.print("Origin Array: ");
        for (int i=0; i<array1.length; i++)
System.out.printf("%d ", array1[i]);
        System.out.print("\n");

        long startTime;
        long endTime;

        // QuickSort
        QuickSort quick = new QuickSort(array1, 0,
array1.length - 1);
        startTime = System.currentTimeMillis();

        pool.invoke(quick); // Start execution and wait
for result/return

        endTime = System.currentTimeMillis();

        System.out.print("QuickSort: ");
        for (int i=0; i<array1.length; i++)
System.out.printf("%d ", array1[i]);
        System.out.print("\n");
        System.out.println("Time taken: " + (endTime -
startTime) + " millis");

        // MergeSort
        MergeSort merge = new MergeSort(array2, 0,
array2.length - 1);
        startTime = System.currentTimeMillis();

        pool.invoke(merge); // Start execution and wait
for result/return

        endTime = System.currentTimeMillis();

        System.out.print("MergeSort: ");
        for (int i=0; i<array2.length; i++)
System.out.printf("%d ", array2[i]);
        System.out.print("\n");
        System.out.println("Time taken: " + (endTime -
startTime) + " millis");
    }
    private static int[] createArray(final int size) {
        int[] array = new int[size];
        Random rand = new Random();
        for (int i = 0; i < size; i++) {
            array[i] = rand.nextInt(1000);
        }
        return array;
    }
}
```

Merge Sort

We only need to override the `compute()` function in a `RecursiveAction` class to implement merge sort.

```
class MergeSort extends RecursiveAction {
    private int array[];
    private int left;
    private int right;

    public MergeSort(int[] array, int left, int right) {
        this.array = array;
        this.left = left;
        this.right = right;
    }

    /**
     * Inherited from RecursiveAction.
     * Compare it with the run method of a Thread.
     */
    @Override
    protected void compute() {
        if (left < right) {
            int mid = (left + right) / 2;
            RecursiveAction leftSort = new
MergeSort(array, left, mid);
            RecursiveAction rightSort = new
MergeSort(array, mid + 1, right);
            invokeAll(leftSort, rightSort);
            merge(left, mid, right);
        }
    }

    /**
     * Merge two parts of an array in sorted manner.
     * @param left  Left side of left array.
     * @param mid   Middle of separation.
     * @param right Right side of right array.
     */
    private void merge(int left, int mid, int right) {
        int temp [] = new int[right - left + 1];
        int x = left;
        int y = mid + 1;
        int z = 0;
        while (x <= mid && y <= right) {
            if (array[x] <= array[y]) {
                temp[z] = array[x];
                z++;
                x++;
            } else {
                temp[z] = array[y];
                z++;
                y++;
            }
        }
        while (y <= right) {
            temp[z++] = array[y++];
        }
        while (x <= mid) {
            temp[z++] = array[x++];
        }

        for (z = 0; z < temp.length; z++) {
            array[left + z] = temp[z];
        }
    }
}
```

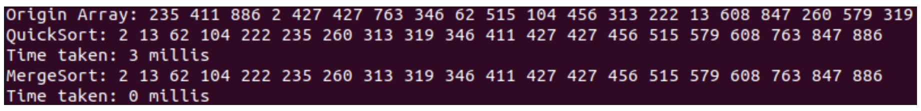
Quick Sort

Similarly, override the `compute()` function in a `RecursiveAction` class.

```
class QuickSort extends RecursiveAction {
    private int array[];
    private int left;
    private int right;

    public QuickSort(int[] array, int left, int right) {
        this.array = array;
        this.left = left;
        this.right = right;
    }

    /**
```

```
      * Inherited from RecursiveAction.
      * Compare it with the run method of a Thread.
      */
     @Override
     protected void compute() {
         if (left < right) {
             // int mid = (left + right) / 2;
             // RecursiveAction leftSort = new
 MergeSort(array, left, mid);
             // RecursiveAction rightSort = new
 MergeSort(array, mid + 1, right);
             // invokeAll(leftSort, rightSort);
             // merge(left, mid, right);
             int pivot = array[left];
             int ll = left+1;
             for (int i=left + 1; i<=right; i++){
                 if (array[i] < array[left]){
                     int tmp = array[i];
                     array[i] = array[ll];
                     array[ll] = tmp;
                     ll+=1;
                 }
             }
             int tmp = array[left];
             array[left] = array[ll-1];
             array[ll-1] = tmp;
             RecursiveAction leftSort = new
 QuickSort(array, left, ll-2);
             RecursiveAction rightSort = new
 QuickSort(array, ll, right);
             invokeAll(leftSort, rightSort);
         }
     }

 }
```

Result



```
Origin Array: 235 411 886 2 427 427 763 346 62 515 104 456 313 222 13 608 847 260 579 319
QuickSort: 2 13 62 104 222 235 260 313 319 346 411 427 427 456 515 579 608 763 847 886
Time taken: 3 millis
MergeSort: 2 13 62 104 222 235 260 313 319 346 411 427 427 456 515 579 608 763 847 886
Time taken: 0 millis
```