Project3: **Multithreaded Sorting Application & Fork-Join Sorting Application**

Yanjie Ze 519021910706

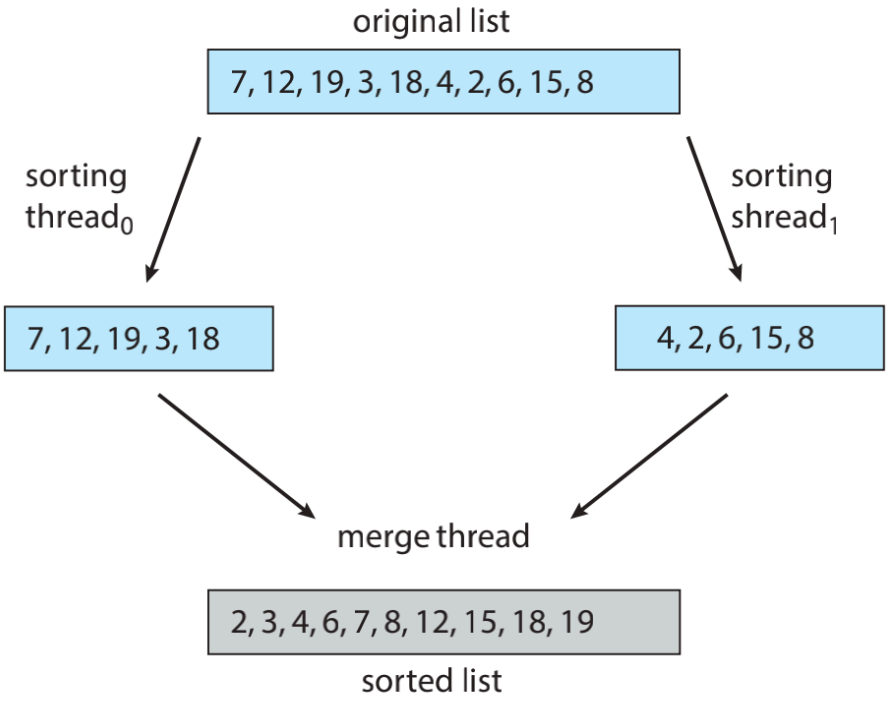
Project3-1: Multithreaded Sorting Application

1 Requirements

In this project, we write a multithreaded sorting program that works as follows:

A list of integers is divided into two smaller lists of equal size. Two separate threads (which we will term **sorting threads**) sort each sublist using a sorting algorithm of your choice. The two sublists are then merged by a third thread—a **merging thread** —which merges the two sublists into a single sorted list.

The program is structured as the figure below.



This programming project will require passing parameters to each of the sorting threads. In particular, it will be necessary to identify the starting index from which each thread is to begin sorting.

The **parent thread** will output the sorted array once all sorting threads have exited.

2 Implementation Details and Methods

This section introduces the implementation details and methods.

2.1 Global Variable and Struct

Because global data are shared across all threads, the easiest way to set up the data is to create a global array.

```
int *array;
int *array_copy;
```

To make the passing of parameters more flexible, we define a struct called **range**, to record the working space of one sorting algorithm.

```
typedef struct
{
    /* data */
    int left;
    int right;
    int mid;
} range;
```

2.2 Basic Code Frame

Defining the basic code frame and determining what functions we need to implement are the most important thing.

First, the main part takes the input and stores the input in **array**.

```
int n;
    scanf("%d", &n);
    printf("Your input: %d, then please input array. \n", n);
    array = (int*)malloc(sizeof(int)*n);
    array_copy = (int*)malloc(sizeof(int)*n);

    for(int i=0;i<n;++i)
    {
        scanf("%d", &array[i]);
    }
```

Second, we need to define **3 threads**:

- One sorts the left part of the array
- One sorts the right part of the array
- One merges the left part and the right part of the array

```
   pthread_t tid1, tid2, tid3;
    pthread_attr_t attr1, attr2, attr3;
    pthread_attr_init(&attr1);
    pthread_attr_init(&attr2);
    pthread_attr_init(&attr3);
```

Third, we assign the working range of each thread:

```
    range *r[3];
    for(int i=0;i<3;++i)
        r[i] = (range*)malloc(sizeof(range));

    int mid = (n-1)/2;
    if(n>1){
      // thread 1
    r[0]->left = 0;
    r[0]->right = mid;
```

```
    //thread 2
    r[1]->left = mid+1;
    r[1]->right = n-1;

    //thread 3
    r[2]->left = 0;
    r[2]->right = n-1;
    r[2]->mid = mid;
    }
```

Forth, we pass the functions that we want each thread to execute, using **pthread_create()** and **pthread_join()**. The two **pthread** operation functions are provided by the library **pthread**.

```
    pthread_create(&tid1, &attr1, sort, r[0]);//sort
    pthread_create(&tid2, &attr2, sort, r[1]); //sort
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    pthread_create(&tid3, &attr3, merge, r[2]);//merge
    pthread_join(tid3, NULL);
```

In this part, the functions we need to implement are **sort()** and **merge()**, which we will clarify in the following section.

Fifth, after finishing sorting, we display the result of sorting and free the space(a necessary coding habit).

```
 printf("Sort Result: ");
    for(int i=0;i<n;++i)
    {
        printf("%d ", array[i]);
    }
    printf("\n");
    // free space
    for(int i=0;i<3;++i)
        free(r[i]);
    free(array_copy);
    free(array);
```

2.3 Quicksort Implementation

Quicksort has the time complexity:

- Average: O(nlogn)
- Worst: O(n^2)
- Best:O(nlogn)

The basic idea of Quicksort is **Divide and Conquer**, which first divides the question into small questions and solves them in a small scale.

The algorithm of Quicksort is:

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi do
        if A[j] < pivot then
            swap A[i] with A[j]
```

```
            i := i + 1
    swap A[i] with A[hi]
    return i
```

In our implementation, we also use a recursive form.

Each time the main part of **quick_sort** selects a **pivot** and divides the array into two part:

- the left part is smaller than the pivot
- the right part is larger than the pivot

Then we have the following implementation:

```
void quick_sort(int *a, int low, int high)
{
    int i = low;
    int j = high;
    int pivot = a[low];
    if (low >= high)
    {
        return ;
    }

    while (low < high)
    {
        while (low < high && pivot <= a[high])
        {
            --high;
        }
        if (pivot > a[high])
        {
            int tmp;
            tmp = a[low];
            a[low] = a[high];
            a[high] =tmp;
```

```
            ++low;
        }
        while (low < high && pivot >= a[low])
        {
            ++low;
        }
        if (pivot < a[low])
        {
            int tmp;
            tmp = a[low];
            a[low] = a[high];
            a[high] =tmp;
            --high;
        }
    }
    quick_sort(a, i, low-1);
    quick_sort(a, low+1, j);
}
```

Then, we cover this function into another function, to make the thread able to call it:

```
void *sort(void *r)
{
    int left = ((range*)r)->left;
    int right = ((range*)r)->right;
    quick_sort(array, left, right);
}
```

2.4 Merge Implementation

The process of merging is similar to the merge sort algorithm.

We compare the elements of two subarrays one by one, and copy the smaller one into **array_copy**.

We repeat this action until one subarray is empty.

Then we copy the rest of another subarray directly into **array_copy**.

Finaly, copy **array_copy** back to **array**.

```
void *merge(void *r)
{
    int left = ((range*)r)->left;
    int right = ((range*)r)->right;
    int mid = ((range*)r)->mid;

    int current1=left;
    int current2=mid+1;
    int copy_id = 0;

    while(current1<=mid && current2<=right)
    {
        if(array[current1]<=array[current2])
            {
                array_copy[copy_id]=array[current1];
                current1++;
                copy_id++;
            }
        else
        {
            array_copy[copy_id]=array[current2];
            current2++;
            copy_id++;
        }
    }
    if(copy_id==right)
    {
        for(int i=left;i<=right;i++)
            array[i]=array_copy[i];
    }
    else if(current1<mid)
    {
        for(int i=current1;i<=mid;i++)
```

```
            {
                array_copy[copy_id]=array[i];
                copy_id++;
            }
        for(int i=0; i<=right;i++)
            array[i]=array_copy[i];
    }
    else if(current2<right)
    {

        for(int i=current2;i<=right;i++)
            {
                array_copy[copy_id]=array[i];
                copy_id++;
            }

        for(int i=0; i<=right;i++)
            array[i]=array_copy[i];
    }
}
```

3 Program Results

Complie the program:

```
gcc multithread_sorting.c -o c_sorting -l pthread
```

Run with 10 elements:

```
zyj@ubuntu:~/Documents/osc10e/ch4_lab$ ./c_sorting
10
Your input: 10, then please input array.
89 98 76 67 45 53 1 2 3 4
Sort Result: 1 2 3 4 45 53 67 76 89 98
```

Run with 15 elements:

```
zyj@ubuntu:~/Documents/osc10e/ch4_lab$ ./c_sorting
15
Your input: 15, then please input array.
20 19 18 17 16 1 2 3 4 5 67 53 33 22 11
Sort Result: 1 2 3 4 5 11 16 17 18 19 20 22 33 53 67
```

Run with 20 elements:

```
zyj@ubuntu:~/Documents/osc10e/ch4_lab$ ./c_sorting
20
Your input: 20, then please input array.
20 19 18 17 16 15 14 13 1 2 3 4 5 66 77 88 99 11 22 25
Sort Result: 1 2 3 4 5 11 13 14 15 16 17 18 19 20 22 25 66 77 88 99
```

Run with repeated elements:

```
zyj@ubuntu:~/Documents/osc10e/ch4_lab$ ./c_sorting
10
Your input: 10, then please input array.
1 1 1 1 1 1 1 1 1 1
Sort Result: 1 1 1 1 1 1 1 1 1 1
```

Project3-2:Fork-Join Sorting Application

1 Requirements

Implement the preceding project (Multithreaded Sorting Application) using Java's fork-join parallelism API. This project will be developed in two different versions. Each version will implement a different divide-and-conquer sorting algorithm:

1. Quicksort
2. Mergesort

The Quicksort implementation will use the Quicksort algorithm for dividing the list of elements to be sorted into a *left half* and a *right half* based on the position of the pivot value. The Mergesort algorithm will divide the list into two evenly sized halves. **For both the Quicksort and Mergesort algorithms, when the list to be sorted falls within some threshold value (for example, the list is size 100 or fewer), directly apply a simple algorithm such as the Selection or Insertion sort.** Most data structures texts describe these two well-known, divide-and-conquer sorting algorithms.

The class SumTask shown in Section 4.5.2.1 extends RecursiveTask, which is a result-bearing ForkJoinTask. As this assignment will involve sorting the array that is passed to the task, but not returning any values, we will instead create a class that extends RecursiveAction, a non result-bearing ForkJoinTask (see Figure 4.19).

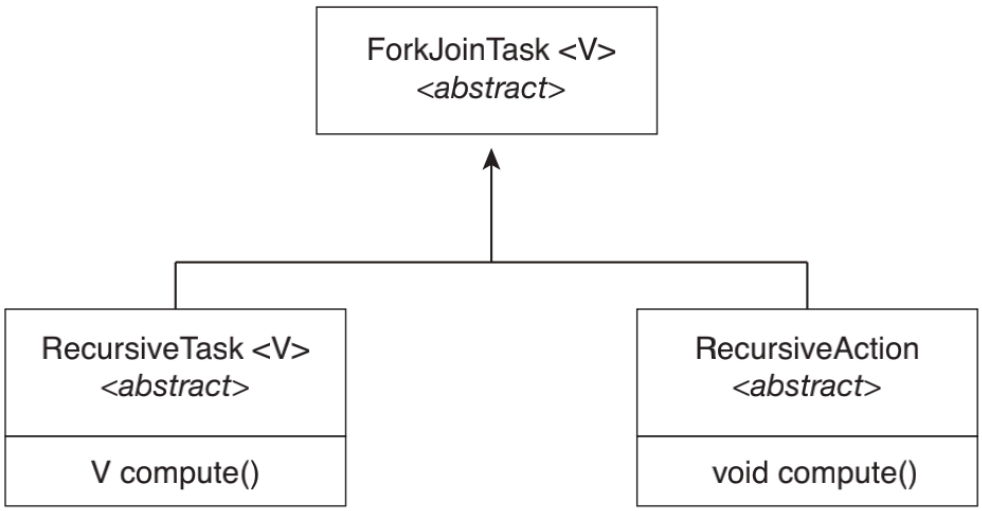


Figure 4.19 UML class diagram for Java's fork-join.

The objects passed to each sorting algorithm are required to implement Java's Comparable interface, and this will need to be reflected in the class definition for each sorting algorithm.

2 Implementation Details and Methods

2.1 Quicksort Implementation

The implementation of Quicksort here is almost the same as what we have implemented in the project3-1. This is because the grammar of Java is very similar to the grammar of C.

The algorithm of Quicksort is shown below:

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi do
        if A[j] < pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

In our implementation, we write the partition function just in the quicksort function.

```
   public void quick_sort_pivot(int low, int high)
    {
        if (end - begin < THRESHOLD) {

            this.selection_sort(begin, end);
            return;
        }
        int i=low;
        int j=high;
        int pivot = array[low];
        if(low>=high)
```

```
        return;

    while(low<high)
    {
        while(low<high && pivot<=array[high])
            high--;
        if(pivot > array[high])
        {
            int tmp;
            tmp = array[low];
            array[low] = array[high];
            array[high] =tmp;
            ++low;
        }
        while (low < high && pivot >= array[low])
        {
        ++low;
        }
        if (pivot < array[low])
        {
            int tmp;
            tmp = array[low];
            array[low] = array[high];
            array[high] =tmp;
            --high;
        }
    }
    quick_sort_pivot(i, low-1);
    quick_sort_pivot(low+1, j);
}
```

2.2 Mergesort Implementation

Conceptually, a merge sort works as follows:

1. Divide the unsorted list into *n* sublists, each containing one element (a list of one element is considered sorted).
2. Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.

Mergesort has the time complexity **O(nlogn)** in average.

In our implementation, we also utilize the part of our implementation in the project3-1.

We use **array_copy** to copy the element after the comparison one by one.

```
private void merge(int left, int mid, int right)
    {
        int current1=left;
        int current2=mid+1;
        int copy_id =left;
        int[] array_copy = new int[SIZE];

        while(current1<=mid && current2<=right)
        {
            if(array[current1]<=array[current2])
                {
                    array_copy[copy_id]=array[current1];
                    current1++;
                    copy_id++;
                }
            else
            {
                array_copy[copy_id]=array[current2];
                current2++;
                copy_id++;
            }
```

```
        }
        if(copy_id==right)
        {
            for(int i=left;i<=right;i++)
                array[i]=array_copy[i];
        }
        else if(current1<mid)
        {
            for(int i=current1;i<=mid;i++)
                {
                    array_copy[copy_id]=array[i];
                    copy_id++;
                }
            for(int i=left; i<=right;i++)
                array[i]=array_copy[i];
        }
        else if(current2<right)
        {
                for(int i=current2;i<=right;i++)
                {
                    array_copy[copy_id]=array[i];
                    copy_id++;
                }

                for(int i=left; i<=right;i++)
                    array[i]=array_copy[i];
        }

    }
```

2.3 Selection Sort Implementation

Selection Sort is one of the most simplest sorting algorithm, with the time complexity **O(n^2)**.

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

- The subarray which is already sorted.
- Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

The implementation is shown here.

```
private void selection_sort(int low, int high)
    {
        for(int i=low;i<high;i++)
        {
            int min = array[i];
            for(int j=i+1;j<=high;j++)
                {
                    if(array[j]<min)
                    {
                        min = array[j];
                        array[j] = array[i];
                        array[i] = min;
                    }
                }
        }
    }
```

2.4 compute()

To call the algorithm we implement before, we need to finish the function **compute()**.

This function will call **Selection Sort** when the subarray's size reaches the **threshold**, and will call **Quicksort** or **Mergesort** when the subarray's size is beyond the **threshold**.

```
protected void compute()
    {
```

```
        if (end - begin < THRESHOLD) {

            this.selection_sort(begin, end);
            return;
        }
        else{
            // divide stage
            int mid = (end+begin)/2;
            SortTask leftTask = new SortTask(begin, mid, array);
            SortTask rightTask = new SortTask(mid + 1, end, array);

            leftTask.fork();
            rightTask.fork();

            leftTask.join();
            rightTask.join();
            this.merge(begin, mid, end);
            return;
        }
    }
```

2.5 Main()

In main(), we utilize what we have implemented, and finish the input part and the output part.

Our input is randomized by the function **rand.nextInt()**.

```
// create SIZE random integers between 0 and 9
    java.util.Random rand = new java.util.Random();

    System.out.println("Init array is: ");

    for (int i = 0; i < SIZE; i++) {
      array[i] = rand.nextInt(10);

            System.out.print(array[i]);
            System.out.print(' ');
    }
```

We use fork-join parallelism to get the array.

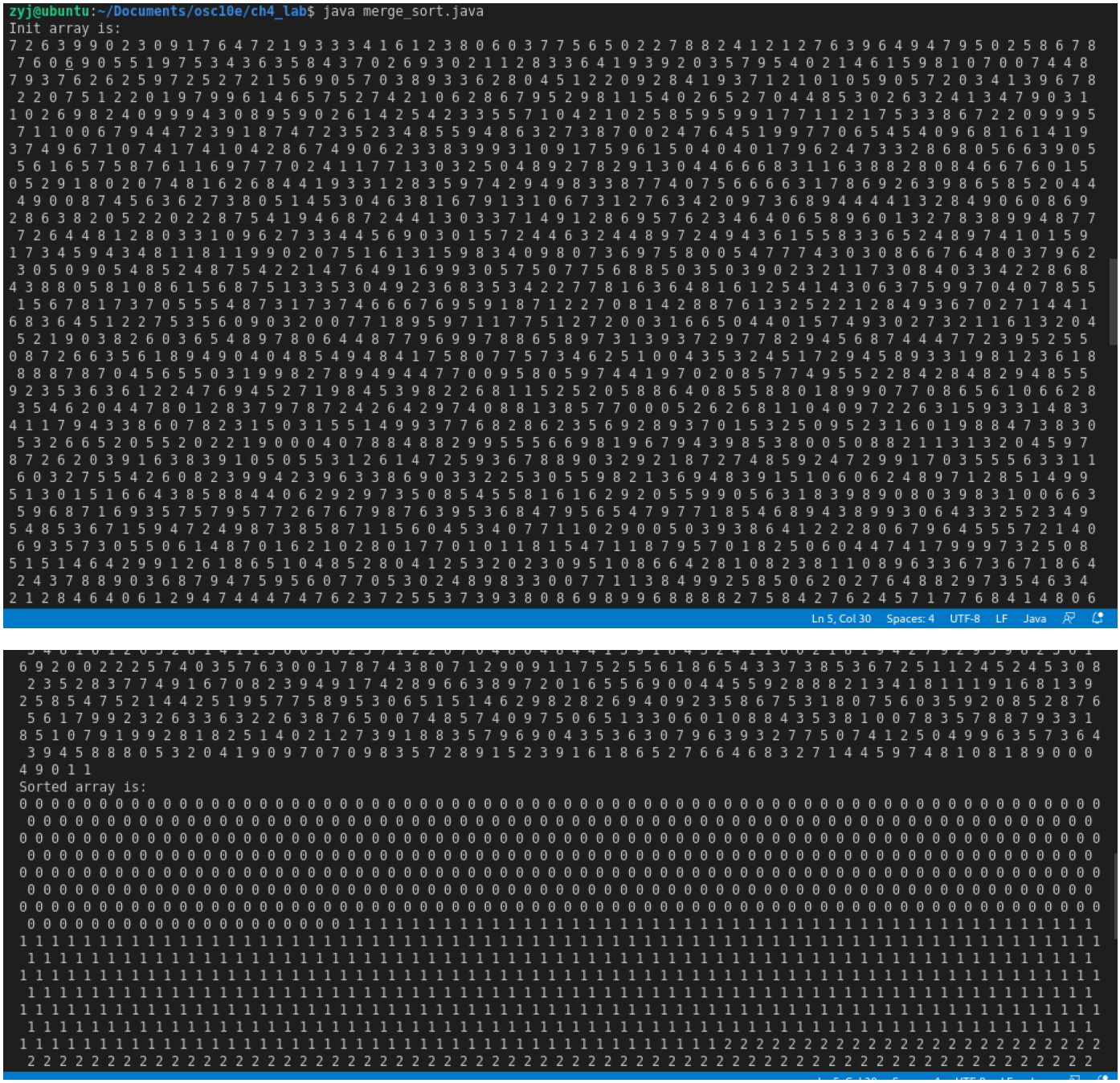
```
// use fork-join parallelism to get the array
    SortTask task = new SortTask(0, SIZE-1, array);

    pool.invoke(task);
```

Finally, we output the sorted array.

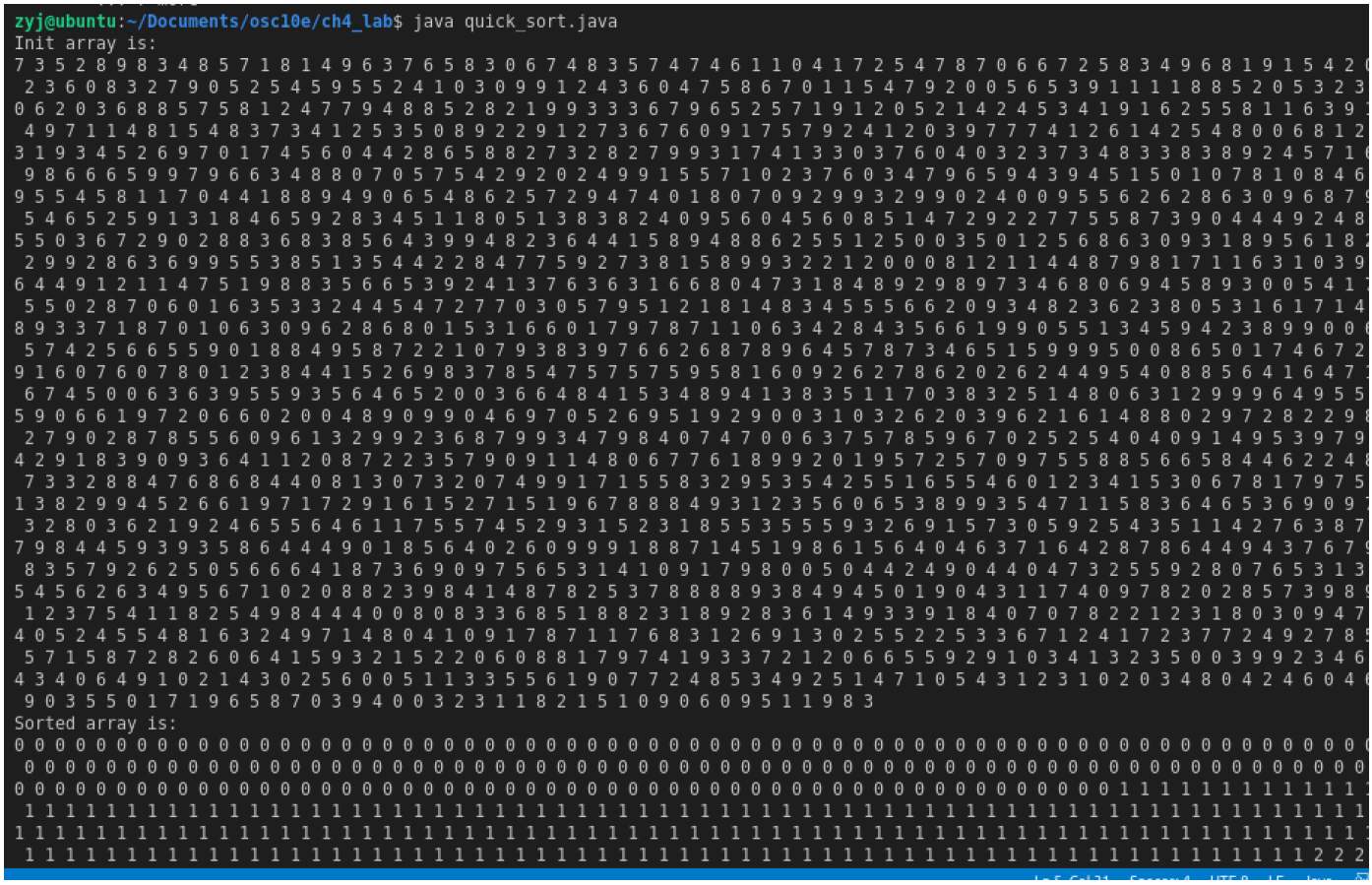
```
  System.out.println("Sorted array is: ");
        for (int i = 0; i < SIZE; i++) {
            System.out.print(array[i]);
            System.out.print(' ');
    }
```

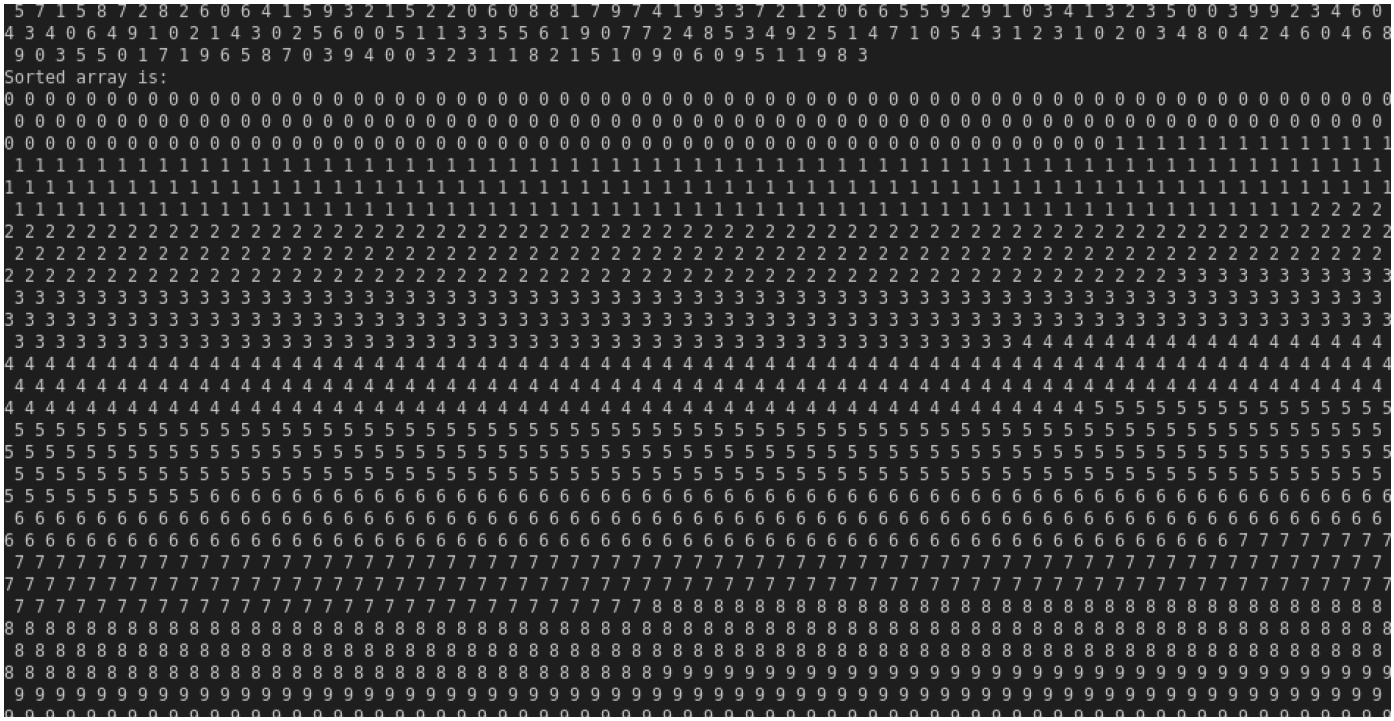
3 Program Results

Mergesort:


zyj@ubuntu:~/Documents/osc10e/ch4_lab$ java merge_sort.java
Init array is:
Sorted array is:
Ln 5, Col 30
Spaces: 4
UTF-8
LF
Java

Quicksort :





Conclusion and Thoughts

In project 3-1, we learned about using C language to implement a multithread program, based on the **pthread** library.

In project 3-2, we utilize **fork-join** in Java, similar to the multithread in C.

In the two projects, we not only learn more about **multithread** , but also have the elementary ability to write a multithread program by ourselves.

What's more, I get to know about **Java**, which I haven't used before. And I get more familar with the sorting algorithm: **quicksort, selection sort, mergesort.**

The process of finishing the project is also a process that advances my programming ability.