Project 3

Name: 刘 涵 之
ID : 519021910102

Project 3: Multithreaded Sorting Application and Fork-Join Sorting Application

1 Multithreaded Sorting Application

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 ***soring threads***) sort each sublist using a sorting algorithm of your choice. The two sublists are merged by a third thread - a ***merging thread*** - which merges the two sublists into a single sorted list.

Because global data are shared across all threads, perhaps the easiest way to set up the data is to create a global array. Each sorting thread will work on one half of this array. A second global array of the same size as the unsorted integer array will also be established. The merging thread will then merge the two sublists into this second array. Graphically, this program is structured as follows (Fig. 1).

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.

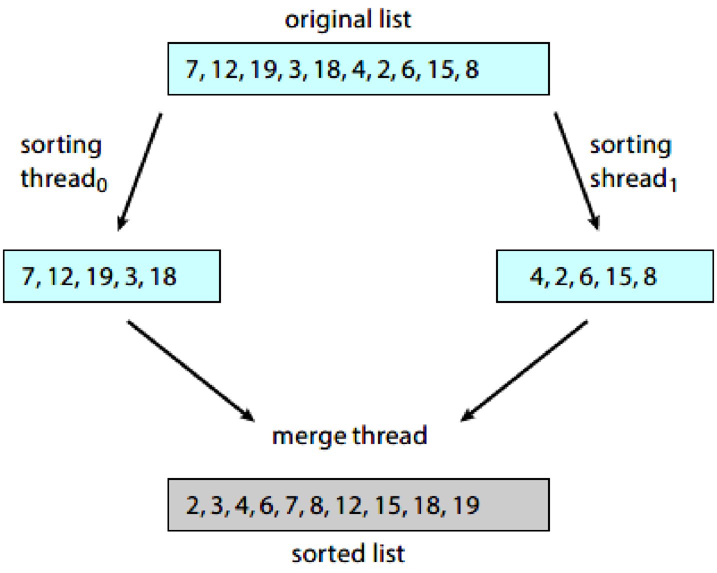


图 1: Multithreaded sorting

Design: My design for this task is shown as follows:

- Global array `array` and `result` are shared between all thread.
- First, create two threads to sort `array`. Each thread sorts a subset of the whole `array`.
- Then, **sort_thread()** function is designed to sort `array` from index **start** to index **end**. Index **start** to index **end** will be passed to it when the thread is created.

- After two sorting threads are finished (after we join them in the main()). **merge_thread()** function is used to create a thread that merge the sorting result to `result`.
- The length of `array` is given by the user and the data is generated by **rand()** function.

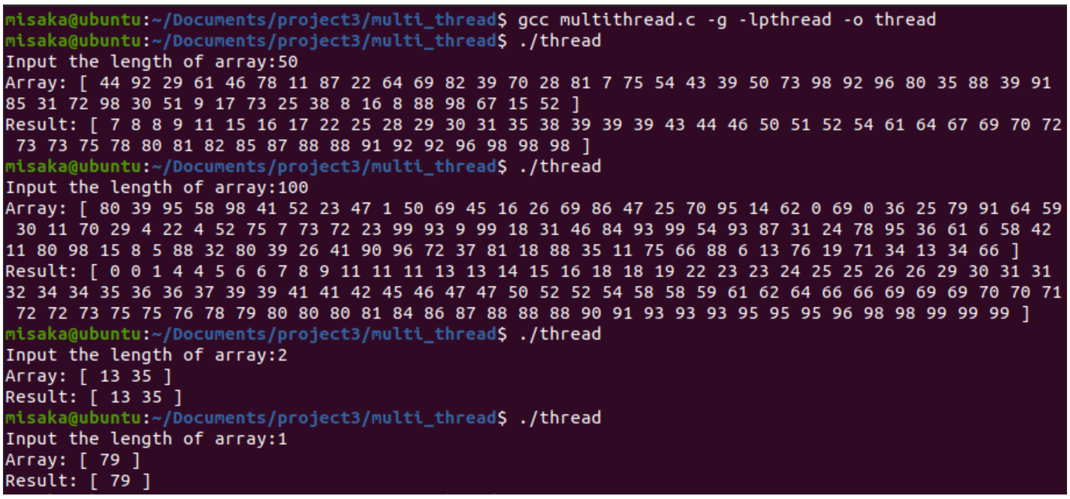
The code for this task is shown as follows.

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

int *array;
int *result;

struct function_args {
    int start, end, mid;
};

int cmp(const void *a, const void *b) {
        return *((int *)a) - *((int *)b);
}

void* sort_thread(void *arg) {
    struct function_args * args = (struct function_args *)arg;
    int start = args -> start;
    int end = args -> end;
    if (start > end) return NULL;

    qsort(array + start, end - start + 1, sizeof(int), cmp); // sort.

    return NULL;
}

void* merge_thread(void *arg) {
    struct function_args * args = (struct function_args *)arg;
    int start = args -> start;
    int mid = args -> mid;
    int end = args -> end;

    int loc1, loc2, i;
    loc1 = start;
    loc2 = mid + 1;
    i = 0;

    while (i <= end) {
        int flag = 1; // select which sub-array
        if (loc1 <= mid && (array[loc1] < array[loc2])) flag = 0;
        if (loc1 > mid) flag = 1;
        if (loc2 > end) flag = 0;
        if (flag == 0) {
            result[i++] = array[loc1++];
        } else {
            result[i++] = array[loc2++];
        }
    }
    return NULL;
}


int main() {
    int n;
    printf("Input the length of array:");
    scanf("%d", &n); // get the length of array.

    array = (int *) malloc (n * sizeof(int)); // allocate memory.
    result = (int *) malloc (n * sizeof(int));

    srand((unsigned int)time(0));
    printf("Array: [ "); // generate random array and print it.
    for(int i = 0; i < n; i++) {
        array[i] = rand() % 100;
        printf("%d ", array[i]);
    }
    printf("]\n");

    int start, mid, end; // divide array into two sub-array.
    start = 0;
    mid = n / 2;
    end = n - 1;

    struct function_args args[2]; // arguments for sort threads.
    args[0].start = start;
    args[0].end = mid;
    args[1].start = mid + 1;
    args[1].end = end;

    pthread_t sort_th[2]; // create 2 sort thread to sort sub-array.
    for (int i = 0; i < 2; i++) {
        if (pthread_create(&sort_th[i], NULL, sort_thread, &args[i])) {
            printf("Can't create thread.\n");
            return 1;
        }
    }

    for (int i = 0; i < 2; i++) { // join 2 sort thread.
        void *out;
        if (pthread_join(sort_th[i], &out)) {
            printf("Can't join thread.\n");
            return 1;
        }
    }

    args[0].start = start; // arguments for merge thread.
    args[0].mid = mid;
    args[0].end = end;

    pthread_t merge_th; // create merge thread.
    if (pthread_create(&merge_th, NULL, merge_thread, &args[0])) {
        printf("Can't create thread.\n");
        return 1;
    }

    void *out;
    if (pthread_join(merge_th, &out)) { // join merge thread.
        printf("Can't join thread.\n");
        return 1;
    }

    printf("Result: [ "); // print the result.
    for(int i = 0; i < n; i++) {
        printf("%d ", result[i]);
    }
    printf("]\n");

    free(array);
    free(result);
    return 0;
}
```

As there is only a single file, there is no need to write Makefile for it. The execution command for this task is shown as follows.

```
gcc multithread.c -g -lpthread -o thread
```

The result is shown as follows. The randomized array is sorted correctly by this program.



```
misaka@ubuntu:~/Documents/project3/multi_thread$ gcc multithread.c -g -lpthread -o thread
misaka@ubuntu:~/Documents/project3/multi_thread$ ./thread
Input the length of array:50
Array: [ 44 92 29 61 46 78 11 87 22 64 69 82 39 70 28 81 7 75 54 43 39 50 73 98 92 96 80 35 88 39 91
85 31 72 98 30 51 9 17 73 25 38 8 16 8 88 98 67 15 52 ]
Result: [ 7 8 8 9 11 15 16 17 22 25 28 29 30 31 35 38 39 39 39 43 44 46 50 51 52 54 61 64 67 69 70 72
 73 73 75 78 80 81 82 85 87 88 88 91 92 92 96 98 98 98 ]
misaka@ubuntu:~/Documents/project3/multi_thread$ ./thread
Input the length of array:100
Array: [ 80 39 95 58 98 41 52 23 47 1 50 69 45 16 26 69 86 47 25 70 95 14 62 0 69 0 36 25 79 91 64 59
 30 11 70 29 4 22 4 52 75 7 73 72 23 99 93 9 99 18 31 46 84 93 99 54 93 87 31 24 78 95 36 61 6 58 42
11 80 98 15 8 5 88 32 80 39 26 41 90 96 72 37 81 18 88 35 11 75 66 88 6 13 76 19 71 34 13 34 66 ]
Result: [ 0 0 1 4 4 5 6 6 7 8 9 11 11 11 13 13 14 15 16 18 18 19 22 23 23 24 25 25 26 26 29 30 31 31
32 34 34 35 36 36 37 39 39 41 41 42 45 46 47 47 50 52 52 54 58 58 59 61 62 64 66 66 69 69 69 70 70 71
 72 72 73 75 75 76 78 79 80 80 80 81 84 86 87 88 88 88 90 91 93 93 93 95 95 95 96 98 98 99 99 99 ]
misaka@ubuntu:~/Documents/project3/multi_thread$ ./thread
Input the length of array:2
Array: [ 13 35 ]
Result: [ 13 35 ]
misaka@ubuntu:~/Documents/project3/multi_thread$ ./thread
Input the length of array:1
Array: [ 79 ]
Result: [ 79 ]
```

图 2: Multithreaded Sorting

2 Fork-Join Sorting Application

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:

- QuickSort
- 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 source code download for this text includes Java code that provides the foundations for beginning this project.

Design: My design for this task is shown as follows:

- `RecursiveAction` is extended to implement the sorting process
- In the `compute()` function, the concrete sorting algorithm is implemented and called recursively.
- If the length of array is less than **THRESHOLD**, the bubble sort is used to sort the array.

2.1 QuickSort

The code for Fork-Join QuickSort is shown as follows.

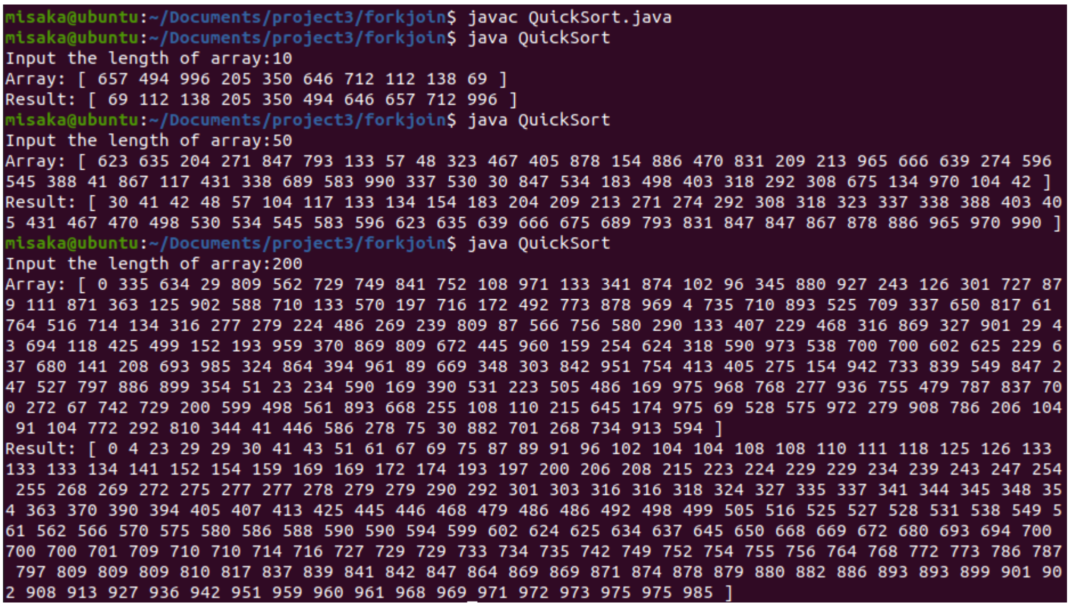
```java
import java.util.concurrent.*;
import java.util.Scanner;

public class QuickSort extends RecursiveAction {
    private static final long serialVersionUID = 1L;
    static final int THRESHOLD = 10;
    private int start, end;
    private Integer[] array;

    public QuickSort(int start, int end, Integer[] array) {
        this.start = start;
        this.end = end;
        this.array = array;
    }

    protected void compute() {
        if (end - start < THRESHOLD) {
            for (int i = start; i < end; i++) {
                for (int j = i; j <= end; j++) {
                    if (array[i].compareTo(array[j]) > 0) {
                        Integer t = array[i];
                        array[i] = array[j];
                        array[j] = t;
                    }
                }
            }
        } else { // divide into sub-sort
            int mid = start + (end - start) / 2;
            Integer comp = array[start];
            int l, h;
            l = start;
            h = end;

            while (l < h) {
                while (l < h && array[h].compareTo(comp) >= 0) h--;
                if (l < h) array[l++] = array[h];
                                while (l < h && array[l].compareTo(comp) <= 0) l++;
                                if (l < h) array[h--] = array[l];
            }
            array[l] = comp;

            QuickSort taskl = new QuickSort(start, l - 1, array);
            QuickSort taskr = new QuickSort(l + 1, end, array);

            taskl.fork();
            taskr.fork();

            taskl.join();
            taskr.join();
        }
    }

    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool();
        int n;

        Scanner input = new Scanner(System.in);
        System.out.print("Input the length of array:");
        n = input.nextInt();
        input.close();

        Integer[] array = new Integer [n];
        java.util.Random rand = new java.util.Random();

        System.out.print("Array: [ ");
        for (int i = 0; i < n; ++ i) {
            array[i] = rand.nextInt(1000);
            System.out.print(array[i] + " ");
        }
                System.out.println("]");

        QuickSort task = new QuickSort(0, n - 1, array);
        pool.invoke(task);

        System.out.print("Result: [ ");
```

```java
        for (int i = 0; i < n; ++ i) {
            System.out.print(array[i] + " ");
        }
                System.out.println("]");
    }
}
```

The program can be compiled and executed with the following commands.

```
javac QuickSort.java
java QuickSort
```

The result is shown as follows. The randomized array is sorted correctly by this program.



```
misaka@ubuntu:~/Documents/project3/forkjoin$ javac QuickSort.java
misaka@ubuntu:~/Documents/project3/forkjoin$ java QuickSort
Input the length of array:10
Array: [ 657 494 996 205 350 646 712 112 138 69 ]
Result: [ 69 112 138 205 350 494 646 657 712 996 ]
misaka@ubuntu:~/Documents/project3/forkjoin$ java QuickSort
Input the length of array:50
Array: [ 623 635 204 271 847 793 133 57 48 323 467 405 878 154 886 470 831 209 213 965 666 639 274 596
545 388 41 867 117 431 338 689 583 990 337 530 30 847 534 183 498 403 318 292 308 675 134 970 104 42 ]
Result: [ 30 41 42 48 57 104 117 133 134 154 183 204 209 213 271 274 292 308 318 323 337 338 388 403 40
5 431 467 470 498 530 534 545 583 596 623 635 639 666 675 689 793 831 847 847 867 878 886 965 970 990 ]
misaka@ubuntu:~/Documents/project3/forkjoin$ java QuickSort
Input the length of array:200
Array: [ 0 335 634 29 809 562 729 749 841 752 108 971 133 341 874 102 96 345 880 927 243 126 301 727 87
9 111 871 363 125 902 588 710 133 570 197 716 172 492 773 878 969 4 735 710 893 525 709 337 650 817 61
764 516 714 134 316 277 279 224 486 269 239 809 87 566 756 580 290 133 407 229 468 316 869 327 901 29 4
3 694 118 425 499 152 193 959 370 869 809 672 445 960 159 254 624 318 590 973 538 700 700 602 625 229 6
37 680 141 208 693 985 324 864 394 961 89 669 348 303 842 951 754 413 405 275 154 942 733 839 549 847 2
47 527 797 886 899 354 51 23 234 590 169 390 531 223 505 486 169 975 968 768 277 936 755 479 787 837 70
0 272 67 742 729 200 599 498 561 893 668 255 108 110 215 645 174 975 69 528 575 972 279 908 786 206 104
 91 104 772 292 810 344 41 446 586 278 75 30 882 701 268 734 913 594 ]
Result: [ 0 4 23 29 29 30 41 43 51 61 67 69 75 87 89 91 96 102 104 104 108 108 110 111 118 125 126 133
133 133 134 141 152 154 159 169 169 172 174 193 197 200 206 208 215 223 224 229 229 234 239 243 247 254
 255 268 269 272 275 277 277 278 279 279 290 292 301 303 316 316 318 324 327 335 337 341 344 345 348 35
4 363 370 390 394 405 407 413 425 445 446 468 479 486 486 492 498 499 505 516 525 527 528 531 538 549 5
61 562 566 570 575 580 586 588 590 590 594 599 602 624 625 634 637 645 650 668 669 672 680 693 694 700
700 700 701 709 710 710 714 716 727 729 729 733 734 735 742 749 752 754 755 756 764 768 772 773 786 787
 797 809 809 809 810 817 837 839 841 842 847 864 869 869 871 874 878 879 880 882 886 893 893 899 901 90
2 908 913 927 936 942 951 959 960 961 968 969 971 972 973 975 975 985 ]
```

图 3: Fork-Join QuickSort

2.2 MergeSort

The code for Fork-Join QuickSort is shown as follows.

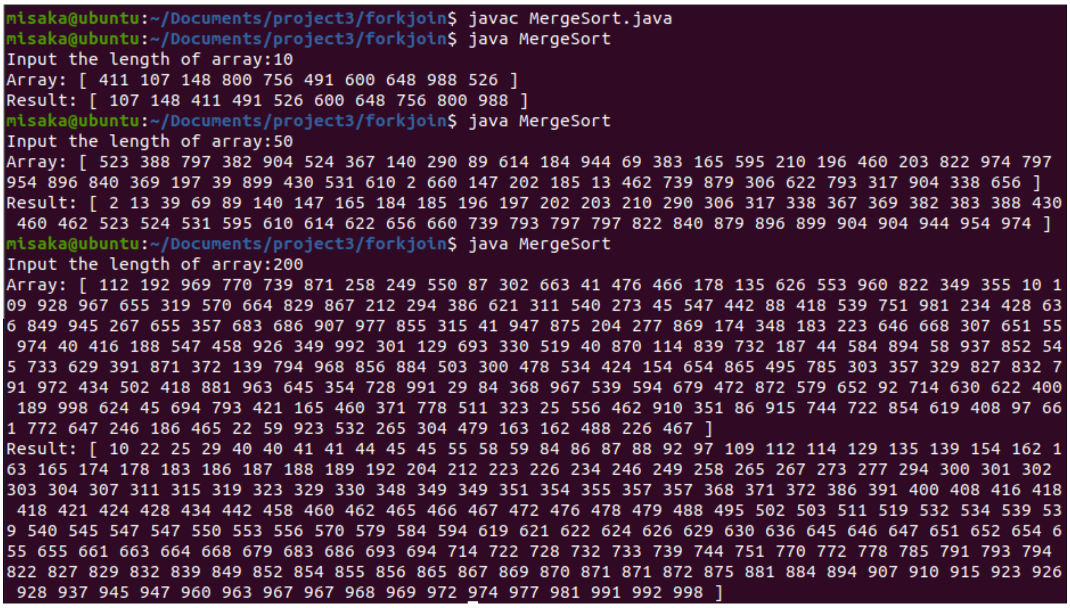
```java
import java.util.concurrent.*;
import java.util.Scanner;

public class MergeSort extends RecursiveAction {
    private static final long serialVersionUID = 1L;
    static final int THRESHOLD = 10;
    private int start, end;
    private Integer[] array;

    public MergeSort(int start, int end, Integer[] array) {
        this.start = start;
        this.end = end;
        this.array = array;
    }

    protected void compute() {
        if (end - start < THRESHOLD) {
            for (int i = start; i < end; i++) {
                for (int j = i; j <= end; j++) {
                    if (array[i].compareTo(array[j]) > 0) {
                        Integer t = array[i];
                        array[i] = array[j];
                        array[j] = t;
                    }
                }
            }
        } else { // divide into sub-merge
            int mid = start + (end - start) / 2;

            MergeSort taskl = new MergeSort(start, mid, array);
            MergeSort taskr = new MergeSort(mid + 1, end, array);

            taskl.fork();
            taskr.fork();

            taskl.join();
            taskr.join();

            Integer[] result = new Integer [end - start + 1];

            int loc1, loc2, i; // merge taskl and taskr
            loc1 = start;
            loc2 = mid + 1;
            i = 0;

            while (i < end - start + 1) {
                int flag = 1; // select which sub-array
                if (loc1 > mid) flag = 1;
                else if (loc2 > end) flag = 0;
                else if (loc1 <= mid && (array[loc1] < array[loc2])) flag = 0;
                if (flag == 0) {
                    result[i++] = array[loc1++];
                } else {
                    result[i++] = array[loc2++];
                }
            }

            for (int j = 0; j < end - start + 1; j++) { // merge array from result
                array[start + j] = result[j];
            }
        }
    }

    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool();
        int n;

```

```java
        Scanner input = new Scanner(System.in);
        System.out.print("Input the length of array:");
        n = input.nextInt();
        input.close();

        Integer[] array = new Integer [n];
        java.util.Random rand = new java.util.Random();

        System.out.print("Array: [ ");
        for (int i = 0; i < n; ++ i) {
            array[i] = rand.nextInt(1000);
            System.out.print(array[i] + " ");
        }
                System.out.println("]");

        MergeSort task = new MergeSort(0, n - 1, array);
        pool.invoke(task);

        System.out.print("Result: [ ");
        for (int i = 0; i < n; ++ i) {
            System.out.print(array[i] + " ");
        }
                System.out.println("]");
    }
}
```

The program can be compiled and executed with the following commands.

```
javac MergeSort.java
java MergeSort
```

The result is shown as follows. The randomized array is sorted correctly by this program.



```
misaka@ubuntu:~/Documents/project3/forkjoin$ javac MergeSort.java
misaka@ubuntu:~/Documents/project3/forkjoin$ java MergeSort
Input the length of array:10
Array: [ 411 107 148 800 756 491 600 648 988 526 ]
Result: [ 107 148 411 491 526 600 648 756 800 988 ]
misaka@ubuntu:~/Documents/project3/forkjoin$ java MergeSort
Input the length of array:50
Array: [ 523 388 797 382 904 524 367 140 290 89 614 184 944 69 383 165 595 210 196 460 203 822 974 797
954 896 840 369 197 39 899 430 531 610 2 660 147 202 185 13 462 739 879 306 622 793 317 904 338 656 ]
Result: [ 2 13 39 69 89 140 147 165 184 185 196 197 202 203 210 290 306 317 338 367 369 382 383 388 430
 460 462 523 524 531 595 610 614 622 656 660 739 793 797 797 822 840 879 896 899 904 904 944 954 974 ]
misaka@ubuntu:~/Documents/project3/forkjoin$ java MergeSort
Input the length of array:200
Array: [ 112 192 969 770 739 871 258 249 550 87 302 663 41 476 466 178 135 626 553 960 822 349 355 10 1
09 928 967 655 319 570 664 829 867 212 294 386 621 311 540 273 45 547 442 88 418 539 751 981 234 428 63
6 849 945 267 655 357 683 686 907 977 855 315 41 947 875 204 277 869 174 348 183 223 646 668 307 651 55
 974 40 416 188 547 458 926 349 992 301 129 693 330 519 40 870 114 839 732 187 44 584 894 58 937 852 54
5 733 629 391 871 372 139 794 968 856 884 503 300 478 534 424 154 654 865 495 785 303 357 329 827 832 7
91 972 434 502 418 881 963 645 354 728 991 29 84 368 967 539 594 679 472 872 579 652 92 714 630 622 400
 189 998 624 45 694 793 421 165 460 371 778 511 323 25 556 462 910 351 86 915 744 722 854 619 408 97 66
1 772 647 246 186 465 22 59 923 532 265 304 479 163 162 488 226 467 ]
Result: [ 10 22 25 29 40 40 41 41 44 45 45 55 58 59 84 86 87 88 92 97 109 112 114 129 135 139 154 162 1
63 165 174 178 183 186 187 188 189 192 204 212 223 226 234 246 249 258 265 267 273 277 294 300 301 302
303 304 307 311 315 319 323 329 330 348 349 349 351 354 355 357 357 368 371 372 386 391 400 408 416 418
 418 421 424 428 434 442 458 460 462 465 466 467 472 476 478 479 488 495 502 503 511 519 532 534 539 53
9 540 545 547 547 550 553 556 570 579 584 594 619 621 622 624 626 629 630 636 645 646 647 651 652 654 6
55 655 661 663 664 668 679 683 686 693 694 714 722 728 732 733 739 744 751 770 772 778 785 791 793 794
822 827 829 832 839 849 852 854 855 856 865 867 869 870 871 871 872 875 881 884 894 907 910 915 923 926
 928 937 945 947 960 963 967 967 968 969 972 974 977 981 991 992 998 ]
```

图 4: Fork-Join MergeSort