3100 Assignment1 Report

Zhixuan LIU

118010202

1 Introduction

In the previous study in CSC3100, we learned a so-called "divide and conquer" method and how to apply it into sorting algorithm. Furthermore, we also learned some linear structures. In this assignment, we will mainly focus on those two great idea and implement them in to "merge sort" problem and "Prefix" problem.

2 Merge Sort

In this problem, we should use merge sort to sort a sequence of numbers.

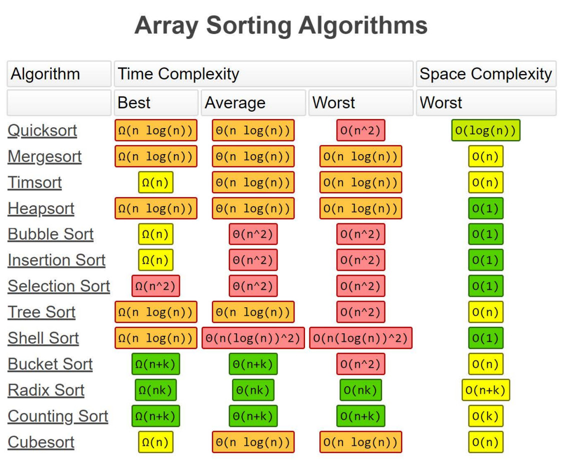
In sorting n objects, merge sort has an average and worst-case performance of $O(nlogn)$. If the running time of merge sort for a list of length n is $T(n)$, then the recurrence $T(n) = 2T(\frac{1}{2}) + n$ follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the n steps taken to merge the resulting two lists). The closed form follows from the master theorem for divide-and-conquer recurrences.

In the worst case, merge sort does about 39 percent fewer comparisons than quicksort does in the average case. In terms of moves, merge sort's worst case complexity is O(n log n)—the same complexity as quicksort's best case, and merge sort's best case takes about half as many iterations as the worst case.

2.1 Possible Solutions for this Problem

There are several different algorithm to implement merge sort with their pros and cons. The method taught in the lecture is Top-Down implementation. Main idea of merge sort contains two steps: First, Divide the unsorted list into n sublists, each containing one element (a list of one element is considered sorted). Second, Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.

We can use recursive method and non-recursive method to realize this idea.



Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Figure 1: Time and Space Complexity of sort algorithms.

2.2 My Solution for this Problem

I use the recursive way to solve the problem.

First, i get the number of elements that needed to be sorted by "reader.nextInt()". Then i create an array to contain those numbers. I also use "reader.nextInt()" to catch the input element, and put them into my array one by one.

```
        Reader in = new Reader();
        int k = in.nextInt();

        // Use .nextInt() to get next integer
        long arr[] = new long[k];
        for (int i = 0; i < k; i++) {
            arr[i] = in.nextInt();
        }
```

Next, i call the merge sort function. I implement merge sort function recursively. The base case of this recursive function is when there is only one element in the array. Otherwise, i implement merge_sort function to the left half and right half of the array separately. After the left and right half are both sorted, i call the merge function.

```
public static void merge_sort(long[] A,
            int left, int right){
        if(right>left) {
                int center = (left + right)/2;

                //divide
```

```
                        merge_sort(A, left,center);
                        merge_sort(A, center+1, right);

                        //conquer
                        merge(A,left,center,right);
                }
                else {
                        return;
                }
}
```

I have two index to go through those two arrays and let the smaller one to go back to the previous array first.

```
private static void merge(long[] A, int left,
                        int center, int right) {
                int n1 = center - left + 1;
                int n2 = right - center;
                long[] L = new long[n1+1];
                long[] R = new long[n2+1];
                for (int i=0; i<n1; i++) {
                        L[i] = A[left+i];
                }
                for (int j=0; j<n2; j++) {
                        R[j]=A[center+1+j];
                }
                L[n1] = 9223372036854775807L;
                R[n2] = 9223372036854775807L;
                int m = 0;
                int n = 0;
                for(int k=left; k<=right; k++) {
                        if(L[m]<=R[n]) {
                                A[k] = L[m];
                                m++;
                        }else {
                                A[k] = R[n];
                                n++;
                        }
                }
}
```

2.3 Why My Solution is Better

In other cases, i use non-recursive method to implement merge sort problem. Different from recursive method, which is a Top-Down implementation, the non-recursive is a Down-to-Top method. In the merge_sort function in the non-recursive algorithm, at first step, we only have one element, and the length of the data is gradually multiplied by two from the beginning to reach the length of the original array. The sample codes are shown below:

```java
public static void merge_sort(long[] arr) {

        int len = 1;

        while (arr.length > len) {

            for (int i = 0;
                i + len <= arr.length - 1;
                i += len * 2) {
                int left = i;
                int mid = i + len - 1;
                int right = i + len * 2 - 1;

                if (right > arr.length - 1)
                    right = arr.length - 1;

                merge(arr, left, mid, right);
            }
            len *= 2;
        }
}
```

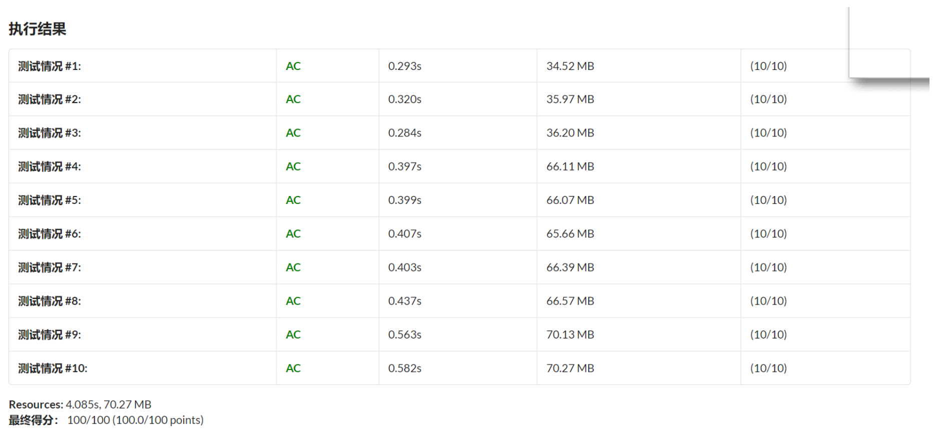
2.3.1 Methods Analysis

The recursive and non-recursive merge sort algorithm have same implementation idea. During each pass, the array is divided into blocks of size m. (Initially, m = 1). Every two adjacent blocks are merged (as in normal merge sort), and the next pass is made with a twice larger value of m. Both recursive and non-recursive merge sort have same time complexity of O(nlog(n)). We can see the data from LGU Online Judge platform:

We can see that recursive algorithm use less time than non-recursive algorithm in all testing cases. Moreover, the structure of recursive method is easy to maintained. Therefore, the recursive method is better than non-recursive method.

2.3.2 Other Issues

In my test function, i used $scanf()$ function to get the input from user. However, this method did not pass LGU Online Judgement. This is due to the great time

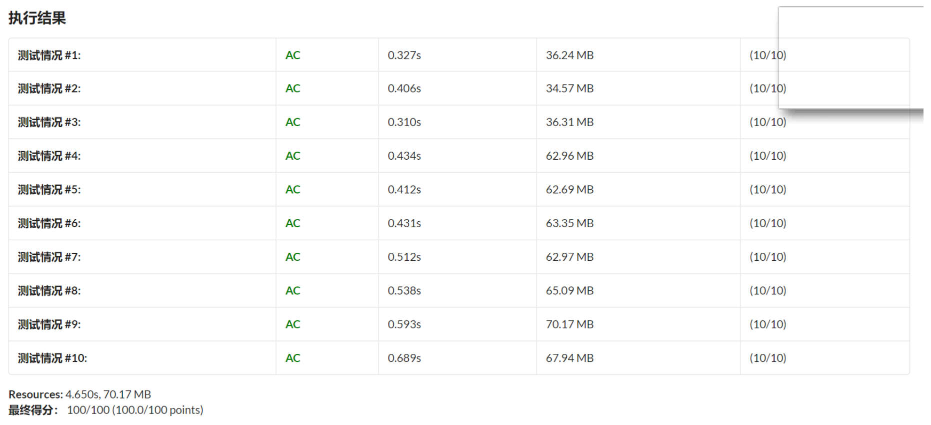


执行结果

测试情况 #1:	AC	0.293s	34.52 MB	(10/10)
测试情况 #2:	AC	0.320s	35.97 MB	(10/10)
测试情况 #3:	AC	0.284s	36.20 MB	(10/10)
测试情况 #4:	AC	0.397s	66.11 MB	(10/10)
测试情况 #5:	AC	0.399s	66.07 MB	(10/10)
测试情况 #6:	AC	0.407s	65.66 MB	(10/10)
测试情况 #7:	AC	0.403s	66.39 MB	(10/10)
测试情况 #8:	AC	0.437s	66.57 MB	(10/10)
测试情况 #9:	AC	0.563s	70.13 MB	(10/10)
测试情况 #10:	AC	0.582s	70.27 MB	(10/10)

Resources: 4.085s, 70.27 MB
最终得分: 100/100 (100.0/100 points)

Figure 2: This figure shows the time taken by recursive algorithm.



执行结果

测试情况 #1:	AC	0.327s	36.24 MB	(10/10)
测试情况 #2:	AC	0.406s	34.57 MB	(10/10)
测试情况 #3:	AC	0.310s	36.31 MB	(10/10)
测试情况 #4:	AC	0.434s	62.96 MB	(10/10)
测试情况 #5:	AC	0.412s	62.69 MB	(10/10)
测试情况 #6:	AC	0.431s	63.35 MB	(10/10)
测试情况 #7:	AC	0.512s	62.97 MB	(10/10)
测试情况 #8:	AC	0.538s	65.09 MB	(10/10)
测试情况 #9:	AC	0.593s	70.17 MB	(10/10)
测试情况 #10:	AC	0.689s	67.94 MB	(10/10)

Resources: 4.650s, 70.17 MB
最终得分: 100/100 (100.0/100 points)

Figure 3: This figure shows the time taken by non-recursive algorithm.

consumption of *scanf*() function. Therefore, i changed to a fast reader and i use the *nextInt*() to get the integer from user.

```
public int nextInt() throws IOException {
    int ret = 0;
    byte c = read();
    while (c <= ' ')
        c = read();
    boolean neg = (c == '-');
    if (neg)
        c = read();
    do {
        ret = ret * 10 + c - '0';
    } while ((c = read()) >= '0' && c <= '9');
        if (neg)
        return -ret;
    return ret;
}
```

This method helps us to path the time limitation.

2.4 How I Test My Program

I first tested it on my platform. Then sent to LGU Online Judge platform to test the whether my program passed all tests.

2.5 Possible Further Improvement

2.5.1 Memory space

For both of my merge sort algorithms, they are not in-place sorted. This will cause memory storage problem. Further improvement can focus on how to implment merge sort into a in-place sorting algorithm, which will save a lot of memory space.

2.5.2 I/O Time Consumption

In my program, i use *System.out.print*() function to do the output. However, it is still not fast enough. Therefore, to further improve the algorithm, a more efficient output method is needed.

2.5.3 Time Improvement

Another way to improve merge sort is to use insertion sort on small subarrays. We can improve most recursive algorithms by handling small cases differently. Switching to insertion sorting of small sub-arrays will shorten the drive time for typical implementations.

3 Prefix

In this problem, we will implement prefix expression to a sequence of numbers.

3.1 Possible Solution for this Problem

The main idea is to use a stack to implement this problem and this can be divided into several steps: First, put a pointer P at the end of the end. Second,if character at P is an operand push it to Stack. Third, if the character at P is an operator pop two elements from the Stack. Operate on these elements according to the operator, and push the result back to the Stack. Forth, decrement P by 1 and go to Step 2 as long as there are characters left to be scanned in the expression. Fifth, the Result is stored at the top of the Stack, return it.

Another way to solve the prefix problem is to simply use the array to solve.

3.2 My Solution

I use array to implement prefix instead of using stack. In my main function, first i use *BufferReader* and *readLine*() method to get the number of imput element and store it into k. Then i create an array with length k to store the input number in the term of string.

```java
BufferedReader in = new BufferedReader
              (new InputStreamReader(System.in));
int k = Integer.valueOf(in.readLine());
int size = -1;
long a;
long b;
long c;
// all the string is stored in the init_arr
String init_arr[] = new String[k];
String init_target;
for(int j = 0 ; j<k ; j++) {
          init_target = in.readLine();
     init_arr[j] = init_target;
}
```

Then i use an array *arr*[] to implement the function of stack. I have a pivot to store the exact size of the array. To begin with, I select a pivot from the end of previous array to the start. If this pivot is a number, then put it into the array. I it is a operator, select the last two element from *arr*[] and implement the operator on them. After that, put the result into the array.

```java
long arr[] = new long [k];
long num_target;
String target;
```

```
for (int i = init_arr.length-1; i>=0; i--) {
        target = init_arr[i];


        if(target.charAt(0) == '+') {
                // get the + operation
                if(size <= 0) {
                        System.out.print
                                ("Invalid\n");
                        return;
                }
                else {
                        // there are at least
                        //two element in an array
                        a = arr[size-1];
                        b = arr[size];
                        c = Math.floorMod(a+b,
                                    1000000007L);
                        size -= 1;
                        arr[size] = c;
                }
        }
        else if(target.charAt(0)=='-') {
                //get a - operation
                if(size <= 0) {
                        System.out.print
                                ("Invalid\n");
                        return;
                }
                else {
                        a = arr[size-1];
                        b = arr[size];
                        c = Math.floorMod(b-a,
                                    1000000007L);
                        size -= 1;
                        arr[size] = c;
                }
        }
        else if(target.charAt(0)=='*') {
                //get a - operation
                if(size <= 0) {
                        System.out.print
                                ("Invalid\n");
                        return;
                }
                else {
```

```
                                a = arr[size -1];
                                b = arr[size];
                                c = Math.floorMod(a*b,
                                            1000000007L);
                                size -= 1;
                                arr[size] = c;
                        }
                }
                else {
                        num_target = Long.valueOf(target);
                        size += 1;
                        arr[size] = Math.floorMod
                                    (num_target,1000000007L);
                }
}
```

3.3 Why my Solution is Better

The main idea to implement a stack to this problem is as follows:

```
for (int j = exprsn.length() - 1; j >= 0; j--) {

        // Push operand to Stack
        // To convert exprsn[j] to digit subtract
        // '0' from exprsn[j].
        if (isOperand(exprsn.charAt(j)))
                Stack.push((double)(exprsn.charAt(j) - 48));

        else {

                // Operator encountered
                // Pop two elements from Stack
                double o1 = Stack.peek();
                Stack.pop();
                double o2 = Stack.peek();
                Stack.pop();

                // Use switch case to operate on o1
                // and o2 and perform o1 O o2.
                switch (exprsn.charAt(j)) {
                case '+':
                        Stack.push(o1 + o2);
                        break;
                case '-':
                        Stack.push(o1 - o2);
```

```
                            break;
                    case '*':
                            Stack.push(o1 * o2);
                            break;
                    }
            }
}
```

Instead of using linked list to implement stack, i use array to have the same function as stack. For both of them, the complexity is $O(N)$, but the array method is more easy to understand.

3.4 How to Test my Program

I first tested it on my platform. Then sent to LGU Online Judge platform to test the whether my program passed all tests.

3.5 Possible Further Improvement

3.5.1 Algorithm Improvement

In both methods mentioned above, the calculation are implemented only after we get all the numbers from the user. Then implement the calculation from the end to the start. This is a waste of time. Therefore, if we can solve the problem in order. In other words, we can perform calculation while user input, this will save lots of time and also save memory spaces.

3.5.2 I/O Time Consumption

To further improve the running time, a faster scanner is needed to improve the time by reader.