My Java Solution to share

103

R ran3
Reputation: ★ 151

The idea here is basically implement a String comparator to decide which String should come first during concatenation. Because when you have 2 numbers (let's convert them into String), you'll face only 2 cases:
For example:

```
String s1 = "9";
String s2 = "31";

String case1 =  s1 + s2; // 931
String case2 = s2 + s1; // 319
```

Apparently, case1 is greater than case2 in terms of value.
So, we should always put s1 in front of s2.

I have received many good suggestions from you in this discussion. Below is the modified version of codes based on your suggestions:

```
public class Solution {
     public String largestNumber(int[] num) {
                if(num == null || num.length == 0)
                    return "";

                // Convert int array to String array, so we can sort later on
                String[] s_num = new String[num.length];
                for(int i = 0; i < num.length; i++)
                    s_num[i] = String.valueOf(num[i]);

                // Comparator to decide which string should come first in concatenation
                Comparator<String> comp = new Comparator<String>(){
                    @Override
                    public int compare(String str1, String str2){
                        String s1 = str1 + str2;
                        String s2 = str2 + str1;
                        return s2.compareTo(s1); // reverse order here, so we can do append() lat
                    }
                };

                Arrays.sort(s_num, comp);
                // An extreme edge case by lc, say you have only a bunch of 0 in your int array
                if(s_num[0].charAt(0) == '0')
                    return "0";

                StringBuilder sb = new StringBuilder();
                for(String s: s_num)
                    sb.append(s);

                return sb.toString();


        }
}
```

In terms of Time and Space Complexity:
Let's assume:
the length of input array is n,
average length of Strings in s_num is k,
Then, compare 2 strings will take O(k).
Sorting will take O(nlgn)
Appending to StringBuilder takes O(n).
So total will be O(n*k*lgnk) + O(n) = O(n*k*lgnk).

Space is pretty straight forward: O(n).

0

wgtmac
Reputation: ★ 0

I wrote almost the same as you.

You can put
if(sb.charAt(0)=='0')
return "0";
before
StringBuilder sb = new StringBuilder();
for(String s: Snum)
sb.insert(0, s);
It is better :)

0

ran3
Reputation: ★ 151

I've changed the code according to your suggestion.
Thanks.

14

bing10
Reputation: ★ 22

```
sort the opposite way
return s2.compareTo(s1);

which makes this easier
if(Snum[0]=="0") return "0"; //if intteger was 0. the string has to be "0"

and this faster
sb.append(s);
```

0

ran3 ↩ @bing10
Reputation: ★ 151

You are totally right on this, append is faster than insert in StringBuilder Operations.
But I may still keep the original solution for easier understanding.

Thanks,
Ryan

0

dAb0 ↩ @bing10
Reputation: ★ 21

Or probably we could implement the comparator as a reverse comparator which means we return s2.compareTo(s1) so in this case we can apply append as you wish and also have a check at the first element is "0" or not.

0

zealot @bing10
Reputation: ★ 28

There is a mistake, try this: if(Snum[0].equals("0")) return "0";

0

qlan2
Reputation: ★ 49

This post is deleted!

0

amay
Reputation: ★ 0

excellent code

0

mouse77
Reputation: ★ 0

what's the time complexity of this algorithm??

▲ 0 ▼

CodingGeek
Reputation: ★ 2

Same code by user Daring with comments : https://leetcode.com/discuss/21488/java-code-by-providing-comparator-with-explaination-nlogn

▲ 0 ▼

VectorChange
Reputation: ★ 1

Compare between the string needs O(len) and sort needs O(len*log(len)), the time complexity is O(len^2*log(len))

▲ 0 ▼

codecola
Reputation: ★ 5

may be Integer.toString(num[i]) is better than num[i] + ""

▲ 2 ▼

fun4LeetCode
Reputation: ★ 1247

I would recommend using the following comparator if you wish to reduce memory footprint, since the comparator above generates a lot of concatenated strings (of the order of O(n^2), with n the total number of elements). Also for large-size input arrays, the following comparator will have a smaller chance to trigger GC, which is detrimental to the time performance.

```
Comparator<String> cmp = new Comparator<String>() {
        @Override
        public int compare(String str1, String str2) {
                sb1.delete(0, sb1.length()).append(str1).append(str2);
                sb2.delete(0, sb2.length()).append(str2).append(str1);

                for (int i = 0; i < sb1.length(); i++) {
                        if (sb1.charAt(i) == sb2.charAt(i)) continue;
                        return (sb1.charAt(i) > sb2.charAt(i) ? -1 : 1);
                }

                return 0;
        }
};
```

I assume you have initialized two final StringBuilder objects sb1 and sb2 somewhere before the comparator.

▲ 0 ▼

F **freezaku**

Reputation: ★ 0

This post is deleted!

▲ 0 ▼

S **saharH** ↰ @ran3

Reputation: ★ 0

@ran3 could you explain how you get nklognk ?

▲
0
▼

T **touchdown**
Reputation: ★ 5

Great Solution! You have a deep understanding of Arrays.sort()

▲
1
▼

W **whglamrock**
Reputation: ★ 19

Maybe the time complexity is O(knlogn)? The typical sort takes O(nlogn) when the comparison between two elements takes O(1). Then in this case each comparison takes O(k), I guess the overall time complexity should be O(knlogn)?

▲
0
▼

T **tankztc** ↰ @whglamrock
Reputation: ★ 52

@whglamrock I think you're right, I've the same opinion. Typical sort like merge sort have O(logn) level, and each level have O(n) times compare, so the overall time complexity is O(knlogn)

19
POSTS

10588
VIEWS

Log in to reply