Previous (/articles/find-the-town-judge/) Next (/articles/search-insert-position/)

451. Sort Characters By Frequency (/problems/sort-characters-by-frequency/)

Feb. 29, 2020 | 5.1K views

Average Rating: 5 (19 votes)

Given a string, sort it in decreasing order based on the frequency of characters.

Example 1:

```
Input:
"tree"

Output:
"eert"

Explanation:
'e' appears twice while 'r' and 't' both appear once.
So 'e' must appear before both 'r' and 't'. Therefore "eetr" is also a valid ans
```

Example 2:

```
Input:
"cccaaa"

Output:
"cccaaa"

Explanation:
Both 'c' and 'a' appear three times, so "aaaccc" is also a valid answer.
Note that "cacaca" is incorrect, as the same characters must be together.
```

Example 3:

```
Input:
"Aabb"

Output:
"bbAa"

Explanation:
"bbaA" is also a valid answer, but "Aabb" is incorrect.
Note that 'A' and 'a' are treated as two different characters.
```

Solution

Remember, Strings are Immutable!

The input type for this question is a `String`. When dealing with `String`s, we need to be careful to not inadvertently convert what should have been an $O(n)$ algorithm into an $O(n^2)$ one.

`String`s in most programming languages are **immutable**. This means that once a `String` is created, we cannot modify it. We can only create a new `String`. Consider the following Java code.

```
String a = "Hello ";
a += "Leetcode";
```

This code creates a `String` called `a` with the value `"Hello "`. It then sets `a` to be a *new* `String`, made with the letters from the old `a` and the additional letters `"Leetcode"`. It then assigns this new `String` to the variable `a`, throwing away the reference to the old one. It does NOT actually "modify" `a`.

For the most part, we don't run into problems with `String`s being treated like this. But consider this code for *reversing* a `String`.

```java
String s = "Hello There";
String reversedString = "";
for (int i = s.length() - 1; i >= 0; i--) {
    reversedString += s.charAt(i);
}
System.out.println(reversedString);
```

Each time a character is added to `reverseString`, a *new String* is created. Creating a new `String` has a cost of n, where n is the length of the `String`. The result? Simply reversing a `String` has a cost of $O(n^2)$ using the above algorithm.

The solution is to use a `StringBuilder`. A `StringBuilder` collects up the characters that will be converted into a `String` so that only one `String` needs to be created—once all the characters are ready to go. Recall that inserting an item at the end of an `Array` has a cost of $O(1)$, and so the total cost of inserting the n characters into the `StringBuilder` is $O(n)$, and it is also $O(n)$ to then convert that `StringBuilder` into a `String`, giving a total of $O(n)$.

```java
String s = "Hello There";
StringBuilder sb = new StringBuilder();
for (int i = s.length() - 1; i >= 0; i--) {
    sb.append(s.charAt(i));
}
String reversedString = sb.toString();
System.out.println(reversedString);
```

If you're unsure what to do for your particular programming language, it shouldn't be too difficult to find using a web search. The algorithms provided in the solutions here all do string building efficiently.

Approach 1: Arrays and Sorting

Intuition

In order to sort the characters by frequency, we firstly need to know how many of each there are. One way to do this is to sort the characters by their numbers so that identical characters are side-by-side (all characters in a programming language are identified by a unique number). Then, knowing how many times each appears will be a lot easier.

Because `String` s are **immutable** though, we cannot sort the `String` directly. Therefore, we'll need to start by converting it *from a* `String` *to an Array of* characters.

"welcometoleetcode" → **To Array** → ['w', 'e', 'l', 'c', 'o', 'm', 'e', 't', 'o', 'l', 'e', 'e', 't', 'c', 'o', 'd', 'e']

Now that we have an `Array` , we can sort it, which will make all identical characters side-by-side.

['w', 'e', 'l', 'c', 'o', 'm', 'e', 't', 'o', 'l', 'e', 'e', 't', 'c', 'o', 'd', 'e'] → **Sorted** → ['c', 'c', 'd', 'e', 'e', 'e', 'e', 'e', 'l', 'l', 'm', 'o', 'o', 'o', 't', 't', 'w']

There are a few different ways we can go from here. One easy-to-understand way is to create a *new* `Array` of `String` s. Each `String` in the list will consist of one of the unique characters from the sorted characters `Array` .

['c', 'c', 'd', 'e', 'e', 'e', 'e', 'e', 'l', 'l', 'm', 'o', 'o', 'o', 't', 't', 'w'] → **Group** → ["cc", "d", "eeeee", "ll", "m", "ooo", "tt", "w"]

Remember: do this process using `StringBuilder` s, not naïve `String` appending! (See the first section of this article if you're confused).

The next step is to sort this `Array` of `String` s by length. To do this, we'll need to implement a suitable **Comparator**. Recall that there is *no requirement* for characters of the same frequency to appear in a specific order.

["cc", "d", "eeeee", "ll", "m", "ooo", "tt", "w"] → **Length sort** → ["eeeee", "ooo", "cc", "ll", "tt", "d", "m", "w"]

Finally, we can convert this `Array` of `String` s into a single `String` . In Java, this can be done by passing the `Array` into a `StringBuilder` and then calling `.toString(...)` on it.

["eeeee", "ooo", "cc", "ll", "tt", "d", "m", "w"] → **String build** → "eeeeeooocclttdmw"

Algorithm

Java Python Copy

```
def frequencySort(self, s: str) -> str:
    if not s: return s

    # Convert s to a list.
    s = list(s)

    # Sort the characters in s.
    s.sort()

    # Make a list of strings, one for each unique char.
    all_strings = []
    cur_sb = [s[0]]
    for c in s[1:]:
        # If the last character on string builder is different...
        if cur_sb[-1] != c:
            all_strings.append("".join(cur_sb))
            cur_sb = []
        cur_sb.append(c)
    all_strings.append("".join(cur_sb))

    # Sort the strings by length from *longest* to shortest.
    all_strings.sort(key=lambda string : len(string), reverse=True)

    # Convert to a single string to return.
    # Converting a list of strings to a string is often done
    # using this rather strange looking python idiom.
    return "".join(all_strings)
```

Complexity Analysis

Let n be the length of the input `String`.

- Time Complexity : $O(n \log n)$.

  The first part of the algorithm, converting the `String` to a `List` of characters, has a cost of $O(n)$, because we are adding n characters to the end of a `List`.

  The second part of the algorithm, sorting the `List` of characters, has a cost of $O(n \log n)$.

  The third part of the algorithm, grouping the characters into `String`s of similar characters, has a cost of $O(n)$ because each character is being inserted once into a `StringBuilder` and once converted into a `String`.

  Finally, the fourth part of the algorithm, sorting the `String`s by length, has a worst case cost of $O(n)$, which occurs when *all* the characters in the input `String` are unique.

Because we drop constants and insignificant terms, we get $O(n \log n) + 3 \cdot O(n) = O(n \log n)$.

Be careful with your own implementation—if you didn't do the string building process in a sensible way, then your solution could potentially be $O(n^2)$.

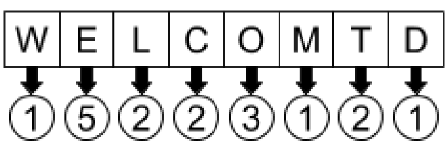
- Space Complexity : $O(n)$.

  It is impossible to do better with the space complexity, because `String` s are immutable. The `List` of characters, `List` of `String` s, and the final output `String` , are all of length n, so we have a space complexity of $O(n)$.

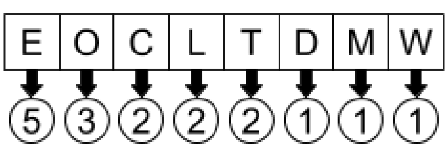
Approach 2: HashMap and Sort

Intuition

Another approach is to use a `HashMap` to count how many times each character occurs in the `String` ; the keys are characters and the values are frequencies.



Next, extract a copy of the *keys* from the `HashMap` and *sort* them by frequency using a `Comparator` that looks at the `HashMap` values to make its decisions.



Finally, initialise a new `StringBuilder` and then iterate over the list of sorted characters (sorted by frequency). Look up the values in the `HashMap` to know how many of each character to append to the `StringBuilder` .

Algorithm

Java Python Copy

```python
def frequencySort(self, s: str) -> str:

    # Count up the occurances.
    counts = collections.Counter(s)

    # Build up the string builder.
    string_builder = []
    for letter, freq in counts.most_common():
        # letter * freq makes freq copies of letter.
        # e.g. "a" * 4 -> "aaaa"
        string_builder.append(letter * freq)
    return "".join(string_builder)
```

Complexity Analysis

Let n be the length of the input `String` and k be the number of unique characters in the `String`.

We know that $k \leq n$, because there can't be more unique characters than there are characters in the `String`. We also know that k is somewhat bounded by the fact that there's only a finite number of characters in Unicode (or ASCII, which I suspect is all we need to worry about for this question).

There are two ways of approaching the complexity analysis for this question.

1. We can disregard k, and consider that *in the worst case,* `k = n`.
2. We can consider k, recognising that the number of unique characters is not infinite. This is more accurate for real world purposes.

I've provided analysis for both ways of approaching it. I choose not to bring it up for the previous approach, because it made no difference there.

- Time Complexity : $O(n \log n)$ OR $O(n + k \log k)$.

  Putting the charactersss into the `HashMap` has a cost of $O(n)$, because each of the n charactersss must be put in, and putting each in is an $O(1)$ operation.

  Sorting the `HashMap` keys has a cost of $O(k \log k)$, because there are k keys, and this is the standard cost for sorting. If only using n, then it's $O(n \log n)$. For the previous question, the sort was carried out on n items, not k, so was possibly a *lot* worse.

Traversing over the sorted keys and building the `String` has a cost of $O(n)$, as n characters must be inserted.

Therefore, if we're only considering n, then the final cost is $O(n \log n)$.

Considering k as well gives us $O(n + k \log k)$, because we don't know which is largest out of n and $k \log k$. We do, however, know that in total this is less than or equal to $O(n \log n)$.

- Space Complexity : $O(n)$.

  The `HashMap` uses $O(k)$ space.

  However, the `StringBuilder` at the end dominates the space complexity, pushing it up to $O(n)$, as every character from the input `String` must go into it. Like was said above, it's impossible to do better with the space complexity here.

What's interesting here is that if we only consider n, the time complexity is the same as the previous approach. But by considering k, we can see that the difference is potentially substantial.

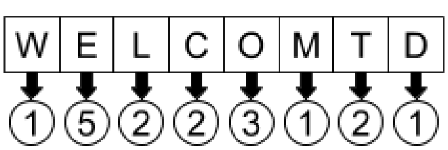
Approach 3: Multiset and Bucket Sort

Intuition

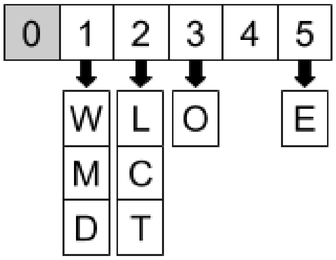
While the second approach is probably adequate for an interview, there is actually a way of solving this problem with a time complexity of $O(n)$.

Firstly, observe that because all of the characters came out of a `String` of length n, the maximum frequency for any one character is n. This means that once we've determined all the letter frequencies using a `HashMap`, we can sort them in $O(n)$ time using **Bucket Sort**. Recall that for our previous approaches, we used comparison-based sorts, which have a cost of $O(n \log n)$.

This was the `HashMap` from earlier.



Recall that **Bucket Sort** is the sorting algorithm where items are placed at `Array` indexes based on their values (the indexes are called "buckets"). For this problem, we'll need to have a `List` of characters at each index. For example, here is how our `String` from before goes into the buckets.



While we could simply make our bucket `Array` length n, we're best to just look for the maximum value (frequency) in the `HashMap`. That way, we only use as much space as we need, and won't need to iterate over heaps of empty buckets during the next phase.

Finally, we need to iterate over the buckets, starting with the largest and ending with the smallest, building up the string in much the same way as we did before.

Algorithm

Java Python Copy

```
def frequencySort(self, s: str) -> str:
    if not s: return s

    # Determine the frequency of each character.
    counts = collections.Counter(s)
    max_freq = max(counts.values())

    # Bucket sort the characters by frequency.
    buckets = [[] for _ in range(max_freq + 1)]
    for c, i in counts.items():
        buckets[i].append(c)

    # Build up the string.
    string_builder = []
    for i in range(len(buckets) - 1, 0, -1):
        for c in buckets[i]:
            string_builder.append(c * i)

    return "".join(string_builder)
```

Complexity Analysis

Let n be the length of the input `String`. The k (number of unique characters in the input `String` that we considered for the last approach makes no difference this time).

- Time Complexity : $O(n)$.

Like before, the `HashMap` building has a cost of $O(n)$.

The bucket sorting is $O(n)$, because inserting items has a cost of $O(k)$ (each entry from the `HashMap`), and building the buckets initially has a worst case of $O(n)$ (which occurs when $k = 1$). Because $k \leq n$, we're left with $O(n)$.

So in total, we have $O(n)$.

It'd be impossible to do better than this, because we need to look at each of the n characters in the input `String` at least once.

- Space Complexity : $O(n)$.

  Same as above. The bucket `Array` also uses $O(n)$ space, because its length is at most n, and there are k items across *all* the buckets.

Rate this article:

Previous (/articles/find-the-town-judge/)

Next (/articles/search-insert-position/)

Comments: 6

Sort By

Type comment here... (Markdown is supported)

Preview

Post

(/neosdeus)

NeosDeus (/neosdeus) 182 March 5, 2020 10:49 PM

Super clean written article! Looking forward to your next one @Hai_dee (https://leetcode.com/Hai_dee) !

4 Share Reply

SHOW 1 REPLY

(/my3m)

my3m (/my3m) 156 March 14, 2020 9:43 PM

Approach 1 last section can be simplified to `return String.join("", charStrings);`

1 Share Reply

(/chaowan_2020)

ChaoWan_2020 (/chaowan_2020) ★ 28 April 14, 2020 10:04 AM

This should be easy level

-1 Share Reply

SHOW 1 REPLY

(/loganle)

loganle (/loganle) ★ 19 April 7, 2020 6:34 PM

I think the last approach is not quite correct. It should use LinkedHashMap instead of HashMap because HashMap doesn't maintain the original order of insertion. Hence the sort is not stable.

0 Share Reply

SHOW 1 REPLY

(/innerpieces)

innerpieces (/innerpieces) ★ 0 March 17, 2020 1:19 AM

awesome

0 Share Reply

(/archcorsair)

archcorsair (/archcorsair) ★ 0 May 15, 2020 7:59 PM

Great write up!
It looks like Approach 2 in Java won't work the way its written because in this block:

```
for (char c : characters) {
        int copies = counts.get(c);
```

Read More

-1 Share Reply

 Help Center (/support/) | Terms (/terms/) | Privacy (/privacy/) United States (/region/)