

## Rearrange String (hard)

### We'll cover the following ^

- Problem Statement
- Try it yourself
- Solution
- Code
  - Time complexity
  - Space complexity

## Problem Statement #

Given a string, find if its letters can be rearranged in such a way that no two same characters come next to each other.

### Example 1:

```
Input: "aappp"
Output: "papap"
Explanation: In "papap", none of the repeating characters come next to each other.
```

### Example 2:

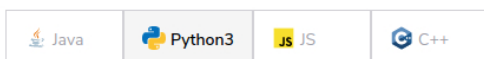
```
Input: "Programming"
Output: "rgmrpmPiano" or "gmringmrPoa" or "gmrPagimnor", etc.
Explanation: None of the repeating characters come next to each other.
```

### Example 3:

```
Input: "aapa"
Output: ""
Explanation: In all arrangements of "aapa", atleast two 'a' will come together e.g., "apaa", "pa
aa".
```

## Try it yourself #

Try solving this question here:



```
1 from heapq import *
2
3
4 def rearrange_string(str):
5     # TODO: Write your code here
6     return ""
7
8
9 def main():
10    print("Rearranged string: " + rearrange_string("aappp"))
11    print("Rearranged string: " + rearrange_string("Programming"))
12    print("Rearranged string: " + rearrange_string("aapa"))
13
14
15 main()
16
17
```

Run

Save

Reset



## Solution #

This problem follows the [Top 'K' Numbers](#) pattern. We can follow a greedy approach to find an arrangement of the given string where no two same characters come next to each other.

We can work in a stepwise fashion to create a string with all characters from the input string. Following a greedy approach, we should first append the most frequent characters to the output strings, for which we can use a **Max Heap**. By appending the most frequent character first, we have the best chance to find a string where no two same characters come next to each other.

So in each step, we should append one occurrence of the highest frequency character to the output string. We will not put this character back in the heap to ensure that no two same characters are adjacent to each other. In the next step, we should process the next most frequent character from the heap in the same way and then, at the end of this step, insert the character from the previous step back to the heap after decrementing its frequency.

Following this algorithm, if we can append all the characters from the input string to the output string, we would have successfully found an arrangement of the given string where no two same characters appeared adjacent to each other.

## Code #

Here is what our algorithm will look like:

Java

Python3

C++

JS

```

1  from heapq import *
2
3
4  def rearrange_string(str):
5      charFrequencyMap = {}
6      for char in str:
7          charFrequencyMap[char] = charFrequencyMap.get(char, 0) + 1
8
9      maxHeap = []
10     # add all characters to the max heap
11     for char, frequency in charFrequencyMap.items():
12         heappush(maxHeap, (-frequency, char))
13
14     previousChar, previousFrequency = None, 0
15     resultString = []
16     while maxHeap:
17         frequency, char = heappop(maxHeap)
18         # add the previous entry back in the heap if its frequency is greater than zero
19         if previousChar and -previousFrequency > 0:
20             heappush(maxHeap, (previousFrequency, previousChar))
21         # append the current character to the result string and decrement its count
22         resultString.append(char)
23         previousChar = char
24         previousFrequency = frequency+1 # decrement the frequency
25
26     # if we were successful in appending all the characters to the result string, return it
27     return ''.join(resultString) if len(resultString) == len(str) else ""
28
29
30 def main():
31     print("Rearranged string: " + rearrange_string("aapppp"))
32     print("Rearranged string: " + rearrange_string("Programming"))
33     print("Rearranged string: " + rearrange_string("aapa"))
34
35
36 main()
37

```

Run

Save

Reset

## Time complexity #

The time complexity of the above algorithm is  $O(N * \log N)$  where 'N' is the number of characters in the

input string.

## Space complexity #

The space complexity will be  $O(N)$ , as in the worst case, we need to store all the 'N' characters in the **HashMap**.



[< Back](#)[Next >](#)

Sum of Elements (medium)

Problem Challenge 1

 Completed

---

 Report an Issue  Ask a Question