1. Autoscale Policy

A scaling computing system checks its average *utilization* every second while it monitors. It implements an autoscale policy to increase or reduce instances depending on the current load as described below. Once an *action* of increasing or reducing the number of instances is performed, the system will stop monitoring for *10* seconds. During that time, the number of instances does not change.

- If the *average utilization* $< 25\%$, then an action is instantiated to reduce the number of instances by half *if* the number of instances is greater than *1*. Take the ceiling if the number is not an integer. If the number of instances is *1*, take no action.
- If $25\% \le$ *average utilization* $\le 60\%$, take no action.
- If the *average utilization* $> 60\%$, then an action is instantiated to double the number of instances *if* the doubled value does not exceed $2 * 10^8$. If the number of instances exceeds this limit upon doubling, take no action.

Given an array of integers that represent the average utilization at each second, determine the number of instances at the end of the time frame.

Example
instances = 2
averageUtil = [25, 23, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 76, 80]

Example
instances = 2
averageUtil = [25, 23, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 76, 80]

- At second *1*, the *average utilization averageUtil[0]* $= 25 \le 25$, so take no action.
- At second 2, the *average utilization averageUtil[1]* $= 23 < 25$, so reduce the number of instances by half to get $2 / 2 = 1$.
- Since an action was taken, the system will stop checking for *10* seconds, from *averageUtil[2]* through *averageUtil[11]*.
- At *averageUtil[12]* $= 76$, $76 > 60$, so the number of instances is doubled from *1* to *2*.

There are no more readings to consider and *2* is the final answer.

Function Description
Complete the function *finalInstances* in the editor below.

finalInstances has the following parameter(s):
  int instances: the original number of instances running
  int averageUtil[n]: the average utilization at each second of the time frame

Returns:
  int: the final number of instances running

Constraints

- $1 \le instances < 10^5$
- $1 \le n < 10^5$
- $0 \le averageUtil[i] \le 100$

2. Almost Equivalent Strings

Two strings are considered “almost equivalent” if they have the same length AND for each lowercase letter *x*, the number of occurrences of *x* in the two strings differs by no more than 3. *There are two string arrays, s and i*, that each contains *n* strings. Strings *s[i]* and *t[i]* are the i^{th} pair of strings. They are of equal length and consist of lowercase English letters. For each pair of strings, determine if they are almost equivalent.

Example
s = ['aabaab', 'aaaaabb']
t = ['bbabbc', 'abb']

Occurrences of '*a*' in *s[0]* = 4 and in *t[0]* = 1, difference is 3
Occurrences of '*b*' in *s[0]* = 2 and in *t[0]* = 4, difference is 2
Occurrences of *'c'* in *s[0]* = 0 and in *t[0]* = 1, difference is 1
The number of occurrences of '*a', 'b'*, and *'c'* in the two strings never differs by more than 3. This pair is almost equivalent so the return value for this case is '*YES'*.
Occurrences of *'a'* in *s[1] = 5* and in *t[1] = 1,* difference is 4
Occurrences of *'b'* in *s[1] = 2* and in *t[1] = 2,* difference is 0
The difference in the number of occurrences of *'a'* is greater than 3 so the return value for this case is '*NO'.*

The return array is *['YES', 'NO'].*

Function Description

Complete the function *areAlmostEquivalent* in the editor below.

areAlmostEquivalent has the following parameters:
string s[n]: an array of strings
string t[n]: an array of strings

Returns:
string[n]: an array of strings, either 'YES' or 'NO' in answer to each test case

Constraints

- $1 \le n \le 5$
- 1 ≤ length of any string in the input ≤ 10^5

3. Oscillating String

Sam and Alex are competitive coders working together on strings. They like to create challenges for each other as practice. In one challenge, Sam asks Alex to create a function to sort a string. The terms *smaller* and *larger* refer to the alphabetically lower or higher character. For example 'a' is smaller than 'b' and 'b' is larger than 'a'. To make the challenge more complex, the following string formation rules are stated:

1. **The sorted string, *ss*, begins with the smallest character in the original string, *s*.**
2. **Choose the smallest character from the remaining string that is larger than the previous letter, and append it to *ss*.**
3. **Repeat step 2 until it is no longer possible.**
4. **Choose the largest character from the remaining string that is smaller than the previous one, and append it to *ss*.**
5. **Repeat step 4 until it is no longer possible.**
6. **If no character was chosen in steps 2 through 5 because all characters are equal, choose any one of the characters and append it to *ss*.**
7. **Repeat steps 2 through 6 until the entire string has been processed.**

Example
s = ababyz.

Example
s = ababyz.

The following table shows the method:

```
s      choose ss
ababyz a      a       <- step 1
babyz  b      ab      <- step 2
abyz   y      aby     <- repeat step 2
abz    z      abyz    <- repeat step 2, reached the end
ab     b      abyzb   <- step 4
a      a      abyzba  <- repeat step 4, reached the end
```

The final sorted string is *abyzba.*

Function Description
Complete the function *formString* in the editor below. The function must return a string which is formed per the rules.

formString has the following parameter:
string s: a string of characters

Constraints

- **$0 < |s| \leq 10^5$**
- **All letters in *s* ∊ *ascii[a-z]***