STRINGS

String is an array of characters. '\n' '\t'

vector<char> str. ×

String:

→ Array has a pre-defined size and through that size, we get to know what the last element in an array is.

→ Strings are terminated by a special character and that special character defines the end of a string.

(special character: '\0')

[NOTE: Valid for C/C++]

→ In strings, maximum of 256 unique characters can be present.
↓
ASCII characters

O(1)

(a, b, c, ----, z) (A, B, C, ----, Z) (@, !, #, $ ...)

(int)'a'

→ Check if lowercase(ch):

if ((int)ch >= 97 && (int)ch <= 122) ✓

if (ch >= 'a' && ch <= 'z') ✓

→ Check if uppercase(ch):

if (ch >= 'A' && ch <= 'Z')

→ Check if digit(ch):

if (ch >= '0' && ch <= '9')

→ Convert lowercase(ch) to its corresponding uppercase:

'a' → 'A' 'n' → 'N'

= (ch − 32)

= (ch − ('a' − 'A')) = $\boxed{ch - 'a' + 'A'}$ ✓

→ Uppercase to lowercase:

⇒ (ch − 'A' + 'a')

[ch + ('a' − 'A')
ch − ('A' − 'a')]

Palindrome: A string equivalent to its reverse is called a palindromic string.

"abc" ×

"aba" ✓

"abcdcba" ⇒ True.

"abcdba" ⇒ False

"A man, a plan, a canal: Panama" ✓

"amanaplanacanalpanama"

TC: O(n)
Aux.space: O(n)

Anagrams: "act", "tac" → ✓
"aac", "cca" → ×

#1 Sort the strings: If two strings are anagrams, then their sorted versions will be the same.

"act", "tac"
↓ ↓
(act == act) → True

TC: O(n log n)
Aux.space: O(1)

#2: 2 Frequency Maps: maps will take O(26) = O(1) amount of space.

act: a→1, c→1, t→1
tac: a→1, c→1, t→1
True

aac: a→2, c→1
cca: a→1, c→2
False

[~~ab~~ ~~abc~~]

TC: O(n+n+26) = O(n)
Aux. space: O(26) = O(1)

#3: 1 Frequency Map

act
tac

a→~~1~~ 0
c→~~1~~ 0
t→~~1~~ 0
True

aac
cca

a→~~2~~ 1
c→~~1~~ −1
False

TC: O(n+n+26) = O(n)
Aux. space: O(26) = O(1)

a[i] = 'a' ⇒ (a[i] − 'a') = 0
a[i] = 'b' ⇒ (a[i] − 'a') = 1
⋮
a[i] = 'z' ⇒ (a[i] − 'a') = 25

Reverse a strings: "geeks" → "skeeg"

TC: O(n)
Aux.space: O(n)

i.like.this.program

word = ~~i like this~~ program

[i, like, this, program]

Lexicographical Rank of Strings

O(1)
O(n log n)

[abc, ach, bar, bca, cab, cba]
1, 2, 3, 4, 5, 6

e i v w

Brute force:

Length of string = n
No. of permutations = n!
For building all perm(s) = O(n*n!)
Lexicographical sorting = O(n*n!*log(n!))
Searching: O(n*n!)

e ~~i~~ v w

view:
e _ _ _ 3!
i _ _ _ 3!
2×3! = $\boxed{12}$

[v]iew:
e _ _ 2!
1×2! = $\boxed{2}$

[vi]ew: $\boxed{0}$

[view]

12+2+0 = 14 permutations are lexicographically smaller than "view".

⇒ Lexicographic rank of "view" = 15.

---

s: "cdba"

s_sorted = ~~a~~ ~~b~~ ~~c~~ ~~d~~

a _ _ _ 3!
b _ _ _ 3!
2×3! = $\boxed{12}$

'#'
a b ~~c~~ d

[cnt_smaller = ~~0~~ ~~1~~ 2]

[c]dba:
a _ _ 2!
b _ _ 2!
2×2! = $\boxed{4}$

[cd]ba:
a _ 1!
1 = $\boxed{1}$

[cdb]a: $\boxed{0}$

12+4+1+0 = 17

⇒ Lexicogr. Rank = 17+1 = 18.

5%2 = 1 8%3 = 2

x % n ∈ [0, n−1]

(a+b) % mod = (a%mod + b%mod) % mod

(a*b) % mod = (a%mod * b%mod) % mod

NOTE: This distribution does not work for division. Therefore, whenever you are asked to return (ans % mod), always avoid doing division operations at any step.

TC:
Precomputing factorials: O(n)
Sorting s_sorted: O(n log n)
Nested loops to get the no. of lesser strings: O(n²)

TC: O(n²)

Aux space: O(n + n) = O(n)
(fact[]) (s_sorted)