STRINGS

Q.1 Convert Strings:

↳ The problem is simple, here we have to convert 1st char of every word into upper case.

↳ One might think just to split string on space, capitalize 1st letter of word & join again. But the case that has to be taken is whitespace count must not reduce

e.g.:

```
aAb     dD12
```

o/p: `AAb DD12` ✓

`AAb DD12` ✗

↳ Hence the logic I followed is to generate a new string on the go. If prev. char is space & new/curr char is not space, capatilize it.

Pseudocode:

```
l = len(string)
ans = ""
ans += string[0].upper()

for i in range(1, len(string)):
    if ans[-1] == ' ' and string[i] != ' ':
        ans += string[i].upper()
    else:
        ans += string[i]

print(ans).
```

Q.2 Encode Message

↳ Here we have to create run length encoding of message.

↳ The logic is simple, if prev & curr char are diff, we append the count of prev. char to string, the curr char & begin new count.

Pseudocode:

```
l = len(message)
ans = message[0]
count = 1

for i in range(1, l):
    if message[i] == ans[-1]:
        count += 1
    else: // Diff char begins
        ans += str(count) + message[i]
        count = 1 // New count for curr char

ans += str(count) // count for last char

return ans.
```

Q.4 minimum Parenthesis:

↳ Here we have to find how many additional parenthesis are need to make/match all parenthesis in the pattern.

↳ we can use stack method & at end of parsing whatever is left those many additional brackets we shall need.

Pseudocode:

```
stack = deque()

for bracket in pattern:
    if bracket == '(':
        // opening bracket add to stack
        stack.append(bracket)

    elif bracket == ')':
        // for closing bracket, check if an
        // opening bracket is present on stack
        // top, if yes then cancel it with that
        // else add to stack.

        if stack & stack[-1] == '(':
            stack.pop()

        else:
            stack.append(bracket)

return len(stack).
```

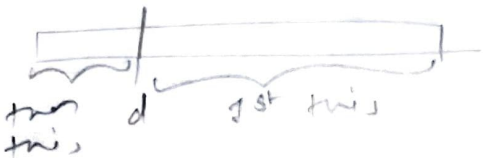
Q.5 Left and Right Rotation:

- This problem is simple appⁿ of string slicing.
- Here we have to just figure out at what idx we need to cut & join.
- For left rotⁿ we can directly use the rotⁿ no. given for right rotⁿ since we need to take end section at start, we shall use $(n-d)$ to get the idx.
- Also, #rotⁿ (d) can be $> n$ (len of string) hence we use mod to bring in range.

Pseudocode:


```
LeftRotate(strr, d):
    n = len(strr)
    return strr[d%n:] + strr[:d%n]
```

(till end / from 0)




```
RightRotate(strr, d)
    n = len(strr)
    d = n - (d%n)   // to get idx from right
    return strr[d:] + strr[:d]
```

Since already mod. ∴ no mod while slicing

B) MIXED PROBLEMS:

Q.1 Largest substring with k-distinct chars:

↳ Here we have to find max len of substring that has at most k distinct chars.

e.g:- abac & k = 3,

largest substring is: abac

e.g:- araaci. k = 2

substrings are: a, ar, ra, raa, araa, raac, aac, aci, etc.

out of these araa has max len with almost 2 distinct chars.

⇒ we can generate all possible substrings in $O(n^2)$ time & for each substring count unique chars $O(n)$ But this will be $O(n^3)$. Can we optimize it?

⇒ Since we have to form substrings we can use dynamic sliding window, by changing pos^n of window start & end we can get all possible substrings. Also the freq. count can be maintained in a hash map where keys will tell # distinct keys in the curr window, & their freq will give us the count.

⇒ So what we shall do is, go on adding elements into window until distinct count ≤ k. Once it reaches k we shall count the window len (window_end − window_start +1)

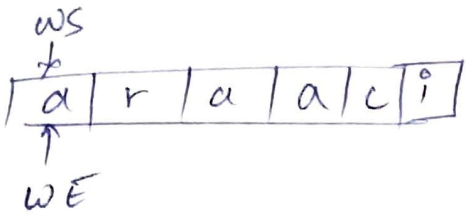
When window has ele > k we start removing from window until distinct count < k & add new ele later on to now check new length. If this len is > max update it.

e.g.:-

window_start = ws
window_end = we
max_len = ML
curr_len = cl , curr_k = ck

let word be 'araaci', k = 2

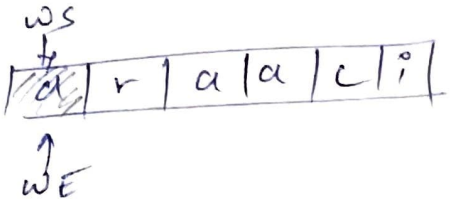
1)



ML = 0

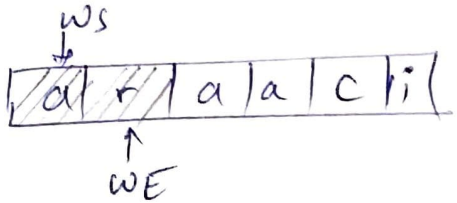
dict = { }
ck = 0

2) ML = 1



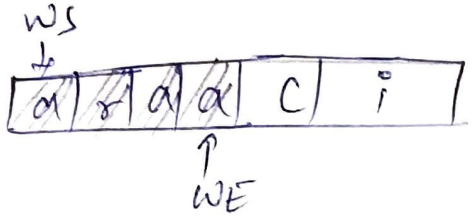
dict = {a : 1}
ck = 1
cl = 1

3) ML = 2



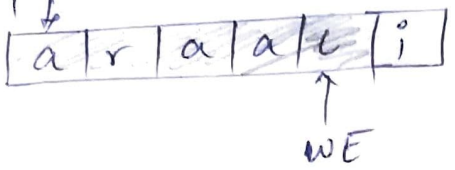
dict = { a : 1
r : 1
}
ck = 2
cl = 2

3) ML = 4



dict = { a : 2
r : 1
}
ck = 2
cl = 4

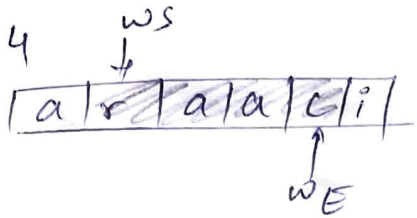
4) ML = 4



dict = { a : 3
r : 1
c : 1
}
ck = 3
cl = 4 (No update ML)

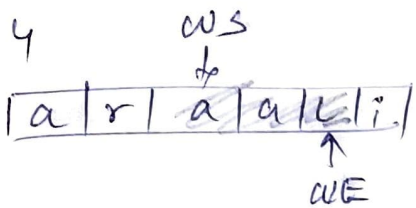
ck > k
∴ decrement window until
ck ≤ k

5) ML = 4



dict = { a : 2
r : 1
c : 1
}
ck = 3
cl = 4

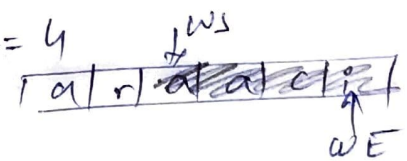
6) ML = 4



dict = { a : 2
c : 1 }
ck = 2
cl = 3

Now ck = 2, again start inc. window

7) ML = 4



dict = { a : 2
c : 1
i : 1 }
ck = 3
cl = 4

ck > k,
also reached end hence stop.

Pseudocode: (s, k)

```
hash_map = defaultdict(int)
l = len(string) or len(s)
window_start = 0
max_len = 0

// i -> window_end
for i in range(l):
    // Add char to map if not present
    // else inc. its freq.
    if not hash_map.get(s[i], None):
        hash_map[s[i]] += 1
    else:
        hash_map[s[i]] += 1

    // If map has more than k keys,
    // reduce window
    while len(hash_map) > k:
        hash_map[s[window_start]] -= 1
        // If freq becomes 0, remove key
        if hash_map[s[window_start]] == 0
            hashmap.pop(s[window_start])
        window_start += 1
    max_len = max(max_len, i - windowstart + 1)
```

Q. 2 Anagram Difference:

→ Solⁿ to this problⁿ is simple, first we keep freq count of all char's in Str1. Now anagram is created when we have atleast as many chars in str2 as there are in str1 both in freq & value.

→ Hence for every char in str2 we check if that is available in Str1, if yes this can be used as anagram char. We reduce it's count to notify it. If count becomes 0 or the char is not present in str1, we ~~needs~~ ~~add~~ can consider this as a manipulation.

e.g:- accept, except

a: 1
c: ~~1~~
e: ~~1~~ 0
p: ~~1~~
t: 1

~~except~~ except
↑ not present
2 ct=0

manipul: 11
- ans = 2

Pseudocode:

```
freq_map_s1 = defaultdict(int)
manipuln = 0

for char in s1:
    freq_map_s1[char] += 1

for char in s2:
    if freq_map_s1.get(char, None):
        if freq_map_s1[char] == 0:
            freq_map_s1.pop(char)
            manipuln ++
        else:
            freq_map_s1[char] -= 1

    else:
        manipuln += 1

return manipuln.
```


Q3 Minimum Oper? to make strings Equal:

↳ In this prob^m we have to make 2 strings equal by doing swap & replacement oper^n.

↳ swap oper^n do not account for cost, but replacement opers do account for cost.

↳ we are allowed to swap bcm {a[i], a[n-i-1], b[i], b[n-i-1]} thus what we can do is form groups of them & check if letters can be rearranged among themselves without replacement. Since, each group is separate from other group, we can calculate preprocessing moves separately for each group & add them together.

↳ Let's see cases where we don't need any preprocessing (replacing with a char which is not in the group)

Since, index notation becomes lengthy let me make them with variables.

$c_1 = a[i]$, $c_2 = a[n-i-1]$
$c_3 = b[i]$, $c_3 = b[n-i-1]$

1) If all chars are equal then no swaps & no preprocessing

2) If any 2 chars are equal pair wise then no preprocessing.

$c_1 = c_2$ & $c_3 = c_4$,
or $c_1 = c_3$ & $c_2 = c_4$,
or $c_1 = c_4$ & $c_3 = c_4$,

3) If any 2 chars are equal & rest are unequal then 1 preprocessing as we can replace one char with value of another.

eg:- c_1, c_3 are equal &
c_2, c_4 are unequal. replace c_2 with value of c_4.

pairs: $(c_1 == c_3)$ or $(c_1 == c_4)$ or $(c_2 == c_3)$
or $(c_2 == c_4)$ or $(c_3 == c_4)$

In these pairs we are not taking $(c_4 = c_2)$ as we are not allowed to make any changes to string B.

eg:- a a $\Rightarrow$ A
b c $\Rightarrow$ B

Here we would need to change both a's of A to b, c — 2 more.

3) For rest all cases we would need 2 moves (eg:- like one shown above)

→ If string is odd then middle char of both strings will form a separate grp. If the 2 chars are equal then no replacement, else 1 replacement.

Pseudocode:

```
l1, l2 = len(a), len(b)
if l1 != l2:
    return -1
moves = 0
for i in range(l1//2):        # only till 1/2 as other part is being considered in (n-i-1)
    c1, c2, c3, c4 = a[i], a[n-i-1], b[i], b[n-i-1]
    // No preprocessing:
    if ((c1==c2) and (c3==c4)) or
    ((c1==c3) and (c2==c4)) or
    ((c1==c4) and (c2==c3)):
        continue
    // one preprocessing
    if (c1==c3) or (c1==c4) or (c2==c3) or
        (c2==c4) or (c3==c4):
        moves+1
    else:
        moves += 2
// Outside for, odd len logic
if l1%2 and a[l1//2] != b[l2//2]:
    moves++
return moves.
```


Q-4 Match Specific Pattern:

↳ The problem is simple, for pattern we will generate unique hash value.

eg:- pqr : 123
aabc : 1123
xyyzaa : 122344

↳ we shall also generate a hash for word, if both hash match. it is our ans.

↳ Why hash? Internally when we are matching a pattern with a word eg:- foo with baa we see a f occured once, similarly b o occured twice ——— a we also want to preserve order as foo ≠ aab

Thus if we replace chars with nos, then we can compare only based on # of occurences & place of occurence. Hence hash.

Pseudocode:

```
generate-hash(word):
    freq_dict = {}
    counter = 1  // Initialize to 1
    word_hash = ""
    for char in word:
        // If unique word found which is
        // not in dict
        if not freq_dict.get(char, None):
            freq_dict[char] = counter
            counter += 1
        word_hash += str(freq_dict[char])
    return word_hash

// Now for each word gen. hash & compare with hash of pattern to check for matching
```

Q.5 Find Next Smallest Palindrome.

1) For a given string of numer we need to find the next greater palindromic string.

eg:- 1221 ⇒ 1331
999 ⇒ 1001

There are various cases for the problem let's see

2456 → 24|42 → Override secondhalf with mirror of 1st
2456 → 65|56 → Override 1st half with mirror 2nd

Since we need the next number ans will be 6556 (atleast for now)

(real ans: 2552)

Even len digits:

2 4 3 6 ⇒ 4 > 3 — Override middle left with midrigh
2 4 4 6

24 46 → Now override left's mirror onto right

↳ 2442

e.g.:- 628798 : 8 > 7

↳ 628898

↳ 628826

∴ for even:

↳ middle left > middle right

↳ Override left's mirror onto right

While changing now we go from R to L

Another case: Even len digit, mid left < mid right

2 8 3 8 49 → 3 < 8

By using prev logic: 283282 ✗

What we do is we inc the mid left & override right part with mirror of left.

28 5 849

28 4 | 849 ⇒ 284884 2

↳ Even len

↳ mid left < mid right

↳ mid left + 1

↳ Override right with left's mirror

Case: Odd len:

Logic almost similar to prev. 2 cases:

mid left & mid right would be 2 numbers on left & right of mid.

28 7 49 : 8 > 4 ⇒ Apply prev. logic (left > right)

2 3 6 8 2 : 7 < 8 ⇒ Apply prev. logic (left < right) (6+1=7)

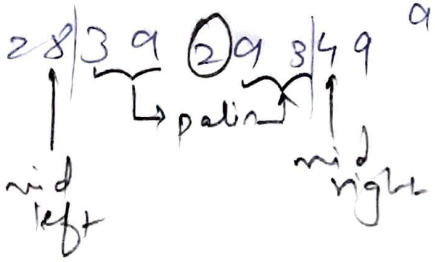
↳ 23732

Exceptions:

- Some nos are palindromic - we need to choose mid left & right accordingly.

Lovesech

e.g:-



$8 > 4 \Rightarrow 28392938 2$

- middle dig. can be 9 & mid left < mid right

e.g:- 24983 : 4 < 8

∴ As per algo 9+1 = 10
⇒ we hold 0 carry & add to 4

→ 25083 → Now overide

25052

- All dig contain 9

⇒ 99, 9 ⇒ nearest palindrome will be +2

Pseudocode:

```
// consider arr to be arr of integers.
// s ---- s to be number in string.
if s[0] == '9' and s[0]*len == s:
    // All 9's case
    return str(int(s) + 2)

mid = len//2
i = mid-1
j = mid+1 if len%2 else mid   ] left & right or from mid

// Reach to non-palindromic section.
while (i>=0 & arr[i] == arr[j]):
    i -= 1
    j += 1

left_small = False
if i<0 or arr[i] < arr[j]):
    // leftmid < right mid
    left_small = True

// copy mirror of left to right
while (i >= 0):
    arr[j] = arr[i]
    i -= 1
    j += 1
```

```
// End while

// Case when mid must be incremented
if left_small:
    carry = 1
    i = mid - 1
    if i >= 0:
        arr[mid] += 1
        carry = arr[mid] // 10
        arr[mid] %= 10
        j = mid + 1

    else:
        j = mid

    while (i >= 0)
        arr[i] += carry
        carry = arr[i] // 10
        arr[i] %= 10
        arr[j] = arr[i]
        i -= 1
        j += 1

// End if                    return (string of arr)
```


Q.6 First Unique Char

1) The question is simple we need to check first char which is unique.
2) In first pass we can count freq. of all char.
3) In second pass, if we find any char with freq 1 we return it, if none found return '#'

Pseudocode:

```
UniqueChar (string):
    char_hash = [0]*26

    for char in string:
        char_hash[ord(char) - ord('a')] += 1

    for char in string:
        if char_hash[ord(char) - ord('a')] == 1
            return char

    return '#'
```

Q.7 Compare Versions:

↳ Here we convert string versions into numbers & split on dot & compare one by one, until either of string ends.

Pseudocode:

```
CompareVersions(a,b):
    a1 = [int(x) for x in a.split('.')]
    b1 = [int(x) for x in b.split('.')]
    l1,l2 = len(a1), len(b1)
    i = 0
    while i<l1 and i<l2:
        if a1[i] > a2[i]:
            return 1
        elif a1[i] < a2[i]:
            return -1
        i += 1
    if i == l1:
        while i<l2:
            if a2[i] > 0:
                return -1
            i += 1
    elif i == l2:
        while i<l1:
            if a2[i] > 0:
                return 1
            i += 1
    return 0
```

Q.8 Validate IP:

↳ we need to check foll cond^n:

i) Split on '.' & total components must be 4

ii) If len of a component (octate) is 1 then it must be bet '0' & '9'

iii) If len > 1 then it must be digit & range of value 1 and 255

Pseudocode:

```
lst = IP.split(".")
if len(lst) = 4 and
all( len(octate) > 0 and
    [(len(octate) == 1 and "0" <= octate <= "9")
    or (octate.isdigit() and octate[0] != 0
        and 1 <= int(octate) <= 255)]
    for octate in lst ): return True
```

no: cannot be like 01, 023, 001, etc

Q.9 Decrypt kth character:

a) Idea is simple, initially take empty decrypted string then decompress the string by reading substring & it's freq. finally append that substring those freq. number of times.

b) First we find substring, next we find the number.

Pseudocode:

```
decrypted_str = ""
i = 0
l = len(s)
while i < l:
    temp_str = ""
    freq = 0
    while i < l and ord('a') <= ord(s[i])
                and ord(s[i]) <= ord('z'):
        temp_str += s[i]
        i += 1
    // End while
    while i < l and ord('0') <= ord(s[i])
                and ord(s[i]) <= ord('9'):
        freq = freq * 10 + (ord(s[i]) - ord('0'))
        i += 1
    decrypted_str += (temp_str * freq)
    // End while
// End while
len_dec = len(decrypted_str)
if len_dec > 0 and k-1 < len_dec:
    return decrypted_str[k-1]
return '$'
```

Q: 10 Multiply 2 strings:

High level:

num1 = 123

num2 = 45

num1 * num2 = $123 \times 4 \times 10^1 + 123 \times 5 \times 10^0$

= 4920 + 615

= 5535

we multiply num1 with each digit of num2 & decimal left shift by 'n' positions where,

n = len(num2) - index pos of dig. in num2 in array - 1

∴ When we multiply,

123 * 4 ⇒ we need to shift result by len(num2) - idx pos - 1

2 — 4 — 0

= 1 pos to left.

∴ mult. by 10^1

Deep: Instead of multiplying 123 by 4 we could use prev. technique for mult. digit by dig.

eg:

prod1 = 123 * 4

$= 4 * 1 \times 10^2 + 4 \times 2 \times 10^1 + 4 \times 3 \times 10^0$

= 400 + 80 + 12

= 492

prod2 = 123 * 5

$= 5 * 1 \times 10^2 + 5 * 2 \times 10^1 + 5 * 3 \times 10^0$

= 500 + 100 + 15

= 615

How to combine prod1 & prod2

we need to shift prod1 by

len(num2) - idx * 4 - 1 pos

= 2 - 0 - 1 = 1 pos

& simil. prod2 by len(num2) - idx * 5 - 1

= 2 - 1 - 1 = 0 pos

∴ Ans = $492 \times 10^1 + 615 * 10^0$

= 4920 + 615 = 5535

Pseudocode:

```
len_n1 = len(n1)
len_n2 = len(n2)
prod = 0
for idx, char2 in enumerate(n2):
    interned_prod = 0
    dig2 = ord(char2) - ord('0')
    for idx, char1 in enumerate(n1):
        dig1 = ord(char1) - ord('0')
        interned_prod += dig1 * dig2 *
                         10**(len_n1 - idx - 1)
    prod += interned_prod * 10**(len_n2 - idx
                                 - 1)
return str(prod)
```