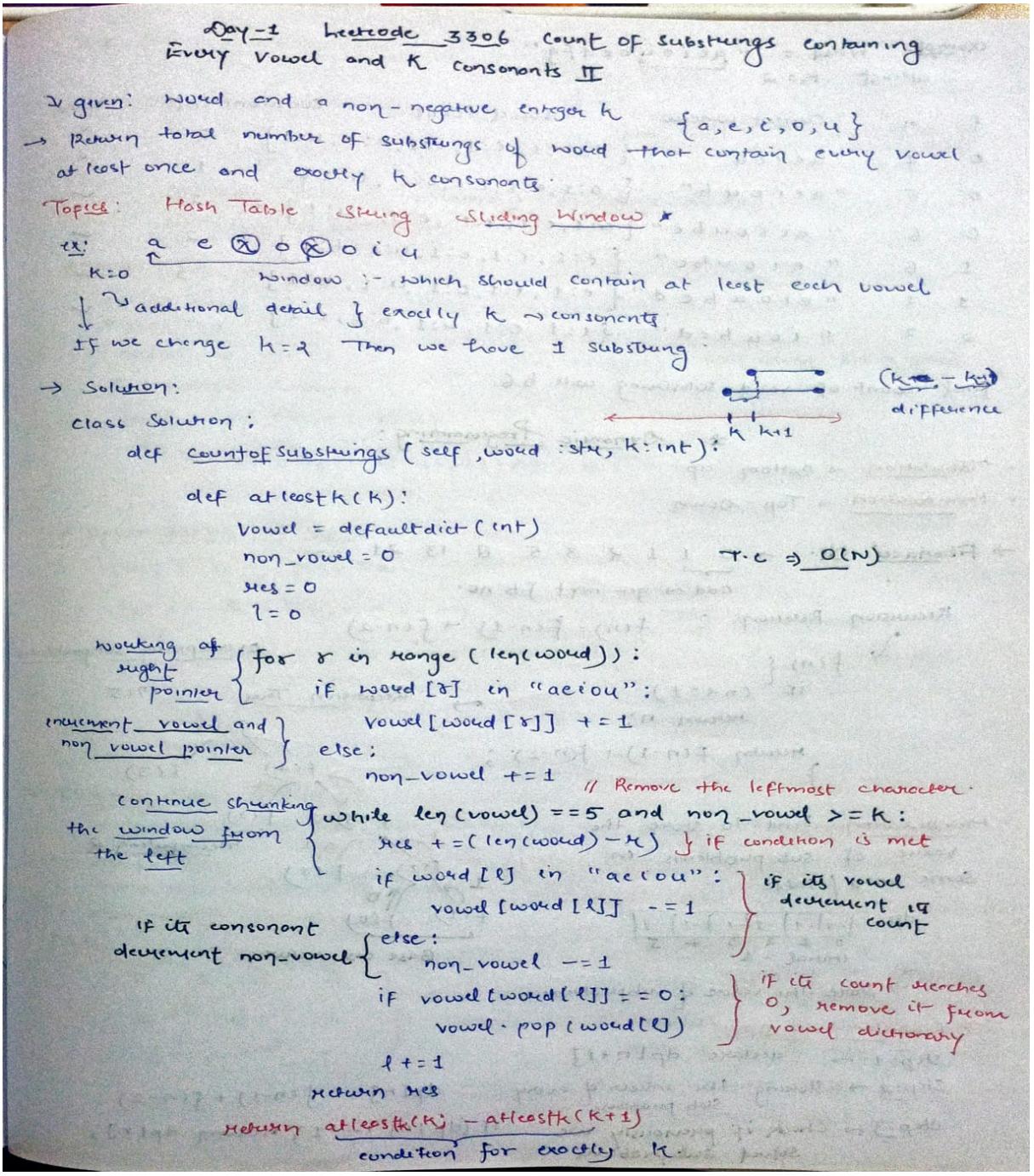
Day_1 : DSA


Day-1 Leetcode 3306 Count of Substrings Containing Every Vowel and K Consonants II
Given: Word and a non-negative integer k
Return total number of substrings of word that contain every vowel {a,e,i,o,u} at least once and exactly k consonants.
Topics: Hash Table String Sliding Window
Ex: a e o o i u
k=0
window :- which should contain at least each vowel
additional detail } exactly k → consonants
If we change k=2 Then we have 1 substring
Solution:
class Solution:
def countofSubstrings(self, word: str, k: int):
def atleastk(K):
vowel = defaultdict(int)
non_vowel = 0
res = 0
l = 0
T.C ⇒ O(N)
working of right pointer
for r in range(len(word)):
if word[r] in "aeiou":
increment vowel and non vowel pointer
vowel[word[r]] += 1
else:
non_vowel += 1
// Remove the leftmost character.
continue shrinking the window from the left
while len(vowel) == 5 and non_vowel >= K:
res += (len(word) - r) } if condition is met
if word[l] in "aeiou":
vowel[word[l]] -= 1
if its vowel decrement it count
if its consonant decrement non_vowel
else:
non_vowel -= 1
if vowel[word[l]] == 0:
vowel.pop(word[l])
if its count reaches 0, remove it from vowel dictionary
l += 1
return res
return atleastk(K) - atleastk(K+1)
condition for exactly k.
difference
k k+1

example word = "aeioybcdfg"
atleast K = 2

i	j	Current window	Vowel Count	Consonant cnt	Valid
0	4	"aeiou"	{"a":1, e:1, i:1, o:1, u:1}	0	x
0	5	"aeioub"	{a:1, e:1, i:1, o:1, u:1}	1	x
0	6	"aeioubc"	{a:1, e:1, i:1, o:1, u:1}	2	✓
1	6	"eioubc"	{e:1, i:1, o:1, u:1, b:1}	2	✓
1	7	"eioubcd"	{e:1, i:1, o:1, u:1, b:1}	3	✓
2	7	"ioubcd"	{i:1, o:1, u:1, b:1, c:1}	3	✓

final count of valid substring will b 6

Dynamic Programming :

- Tabulation → Bottom up
- Memoization → Top-down

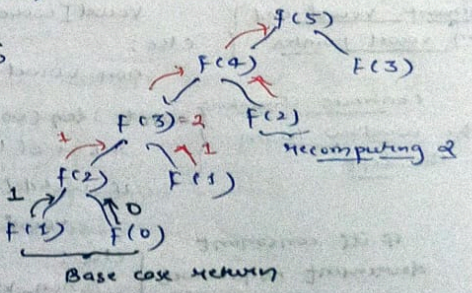
→ Fibonacci No. → 0 1 1 2 3 5 8 13 21 ....
add → get next fib no.

Recursion Relation : $f(n) = f(n-1) + f(n-2)$

```
f(n) {
  if (n<=1)
    return n;
  return f(n-1) + f(n-2);
}
```

overlapping subproblems

Recursion Tree n=5



- Memoization : tend to store the value of sub problems in some map / table.

dp

-1	-1	-1	-1	-1	-1
0	1	2	3	4	5

initial - 1

→ store the value of subproblems

→ Recursion → DP

Step:1 → declare dp[n+1]

Step:2 → Storing the answer of every sub problem → dp[n] = f(n-1) + f(n-2)

Step:3 → check if previously we solved sub problem → if (dp[n] != -1) return dp[n];

```
def Fib(n, memo={}):
    if n in memo:
        return memo[n]
    if n<=1:
        return n
    memo[n] = fib(n-1, memo) + fib(n-2, memo)
    return memo[n]
```

T.C → O(N)
S.C → O(N) + O(N)
stack space array (memo)

→ Recursion → Tabulation

```
dp[n+1]
// Base case
dp[0] = 0   dp[1] = 1
// Recursion Relation
for(i=2; i<=n; i++)
  { dp[i] = dp[i-1] + dp[i-2];
  }
```

T.C → O(N)
S.C → O(N) Not using stack space

* Optimization of Space.

```
prev2 = 0   prev = 1
for (i=2; i<=n; i++)
  {  curi = prev + prev2;
     prev2 = prev;
     prev = curi
  }
print(prev)
```

T.C = O(N)
S.C → O(1) * optimized.

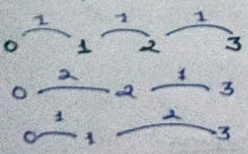
Climbing Stairs {Qus}

Tell n distinct ways in which you can reach nth stair

→ 2 problems:
- Count the total no. of ways
- Minimum / Maximum

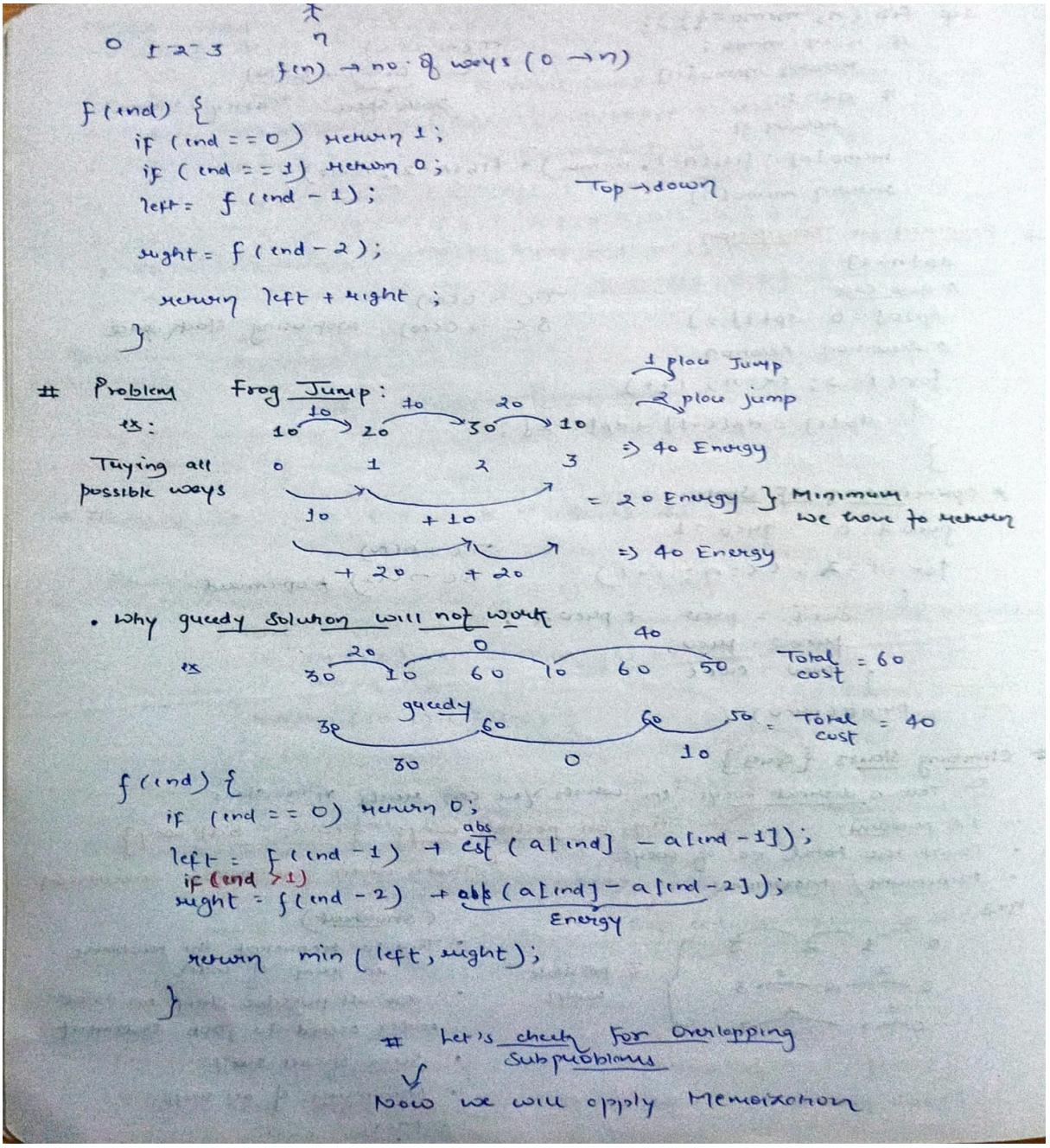
Try all possible ways {count, best way}
↳ That is when we apply recursion

n=3



3 possible ways

(Shortcut)
- Try to represent the problem in term of index
- Do all possible stuff on that index accord. to prob statement
- Sum of all stuff
- Minimum of all stuff
- Max


0 1 2 3 n
f(n) → no. of ways (0 → n)
f(ind) {
if (ind == 0) return 1;
if (ind == 1) return 0;
left = f(ind - 1);
right = f(ind - 2);
return left + right;
}
Top → down
Problem Frog Jump:
1 place Jump
2 place jump
ex: 10 20 30 10
10 10 20
=> 40 Energy
Trying all possible ways
0 1 2 3
10 + 10
= 20 Energy } Minimum we have to return
+ 20 + 20
=> 40 Energy
• Why greedy solution will not work
ex 30 20 10 0 60 10 60 40 50
Total cost = 60
greedy
30 60 60 50
30 0 10
Total cost = 40
f(ind) {
if (ind == 0) return 0;
left = f(ind - 1) + abs(a[ind] - a[ind - 1]);
if (ind > 1)
right = f(ind - 2) + abs(a[ind] - a[ind - 2]);
Energy
return min(left, right);
}
# Let's check for Overlapping Subproblems
Now we will apply Memoization

Recursion → dp

Memoization → Look at the parameter changing as index

dp[i] or dp[n+1]

Tc - O(N)
Sp - O(N) + O(N)

```
f(ind) {
  if (ind == 0) return 0;
  if (dp[ind] != -1) return dp[ind];
  left = f(ind-1) + abs(a[ind] - a[ind-1]);
  if (ind > 1)
  right = f(ind-2) + abs(a[ind] - a[ind-2]);
  return dp[ind] = the min (left, right);
```

Memoization → Tabulation (Bottom up)

```
int dp[n] → 0
// Base case
if ind == dp[0] = 0
for (i = 1 → n-1)
  {
    fs = dp[ind-1] + abs(a[ind] - a[ind-1])
    if (i > 1)
      ss = dp[ind-2] + abs(a[ind] - a[ind-2]); }
  dp[ind] = min(fs, ss); }
```

* stack space is removed.

Space Optimization:

```
int dp[n] → 0  no more dp required.
prev = 0
prev2 = 0
for (i = 1 → n-1) {
    fs = prev + abs(a[ind] - a[ind-1]);
    if (i > 1)
    ss = prev2 + abs(a[ind] - a[ind-2]);
    curi = min(fs, ss);
    prev2 = prev;
    prev = curi;
  }
  print(prev)
```