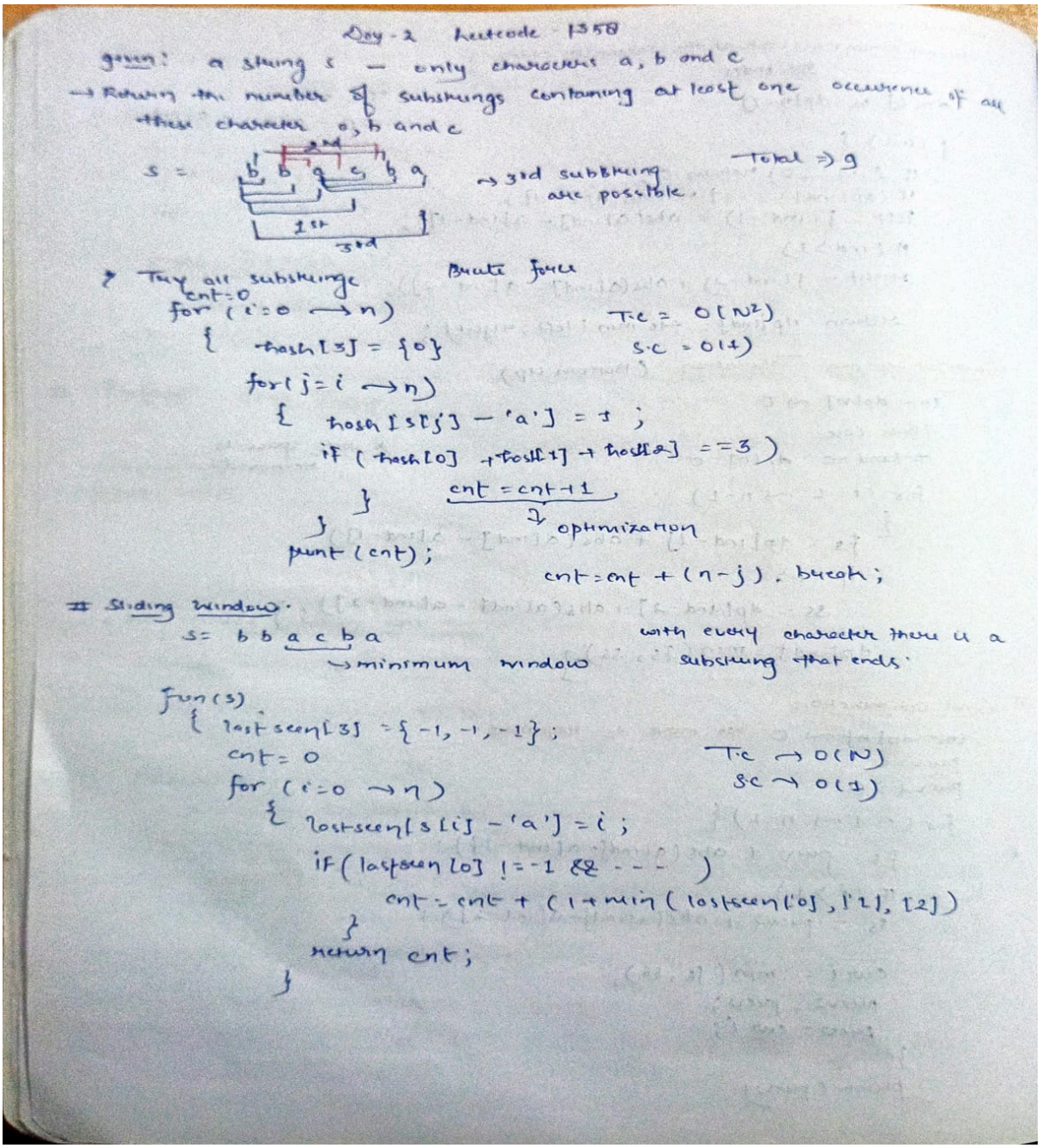
Day_2: DSA


Day-2 leetcode 1358
given: a string s — only characters a, b and c
→ Returning the number of substrings containing at least one occurrence of all these characters a, b and c
s = b b a c b a
→ 3rd substring are possible
Total ⇒ 9
1st
3rd
* Try all substrings
Brute force
cnt=0
for (i=0 → n)
{ hash[3] = {0}
for(j=i → n)
{ hash[s[j] - 'a'] = 1 ;
if (hash[0] + hash[1] + hash[2] == 3)
{ cnt = cnt+1
→ optimization
cnt=cnt + (n-j) . break;
}
}
print(cnt);
T.C = O(N²)
S.C = O(1)
# Sliding window.
s= b b a c b a
→ minimum window
with every character there is a substring that ends.
Fun(s)
{ last seen[3] = {-1, -1, -1};
cnt= 0
for (i=0 → n)
{ lastseen[s[i] - 'a'] = i ;
if(lastseen[0] != -1 && - - -)
cnt = cnt + (1 + min(lastseen['0'], '1', [2]))
}
return cnt;
}
T.C → O(N)
S.C → O(1)

- Frog Jump with K-distance
  - In this frog can jump K steps at each time
- Pseudocode for simple frog jump

```
f(ind, height[]){
    if(ind == 0) return 0
    jumpOne = f(ind-1, height) +
              abs(height[ind] + height[ind-1])
    if(ind > 1)
      jumpTwo = f(ind-2, height) +
                abs(height[ind] + height[ind-1])
    return min(jumpOne, jumpTwo)
}
```

- In this we are taking single step or two step at a time
- Now we need to try K-option in order to try out all possible ways.

Final pseudocode.

```
f(ind, height[]){
    if(ind == 0) return 0
    mmsteps = INT_MAX
    for(j=1; j<=k; j++){
      if(ind-j >= 0){
        jump = f(ind-j, height) + abs(height[ind] - height[ind-j])
        mmsteps = min(jump, mmsteps)
      }
    }
    return mmsteps
}
```

This is recursive solution.

Now convert into dp sol.

- Create a dp[n] array initialized to -1
- whenever we want to find the answer of a particular value (say n) we first check whether the answer is already calculated using the dp array (i.e. dp[n] != -1)
  If yes simply return the value from the dp array
- If not, we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning the function, we will set dp[n] to the solution we get.

Code:

```
def solveUtil(ind, height, dp, k):
    # Base case: If we are at the beginning (index 0), no cost is needed
    if ind == 0:
        return 0
    # If the result is already calculated
    if dp[ind] != -1:
        return dp[ind]

    mmsteps = sys.maxsize
    # Loop to try all possible jumps from '1' to 'k'
    for j in range(1, k+1):
        # Ensure that we do not jump beyond the beginning of the array
        if ind - j >= 0:
            jump = solveUtil(ind - j, height, dp, k) + abs(height[ind] - height[ind - j])

            mmsteps = min(jump, mmsteps)

    # Third step
    dp[ind] = mmsteps
    return dp[ind]

# Tabulation Approach
import sys
def solve_util(n, height, dp, k):
    dp[0] = 0

    for i in range(1, n):
        mmsteps = sys.maxsize
        for j in range(1, k+1):
            if i - j >= 0:
                # calculate the no. of steps to reach current position from previous position
                jump = dp[i-j] + abs(height[i] - height[i-j])

                mmsteps = min(jump, mmsteps)

        dp[i] = mmsteps

    # Return the min steps needed to reach the last position
    return dp[n-1]
```

TC: $O(N*K) \approx O(N)$

Problem: Maximum Sum of Non-adjacent Element

ex 1 2 3 9 → Sum = 11

(Let's try out all subsequence with the given constraint)
↳ pick the subsequence with maximum

Recursion Sol:

```
f(ind) {                                   dp[n] = -1
  if(ind == 0) return a[ind];
  if (ind < 0) return 0;
  if (dp[ind] != -1) return dp[ind];
  pick = a[ind] + f(ind-2);
  not pick = 0 + f(ind-1);
  dp[ind] =
  return max(pick, not pick);
}
```

T.C = O(2^n)

→ This is dp Solution

T.C ⇒ O(N)

S.C ⇒ O(N) + O(N)
Stack space, Array space

Tabulation: → Bottom-Up

```
dp[n] = 0
dp[0] = a[0];   // Base case
int neg = 0;    // Base case

for (i=1; i<n; i++){
    take = a[ind] + dp[i-2];   → if (i>1) take += dp[i-1];
    not take = 0 + dp[i-1];
    dp[i] = max(take, not take);
}
```

If can reach to negative index

T.C = O(N)
S.C = O(N)

Space Optimisation

```
prev = a[0];
prev2 = 0;
for (i=1; i<n; i++){
    take = a[ind]; + if (i>1) take += prev2;
    not take = 0 + prev
    cur[i]
    dp[i] = max(take, not take);
    prev2 = prev; prev = cur[i];
```

T.C = O(N)
S.C = O(1)