Question 1:

Problem: Given an array of integers representing payments for n consecutive days, we are not allowed to select adjacent payments and find the maximum profit that we can get. (One-Day-Gap)

My Algorithm:

Algorithm maxPayment(arr[], n, memo)

Input : arr, is the input array containing payments
n, is the index of the array that we are present
memo, is the array that value of the function calls for future reference
Output : The maximum profit the speaker could get

if (n<0) return 0

// Here n might be from (n-2) or (n-1) (and n might be picked or may not be picked)

If memo[n] != -1 return memo[n]
// We first check if the present function is computed before

pick <- arr[n] + maxPayment(arr, n-2, memo)

notPick <- maxPayment(arr, n-1, memo)

return memo[n] = max(pick, notPick)
//store the maxPayment for n days

Analysis of my Algorithm:

Pros: My Algorithm uses Top-Down dynamic Programming approach. It stores all the results of function calls in a memo array and when we need to call the function again, we can just use the value from the memo array. Here the memo[i] represents the maximum profits that can be achieved in i days following the constraint.

Recurrence Relation: T(n) = max(T(n-1) , arr[n] + T(n-2)) => This is correct

Time Complexity: My Algorithm follows the Top-Down approach where it stores the maximum profits in the memo array. So by storing the results the time complexity is reduced from O(2^N) to O(N) as we need to call the function again if we already computed it making it O(1).

Space Complexity: O(N) I use a memo array of size n (size of the arr array) where I store the maximum profits. Thus, making it O(N) space complexity.

Cons: Even Though my algorithm calculates the maximum profits correctly, it does not track the indices giving the maximum profit making my algorithm an incomplete one.

Solution for the Incompleteness:

I need to somehow keep track of the indices giving the maximum profit. So to achieve that I learnt that by creating a function recordIndices that will return the indices using memo and arr.

Algorithm recordIndices(arr[], memo[], res[])

Input:

arr: the input array containing payments

memo: the array containing maximum profit values from maxPayment

res: a boolean array to mark selected payments (true if picked)

Output:

res array filled with selections that yield the maximum profit

```
    n <- arr.length - 1     // Start from the last index


    if (n < 0) return       // If array is empty, no selections
    if (memo[n] <= 0) return  // If max profit is non-positive, pick nothing


    while (n >= 0)
        if (n = 0 and memo[0]>0) then
Do res[0] <- true;
Stop here
If (n >= 2) then
                        do pick <- arr[n] + memo[n-2]
        else
                        do pick <- arr[n]
notPick <- memo[n-1]
If (pick >= notPick and memo[n] = pick) then
do res[n] <- true
n <- n-2
else
do res[n] = false
n <- n-1
```

Explanation: we start from the last index of the array and handle the base cases like 0 days and memo[n] <= 0. First we check whether a day comes under the optimal selection. At each step, it compares the profit from picking the current day versus notPick. If picking is optimal, the day is marked as ture, and n moves back by 2 to avoid adjacent days. Otherwise, n moves back by 1. This process continues until n becomes 0, ensuring that the **maximum profit** is recorded efficiently.

Time Complexity: O(N) => we iterate through the house in reverse order exactly once and mark the status in the res array, thus making it O(N)

Space Complexity: O(1) => No additional space used as already computed memo, res array filled with false and input given arr arrays are used

Note: The following two are implemented in the code

1. In the question we are asked to write a function which will take only the input array arr, but my algorithm takes arr, n, memo. So, to solve this we can use a helper function and call our function inside it.
2. The res is the reference for the Boolean array, we use that and iterate over it and if the current index is true then we store the array[current] in a separate array .

Question 2:

Given an array of integers representing the daily changes in energy levels, we should determine the highest cumulative increase over any continuous period and identify the specific sequence of energy changes that led to this maximum gain.

My Algorithm:

Algorithm maxEnergy(energy, start, end)

Input : energy is the input array, start and end are the two numbers for indicating starting and ending indices

Output: This algorithm finds the maximum cumulative energy gain over a continuous period

If (start < end)then

Do mid <- start + (end – start)/2

P1 = maxEnergy(energy, start, mid)

P2 = maxEnergy(energy, mid+1, end)

P3 = crossSumEnergy(energy,start,mid,end)

Return max(p1, p2, p3)

Algorithm crossSumEnergy(energy, start, mid, end)

Input :

Output :

// left suffix sum

Sum <- 0

For i <- start to mid

sum1 <- sum1 + energy[i]

// right suffix sum

Sum2 <- 0

For i <- mid+1 to end

sum2 <- sum2 + energy[i]

Return sum(sum1 + sum2)

Analysis: I am using a Divide and Conquer Approach

Divide:

maxEnergy() recursively divides the array into smaller segments.
CrossSumEnergy(): This function calculates the sum of left sum and right sum

Conquer

While conquering we check take the maximum of the left sum, right sum and the total sum. Like this we go to the top and we get the maximum sum possible by array.

Recurrence Relation:

T(n) = 2T(n/2) + O(n)

Reason: We are dividing the array into two parts every time and we are computing the sum of the left and right suffixes which takes O(N) time complexity

Time Complexity: O(Nlog(N))

The corssSumEnergy function computes for N times and there are Log(n) times that function being called.

Space Complexity: O(log(N) There are log(n) function calls so stack size is log(n), meanwhile the crossSumEnergy function is calculating sum in O(1) Time Complexity. Thus, overall Space Complexity is O(log(N)

My Algorithm need Change:

While I believe that using Divide and Conquer to solve this problem is a good idea but it takes O(nlog(n)) time complexity, but this problem can be solved in O(N) using a Kadane's algorithm.

```
public class MaxSumSubarray {
    public static void findMaxSumSubarray(int[] arr) {
        int maxSum = Integer.MIN_VALUE;
        int currentSum = 0;
        int tempStart = 0;
        int start = 0, end = 0;

        for (int i = 0; i < arr.length; i++) {
            currentSum += arr[i];

            if (currentSum > maxSum) {
                maxSum = currentSum;
                start = tempStart;
                end = i;
            }

            if (currentSum < 0) {
                currentSum = 0;
```

```
                tempStart = i + 1;
            }
        }

        // Print the maximum sum
        System.out.println("Maximum Sum: " + maxSum);

        // Print the subarray that contributes to the maximum sum
        System.out.print("Subarray: ");
        for (int i = start; i <= end; i++) {
            System.out.print(arr[i] + " ");
        }
    }

    public static void main(String[] args) {
        int[] arr = {1, -2, 3, 4, -1, 2, 1, -5, 4};
        findMaxSumSubarray(arr);
    }
}
```

Time Complexity Analysis – O(N)

Space Complexity is correct - O(1)