# Prefix Sum

Prefix sum is sum of all element of array upto that index
Example:
Input: arr[] = {10, 20, 10, 5, 15}
Output: prefixSum[] = {10, 30, 40, 45, 60}

code to calculate prefix sum
```
void fillPrefixSum(int[] arr, int n, int[] prefixSum) {
    prefixSum[0] = arr[0];
    // Adding present element with previous element
    for (int i = 1; i < n; i++) {
        prefixSum[i] = prefixSum[i - 1] + arr[i];
    }
}
```

Time Complexity: O(N), traversing the array once
Auxiliary Space: O(N), creating a new array to store prefix sum

Prefix array is very useful in solving many DSA problems
Let’s have a look at some important ones

Ques: Running sum of 1D Array
Given an array nums. We define a running sum of an array as runningSum[i] = sum(nums[0]…nums[i]).

Return the running sum of nums.

Example 1:

Input: nums = [1,2,3,4]
Output: [1,3,6,10]
Explanation: Running sum is obtained as follows: [1, 1+2, 1+2+3, 1+2+3+4].

Code:
```
class Solution {
    public int[] runningSum(int[] nums) {
        int n = nums.length;

        if (n < 2) {
            return nums;
        }

        int[] rSum = new int[n];
        rSum[0] = nums[0];

        for (int i = 1; i < n; ++i) {
            rSum[i] = nums[i] + rSum[i - 1];
        }

        return rSum;
    }
}
```

Explanation: start from zero index and running sum here is the element itself from next element keep adding the previous running sum and current element
Time: O(n), to traverse the array
Space: O(n), created a new array of size n

Ques: Check if array can be partitioned into 2 continuous arrays of equal sum.
Code:

```
public boolean canPartition(int[] nums) {
    int total = 0, current = 0;
    for (int num : nums) {
        total += num;
    }

    if (total % 2 != 0) {
        return false;
    }

    for (int num : nums) {
        current += num;
        if (current == total / 2) {
            return true;
        }
    }

    return false;
}
```

Time Complexity: O(n) where n is the number of elements in the array.
Space Complexity: O(1) as the algorithm uses a constant amount of extra space regardless of the input size.

Explanation:

The canPartition function checks if the input array can be divided into two continuous arrays of equal sum.
It calculates the total sum of the array elements.
If the total sum is odd, it returns false since it's impossible to divide odd numbers into two equal sums.
Then, it iterates through the array, keeping a running sum (current). If current equals half of the total, it returns true indicating that the array can be partitioned, otherwise false.

Ques.Minimum Penalty for a Shop    (leetcode 2483)
You are given the customer visit log of a shop represented by a 0-indexed string customers consisting only of characters 'N' and 'Y':

if the ith character is 'Y', it means that customers come at the ith hour
whereas 'N' indicates that no customers come at the ith hour.
If the shop closes at the jth hour (0 <= j <= n), the penalty is calculated as follows:

For every hour when the shop is open and no customers come, the penalty increases by 1.
For every hour when the shop is closed and customers come, the penalty increases by 1.
Return the earliest hour at which the shop must be closed to incur a minimum penalty.

Note that if a shop closes at the jth hour, it means the shop is closed at the hour j.

Example 1:

Input: customers = "YYNY"
Output: 2
Explanation:
- Closing the shop at the 0th hour incurs in 1+1+0+1 = 3 penalty.
- Closing the shop at the 1st hour incurs in 0+1+0+1 = 2 penalty.
- Closing the shop at the 2nd hour incurs in 0+0+0+1 = 1 penalty.
- Closing the shop at the 3rd hour incurs in 0+0+1+1 = 2 penalty.
- Closing the shop at the 4th hour incurs in 0+0+1+0 = 1 penalty.
Closing the shop at 2nd or 4th hour gives a minimum penalty. Since 2 is earlier, the optimal closing time is 2.

```
class Solution {
    public int bestClosingTime(String customers) {
        int minPenalty = 0;
        int ans = 0;
        int n = customers.length();

        for (int i = 0; i < n; i++) {
            if (customers.charAt(i) == 'Y') {
                minPenalty++;
            }
        }
        int penalty = minPenalty;
        for (int hour = 1; hour <= n; hour++) {
            if (customers.charAt(hour - 1) == 'N') {
                penalty++;
            } else {
                penalty--;
            }

            if (minPenalty > penalty) {
                minPenalty = penalty;
                ans = hour;
            }
        }
        return ans;
    }
}
```

Time complexity: O(n), traversing the array 2 times, so linear time
Space complexity: O(1), only const space being used

Explanation: First, we'll determine the penalty at hour O. Afterward, we'll calculate penalties for hours 1 to N based on the following rule: If the customer's status at the previous hour (hour-1) is 'Y', the penalty will decrease; if it's 'N', the penalty will increase.

Q. Product of Array Except Self (Leetcode 238)
Given an integer array nums, return an array answer such that answer[i] is equal to the product of all the elements of nums except nums[i].

The product of any prefix or suffix of nums is guaranteed to fit in a 32-bit integer.

You must write an algorithm that runs in O(n) time and without using the division operation.

Example 1:
Input: nums = [1,2,3,4]
Output: [24,12,8,6]

Example 2:
Input: nums = [-1,1,0,-3,3]
Output: [0,0,9,0,0]

```
Code: class Solution {
    public int[] productExceptSelf(int[] nums) {
        int[] output = new int[nums.length];
        int total = 1;

        for (int i = 0; i < nums.length; i++) {
            total *= nums[i];
            output[i] = total;
        }

        total = 1;
        for (int i = nums.length - 1; i > 0; i--) {
            output[i] = output[i - 1] * total;
            total *= nums[i];
        }

        output[0] = total;
        return output;
    }
}
```

Time complexity: O(n), traversing the array 2 times, so linear
Space complexity: O(1), only constant space is used

Explanation: In this problem since we know we are dealing with multiplying we can deduce that we will be holding more of a "prefix product" rather than a prefix sum. Now that we know this we also know that we can create a single array as our output without violating the O(1) space requirement. This means we can hold a single extra array. This is no problem. One way we can solve this problem is by doing 2 passes. This is when we pass through the input array first in order and record the prefix product into our created output array. We can then pass through the input array again backwards and hold a postfix product. We can multiply our variable we are holding by the index - 1 in our output array and assign that to our current position in the output array making sure to update our variable after the assignment so that our first value can be the multiplied by 1 (our total). However in order to avoid going out of bounds we stop at index 1 on our backwards pass and we then simply assign the index of 0 of our output array to the value of our postfix variable.

Ques: Longest subsequence with limited sum (leetcode 2389)
Code:
```
class Solution {
    public int[] answerQueries(int[] nums, int[] queries) {
        Arrays.sort(nums);
        for (int i = 1; i < nums.length; i++)
            nums[i] += nums[i - 1];

        int m = queries.length;
        int[] ans = new int[m];
        for (int i = 0; i < m; i++) {
            int index = binarySearch(nums, queries[i]);
            ans[i] = index;
        }
        return ans;
    }

    private int binarySearch(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] == target)
                return mid + 1;
            if (nums[mid] < target)
                left = mid + 1;
            else
                right = mid - 1;
        }
        return nums[left] > target ? left : left + 1;
    }
}
```

Time Complexity: O(NlogN), n for the linear loop on the array the logn for the nested upper_bound
Space Complexity: O(N)

Explanation:
Since we want to find the subarray sum that satisfies the condition, we can

sort all the numbers and then find their cummulative sum. This ensures that we
get cummulative sum in ascending order.
Then using binary search we look for the 1st sum that is greater than the query number.
The distance from 0th index to that index-1 is the length of the longest such subarray.

1402. Reducing Dishes
A chef has collected data on the satisfaction level of his n dishes. Chef can cook any dish in 1 unit of time.
Like-time coefficient of a dish is defined as the time taken to cook that dish including previous dishes multiplied by its satisfaction level i.e. time[i] * satisfaction[i].
Return the maximum sum of like-time coefficient that the chef can obtain after preparing some amount of dishes.
Dishes can be prepared in any order and the chef can discard some dishes to get this maximum value.

Example 1:

Input: satisfaction = [-1,-8,0,5,-9]
Output: 14
Explanation: After Removing the second and last dish, the maximum total like-time coefficient will be equal to (-1*1 + 0*2 + 5*3 = 14).
Each dish is prepared in one unit of time.
Example 2:

Input: satisfaction = [4,3,2]
Output: 20
Explanation: Dishes can be prepared in any order, (2*1 + 3*2 + 4*3 = 20)
Example 3:

Input: satisfaction = [-1,-4,-5]
Output: 0
Explanation: People do not like the dishes. No dish is prepared.

Code:
```
class Solution {
    public int maxSatisfaction(int[] A) {
        Arrays.sort(A);
        int res = 0, total = 0, n = A.length;
        for (int i = n - 1; i >= 0 && A[i] > -total; --i) {
            total += A[i];
            res += total;
        }
        return res;
    }
}
```
Time O(NlogN), to sort the array
Space O(1), only constant space is used

Explanation:If we cook some dishes, they must be the most satisfied among all choices.
Another important observation is that, we will cook the dish with small satisfication,
and leave the most satisfied dish in the end.
We choose dishes from most satisfied.
Everytime we add a new dish to the menu list,all dishes on the menu list will be cooked one time unit later, so the result += total satisfaction on the list. We'll keep doing this as long as A[i] + total > 0.