1. Given an array prices[] of length N, representing the prices of the stocks on different days, the task is to find the maximum profit possible for buying and selling the stocks on different days using transactions where at most one transaction is allowed.

Note: Stock must be bought before being sold.

Input: prices[] = {7, 1, 5, 3, 6, 4]



Output: 5

Explanation:

The lowest price of the stock is on the 2nd day, i.e. price = 1. Starting from the 2nd day, the highest price of the stock is witnessed on the 5th day, i.e. price = 6.

Therefore, maximum possible profit = 6 – 1 = 5.

Solution:

**static** **int** maxProfit(**int** prices[], **int** n)

  {

**int** buy = prices[0], max\_profit = INT\_MIN;

**for** (**int** i = 1; i < n; i++) {

      // Checking for lower buy value

**if** (prices[i]<buy)

        buy = prices[i];

      // Checking for higher profit

**else** **if** (prices[i] - buy > max\_profit)

        max\_profit = prices[i] - buy;

    }

**return** max\_profit;

  }

TC: O(n)

SC: O(1)

1. Given a list containing future prediction of share prices, find the maximum profit earned by buying and selling shares any number of times with the constraint, a new transaction can only start after the previous transaction is complete, i.e., we can only hold at most one share at a time.

For example:

Stock Prices: {1, 5, 2, 3, 7, 6, 4, 5}

Total profit earned is 10

Buy on day 1 and sell on day 2

Buy on day 3 and sell on day 5

Buy on day 7 and sell on day 8

Solution:

public static int findMaxProfit(int[] price)

    {

        // keep track of the maximum profit gained

        int profit = 0;



        // initialize the local minimum to the first element's index

        int j = 0;

        // start from the second element

        for (int i = 1; i < price.length; i++)

        {



            // update the local minimum if a decreasing sequence is found

            if (price[i]<price[i-1]) {

                j = i;

            }

            // sell shares if the current element is the peak,

            // i.e., (`previous <= current > next`)

            if (price[i - 1] <= price[i] &&

                (i + 1 == price.length || price[i] > price[i + 1]))

            {

                profit += (price[i] - price[j]);

                System.out.printf("Buy on day %d and sell on day %d\n", j + 1, i + 1);

            }



        }

        return profit;

    }

TC: O(n)

SC: O(1)

1. Given an unsorted array of integers, find a subarray that adds to a given number. If there is more than one subarray with the sum of the given number, print any of them.

Examples:

Input: arr[] = {1, 4, 20, 3, 10, 5}, targetsum = 33

Output: Sum found between indexes 2 and 4

Explanation: Sum of elements between indices

2 and 4 is 20 + 3 + 10 = 33

**Solution:**

**public** **static** **void** subArraySum(**int**[] arr, **int** n, **int** targetsum)

    {

        // cur\_sum to keep track of cumulative sum till that

        // point

**int** cur\_sum = 0;

**int** start = 0;

**int** end = -1;

        HashMap<Integer, Integer> hashMap = **new** HashMap<>();

**for** (**int** i = 0; i < n; i++) {

            cur\_sum = cur\_sum + arr[i];

            // check whether cur\_sum - sum = 0, if 0 it

            // means the sub array is starting from index 0-

            // so stop

**if** (cur\_sum - targetsum == 0) {

                start = 0;

                end = i;

**break**;

            }

            // if hashMap already has the value, means we

            // already

            // have subarray with the sum - so stop

**if** (hashMap.containsKey(cur\_sum - targetsum)) {

                start = hashMap.get(cur\_sum - targetsum) + 1;

                end = i;

**break**;

            }

            // if value is not present then add to hashmap

            hashMap.put(cur\_sum, i);

        }

        // if end is -1 : means we have reached end without

        // the sum

**if** (end == -1) {

            System.out.println(

                "No subarray with given sum exists");

        }

**else** {

            System.out.println("Sum found between indexes "

                               + start + " to " + end);

        }

    }

TC: O(n)

SC: O(n)

1. Given an array containing only 0s and 1s, find the largest subarray which contains equal no of 0s and 1s.

Input: arr[] = {0, 0, 1, 1, 0}



Output: 0 to 3 Or 1 to 4

Solution:

// Returns largest subarray with

    // equal number of 0s and 1s

**int** maxLen(**int** arr[], **int** n)

    {

        // Creates an empty hashMap hM

        HashMap<Integer, Integer> hM

            = **new** HashMap<Integer, Integer>();

        // Initialize sum of elements

**int** sum = 0;

        // Initialize result

**int** max\_len = 0;

**int** ending\_index = -1;

**int** start\_index = 0;

**for** (**int** i = 0; i < n; i++) {

            arr[i] = (arr[i] == 0) ? -1 : 1;

        }

        // Traverse through the given array

**for** (**int** i = 0; i < n; i++) {

            // Add current element to sum

            sum += arr[i];

            // To handle sum=0 at last index

**if** (sum == 0) {

                max\_len = i + 1;

                ending\_index = i;

            }

            // If this sum is seen before,

            // then update max\_len if required

**if** (hM.containsKey(sum)) {

**if** (max\_len < i - hM.get(sum)) {

                    max\_len = i - hM.get(sum);

                    ending\_index = i;

                }

            }

**else** // Else put this sum in hash table

                hM.put(sum, i);

        }

**for** (**int** i = 0; i < n; i++) {

            arr[i] = (arr[i] == -1) ? 0 : 1;

        }

**int** end = ending\_index - max\_len + 1;

        System.out.println(end + " to " + ending\_index);

**return** max\_len;

    }

TC: O(n)

SC: O(n)

1. Given an integer array arr[], the task is to find the length of the longest subarray with an equal number of odd and even elements.

Input: arr[] = {1, 2, 1, 2}

Output: 4

Explanation:

Subarrays in the given array are –

{{1}, {1, 2}, {1, 2, 1}, {1, 2, 1, 2}, {2}, {2, 1}, {2, 1, 2}, {1}, {1, 2}, {2}}

where the length of the longest subarray with an equal number of even and odd elements is 4 – {1, 2, 1, 2}

**Solution:**

**static** **int** maxSubarrayLength(**int** []A, **int** N)

{

    // Initialize variable to store result

**int** maxLen = 0;

    // Initialize variable to store sum

**int** curr\_sum = 0;

    // Create an empty map to store

    // index of the sum

    HashMap<Integer,Integer> hash = **new** HashMap<Integer,Integer>();

    // Loop through the array

**for** (**int** i = 0; i < N; i++)

    {

**if** (A[i] % 2 == 0)

            curr\_sum -= 1;

**else**

            curr\_sum += 1;

        // Check if number of even and

        // odd elements are equal

**if** (curr\_sum == 0)

            maxLen = Math.max(maxLen, i + 1);

        // If curr\_sum already exists in map

        // we have a subarray with 0 sum, i.e,

        // equal number of odd and even number

**if** (hash.containsKey(curr\_sum))

             maxLen = Math.max(maxLen,i - hash.get(curr\_sum));

        // Store the index of the sum

**else** {

             hash.put(curr\_sum, i);

         }

     }

**return** maxLen;

}

TC: O(n)

SC: O(n)

1. Find the majority element in the array. A majority element in an array A[] of size n is an element that appears more than n/2 times (and hence there is at most one such element).

Input : A[]={3, 3, 4, 2, 4, 4, 2, 4, 4}

Output : 4

Explanation: The frequency of 4 is 5 which is greater than the half of the size of the array size.

**Solution**

/\* Function to print Majority Element \*/

void printMajority(int a[], int size)

{

/\* Find the candidate for Majority\*/

int cand = findCandidate(a, size);

/\* Print the candidate if it is Majority\*/

if (isMajority(a, size, cand))

System.out.println(" " + cand + " ");

else

System.out.println("No Majority Element");

}

/\* Function to find the candidate for Majority \*/

int findCandidate(int a[], int size)

{

int maj\_index = 0, count = 1;

int i;

for (i = 1; i < size; i++) {

if (a[maj\_index] == a[i])

count++;

else

count--;

if (count == 0) {

maj\_index = i;

count = 1;

}

}

return a[maj\_index];

}

/\* Function to check if the candidate occurs more

than n/2 times \*/

boolean isMajority(int a[], int size, int cand)

{

int i, count = 0;

for (i = 0; i < size; i++) {

if (a[i] == cand)

count++;

}

if (count > size / 2)

return true;

else

return false;

}

TC: O(n)

SC: O(1)