Two Pointers Technique

Two pointers is really easy and effective technique that is typically used for

- Two Sum in Sorted Arrays
- Closest two sum
- Three Sum
- Four Sum
- Trapping Rain water and many other popular interview questions.

Two Sum in Sorted Array:

Given a sorted array arr (sorted in ascending order) and a target, find if there exists any pair of elements (arr[i], arr[j]) such that their sum is equal to target.

Input: arr[] = {10, 20, 35, 50} target = 70

Output: Yes ⇒ 20+50 = 70

Input: arr[] = {10, 20, 30}, target = 70

output: NO

Input: arr[] = {-8, 1, 4, 6, 10, 45}, target = 16

output: Yes ⇒ 10+6 = 16

Approach #1 :- (Brute Force Approach)

Generate all pairs and check if any of them add up to target.

```
for(int i=0; i<N; i++){
    for(int j=i+1; j<N; j++){
        if (arr[i] + arr[j] == target){
            return true;
        }
    }
}
return false;
```

TC: $O(N^2)$

SC: $O(1)$

Approach #2:- (Binary Search Approach)

See, we know that array is sorted, so let's take help of this condition.

1) pick one element from array say arr[i].
2) calculate it's complement i.e., target - arr[i] = arr[j]
3) BinarySearch for that complement in i+1 to n

```
boolean binary_search(int[] arr, int left, int right, int target){
    while (left <= right){
        int mid = (left + right)/2;
        if (arr[mid] == target)
            return true;
        if (arr[mid] < target){
            left = mid+1;
        }else right = mid-1;
    }
    return false;
```

```
static boolean twoSum(int[] arr, int target){
    for(int i=0;i<N;i++){
        int complement = target-arr[i];
        if(binarySearch(arr, i+1, arr.length-1, complement){
            return true;
        }
    }
    return false;
}
```

T.C: O(NlogN)

S.C: O(1)

Approach #3:- (Hashset Approach)

```
HashSet<Integer> hs= new HashSet<>();
for(int i=0; i< arr.length; i++){
    int complement= target-arr[i];
    if( hs.contains(complement) ){
        return true;
    }
    hs.add(arr[i]);
}
return false.
```

T.C: O(N)

S.C: O(N)

Approach #4: (Two Pointers)

Assume two pointers say left, right

left start at index '0'

right at index 'n-1'

→ calculate sum = arr[left] + arr[right]

→ if sum == target ⇒ return true

→ if sum < target ⇒ then we should increase sum hence ↑ left

→ if sum > target ⇒ then we should decrease sum hence ↓ right

```
int left=0;
int right=n-1;

while (left<right){
        int sum= arr[left] + arr[right];
        if (sum == target) return true;
        else if (sum< target) left++;
        else if (sum>target) right--;
}
return false;
```

T.C: O(N)

S.C: O(1)

2. Best Time to Buy and Sell Stock

You are given an array prices where prices[i] → price of a given stock on the ith day.

You want to maximize your profit by chosing a single day to buy one stock and choosing a different day in the future to sell that stock.

Return the maximum profit you can achieve from this transaction. if you cannot achieve any profit return 0.

Approach #1:- (Brute Force Approach)

Consider all pairs

```
int max=0;
for(int i=0; i<n; i++){
    for(int j=i+1; j<n; j++){
        if(prices[j]-prices[i] > max){
            max = prices[j]-prices[i];
        }
    }
}
return max;
```

T.C: O(N²)

S.C: O(1)

Approach #2:- (Postfix Array Approach)

generate postfix array Pf where Pf[i] → denotes max element in [i to n-1]

```
Pf[n-1] = prices[n-1];
for(int i=n-2; i>=0; i--){
    Pf[i] = Math.max(Pf[i+1], prices[i]);
}
```

```
int max=0;
for(int i=0 ; i<n-1 ; i++ ){
        if ( pf[i+1] - prices[i] > max){
            max = pf[i+1] - prices[i];
        }
}
return max;
```

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

Approach #3:- (min price one pass Approach) & optimal)

Keep track of minprice
maintain max profit achievable so far.

```
int minprice = Integer.MAX_VALUE;
int max = 0;
for (int i=0; i<n ;i++){
        if (prices[i] < minPrice){
            minPrice = prices[i];
        }
        else if ( prices[i] - minPrice > max){
            max = prices[i] - minPrice;
        }
}
return max;
```

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

Approach #4: (Dynamic Programming Interpretation & Kadane's Algo Analogy):

⇒ find difference array

say diff[i] = prices[i] - prices[i-1]

⇒ Now find maximum sum subarray of diff, which gives us the maximum profit

```
int maxEndingHere = 0;
int maxSoFar = 0;

for (int i=1; i<prices.length; i++){
    int diff = prices[i] - prices[i-1];
    maxEndingHere = Math.max(maxEndingHere + diff, 0);
    maxSoFar = Math.max(maxSoFar, maxEndingHere);
}
return maxSoFar;
```

T.C: O(N)

S.C: O(1)

3. Valid Parenthesis

Given string s containing just characters '(', ')', '{', '}', '[' and ']' determine if the input string is valid.

An input string is valid if

→ Open brackets must be closed by the same type of brackets

→ Open brackets must be closed in the correct order.

→ Every closed bracket has a corresponding open bracket of the same type.

Example#1

Input :- s = "()"

Output: true

Example#2

Input :- s = "()[]{}"

output: true

Example #3:-

Input :- s = "(]"

output: false

Example #4:-

Input :- s = "([])"

Output: true.

Approach#1 : (Stack Based Approach).

```
Stack<Character> st = new Stack<>();

for(int i=0; i<s.length(); i++){
    char cur = s.charAt(i);
    if(!st.isEmpty()){
        char last = st.peek();
        if(isPair(cur, last)){
            st.pop();
            continue;
        }
    }
    st.push(cur);
}

private boolean isPair(char cur, char last){
    return (last=='(' && cur==')') ||
           (last=='{' && cur=='}') ||
           (last=='[' && cur==']');
}
```

T.C: O(N)

S.C: O(N)