

## Valid Anagram

### Purpose:

The function checks whether two strings s and t are **anagrams** of each other.

---

### Definition:

Two strings are anagrams if they contain the **same characters** in the **same frequency**, regardless of the order.

---

### Logic Used:

1. Use a **hash map (unordered\_map)** to count the frequency of characters in string s.
  2. Subtract the frequency of characters based on string t.
  3. If all values in the map are 0, then s and t are anagrams.
- 

### Step-by-step Explanation:

#### Step 1: Create a hash map mp

```
unordered_map<int, int> mp;
```

- Used to store frequency of characters.

#### Step 2: Traverse the first string s

```
for (auto ele : s) {  
    mp[ele]++;  
}
```

- Count how many times each character appears in s.

#### Step 3: Traverse the second string t

```
for (auto ele : t) {  
    mp[ele]--;  
}
```

- Decrease the count for each character found in t.

#### Step 4: Check all values in the map

```
for (auto it : mp) {  
    if (it.second != 0) {
```

```
    return false;
}
}
```

- If any character count is not zero, then s and t are not anagrams.

#### Step 5: Return true

```
return true;
```

- All character frequencies matched.
- 

#### ✅ Time Complexity:

- $O(n + m)$  where  $n$  = length of s,  $m$  = length of t.

#### ✅ Space Complexity:

- $O(1)$  (Since there are at most 26 lowercase letters or 128 ASCII characters, considered constant space).
- 

#### 🔄 Example:

s = "listen", t = "silent" → true

s = "aacc", t = "ccac" → false

#### C++ CODE:

##### Class solution {

##### Public:

```
bool isanagram(string s, string t) {
    unordered_map<char, int> mp;
    for (auto ele : s) {
        mp[ele]++;
    }
    for (auto ele : t) {
        mp[ele]--;
    }
    for (auto it : mp) {
```

```
        if (it.second != 0) {  
            return false;  
        }  
    }  
    return true;  
}  
};
```

**Java code:**

```
class Solution {  
    public boolean isAnagram(String s, String t) {  
        if (s.length() != t.length()) return false;  
        HashMap<Character, Integer> counts = new HashMap<>();  
        int n = s.length();  
  
        for (int i = 0; i < n; i++) {  
            counts.put(s.charAt(i), counts.getDefault(s.charAt(i), 0) + 1);  
        }  
        for (int i = 0; i < n; i++) {  
            counts.put(t.charAt(i), counts.getDefault(t.charAt(i), 0) - 1);  
        }  
        for (int count : counts.values()) {  
            if (count != 0)  
                return false;  
        }  
  
        return true;  
    }  
}
```

## Maximum Sum Subarray Ending at $i \rightarrow P1[i]$

### ✓ Brute Force Approach ( $O(N^2)$ )

`vector<int> P1(n, 0); // P1[i] = max subarray sum ending at i`

```
for (int i = 0; i < n; i++) {
    int sum = 0, t = INT_MIN;
    for (int j = i; j >= 0; j--) {
        sum += B[j];    // sum of B[j..i]
        t = max(t, sum); // update max subarray ending at i
    }
    P1[i] = t;
}
```

### ✓ Optimized kadane's approach ( $O(n)$ )

`vector<int> p1(n, 0); // p1[i] = max subarray sum ending at i`

`p1[0] = b[0]; // base case`

```
for (int i = 1; i < n; i++) {
    p1[i] = max(p1[i - 1] + b[i], b[i]); // either extend or restart
}
```

`int maxsum = *max_element(p1.begin(), p1.end()); // overall max`

### ■ logic:

at every index  $i$ , choose:

- $p1[i - 1] + b[i] \rightarrow$  extend previous subarray
- $b[i] \rightarrow$  start new subarray at  $i$

store the maximum of these in  $p1[i]$ .

## MAXIMUM SUBARRAY : KADANE ALGORITHM

```
int maxsubarraysum(vector<int> &arr) {  
    int n=arr.size();  
    int sum=0;  
    int maxi=INT_MIN;  
    for(int i=0;i<n;i++){  
        sum+=arr[i];  
        if(sum>maxi){  
            maxi=sum;  
        }  
        if(sum<0){  
            sum=0;  
        }  
    }  
    return maxi;  
}
```

### KEY POINTS:

- SUM IS RESET TO 0 WHEN IT GOES NEGATIVE.
- TRACKS OVERALL MAXI DURING THE LOOP.
- HANDLES NEGATIVE-ONLY ARRAYS CORRECTLY BECAUSE MAXI STARTS AT INT\_MIN.
- SLIGHTLY SIMPLER TO READ AND OFTEN USED IN INTERVIEWS.

## ANOTHER WAY:

### CODE 1: SAFE & STANDARD KADANE'S

```
int maxsubarraysum(vector<int> &arr) {  
    int res = arr[0];  
    int maxending = arr[0];  
  
    for (int i = 1; i < arr.size(); i++) {  
        maxending = max(maxending + arr[i], arr[i]);  
    }  
    return res;  
}
```

```
    res = max(res, maxending);  
}  
return res;  
}
```

#### KEY POINTS:

- INITIALIZED WITH `ARR[0]`, SO WORKS EVEN IF ALL ELEMENTS ARE NEGATIVE.
- USES `MAXENDING` TO KEEP TRACK OF THE BEST SUM ENDING AT `I`.
- `RES` ALWAYS TRACKS THE OVERALL MAX.
- NEVER RESETS `MAXENDING` TO 0 — ALWAYS CHOOSES THE BETTER OF:
  - EXTENDING PREVIOUS SUBARRAY
  - STARTING NEW AT CURRENT INDEX

#### FINAL CONCLUSION:

BOTH ARE VALID KADANE'S ALGORITHM IMPLEMENTATIONS:

- USE CODE 1 IF:
  - YOU NEED INTERMEDIATE RESULTS (LIKE `P1[I]`)
  - YOU'RE IMPLEMENTING A DP-STYLE SOLUTION
- USE CODE 2 IF:
  - YOU WANT A SIMPLE AND FAST IMPLEMENTATION
  - YOU'RE IN A CODING INTERVIEW OR CONTEST

## **Second Largest Element in an Array**

### **One Pass Search**

```
int getSecondLargest(vector<int> &arr) {  
    int largest=INT_MIN;  
    int seclargest=INT_MIN;  
    for(int i=0;i<arr.size();i++){  
        if(arr[i]>largest){  
            seclargest=largest;  
            largest=arr[i];  
  
        }  
        else if(arr[i]>seclargest && arr[i]!=largest){  
            seclargest=arr[i];  
        }  
    }  
    if(seclargest==INT_MIN){  
        seclargest=-1;  
    }  
    return seclargest;  
}
```

**TIME COMPLEXITY:  $O(N)$ , AS WE ARE TRAVERSING THE ARRAY ONLY ONCE.**

**AUXILIARY SPACE:  $O(1)$**

## **REPLACE ELEMENTS WITH GREATEST ELEMENT ON RIGHT SIDE**

### **APPROACH 1:**

```
vector<int> replaceelements(vector<int>& arr) {  
    for(int i=0;i<arr.size()-1;i++){  
        arr[i]=*max_element(arr.begin()+i+1,arr.end());  
    }  
}
```

```
arr[arr.size()-1]=-1;  
return arr;  
}
```

#### **APPROACH 2:**

```
vector<int> replaceElements(vector<int>& arr) {  
    int num = -1; // Initialize num with -1, since the last element should be replaced with -1  
  
    // Traverse the array from the end (right to left)  
    for (int i = arr.size() - 1; i >= 0; i--) {  
        int temp = arr[i]; // Store the current element  
        arr[i] = num;      // Replace current element with the max value to its right (initially -1)  
        num = max(num, temp); // Update num to be the max of itself and the current element  
    }  
  
    return arr; // Return the modified array  
}
```