Count Inversions of size three in a give array

Given an array arr[] of size n. Three elements arr[i], arr[j] and arr[k] form an inversion of size 3 if a[i] > a[j] >a[k] and i < j < k. Find total number of inversions of size 3.

Example:

```
Input:  {8, 4, 2, 1}
Output: 4
The four inversions are (8,4,2), (8,4,1), (4,2,1) and (8,2,1).

Input:  {9, 6, 4, 5, 8}
Output:  2
The two inversions are {9, 6, 4} and {9, 6, 5}
```

We have already discussed inversion count of size two by merge sort, Self Balancing BST and BIT.

Simple approach :- Loop for all possible value of i, j and k and check for the condition a[i] > a[j] > a[k] and i < j < k.

C++

```
// A Simple C++ O(n^3)  program to count inversions of size 3
#include<bits/stdc++.h>
using namespace std;

// Returns counts of inversions of size three
int getInvCount(int arr[],int n)
{
    int invcount = 0;  // Initialize result

    for (int i=0; i<n-2; i++)
    {
        for (int j=i+1; j<n-1; j++)
        {
            if (arr[i]>arr[j])
            {
                for (int k=j+1; k<n; k++)
                {
                    if (arr[j]>arr[k])
                        invcount++;
                }
            }
        }
    }
    return invcount;
}

// Driver program to test above function
int main()
{
    int arr[] = {8, 4, 2, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Inversion Count : " << getInvCount(arr, n);
    return 0;
}
```

Java

```
// A simple Java implementation  to count inversion of size 3
class Inversion{

    // returns count of inversion of size 3
    int getInvCount(int arr[], int n)
    {
        int invcount = 0; // initialize result

        for(int i=0 ; i< n-2; i++)
        {
            for(int j=i+1; j<n-1; j++)
            {
                if(arr[i] > arr[j])
                {
                    for(int k=j+1; k<n; k++)
                    {
                        if(arr[j] > arr[k])
                            invcount++;
                    }
                }
            }
        }
        return invcount;
    }

    // driver program to test above function
    public static void main(String args[])
    {
        Inversion inversion = new Inversion();
        int arr[] = new int[] {8, 4, 2, 1};
        int n = arr.length;
        System.out.print("Inversion count : " +
                    inversion.getInvCount(arr, n));
    }
}
// This code is contributed by Mayank Jaiswal
```

Python

```
# A simple python O(n^3) program to count inversions of size 3

# Returns counts of inversions of size threee
def getInvCount(arr):
    n = len(arr)
    invcount = 0  #Initialize result
    for i in range(0,n-1):
        for j in range(i+1 , n):
                if arr[i] > arr[j]:
                    for k in range(j+1 , n):
                        if arr[j] > arr[k]:
                            invcount += 1
    return invcount

# Driver program to test above function
arr = [8 , 4, 2 , 1]
print "Inversion Count : %d" %(getInvCount(arr))

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Inversion Count : 4
```

Time complexity of this approach is : O(n^3)

Better Approach :
We can reduce the complexity if we consider every element arr[i] as middle element of inversion, find all the numbers greater than a[i] whose index is less than i, find all the numbers which are smaller than a[i] and index is more than i. We multiply the number of elements greater than a[i] to the number of elements smaller than a[i] and add it to the result. Below is C++ implementation of the idea.

C++

```
// A O(n^2) C++  program to count inversions of size 3
#include<bits/stdc++.h>
using namespace std;

// Returns count of inversions of size 3
int getInvCount(int arr[], int n)
{
    int invcount = 0;  // Initialize result

    for (int i=1; i<n-1; i++)
    {
        // Count all smaller elements on right of arr[i]
        int small = 0;
        for (int j=i+1; j<n; j++)
            if (arr[i] > arr[j])
                small++;

        // Count all greater elements on left of arr[i]
        int great = 0;
        for (int j=i-1; j>=0; j--)
            if (arr[i] < arr[j])
                great++;

        // Update inversion count by adding all inversions
        // that have arr[i] as middle of three elements
        invcount += great*small;
    }

    return invcount;
}

// Driver program to test above function
int main()
{
    int arr[] = {8, 4, 2, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Inversion Count : " << getInvCount(arr, n);
    return 0;
}
```

Java

```
// A O(n^2) Java  program to count inversions of size 3

class Inversion {

    // returns count of inversion of size 3
    int getInvCount(int arr[], int n)
    {
        int invcount = 0; // initialize result

        for (int i=0 ; i< n-1; i++)
        {
            // count all smaller elements on right of arr[i]
            int small=0;
            for (int j=i+1; j<n; j++)
                if (arr[i] > arr[j])
                    small++;

            // count all greater elements on left of arr[i]
            int great = 0;
            for (int j=i-1; j>=0; j--)
                        if (arr[i] < arr[j])
                            great++;

            // update inversion count by adding all inversions
            // that have arr[i] as middle of three elements
            invcount += great*small;
        }
        return invcount;
    }
    // driver program to test above function
    public static void main(String args[])
    {
        Inversion inversion = new Inversion();
        int arr[] = new int[] {8, 4, 2, 1};
        int n = arr.length;
        System.out.print("Inversion count : " +
                       inversion.getInvCount(arr, n));
    }
}

// This code has been contributed by Mayank Jaiswal
```

Output:

```
Inversion Count : 4
```

Time Complexity of this approach : O(n^2)

Binary Indexed Tree Approach :

Like inversions of size 2, we can use Binary indexed tree to find inversions of size 3. It is strongly recommended to refer below article first.

Count inversions of size two Using BIT

The idea is similar to above method. We count the number of greater elements and smaller elements for all the elements and then multiply greater[] to smaller[] and add it to the result.

Solution :

1. To find out the number of smaller elements for an index we iterate from n-1 to 0. For every element a[i] we calculate the getSum() function for (a[i]-1) which gives the number of elements till a[i]-1.
2. To find out the number of greater elements for an index we iterate from 0 to n-1. For every element a[i] we calculate the sum of numbers till a[i] (sum smaller or equal to a[i]) by getSum() and subtract it from i (as i is the total number of element till that point) so that we can get number of elements greater than a[i].

Like we did for inversions of size 2, here also we convert the input array to an array with values from 1 to n so that the size of BIT remains O(n), and getSum() and update() functions take O(Log n) time. For example, we convert arr[] = {7, -90, 100, 1} to arr[] = {3, 1, 4 ,2 }.

Below is the implementation of above idea.

C

```
// C++ program to count inversions of size three using
// Binary Indexed Tree
#include<bits/stdc++.h>
using namespace std;

// Returns sum of arr[0..index]. This function assumes
// that the array is preprocessed and partial sums of
// array elements are stored in BITree[].
int getSum(int BITree[], int index)
{
    int sum = 0; // Initialize result

    // Traverse ancestors of BITree[index]
    while (index > 0)
    {
        // Add current element of BITree to sum
        sum += BITree[index];

        // Move index to parent node in getSum View
        index -= index & (-index);
    }
    return sum;
}

// Updates a node in Binary Index Tree (BITree) at given index
// in BITree.  The given value 'val' is added to BITree[i] and
// all of its ancestors in tree.
void updateBIT(int BITree[], int n, int index, int val)
{
    // Traverse all ancestors and add 'val'
    while (index <= n)
    {
       // Add 'val' to current node of BI Tree
       BITree[index] += val;

       // Update index to that of parent in update View
       index += index & (-index);
    }
}

// Converts an array to an array with values from 1 to n
// and relative order of smaller and greater elements remains
// same.  For example, {7, -90, 100, 1} is converted to
// {3, 1, 4 ,2 }
void convert(int arr[], int n)
{
    // Create a copy of arrp[] in temp and sort the temp array
    // in increasing order
    int temp[n];
    for (int i=0; i<n; i++)
        temp[i] = arr[i];
    sort(temp, temp+n);

    // Traverse all array elements
    for (int i=0; i<n; i++)
    {
        // lower_bound() Returns pointer to the first element
        // greater than or equal to arr[i]
        arr[i] = lower_bound(temp, temp+n, arr[i]) - temp + 1;
    }
}

// Returns count of inversions of size three
int getInvCount(int arr[], int n)
{
    // Convert arr[] to an array with values from 1 to n and
    // relative order of smaller and greater elements remains
    // same.  For example, {7, -90, 100, 1} is converted to
    //  {3, 1, 4 ,2 }
```

```
    convert(arr, n);

    // Create and initialize smaller and greater arrays
    int greater1[n], smaller1[n];
    for (int i=0; i<n; i++)
        greater1[i] = smaller1[i] = 0;

    // Create and initialize an array to store Binary
    // Indexed Tree
    int BIT[n+1];
    for (int i=1; i<=n; i++)
        BIT[i]=0;

    for(int i=n-1; i>=0; i--)
    {
        smaller1[i] = getSum(BIT, arr[i]-1);
        updateBIT(BIT, n, arr[i], 1);
    }

    // Reset BIT
    for (int i=1; i<=n; i++)
        BIT[i] = 0;

    // Count greater elements
    for (int i=0; i<n; i++)
    {
        greater1[i] = i - getSum(BIT,arr[i]);
        updateBIT(BIT, n, arr[i], 1);
    }

    // Compute Inversion count using smaller[] and
    // greater[].
    int invcount = 0;
    for (int i=0; i<n; i++)
        invcount += smaller1[i]*greater1[i];

    return invcount;
}

// Driver program to test above function
int main()
{
    int arr[] = {8, 4, 2, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Inversion Count : " << getInvCount(arr, n);
    return 0;
}
```

Java

```
// Java program to count inversions of size three using
// Binary Indexed Tree
import java.util.Arrays;

class BinaryTree {

   // Returns sum of arr[0..index]. This function assumes
   // that the array is preprocessed and partial sums of
   // array elements are stored in BITree[].
    int getSum(int BITree[], int index) {
        int sum = 0; // Initialize result

        // Traverse ancestors of BITree[index]
        while (index > 0) {

            // Add current element of BITree to sum
            sum += BITree[index];

            // Move index to parent node in getSum View
            index -= index & (-index);
        }
```

```
        }
        return sum;
    }

    // Updates a node in Binary Index Tree (BITree) at given
    // index in BITree.  The given value 'val' is added to
    // BITree[i] and  all of its ancestors in tree.
    void updateBIT(int BITree[], int n, int index, int val) {

        // Traverse all ancestors and add 'val'
        while (index <= n) {

            // Add 'val' to current node of BI Tree
            BITree[index] += val;

            // Update index to that of parent in update View
            index += index & (-index);
        }
    }

    // Converts an array to an array with values from 1 to n
    // and relative order of smaller and greater elements remains
    // same.  For example, {7, -90, 100, 1} is converted to
    // {3, 1, 4 ,2 }
    void convert(int arr[], int n) {

        // Create a copy of arrp[] in temp and sort the temp array
        // in increasing order
        int temp[]= new int[n];
        for (int i = 0; i < n; i++) {
            temp[i] = arr[i];
        }
        Arrays.sort(temp);

        // Traverse all array elements
        for (int i = 0; i < n; i++) {

            // lower_bound() Returns pointer to the first element
            // greater than or equal to arr[i]
            arr[i] = Arrays.binarySearch(temp, arr[i])  + 1;
        }
    }

    // Returns count of inversions of size three
    int getInvCount(int arr[], int n) {

        // Convert arr[] to an array with values from 1 to n and
        // relative order of smaller and greater elements remains
        // same.  For example, {7, -90, 100, 1} is converted to
        //  {3, 1, 4 ,2 }
        convert(arr, n);

        // Create and initialize smaller and greater arrays
        int greater1[]= new int[n];
        int smaller1[]= new int[n];
        for (int i = 0; i < n; i++) {
            greater1[i] = smaller1[i] = 0;
        }

        // Create and initialize an array to store Binary
        // Indexed Tree
        int BIT[]= new int[n+1];
        for (int i = 1; i <= n; i++) {
            BIT[i] = 0;
        }

        for (int i = n - 1; i >= 0; i--) {
            smaller1[i] = getSum(BIT, arr[i] - 1);
            updateBIT(BIT, n, arr[i], 1);
        }

        // Reset BIT
        for (int i = 1; i <= n; i++) {
```

```
            BIT[i] = 0;
        }

        // Count greater elements
        for (int i = 0; i < n; i++) {
            greater1[i] = i - getSum(BIT, arr[i]);
            updateBIT(BIT, n, arr[i], 1);
        }

        // Compute Inversion count using smaller[] and
        // greater[].
        int invcount = 0;
        for (int i = 0; i < n; i++) {
            invcount += smaller1[i] * greater1[i];
        }

        return invcount;
    }

    // Driver program to test above function
    public static void main(String args[]) {
        BinaryTree tree = new BinaryTree();
        int[] arr = new int[]{8, 4, 2, 1};
        int n = arr.length;
        System.out.print( "Inversion Count : " +
                          tree.getInvCount(arr, n));
    }
}

// This code is contributed by Mayank Jaiswal
```

Output:

```
Inversion Count : 4
```

Time Complexity : O(n log n)
Auxiliary Space : O(n)

We can also use Self-Balancing Binary Search Tree to count greater elements on left and smaller on right. Time complexity of this method would also be O(n Log n), But BIT based method is easy to implement.