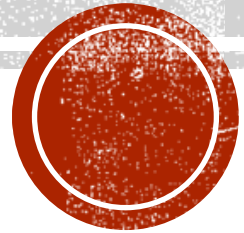
GOOD TRIPLETS

- Jaskamal Kainth



PROBLEM STATEMENT:

- Given 3 array's A[], B[] and C[] of N integers each.
- The task is to find the count of triplets (A[i], B[j], C[k]) such that

  A[i] < B[j] < C[k]

Input:

A[]: {2, 6}

B[]: {3, 5}

C[]: {4, 7}

Output: 3

Triplets are (2,3,4), (2,3,7), (2,5,7)

SOLUTION 1: BRUTE FORCE **O(N^3)**

SOLUTION 2: BRUTE FORCE + BINARY SEARCH **O(N^2*LOG(N))**

SOLUTION 3: BINARY SEARCH – TWICE **O(N*LOG(N))**

SOLUTION 4: SORTING + 3 POINTERS **O(N*LOG(N))**

SOLUTION 5: SORTING + 3 POINTERS + BSEARCH **O(N*LOG(N))**

COUNTING SORT ALGORITHM

SOLUTION 6: SOLUTION 6: C-SORT + 3POINTERS + BSEARCH **O(N)**

SOLUTION 7: CUMULATIVE SUM **O(N)**

SOLUTION 1: BRUTE FORCE

```
for(int i = 1; i <= N; ++i) //Iterate over Array A - O(N)
{
    for(int j = 1; j <= N; ++j)   //Iterate over Array B - O(N)
    {
        for(int k = 1; k <= N; ++k)   //Iterate over Array C - O(N)
        {
            if (A[i] < B[j] && B[j] < C[k])
            {
                count += 1;
            }
        }
    }
}
return count;
// Time Complexity: O(N^3)
```

```
for(int i = 1; i <= N; ++i) // O(N)
{
    for(int j = 1; j <= N; ++j)  // O(N)
    {
        if(A[i] >= B[j]) continue;
        for(int k = 1; k <= N; ++k)  // O(N)
        {
            if (B[j] < C[k])
            {
                count += 1;
            }
        }
    }
}
return count;
// Time Complexity: O(N^3)
```

SOLUTION 2: BRUTE FORCE + BINARY SEARCH

```
sort(C); // O(N*log(N))

for(int i = 1; i <= N; ++i) // O(N)
{
    for(int j = 1; j <= N; ++j)  // O(N)
    {
        if (A[i] >= B[j]) continue;
        // (A[i], B[j], X)
        // Total number of triplets = # of elements in Array C which are greater than B[j]
        // This can be found in O(log(N)) using Binary search, provided array C is sorted.
    }
}
//Time Complexity: O(N*log(N) + O(N^2 * log(N))
```

A: 2 3 5 7 8

B: 3 4 6 7 9

C: 4 6 8 8 9

A: 2 3 5 7 8

B: 3 4 6 7 9

C: 4 6 8 8 9

A: 2 3 5 7 8

B: 3 4 6 7 9

C: 4 6 8 8 9

(2,6,8)
(2,6,8)
(2,6,9)

(3,6,8)
(3,6,8)
(3,6,9)

(5,6,8)
(5,6,8)
(5,6,9)

SOLUTION 3: BINARY SEARCH - TWICE

```
sort(A); // O(N*log(N))
sort(C); // O(N*log(N))

for(int j = 0; j < N; ++j)  // Iterate over Array B - O(N)
{
    int currentElement = B[j];
    // Find the number of elements in Array A which are less    than B[j] = countA
        // Use Binary search on Array A - O(log(N)))

    // Find the number of elements in Array C which are greater than B[j] = countC
        // Use Binary search on Array C - O(log(N)))
    totalCount += (countA * countC);
}
// Time Complexity: O(2*N*log(N)) + O(N * 2 * log(N)) = O(4*N*log(N))
```

A: 2 3 5 7 8

B: 3 4 6 7 9

C: 4 6 8 8 9

```
sort(A); // O(N*log(N))
sort(C); // O(N*log(N))
sort(B); // O(N*log(N))

for(int j = 1; j <= N; ++j)  // Iterate over Array B - O(N)
{
    int currentElement = B[j];

    // OBSERVATION:
    // index "i" is the last index in Array A such that A[i] < B[j]
    // =>
    // There are *exactly* i elements in Array A which are less than B[j]
    // =>
    // There will be *atleast* "i" elements in Array A which are less than B[j+1]
    // because Array B is sorted and B[j+1] > B[j].

    // Similarly for Array C
}
```

SOLUTION 4: SORTING + 3 POINTERS

```
sort(A); // O(N*log(N))
sort(C); // O(N*log(N))
sort(B); // O(N*log(N))

int i = 1; // Index for Array A
int k = 1; // Index for Array C
for(int j = 1; j <= N; ++j)  // Iterate over Array B - O(N)
{
    int currentElement = B[j];
    // Every element in Array A is visited only once
    while(A[i] < B[j]) {
        i += 1;
    }
    // Every element in Array C is visited only once
    while(C[k] < B[j]) {
        k += 1;
    }
    totalCount += (i * (N-k+1));
}
// Time complexity: O(3*N*log(N)) + O(3*N)
```

SOLUTION 5: SORTING + 3 POINTERS +BSEARCH

```
sort(A); // O(N*log(N))
sort(C); // O(N*log(N))
sort(B); // O(N*log(N))

int i = 1; // Index for Array A
int k = 1; // Index for Array C
for(int j = 1; j <= N; ++j)  // Iterate over Array B - O(N)
{
    int currentElement = B[j];
    // Binary search on the Interval [i, N] instead of [1, N]

    // Binary search on the Interval [k, N] instead of [1, N]
}
// Time complexity: O(3*N*log(N)) + O(N + 2*N)
```

COUNTING SORT

```
// countA[] and sortedA[]

for(int i = 1; i <= N; ++i) { // Update counts of all elements of Array A
    countA[ A[i] ] += 1; // CountA[x] = count of number 'x' in Array A
}
for(int i = 1; i <= MAXA; ++i) { // Make cumulative sum array
    countA[i] += count[i-1]; // CountA[x] = number of numbers <= 'x' in Array A
}
for(int i = 1; i <= N; ++i) {
    sortedA[countA[A[i]]] = A[i]; // place A[i] at the position = countA[A[i]]
    countA[A[i]] -= 1; // decrease the position to place the same element
}
// Time complexity: O(2*N + MAXA)
```

SOLUTION 6: C-SORT + 3POINTERS +BSEARCH

```
csort(A); // O(3*N)
csort(C); // O(3*N)
csort(B); // O(3*N)

int i = 1; // Index for Array A
int k = 1; // Index for Array C
for(int j = 1; j <= N; ++j)  // Iterate over Array B - O(N)
{
    int currentElement = B[j];
    // Binary search on the Interval **[i, N]** instead of [1, N]

    // Binary search on the Interval **[k, N]** instead of [1, N]
}
// Time complexity: O(9*N)) + O(N + 2*N) -> O(12*N)
```

SOLUTION 7: CUMULATIVE SUM

```
// Precomputed Cumulative count array for A and C
// countA[] and countC[]
// Above will take O(2*N) + O(2*N)
for(int j = 1; j <= N; ++j)  // Iterate over Array B - O(N)
{
    int currentElement = B[j];
    // C1: #of elements in Array A which are < B[j]
    // => countA[B[j]-1]
    // C2: #of elements in Array C which are >= B[j]
    // => N - #of elements in Array C which are  < B[j]
    // => N - countC[B[j]]
    totalCount += (C1 * C2);
}
// Time complexity: O(4*N)) + O(N) -> O(5*N)
```

FURTHER OPTIMISATIONS:

- Reading *char* is faster than reading *int*.
- Take characters as input using “getchar()”of “fread()” and then convert it into number using empty space as a delimeter.
- User of “register int” instead of “int”.
- Compute answers for only unique values of Array B.
- Compute the maximum and minimum value of Array A,B,C for cleaning the respective count arrays.
- Handle the corner cases where all elements are same in individual Arrays:

```
A: 1 1 1
B: 3 3 3
C: 5 5 5            return -> 3*3*3 = 27;  27 (1,3,5) triplets

A: 3 3 3
B: 1 1 1
C: 5 5 5            return -> 0;
```