Synopsis

Problem Statement

Let A[i...n] be an array of n distinct real numbers. A pair (A[i], A[j]) is said to be an index-value inversion if A[i] = j and A[j] = i . Design an algorithm for counting the number of index-value inversions.

Design Technique

We can obtain an efficient solution for the above problem statement if we use Divide And Conquer Technique. Here, we divide the input array into two halfs. The number of index value inversions from both the halves are added to find the total number of index value inversions in the complete array. This procedure is executed recursively.

We can expect our solution to be similar to that of mergesort.

Data Structure

Array Data Structure is used to solve the above problem statement. We have used this data structure because :

- It is easier to code our algorithm around array data structure.
- It gives an efficient solution.
- We face no notable loss in time or memory efficiency.

Algorithm

```
ALGORITHM  indexValueInversion ( arr[l ..... r] )
  // Input:  array arr[l ... r]
  // Output: count of total index value inversion
             in arr[l ... r]
 (1) count = 0
 (2) if l < r
      (1) mid = l + (r-l)/2
      (2) leftcount = indexValueInversion (arr[l ... m])
      (3) rightcount = indexValueInversion (arr[m+1 ... l])
      (4) count = mergeAndOutput (arr[l ... r], mid)
      (5) count = count + rightcount + leftcount
      (6) return count
 (3) end if
 (4) return count.
```

Algorithm

```
ALGORITHM mergeAndOutput( arr[l.....r], mid )
    // Input: arr[l,.....r] , mid
    // Output: count of index value inversion in
       given array.

    ① count = 0 and i = l
    ② while ( i <= mid )
          ① if ( arr[i] >= m+1 and
                    arr[i] <= r )
                ① j = arr[i];
                ② if ( arr[j] == i )
                      ① count++;
                ③ endif
          ② endif
          ③ i++
    ③ end while
    ④ return count
```

Algorithm Analysis

Our algorithm mergeAndOutput(A[l...r] , mid) will run for n/2 times compulsorily. That is, the while loop(present at line location 2) while run n/2 times without fail.

Hence, we need not check our algorithm's efficiency for best or worst case as it will run for fixed amount of time everytime.

We know that $C(1) = 0$ —— (i)

We can say that,

$$C_{mergeandOutput}(n) = n/2$$

as, comparison takes place only n/2 times compulsorily. (i.e. no best case or worst case)

As the complete array is split into 2 halves every time,

$$C(n) = 2\,C\left(n/2\right) + C_{mergeandoutput}(n)$$

$$C(n) = 2\,C(n/2) + n/2$$

Continued On The Next Page...

Algorithm Analysis

Let $n = 2^k$

$$c(n) = 2\,c(2^{k-1}) + 2^{k-1}$$

$$= 2[2\,c(2^{k-2}) + 2^{k-2}] + 2^{k-1}$$

$$= 2^2 \cdot c(2^{k-2}) + 2(2^{k-1})$$

we can say

$$= 2^i\,c(2^{k-i}) + i(2^{k-1})$$

from (1), if $k = i$ $\quad c(2^{k-k}) = c(1) = 0$

$$= \cancel{2^k\,c(1)} + k(2^{k-1})$$

$$c(n) = k(2^{k-1})$$

$\Rightarrow n = 2^k \qquad k = \log_2 n$

$$\boxed{c(n) = \frac{n}{2}\log_2 n}$$

Therefore, we can say

$$c(n) \in \Theta(n \log_2 n)$$

We have obtained an efficiency class of n*log(n).

Algorithm Analysis

We can justify our efficiency class by the help of Master Theorem.

```
function f(input x size n)
        if(n < k)
                solve x directly and return
        else
                divide x into a subproblems of size n/b
                call f recursively to solve each subproblem
                Combine the results of all sub-problems
```

So, according to master theorem the runtime of the above algorithm can be expressed as:

$$T(n) = aT(n/b) + f(n)$$

Where, n = size of the problem
a = number of subproblems in the recursion and a >= 1
n/b = size of each subproblem
f(n) = cost of work done outside the recursive calls like dividing into subproblems and cost of combining them to get the solution.

Advance version of master theorem that can be used to determine running time of divide and conquer algorithms if the recurrence is of the following form

$$T(n) = aT(n/b) + \theta(n^k \log^p n)$$

Where, n = size of the problem
a = number of subproblems in the recursion and a >= 1
n/b = size of each subproblem
b > 1, k >= 0 and p is a real number.

Then,

1. if $a > b^k$, then $T(n) = \theta(n^{\log_b a})$
2. if $a = b^k$, then
   (a) if $p > -1$, then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$
   (b) if $p = -1$, then $T(n) = \theta(n^{\log_b a} \log\log n)$
   (c) if $p < -1$, then $T(n) = \theta(n^{\log_b a})$
3. if $a < b^k$, then
   (a) if $p >= 0$, then $T(n) = \theta(n^k \log^p n)$
   (b) if $p < 0$, then $T(n) = \theta(n^k)$

Algorithm Analysis

Our Algorithm will follow the reccurence equation,

$$T(n) = 2T(n/2) + O(n)$$

a = 2, b = 2, k = 1, p = 0
$b^k = 2$. So, $a = b^k$ and $p > -1$ [Case 2.(a)]
then, $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$
$T(n) = \theta(\ n\log n\ )$

The efficiency classes obtained by analysis and obtained by Advanced Master Theorem are same. Our Analysis is justified.
Our Algorithm has an efficiency class of **nlogn**.

Source Code

```c
/*
Let A[i…n] be an array of n distinct real numbers.
A pair (A[i],A[j]) is said to be an index-
value inversion if A[i]=j and A[j]=i.
Design an algorithm for counting the number of index-value inversions.
*/

#include<stdlib.h>
#include<stdio.h>

int checkAndOutput(int arr[], int l, int m, int r)
{
    int i, j, count = 0;
    i = l;
    while(i<=m){
        if(arr[i]>=m+1 && arr[i]<=r){
            j = arr[i];
            if(arr[j] == i)
                count++;
        }
        i++;
    }
  return count;
}


int checkIndexValueInversionFunction(int arr[], int l, int r){
  int left = 0, right = 0, count = 0;
    if (l < r){
        int m = l+(r-l)/2;

        left = checkIndexValueInversionFunction(arr, l, m);
        right = checkIndexValueInversionFunction(arr, m+1, r);

        count = checkAndOutput(arr, l, m, r);
        count = count + left + right;
    }
    return count;
}

void printArray(int A[], int size){
    int i;
    printf("Entered Array:\n");
    for (i=0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}
```

Source Code

```
int main(){
    int *arr, i, count, size, mainOption;
    while(1){
        printf("Index Value Iversion Problem\n1. Check For New Input\n2. Exit\
nEnter Option\n");
        scanf("%d", &mainOption);
        switch(mainOption){
            case 1:{
                printf("Enter Size\n");
                scanf("%d", &size);
                arr = (int*)malloc(size*sizeof(int));
                printf("Enter Array\n");
                for(i = 0; i<size; i++){
                    scanf("%d", (arr + i));
                }
                printArray(arr, size);
                count = checkIndexValueInversionFunction(arr, 0, size - 1);
                printf("Index-Value Inversion Count: %d\n\n", count);
                break;
            }
            case 2:{
                exit(0);
            }
            default:{
                break;
            }
        }
    }
    return 0;
}
```

Output

pi@raspberrypi:~/college $ gcc -o pgm pgm.c

pi@raspberrypi:~/college $ ./pgm

Index Value Iversion Problem

1. Check For New Input

2. Exit

Enter Option

1

Enter Size

1

Enter Array

1

Entered Array:

1

Index-Value Inversion Count: 0

Index Value Iversion Problem

1. Check For New Input

2. Exit

Enter Option

1

Enter Size

2

Enter Array

1

0

Entered Array:

1 0

Index-Value Inversion Count: 1

Output

Index Value Iversion Problem

1. Check For New Input

2. Exit

Enter Option

1

Enter Size

8

Enter Array

1

4

2

7

1

6

5

3

Entered Array:

1 4 2 7 1 6 5 3

Index-Value Inversion Count: 3

Index Value Iversion Problem

1. Check For New Input

2. Exit

Enter Option

2

pi@raspberrypi:~/college $

References

https://www.geeksforgeeks.org/merge-sort/

https://www.geeksforgeeks.org/advanced-master-theorem-for-divide-and-conquer-recurrences/

Project Repository

https://github.com/aksharsramesh/ADAproject