INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

-------------------ANÁLISIS DE ALGORITMOS-----------------

ACTIVIDAD

Diseño de soluciones DyV

PROFESOR:

Franco Martínez Edgardo Adrián

ALUMNO:

Meza Vargas Brandon David – 2020630288

GRUPO:

3CM13

índice

Antes de todo, hay que aclarar que en algunos problemas se usaron tipos de datos long, long long int, ya que se requieren para valores grandes y el juez lo acepte al 100%.

Problema: Divide and Conquer 1

Redacción

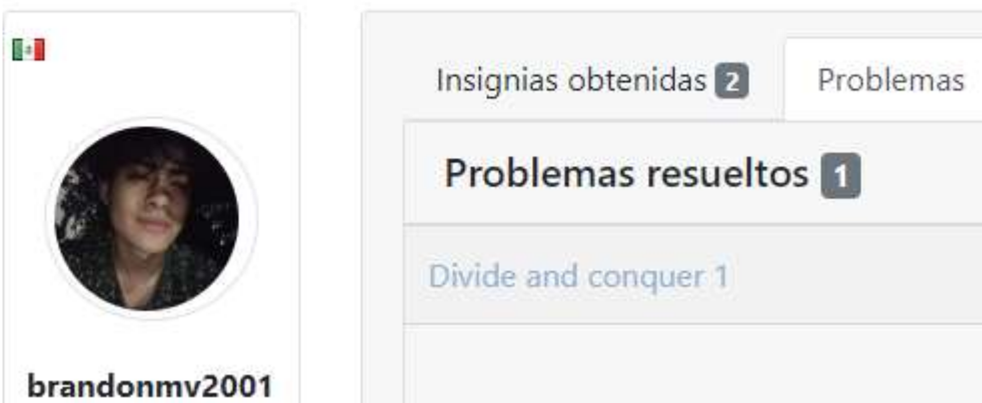
Edgardo se puso un poco intenso este semestre y puso a trabajar a sus alumnos con problemas de mayor dificultad.

La tarea es simple, dado un arreglo A de números enteros debes imprimir cual es la suma máxima en cualquier subarreglo contiguo.

Por ejemplo si el arreglo dado es {-2, -5, **6, -2, -3, 1, 5**, -6}, entonces la suma máxima en un subarreglo contiguo es **7**.

Captura de aceptación por juez

Enviado	GUID	Estatus	Porcentaje	Lenguaje	Memoria	Tiempo	Acciones
2021-11-02 16:17:08	e49ef1f8	AC	100.00%	c11-gcc	2.31 MB	0.10 s	



Explicación Algoritmo

A continuación, se muestra el algoritmo empleado para resolver este problema.

```
long long int sumaMaxima(long long int A[], long long int inicio, long long int final){
    long long int i;                                    //Variable para loops
    if(inicio == final)                                 //Si el inicio es igual al final indica que hay un elemento y lo retornamos
        return A[inicio];

    long long int mitad = (inicio+final) / 2;           //Establecemos la mitad del arreglo

    long long int izq = sumaMaxima(A,inicio,mitad);     //Almacenamos la suma maxima del sub arreglo de la izquierda con una llamada recursiva
    long long int der = sumaMaxima(A,mitad+1,final);    //Almacenamos la suma maxima del sub arreglo de la derecha con una llamada recursiva

    long long int suma = 0;
    long long int sumaIzq = A[mitad];                   //Establecemos la suma de la izquierda desde la mitad del arreglo

    //Se recorre desde la mitad hasta el inicio del arreglo para encontrar la suma
    for(i=mitad; i>=inicio;i--){
        suma += A[i];                                   //Vamos acumulando la suma de los elementos
        if(suma > sumaIzq)                              //Si la suma es mayor a la suma de la parte izquierda, esta sera la suma maxima y se almacena
            sumaIzq = suma;
    }
    suma = 0;

    //Hacemos lo anterior, pero ahora partiendo de la mitad+1 hasta el ultimo elemento del arreglo
    long long int sumaDer = A[mitad+1];
    for(i=mitad+1; i<=final;i++){
        suma += A[i];
        if(suma > sumaDer)
            sumaDer = suma;
    }

    long long int sumaCentral = sumaIzq + sumaDer;      //Posteriormente, se suman las sumas de ambos lados para encontrar la del arreglo

    return mo3(izq,der,sumaCentral);                    //Obtenemos el mayor de los 3 y esta sera la suma maxima
}
```

Primeramente, establecemos la mitad del arreglo, para de eta forma partir el arreglo en dos y empezar a buscar las sumas de sub-arreglos por las mitades, las mismas que serán partidas hasta que no se pueda más.

Posteriormente, se hace la llamada recursiva para encontrar la suma por la parte de la derecha, que va de la mitad del arreglo hasta el primer elemento, y la suma de la parte izquierda va de un elemento después de la mitad hasta el último elemento del arreglo.

Ahora bien, expliquemos lo que sucede en la siguiente parte, solo se explica una vez ya que es lo mismo para derecha e izquierda.

```
long long int suma = 0;
long long int sumaIzq = A[mitad];

//Se recorre desde la mitad hasta el
for(i=mitad; i>=inicio;i--){
    suma += A[i];
    if(suma > sumaIzq)
        sumaIzq = suma;
}
```

Primeramente se iguala al lado izquierda el elemento que se encuentra a la mitad del arreglo, en el caso de la parte derecha se asigna el valor mitad+1. Después se hace el ciclo desde la mitad al inicio del arreglo, recorriendo así la parte izquierda. Posteriormente, se van acumulando las sumas de los elementos, si esta suma es mayor al elemento posicionado en la mitad del sub arreglo se establece como la suma mayor de este lado.

Lo mismo sucede con el lado derecho cambiando los límites.

Finalmente se suman las sumas de ambas mitades, siendo así la suma máxima el mayor elemento de la suma central, la suma de la derecha o la de la izquierda.

Análisis de complejidad en cota O()

Para este problema tenemos que se parte en dos el problema cada vez que se manda a llamar a la función, esto lo hace dos veces, por la parte derecha y la izquierda, de esta manera tenemos.

$T(n) = 2T(n/2)+n$ -→ n es la complejidad de la mezcla que se hace abajo.

Haciendo uso del teorema maestro tenemos:

$f(n) = O(n^{\log_2 2}) \rightarrow n = n$

Por lo tanto la complejidad en cota O es:

O(log(n)n)

Código de solución completo

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

#define mo3(a,b,c) ( (a) > (b) ? ((a) > (c) ? (a) : (c)) : ((b) > (c) ? (b) : (c)))

/*
    Función que calcula la suma Maxima de un sub arreglo del arreglo.
    Recibe el arreglo, el inicio y el final del mismo
    Retorna la suma maxima
*/
long long int sumaMaxima(long long int A[], long long int inicio, long long int final){
    long long int i;                            //Variable para loops
    if(inicio == final)                         //Si el inicio es igual al final indica que hay un elemento y lo retornamos
        return A[inicio];

    long long int mitad = (inicio+final) / 2;          //Establecemos la mitad del arreglo

    long long int izq = sumaMaxima(A,inicio,mitad);     //Almacenamos la suma maxima del sub arreglo de la izquierda con una llamda recursiva
    long long int der = sumaMaxima(A,mitad+1,final);    //Almacenamos la suma maxima del sub arreglo de la derecha con una llamda recursiva

    long long int suma = 0;
    long long int sumaIzq = A[mitad];                  //Establecemos la suma de la izquierda desde la mitad del arreglo

    //Se recorre desde la mitad hasta el inicio del arreglo para encontrar la suma
    for(i=mitad; i>=inicio;i--){
        suma += A[i];                           //Vamos acumulando la suma de los elementos
        if(suma > sumaIzq)                      //Si la suma es mayor a la suma de la parte izquierda, esta sera la suma maxima y se almacena
            sumaIzq = suma;
    }
    suma = 0;

    //Hacemos lo anterior, pero ahora partiendo de la mitad+1 hasta el último elemento del arreglo
    long long int sumaDer = A[mitad+1];
    for(i=mitad+1; i<=final;i++){
        suma += A[i];
        if(suma > sumaDer)
            sumaDer = suma;
    }

    long long int sumaCentral = sumaIzq + sumaDer;     //Posteriormente, se suman las sumas de ambos lados para encontrar la del arreglo

    return mo3(izq,der,sumaCentral);                //Obtenemos el mayor de los 3 y esta sera la suma maxima

}

int main()
{
    long long int N;                            //Variable para longitud del arreglo
    scanf("%lld",&N);                           //Leemos los datos que tendrá el arreglo
    long long int A[N];                          //Declaramos arreglo de tamaño N
    long long int j;                            //Variable para loops
    for(j=0;j<N;j++)                            //Llenamos el arreglo de acuerdo a la longitud
            scanf("%lld", &A[j]);               //Leemos los datos que tendrá el arreglo

    printf("%lld", sumaMaxima(A,0,N-1));       //Imprimimos la suma maxima

    return 0;
}
```

Problema: INVCNT – Inversión Count

Redacción

Let A[0...n - 1] be an array of n distinct positive integers. If i < j and A[i] > A[j] then the pair (i, j) is called an inversion of A. Given n and an array A your task is to find the number of inversions of A.

Input

The first line contains t, the number of testcases followed by a blank space. Each of the t tests start with a number n (n <= 200000). Then n + 1 lines follow. In the ith line a number A[i - 1] is given (A[i - 1] <= 10^7). The (n + 1)th line is a blank space.

Output

For every test output one line giving the number of inversions of A.

Captura de aceptación por juez

ID	DATE	USER	PROBLEM	RESULT	TIME	MEM	LANG
28676345	2021-11-02 22:37:01	Brandon Meza	Inversion Count	accepted edit ideone it	0.08	5.4M	C

Explicación Algoritmo

A continuación se muestra el algoritmo empleado para este problema.

```
long long int conteoInversiones(long long int A[], long long int p, long long int r){

    //Si la parte de la izquierda es menor a la derecha
    if( p < r ){
        long long int q = (p + r) / 2;                          //Partimos a la mi
        long long int invDer = conteoInversiones(A,p,q);        //Contamos las inv
        long long int invIzq = conteoInversiones(A,q+1,r);      //Contamos las inv
        long long int invDiv = conteoDivididas(A,p,q+1,r);      //Hacemos la mezcl
        return invDer + invIzq +invDiv;                         //Devolvemos la su
    }else return 0;                                             //En caso contrar

}
```

La función anterior, se realizará el conteo de inversiones siempre y cuando el elemento inicial sea menor al final, en otro caso el arreglo es de un elemento y no hay inversiones.

Se hacen llamadas recursivas para encontrar las inversiones por el lado de la derecha y por el lado izquierdo, posteriormente se hace la mezcla de estas.

Podemos observar que es un ordenamiento por mezcla, solo que en la función merge, que es nuestra conteoDivididas es donde contabilizaremos las inversiones.

```
long long int conteoDivididas(long long int A[], long long int p, long long int q, long long int r){
    long long int C[200000];            //Establecemos un arreglo temporal del tamaño máximo que indica e
    long long int i = p;                //i será el inicio del arreglo (derecha)
    long long int j = q;                //La mitad del arreglo
    long long int k = p, numDiv = 0;    //K será la mitad del arreglo y se inicializa en 0 el numero de d

    //Se hará el while hasta que i llegue a la mitad del arreglo y j hasta el final del mismo
    while((i <= q-1) && (j <= r)){
        if(A[i] <= A[j])                //Si el elemento i  es menor al j vamos avanzando e n los indices
            C[k++] = A[i++];
        //Cuando el elemento i es mayor al j, se forman (q-i) inversiones por que los sub arreglos ya est
        //De esta forma los elementos sobrantes en el lado izquierdo seran grandes que los de j.
        else{
            C[k++] = A[j++];
            numDiv += q-i;
        }
    }

    //Copiamos los elementos que sobran de la primera mitad en el arreglo auxiliar
    while(i <= q-1)
        C[k++] = A[i++];
    //Copiamos los elementos que sobran de la segunda mitad en el arreglo auxiliar
    while(j <= r)
        C[k++] = A[j++];

    //Ahora pasamos a copiar los elementos mezclados en el arreglo original
    for(i=p; i<=r;i++)
        A[i] =C[i];


    return numDiv;                      //Retornamos el numero de inversiones
}
```

Básicamente, en esta función se cuentan las inversiones al momento de que dos mitades del arreglo son mezcladas, se hace creando los índices del inicio que marcan los límites que tendrán al recorrerse. El índice i empieza desde la derecha y el j desde la mitad representando la izquierda.

Al momento en que el elemento i del arreglo es mayor al elemento j significa que las partes izquierdas y derechas del sub arreglo ya están ordenadas, estas serán q-i en total.

Finalmente se hace la mezcla de los elementos en el arreglo original.

Análisis de complejidad en cota O()

Para este algoritmo se parte en dos el arreglo y así hasta el caso base, se hace dos veces, una para el lado izquierdo y otra para el derecho, de esta forma tenemos:

$T(n) = 2T(n/2) + n$ -→ donde n es el costo de la función que hace la mezcla de las inversiones

Haciendo uso del teorema maestro tenemos:

$f(n) = O(n^{\log_2 2})$ ➔ $n = n$

Por lo tanto la complejidad en cota O es:

O(log(n)n)

Código de solución completo

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


/*
    Funcion que junta los dos sub arreglos, prácticamente es un merge sort
    Recibe el arreglo, la parte de la izquierda(p), la mitad(q) y la derecha(r)
    Recibe el conteo completo del array
*/
long long int conteoDivididas(long long int A[], long long int p, long long int q, long long int r){
    long long int C[200000];            //Establecemos un arreglo temporal del tamaño máximo que indica el juez
    long long int i = p;                //i será el inicio del arreglo (derecha)
    long long int j = q;                //La mitad del arreglo
    long long int k = p, numDiv = 0;    //K será la mitad del arreglo y se inicializa en 0 el numero de divididas

    //Se hará el while hasta que i llegue a la mitad del arreglo y j hasta el final del mismo
    while((i <= q-1) && (j <= r)){
        if(A[i] <= A[j])                //Si el elemento i  es menor al j vamos avanzando e n los indices de cada arreglo
            C[k++] = A[i++];
        //Cuando el elemento i es mayor al j, se forman (q-i) inversiones por que los sub arreglos ya estan ordenados
        //De esta forma los elementos sobrantes en el lado izquierdo seran grandes que los de j.
        else{
            C[k++] = A[j++];
            numDiv += q-i;
        }
    }

    //Copiamos los elementos que sobran de la primera mitad en el arreglo auxiliar
    while(i <= q-1)
        C[k++] = A[i++];
    //Copiamos los elementos que sobran de la segunda mitad en el arreglo auxiliar
    while(j <= r)
        C[k++] = A[j++];

    //Ahora pasamos a copiar los elementos mezclados en el arreglo original
    for(i=p; i<=r;i++)
        A[i] =C[i];


    return numDiv;                      //Retornamos el numero de inversiones
}
```

```c
/*
    Función que realiza el conteo de inversiones.
    Recibe el arreglo, el inicio y el final del mismo
    Retorna el conteto total de inversiones.
*/
long long int conteoInversiones(long long int A[], long long int p, long long int r){

    //Si la parte de la izquierda es menor a la derecha
    if( p < r ){
        long long int q = (p + r) / 2;                          //Partimos a la mitad el arreglo
        long long int invDer = conteoInversiones(A,p,q);        //Contamos las inversiones por la derecha
        long long int invIzq = conteoInversiones(A,q+1,r);      //Contamos las inversiones por la izquierda
        long long int invDiv = conteoDivididas(A,p,q+1,r);      //Hacemos la mezcla de la suma de los sub arreglos derecha e izquierda
        return invDer + invIzq +invDiv;                         //Devolvemos la suma de todas
    }else return 0;                                              //En caso contrario devolvemos 0

}

int main()
{
    long long int t;                    //Número de tests
    scanf("%lld",&t);                   //Leemos el número de tests
    long long int i,j;                  //Variable para loop
    long long int n;                    //Longitud del arreglo
    long long int cont = 0;
    while(t--){                         //Mientras haya número de tests vamos leyendo arreglos
        n = 0;

        scanf("%lld", &n);              //Leemos la longitud

        long long int A[n];             //Declaramos un arreglo del tamaño de la longitud anterior
        for(j=0;j<n;j++){               //Llenamos el arreglo de acuerdo a la longitud
            scanf("%lld", &A[j]);       //Leemos los datos que tendrá el arreglo
        }


        printf("%lld\n",conteoInversiones(A,0,n-1));

    }


    /* code */
    return 0;
}
```

Problema: Cumulo

Redacción

Descripción

Te encuentras con un mapa del cúmulo de estrellas R136. En el mapa, cada estrella aparece como un punto ubicado en un plano cartesiano. Te asalta de pronto una pregunta, ¿cuál será la distancia mínima entre dos estrellas en el mapa?

Entrada

La primera línea tendrá un entero 2<=n<=50000 que indica la cantidad de estrellas en el mapa. Las siguientes n líneas tendrán las coordenadas de las estrellas, dadas por dos reales X y Y. En todos los casos, 0<=X,Y<=40000.

Salida

La distancia mínima entre dos estrellas, expresada con un número real con tres cifras después del punto decimal. (La distancia se calcula como la raíz cuadrada de la suma de los cuadrados de las diferencias en X y Y)

Captura de aceptación por juez

Enviado	GUID	Estatus	Porcentaje	Lenguaje	Memoria	Tiempo	Acciones
2021-11-09 13:58:29	ee219773	TLE	24.00%	c11-gcc	2.26 MB	>14.55 s	

Explicación Algoritmo

A continuación se presenta el algoritmo empleado para esta solución

```
float distanciaEstrellas(float X[], float Y[], long int inicio, long int final){
    if(inicio == final)                    //Si el inicio es igual al final indica que hay un elemento y lo retornamos
        return 0;

    long int i,j, mitad = (inicio + final) / 2;          //  Calculamos la mitad para buscar
    float izq = distanciaEstrellas(X,Y,inicio,mitad);    //  Llamada recursiva para la parte izquierda
    float der = distanciaEstrellas(X,Y,mitad+1,final);   //  Llamada recursiva para la parte derecha

    float distancia = 0, min = 0;                        //  Inicializamos las distancias


    /*
        Vamos recorriendo los elementos de la mitad izquierda y la mitad derecha
        para ir calculando sus distancias
    */
    for(i=inicio; i<=mitad;i++){
        for(j=final; j>=mitad+1;j--){
            distancia = calcularDistancia(X[i],Y[i],X[j],Y[j]);      //  Calculamos distancias
            //  Si la distancia calculada es menor a la minima o la minima es 0, la nueva distancia será la calculada
            if((distancia <= min) || (min == 0))
                min = distancia;
        }
    }
    return minimaDist(izq,der,min);                      //  Regresamos la distancia minima
}
```

Podemos ver que al inicio hacemos la partición del problema en dos, haciendo la parte derecha y la parte izquierda para calcular las respectivas distancias en esos lados a partir de una llamada recursiva asignando los rangos correspondientes.

Posteriormente se hace el calculo de las distancias entre los puntos que hemos ingresado en los arreglos recorriendo los puntos desde el inicio a la mitad y los puntos con los que se calculara la distancia desde el final hasta la mitad+1. Una vez calculada la distancia se pregunta si la distancia resultante es mejor a la mínima (inicialmente 0) o si la mínima sigue siendo 0, si se cumple cualquier condición la nueva distancia mínima será la calculada previamente.

Posteriormente se hace la comparación entre las 3 distancias obtenidas, la del lado derecho, izquierdo y la mínima obtenida que sería la menor entre todos los puntos.

Análisis de complejidad en cota O()

El algoritmo anterior primero se divide en dos partiendo el arreglo a la mitad recursivamente hasta el caso base, de esta forma tenemos:

$T(n) = 2T(n/2) + n$ -→ donde n es el costo de recorrer el arreglo para encontrar la distancia mínima

Haciendo uso del teorema maestro tenemos:

$f(n) = O(n^{\log_2 2})$ → $n = n$

Por lo tanto la complejidad en cota O es:

O(nlog(n))

Código de solución completo

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define MIN(a,b) ((a) < (b) ? (a) : (b))     //  Para encontrar el minimo de dos numeros


float minimaDist(float izq, float der, float min){

    if(der == 0 && izq == 0)                //  Si las distancias derecha e izquierda son 0, retornamos la minima calculada
        return min;
    else if(der != 0)                       //  Si la derecha no es 0, entonces retornamos la menor distancia entre la derecha y la minima calculada
            return MIN(der,min);
        else if(izq != 0)                   //  Si la izquierda no es 0, entonces retornamos la menor distancia entre la izquierda y la minima calculada
                return MIN(izq,min);
    else
        return MIN(MIN(der,izq), min);      //  Si la derecha e izquierda son distancias validas se retorna el menor de los 3
}

float calcularDistancia(float x1,float y1,float x2,float y2){ return sqrt(pow(x1 - x2,2) + pow(y1 - y2,2)); }

float distanciaEstrellas(float X[], float Y[], long int inicio, long int final){
    if(inicio == final)                             //Si el inicio es igual al final indica que hay un elemento y lo retornamos
        return 0;

    long int i,j, mitad = (inicio + final) / 2;                 //  Calculamos la mitad para buscar
    float izq = distanciaEstrellas(X,Y,inicio,mitad);           //  Llamada recursiva para la parte izquierda
    float der = distanciaEstrellas(X,Y,mitad+1,final);          //  Llamada recursiva para la parte derecha

    float distancia = 0, min = 0;                               //  Inicializamos las distancias

    /*
        Vamos recorriendo los elementos de la mitad izquierda y la mitad derecha
        para ir calculando sus distancias
    */
    for(i=inicio; i<=mitad;i++){
        for(j=final; j>=mitad+1;j--){
            distancia = calcularDistancia(X[i],Y[i],X[j],Y[j]);      //  Calculamos distancias
            //  Si la distancia calculada es menor a la minima o la minima es 0, la nueva distancia será la calculada
            if((distancia <= min) || (min == 0))
                min = distancia;
        }
    }
    return minimaDist(izq,der,min);                                  //  Regresamos la distancia minima
}

int main()
{
    long int n;                         //  Cantidad de estrellas
    scanf("%ld", &n);                   //  Leemos la cantidad de estrellas
    if(n < 2) return 0;                 //  Si la cantidad de estrellas es menor a 2 salimos
    float X[n],  Y[n];                  //  Arreglos que guardarán las coordenadas de cada estrella
    long int i;                         //  Variable para loops;

    //  Leemos los puntos
    for(i=0;i<n;i++){
        scanf("%f",&X[i]);
        scanf("%f",&Y[i]);
    }

    printf("%f",distanciaEstrellas(X,Y,0,n-1));  //Imprimimimos la distancia minima de dos estrellas
    return 0;
}
```

Problema: TRIPINV – Mega Inversions

Redacción

The n^2 upper bound for any sorting algorithm is easy to obtain: just take two elements that are misplaced with respect to each other and swap them. Conrad conceived an algorithm that proceeds by taking not two, but three misplaced elements. That is, take three elements ai > aj > ak with i < j < k and place them in order ak; aj ; ai. Now if for the original algorithm the steps are bounded by the maximum number of inversions n(n-1)/2, Conrad is at his wits' end as to the upper bound for such triples in a given sequence. He asks you to write a program that counts the number of such triples.

Input

The first line of the input is the length of the sequence, 1 <= n <= 10^5.
The next line contains the integer sequence a1; a2..an.
You can assume that all ai belongs [1; n].

Output

Output the number of inverted triples.

Captura de aceptación por juez

28711572	2021-11-10 01:21:05	Brandon Meza	Mega Inversions	accepted edit ideone it	0.13	6.7M	CPP14

Explicación Algoritmo

En el algoritmo se hace uso de un árbol de Fenwick, una estructura de datos que nos permite hacer sumas y actualizaciones de manera muy eficiente, pues cada operación vale O(lonN), pues usa operaciones con bits.

```
long long int obtenerSuma(long long int *BITree, int indice){
    long long int s = 0;                //  La suma en 0

    indice = indice + 1;                //  El indice del arbol es uno más que el del arreglo


    while (indice>0){

        s += BITree[indice];              //  Vamos sumando los elementos del arbol

        indice -= indice & (-indice);   //  Nos movemos la nodo padre
    }
return s;                           //Retornamos la suma

}

void actualizar(long long int *BITree, int n, int indice, long long int valor){
    indice = indice + 1;                //  El indice del arbol es uno más que el del arreglo

    while (indice <= n){

        BITree[indice] += valor;           //  Agregamos el valor al nodo actual
        indice += indice & (-indice);    //  Actualizamos el indice al del nodo padre
    }

}
```

Primeramente, en la función de obtener suma se encarga de sumar los elementos del árbol de acuerdo al índice en el que estemos, posteriormente se va moviendo entre nodos para seguir con la suma hasta el índice indicado.

La función de actualizar toma un elemento del árbol y se va actualizado de acuerdo a un valor establecido.

Análisis de complejidad en cota O()

Las operaciones del árbol son O(log(n)), pero al estar recorriendo todo el arreglo de tamaño n, tenemos una complejidad final de:

O(nlog(n))

Código de solución completo

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;

//  Variables globales
const int longitud = 100000;        //  Longitud maxima


long long int BITree[3][longitud];  //  Inicializamos el arbol



long long int obtenerSuma(long long int *BITree, int indice){
    long long int s = 0;                //  La suma en 0

    indice = indice + 1;                //  El indice del arbol es uno mas que el del arreglo


    while (indice>0){

        s += BITree[indice];              //  Vamos sumando los elementos del arbol

        indice -= indice & (-indice);   //  Nos movemos la nodo padre
    }
return s;                           //Retornamos la suma

}

void actualizar(long long int *BITree, int n, int indice, long long int valor){
    indice = indice + 1;                //  El indice del arbol es uno mas que el del arreglo

    while (indice <= n){

        BITree[indice] += valor;          //  Agregamos el valor al nodo actual
        indice += indice & (-indice);    //  Actualizamos el indice al del nodo padre
    }

}
```

```
int main()
{
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    long long int inv;                          //  Cantidad de inversiones
    int n,x;                                    //  Longitud del arreglo
    cin >> n;                                   //  Leemos la longitud del arreglo

    long long int a[longitud];                        //  Arreglo de longitud n
    //  Llenamos el arreglo con datos
    for(int i=1;i<=n;i++){
        cin >> a[i];
    }

    //  Con reverse revertimos el array ingresado
    reverse(a+1,a+n+1);


    //  Con el siguiente for vamos almacenando las sumas delas inversiones triples
    for(int i=1;i<=n;i++){
        x = a[i];
        actualizar(BITree[2],n,x,inv += obtenerSuma(BITree[1],x-1));
        actualizar(BITree[1],n,x, obtenerSuma(BITree[0],x-1));
        actualizar(BITree[0],n,x,1);
    }
    cout << inv << endl;

    return 0;
}
```