

Instituto Politécnico Nacional
Escuela Superior de Cómputo

Problemas de divide y vencerás



Luis Fernando Resendiz Chavez

Análisis de algoritmos

15-01-2020

Problema 01: Divide and Conquer 1

Descripción del problema

Descripción

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.

Entrada

La primera línea contendrá un número N .

En la siguiente línea N enteros representando el arreglo A .

Salida

La suma máxima en cualquier subarreglo contiguo.

Explicación de la solución del problema

Este problema ya lo había resuelto anteriormente utilizando programación Dinámica, específicamente el algoritmo de Kadane, el cual encuentra la longitud del subarreglo con la máxima suma en tiempo lineal, sin embargo, esta vez utilice el paradigma de divide y vencerás:

Primero Tenemos que separar el arreglo dado en mitades de forma recursiva, de la misma manera como si hiciéramos un merge sort, hasta que lleguemos al caso base, el cual es cuando el arreglo sea de tamaño 1, en este punto podemos decir que la máxima suma actual que hemos encontrado es un único elemento del arreglo, sin embargo, aun podemos evaluar los subarreglos de tamaños mayores a 1, es por ello que después de separar el arreglo original, obtenemos la máxima suma del lado izquierdo, la máxima suma del lado derecho y la máxima suma contando la parte del centro, comparamos las tres y retornamos la mayor, como este proceso se ejecuta de forma recursiva,

termina comparando el arreglo completo, una vez que termina retorna la mayor suma encontrada en algún subarreglo del arreglo original.

Análisis de complejidad temporal y espacial

Temporal: $O(n \log n)$

Espacial: $O(n)$

Separar el arreglo original a la mitad de forma recursiva, por su naturaleza lo hace en $O(\log n)$ logaritmo base 2 de n , pero cada vez que se divide se tiene que calcular la suma máxima cuando la recursión regresa, hay que revisar la mitad izquierda, la mitad derecha y la combinación de ambas, el algoritmo lo hace en tiempo $O(n)$, por lo tanto la complejidad temporal final es $O(n \log n)$

En cuanto a la complejidad espacial únicamente utilizamos los n espacios del arreglo dado.

Código de la solución del problema

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

#define int long long int

class Solution {
public:
    void solve() {
        int n;
        cin >> n;
        vector<int> arr(n);
        for(int& x: arr)
            cin >> x;
        cout << LargestSumSubarray(arr, 0, n-1);
    }
};
```

```
}
```

```
int MaxSum(vector<int>& arr, int izq, int der, int mid) {
```

```
    int sum = 0;
```

```
    int izqSum = INT_MIN;
```

```
    // valores de izq a mid
```

```
    for(int i = mid; i >= izq; --i) {
```

```
        sum += arr[i];
```

```
        izqSum = max(izqSum, sum);
```

```
    }
```

```
    // valores de mid a der
```

```
    sum = 0;
```

```
    int derSum = INT_MIN;
```

```
    for(int i = mid+1; i <= der; ++i) {
```

```
        sum += arr[i];
```

```
        derSum = max(derSum, sum);
```

```
    }
```

```
    return max({ izqSum, derSum, izqSum + derSum });
```

```
}
```

```
int LargestSumSubarray(vector<int>& arr, int izq, int der) {
```

```
    if(izq == der) {
```

```
        return arr[izq];
```

```
    }
```

```
    int mid = (izq + der) / 2;
```

```
    return max({
```

```


        LargestSumSubarray(arr, izq, mid),
        LargestSumSubarray(arr, mid+1, der),
        MaxSum(arr, izq, der, mid)
    });
}
};

signed main() {
    (new Solution())->solve();
}

```

Datos de la solución subida

Juez	OmegaUp
Usuario	Lfr0
Status	AC

Enviado	GUID	Estatus	Porcentaje	Lenguaje	Memoria	Tiempo	Acciones
2021-01-13 15:48:56	ad90bd9a	AC	100.00%	cpp17-gcc	3.89 MB	0.31 s	

Problema 02: Amigos y Regalos

Descripción del problema

Descripción

Tienes dos amigos. A ambos quieres regalarles varios números enteros como obsequio. A tu primer amigo quieres regalarle C_1 enteros y a tu segundo amigo quieres regalarle C_2 enteros. No satisfecho con eso, también quieres que todos los regalos sean únicos, lo cual implica que no podrás regalar el mismo entero a ambos de tus amigos.

Además de eso, a tu primer amigo no le gustan los enteros que son divisibles por el número primo X . A tu segundo amigo no le gustan los enteros que son divisibles por el número primo Y . Por supuesto, tu no le regalaras a tus amigos números que no les gusten.

Tu objetivo es encontrar el mínimo número V , de tal modo que puedas dar los regalos a tus amigos utilizando únicamente enteros del conjunto $1, 2, 3, \dots, V$. Por supuesto, tú podrías decidir no regalar algunos enteros de ese conjunto.

Un número entero positivo mayor a 1 es llamado primo si no tiene divisores enteros positivos además del 1 y el mismo.

Entrada

Una línea que contiene cuatro enteros positivos C_1, C_2, X, Y . Se garantiza que X y Y son números primos.

Salida

Una línea. Un entero que representa la respuesta al problema.

Explicación de la solución del problema

Para resolver este problema intente hacerlo de forma lineal, desde 1 hasta donde fuera posible para encontrar todos los números que necesito regalar, sin embargo, me dio TLE y llegue a la siguiente solución:

Suponiendo que los primeros c_1 y c_2 números que necesito regalar son válidos, necesito mínimo $c_1 + c_2$ números, entonces inicie una búsqueda binaria desde ese número mínimo hasta el máximo posible que se puede usar en un long long, entonces me posiciono en el número de en medio y hago el calculo necesario para saber si el numero en el que estoy es suficiente, dependiendo de si lo es o no me vuelvo a mover a la mitad del lado izquierdo o a la mitad de lado derecho, el proceso lo hago hasta que encuentre el mínimo posible y lo regreso.

Análisis de complejidad temporal y espacial

El costo temporal de este algoritmo es el numero de veces que se hace la búsqueda binaria en el rango $[f1+f2, \text{MAX_LONG}]$ multiplicado por el costo computacional para calcular si el numero actual es valido o no. Entonces como la búsqueda binaria cada vez que nos movemos a la izquierda o a la derecha eliminamos la mitad del rango inicial. Entonces la búsqueda binaria tiene una complejidad de $O(\log n)$ siendo n el rango $[f1+f2, \text{MAX_LONG}]$ y el cálculo tiene una complejidad de $O(1)$, por lo tanto la complejidad final del algoritmo total es $O(\log n) * O(1) = O(\log n)$

Código de la solución del problema

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

typedef long long int lint;

class Solution {
public:
    lint f1, f2, x, y;

    void solve() {
        cin >> f1 >> f2 >> x >> y;
        cout << binarySearch(f1+f2, LLONG_MAX);
    }

    lint binarySearch(lint start, lint end) {
        lint mid = start + (end - start) / 2;
        if(isValid(mid))
            return (not isValid(mid - 1)) ?
                mid : binarySearch(start, mid - 1);
    }
};
```

```

else
    return binarySearch(mid + 1, end);
}

bool isValid(int n) {
    int nv = n / (x * y);
    int r1 = (n / x) - nv;
    int r2 = (n / y) - nv;

    int m1 = max(f1 - r2, 0LL);
    int m2 = max(f2 - r1, 0LL);

    return (n - r1 - r2 - nv >= m1 + m2);
}

};

signed main() {
    (new Solution())->solve();
}

```

Datos de la solución subida

Juez	OmegaUp
Usuario	Lfr0
Status	AC

Enviado	GUID	Estatus	Porcentaje	Lenguaje	Memoria	Tiempo	Acciones
2021-01-13 17:18:57	b1329513	AC	100.00%	cpp17-gcc	3.40 MB	0.02 s	

Problema 03: Cumulo

Descripción del problema

Descripción

Te encuentras con un mapa del cúmulo de estrellas R136. En el mapa, cada estrella aparece como un punto ubicada 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 $2 \leq n \leq 50000$ 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, $0 \leq X, Y \leq 40000$.

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)

Explicación de la solución del problema

Este problema es un poco más sencillo porque es básicamente el algoritmo de Closest Pair of Points que vimos ya en clase, solo que vez de puntos son estrellas, ahora, para el algoritmo utilice una estructura Point para almacenar las posiciones de las estrellas, hice una función adicional que me devuelve la distancia entre dos puntos, como en la materia de cálculo, y por último una función que calcula la solución utilizando fuerza bruta, la cual use para pocos pares de puntos.

Ordeno los puntos dados respecto al eje x y comienzo a buscar la distancia más corta mediante la función minDistPoints, si me dan menos de 80 puntos o estrellas use la fuerza bruta, si no comienzo a separar el arreglo de puntos en mitades como en el merge sort, lo hago de forma recursiva hasta llegar al caso base, que es la unidad, con la unidad ya podemos encontrar la solución trivial que es 0, después en una variable vamos guardando la mínima distancia encontrada, si encontramos una distancia menor, lo añadimos a un arreglo de nuevos

puntos para buscar en ellos la menor distancia pero ahora en el eje de las y, para ello se utiliza la función LineClosest, primero ordena los puntos de menor a mayor respecto al eje de las y, busca entre ellos la menor distancia y la regresa.

Análisis de complejidad temporal y espacial

La complejidad de la fuerza bruta es $O(n^2)$ ya que cada punto es comparado con todos los demás.

La complejidad del otro algoritmo es $O(n)$ porque tenemos que comparar al menos una vez todas las distancias entre puntos, y dependiendo de el numero de puntos cercanos para evaluar en el eje y es $O(n)$, mas $O(n \log n)$ por el ordenamiento de los puntos, finalmente la complejidad total es $O(\log n) + O(n) + O(n) = O(\log n)$

Código de la solución del problema

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

struct Point {
    double x, y;
};

double Distance(Point &P1, Point &P2) {
    return (P1.x - P2.x) * (P1.x - P2.x) + (P1.y - P2.y)*(P1.y - P2.y);
}

double BruteForce(Point Points[], size_t Size) {

    double MinimunDistance = numeric_limits<double>::max();

    for (auto i = 0; i < Size; ++i) {
```

```

    for (auto j = i + 1; j < Size; ++j) {
        double CurrentDistance = Distance(Points[i], Points[j]);
        if (CurrentDistance < MinimunDistance)
            MinimunDistance = CurrentDistance;
    }
}

return MinimunDistance;
}

double LineClosest(Point Strip[], size_t Size, double d) {
    double MinimunDistance = d;

    sort(Strip, Strip + Size,
        [](const Point& P1, const Point& P2) {
            return P1.y < P2.y;
        }
    );

    for (int i = 0; i < Size; ++i) {
        for (int j = i + 1; j < Size; ++j) {

            if ((Strip[j].y - Strip[i].y) >= MinimunDistance) break;

            double CurrentDistance = Distance(Strip[i], Strip[j]);
            if (CurrentDistance < MinimunDistance)
                MinimunDistance = CurrentDistance;
        }
    }

    return MinimunDistance;
}

```

```

}

double minDistPoints(Point Points[], size_t n) {

    if (n < 80) return BruteForce(Points, n);

    size_t Middle = n / 2;

    double MinDistanceSide = min(
        minDistPoints(Points, Middle),
        minDistPoints(Points + Middle, n - Middle)
    );

    Point Strip[n];
    int SizeOfStrip = 0;

    for (int i = 0; i < n; i++) {
        if (abs(Points[i].x - Points[Middle].x) < MinDistanceSide)
            Strip[SizeOfStrip++] = Points[i];
    }

    return min(
        MinDistanceSide,
        LineClosest(Strip, SizeOfStrip, MinDistanceSide)
    );
}

double ClosestPairOfPoints(Point Points[], int n) {

```

```

sort(Points, Points + n,
    [](const Point& P1, const Point& P2) {
        return P1.x < P2.x;
    }
);

return minDistPoints(Points, n);
}

int main() {

    int n; scanf("%d", &n);
    Point Points[n];

    for (int i = 0; i < n; ++i) {
        scanf("%lf %lf", &Points[i].x, &Points[i].y);
    }


    printf("%.3lf\n", sqrt(ClosestPairOfPoints(Points, n)) );

    return 0;
}

```

Datos de la solución subida

Juez	OmegaUp
Usuario	Lfr0
Status	AC

Enviado	GUID	Estatus	Porcentaje	Lenguaje	Memoria	Tiempo	Acciones
2021-01-13 19:32:36	bab0b35d	AC	100.00%	cpp17-gcc	3.54 MB	0.78 s	

Problema 07: Inversión Count

Descripción del problema

INVCNT - Inversion Count

[#graph-theory](#) [#number-theory](#) [#shortest-path](#) [#sorting](#) [#bitmasks](#)

Let $A[0 \dots n - 1]$ 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 ($n \leq 200000$). Then $n + 1$ lines follow. In the i th line a number $A[i - 1]$ is given ($A[i - 1] \leq 10^7$). The $(n + 1)$ th line is a blank space.

Output

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

Explicación de la solución del problema

Análisis de complejidad temporal y espacial

Código de la solución del problema

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

```

typedef long long int ll;
#define vll vector<ll>

ll n, t, countInv;

void mergeInversionsCount(vll& l1, vll& l2, vll& arr, ll n1, ll n2, ll n) {
    ll p = 0, q = 0, r = 0, counter = 0;
    while(p < n1 && q < n2) {
        if(l1[p] <= l2[q]) {
            countInv += counter;
            arr[r++] = l1[p++];
        }

        else {
            counter++;
            arr[r++] = l2[q++];
        }
    }

    while(p < n1) {
        countInv += counter;
        arr[r++] = l1[p++];
    }

    while(q < n2) {
        arr[r++] = l2[q++];
    }
}

void mergeSortMod(vll& arr, ll n) {
    if(n > 1) {

```

```

    ll mid = n/2;
    ll k = 0;
    vll l1(mid), l2(n-mid);
    for(ll i = 0; i < mid; ++i)
        l1[i] = arr[k++];
    for(ll i = 0; i < n-mid; ++i)
        l2[i] = arr[k++];
    mergeSortMod(l1, mid);
    mergeSortMod(l2, n-mid);
    mergeInversionsCount(l1, l2, arr, mid, n-mid, n);

}
}

int main() {
    cin >> t;
    while(t--) {
        char ignore[10];
        cin.getline(ignore, 10);

        cin >> n;
        countInv = 0LL;
        vll arr(n);
        for(ll& val: arr)
            cin >> val;
        mergeSortMod(arr, n);
        cout << countInv << endl;
    }
}

```

Datos de la solución subida

Juez	Spoj
Usuario	Lfrc0
Status	AC

ID	DATE	USER	PROBLEM	RESULT	TIME	MEM	LANG
27279487	2021-01-15 16:19:38	lfrc0	Inversion Count	accepted edit ideone.it	0.16	7.8M	CPP14

Problema 08: TRIPINV - Mega Inversions

Descripción del problema

TRIPINV - Mega Inversions

no tags

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 $a_i > a_j > a_k$ with $i < j < k$ and place them in order $a_k; a_j; a_i$. 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, $1 \leq n \leq 10^5$.
The next line contains the integer sequence $a_1; a_2 \dots a_n$.
You can assume that all a_i belongs $[1; n]$.

Output

Output the number of inverted triples.

Explicación de la solución del problema

Para resolver este problema utilice una estructura de datos conocida, la cual si funcionamiento esta basado en la idea del paradigma divide y venceras, esta estructura se llama Binary Indexed Tree or Fenwick Tree, el cual permite calcular sumas y consultarlas en tiempo logaritmico, por lo tanto para este problema de inversiones utilizaremos uno en donde vamos a almacenar las sumas acumuladas de los prefijos del arreglo dado, para así calcular el número de tríos de inversiones que nos convengan o que sean favorables.

Análisis de complejidad temporal y espacial

Como las operaciones del Fenwick Tree tiene complejidad $O(\log n)$ y el arreglo es de tamaño $O(n)$, la complejidad final es $O(n \log n)$

Código de la solución del problema

```
// https://www.spoj.com/problems/TRIPINV/  
// Solution with fenwick Tree (Divide and Conquer aproach)  
// Ifrc0  
  
#include <bits/stdc++.h>  
using namespace std;  
  
const int mx = 100028;  
typedef long long int ll;  
  
int n, x;  
ll ans;  
ll a[mx];  
ll ft[3][mx];  
  
void add(ll *ft, int pos, ll value){  
    for( ; pos < mx; pos += (pos & -pos))  
        ft[pos] += value;
```

```

}

ll get(ll *ft, int pos){
    ll s = 0;
    for( ; pos > 0; pos -= (pos & -pos))
        s += ft[pos];
    return s;
}

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);

    cin >> n;
    for(int i = 1; i <= n; i++)
        cin >> a[i];
    reverse(a + 1, a + n + 1);
    for(int i = 1; i <= n; i++){
        x = a[i];
        add(ft[2], x, ans += get(ft[1], x - 1));
        add(ft[1], x, get(ft[0], x - 1));
        add(ft[0], x, 1);
    }
    cout << ans << endl;

    return 0;
}

```

Datos de la solución subida

Juez	Spoj
Usuario	Lfrc0
Status	AC

ID	DATE	USER	PROBLEM	RESULT	TIME	MEM	LANG
27280104	2024-01-15 18:17:27	lfrc0	Mega Inversions	accepted edit ideone it	0.08	6.6M	CPP14