Instituto Politécnico Nacional

Escuela Superior de Cómputo

DISEÑO DE SOLUCIONES DYV

Ejercicio 05

Análisis de Algoritmos

M. en C. Edgardo Adrián Franco Martínez

Grupo: 3CM3

Fecha: 16 / Mayo / 2018

Alumno:

Calva Hernández José Manuel 2017630201

Índice

Divide and Conquer 1

Redacción del ejercicio

Divide and conquer 1

Puntos		Límite de memoria	32MB
Límite de tiempo (caso)	1s	Límite de tiempo (total)	60s

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 numero N.

En la siguiente línea N enteros representando el arreglo A.

Salida

La suma máxima en cualquier subarreglo contiguo.

Ejemplo

Entrada	Salida
8 -2 -5 6 -2 -3 1 5 -6	7
5 1 2 3 4 5	15

Límites

- $1 \leq N \leq 10^5$
- $|A_i| \leq 10^9$

Captura de aceptación

Submitted	GUID	Status	Percentage	Language	Memory	Runtime	Details
2018-05-15 17:05:26	9f514660	Accepted	100.00%	cpp11	3.21 MB	0.08 s	

Explicación de la solución

Se van a leer los enteros conforme vayan llegando, para ello mantendremos dos variables auxiliares que nos permitirán llevar control de las soluciones, la primera de ellas, llamada aquí “partialSum” mantendrá la suma parcial que llevemos hasta el momento y podrá ser modifica con base a dos situaciones: una, que el siguiente número sea mayor que la suma de la actual y ese número; o dos, sumándole el siguiente número. La siguiente variable usada llevará el control de la máxima suma subsecuente encontrada hasta el momento, que aquí la llamé “maxSum”, y sólo podrá ser modificada en un caso, que es cuando la variable “partialSum” sea mayor que esta, en ese caso, le asignaremos el valor.

Estas dos variables usadas las inicializaremos con base al primer número recibido, y se asume momentáneamente que este es la máxima suma hasta ahora. Al finalizar, la variable maxSum contendrá la respuesta.

Código

```
#include < bits / stdc++.h >
using namespace std;
typedef long long int lli;
int main( ) {
    int n;
    scanf("%d", & n);
    lli partialSum, aux;
    scanf("%lld", & partialSum);
    lli maxSum = partialSum;
    for (int i = 1; i < n; ++i) {
        scanf("%lld", & aux);
        if (aux > partialSum + aux) {
            partialSum = aux;
        } else {
            partialSum += aux;
        }
        if (partialSum > maxSum) {
            maxSum = partialSum;
        }
    }
    printf("%lld\n", maxSum);
    return 0;
}
```

Análisis de complejidad

Es posible notar que la mayoría de las operaciones realizadas se consideran constantes, ya que hacer lectura de datos, impresiones, asignaciones y comparaciones no influye en gran medida en el desempeño del algoritmo. Por ello, nos iremos al ciclo for, que es el único que crecerá con el tamaño del problema al depender de la variable n, y dado que realiza la iteración n-1 veces, podemos asumir que su complejidad es la siguiente:

$$f_t(n) = n - 1 = O(n)$$

Amigos y Regalos

Redacción del ejercicio

Amigos y Regalos

Points	24.47	Memory limit	32MB
Time limit (case)	1s	Time limit (total)	60s

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, ..., 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.

Ejemplo

Entrada	Salida	Descripción
3 1 2 3	5	Teniendo el conjunto de números: {1, 2, 3, 4, 5}, podemos regalarle los enteros {1, 3, 5} al amigo 1, y los enteros {2, 4} al amigo 2. Éste es el conjunto más pequeño que cumple con la solución.

Captura de aceptación

Submitted	GUID	Status	Percentage	Language	Memory	Runtime	Details
2018-05-16 11:08:53	51db2c41	Accepted	100.00%	cpp11	3.16 MB	0.00 s	

Explicación de la solución

El algoritmo de solución consiste en hacer una búsqueda binaria sobre todos los posibles valores positivos hasta encontrar el mínimo elemento que cumpla las condiciones que nos piden.

Al ser una función recursiva, primero debemos de encontrar la condición base para poder retornar el valor, éste será que nuestro valor sea el mínimo posible que cumpla las condiciones, y para comprobar ello, aprovecharemos la condición de que la función es creciente, así, si nuestro valor es válido, los sucesivos lo serán, sin embargo, si no es válido, ninguno de los anteriores lo será. Por lo tanto, nuestro mínimo elemento será el que sea válido, pero su n-1 no lo es.

Usando la misma propiedad de ser una función creciente, podemos saber dónde continuar la búsqueda binaria, debido a que, si no es válido el elemento, sabemos que el mínimo elemento válido estará en la sección derecha. Sin embargo, si el elemento es válido y no es el mínimo, el elemento mínimo estará en la sección izquierda de la búsqueda binaria.

Una vez descrita la función de la búsqueda binaria, procederemos a explicar cómo determinar si el elemento es válido o no, para ello tomaremos en cuenta en primer lugar las condiciones que describe el problema. Tomaremos en cuenta dos conjuntos, uno con el número primo que no le agrada al primer amigo y sus respectivos múltiplos, y el segundo será lo mismo, pero para el segundo amigo, sin embargo, estos dos conjuntos no son disjuntos debido a que existen elementos que pueden ser múltiplos tanto de uno como del otro primo que no es admisible. Por otra parte, los elementos resultantes del complemento de uno de los conjuntos, que serán los números validos para dicho sujeto, incluyen parte de los números primos que no acepta el otro sujeto y viceversa. La última consideración que hay que tener, es que un elemento puedo ser válido para ambos sujetos, sin embargo, no podemos dárselo a ambos.

Así, la función de CheckPosition nos devolverá un valor booleano dependiendo de si el número tiene suficientes elementos con las consideraciones hechas anteriormente. En el código, unvalid es la intersección de ambos conjuntos, number/x son los números primos inválidos para el primer amigo, number/y es lo mismo para el segundo amigo, giftsC1 es el complemento del conjunto de números inválidos para el primer amigo, y giftsC2 es lo mismo para el primero.

Código

```
#include < bits / stdc++.h >
using namespace std;
typedef long long int lli;
const lli INF = 1 LL << 60;
lli c1, c2, x, y;

bool CheckPosition(lli number) {
    lli unvalid = number / (x * y);
    lli giftsC1 = (c1 - ((number / y) - unvalid));
    if (giftsC1 < 0) {
        giftsC1 = 0;
    }
    lli giftsC2 = (c2 - ((number / x) - unvalid));
    if (giftsC2 < 0) {
        giftsC2 = 0;
    }
    lli complement = (number - (number / y) - (number / x) + unvalid);
    return ((giftsC1 + giftsC2) <= complement);
}

lli BinarySearch(lli left, lli right) {
    lli mid = left + ((right - left) / 2);
    if (CheckPosition(mid) && !CheckPosition(mid - 1)) {
        return mid;
    } else if (CheckPosition(mid)) {
        return BinarySearch(left, mid - 1);
```

```
    } else {
        return BinarySearch(mid + 1, right);
    }
}

int main() {
    cin >> c1 >> c2 >> x >> y;
    cout << BinarySearch(c1 + c2, INF) << endl;
    return 0;
}
```

Análisis de complejidad

En el main tenemos un código que consideraremos constante al ser únicamente lectura de datos, por lo que pasaremos a analizar la función relevante en el código que es la Búsqueda Binaria, sin embargo, primero deberemos analizar la complejidad de la función CheckPosition para tomar en cuenta en el sucesivo análisis.

Podemos observar que la función recibe un número y a partir de él calcula otros valores que ocuparemos para determinar si es válido o no, sin embargo, todas las operaciones se consideran constantes debido a su coste, de cómputo, esto incluye las comparaciones, por lo que podemos considerar el coste de la función como:

$$f_t(n) = O(1)$$

Una vez que vemos que el coste de ésta función es constante, regresaremos a analizar la función BinarySearch, pero ya se había analizado la complejidad en ésta, resultando en:

$$T(0) = 1$$

$$T(n) = 1 + T\left(\frac{n}{2}\right)$$

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)\ ;\ donde\ a = 1, b = 2\ y\ f(n) = 1$$

$$O\big(f(n)\big) = O(1)$$

$$Tomando\ el\ caso\ 2, tenemos\ que:$$

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$$

$$Esto\ es\ asintóticamente\ igual\ a\ O\big(f(n)\big)$$

$$\therefore T(n) = \theta\big(n^{\log_b a} \log n\big) = \theta(\log n)$$

Así, el coste total de nuestro programa es:

$$f_t(n) = O(1 + \log n) = O(\log n)$$

INVCNT – Inversion Count

Redacción del ejercicio

INVCNT - Inversion Count

#graph-theory #number-theory #shortest-path #sorting #bitmasks

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.

Example

```
Input:
2

3
3
1
2

5
2
3
8
6
1


Output:
2
5
```

Captura de aceptación

ID	DATE	USER	PROBLEM	RESULT	TIME	MEM	LANG
21666101	2018-05-16 01:50:33	akotadi	Inversion Count	**accepted** edit ideone it	0.20	18M	CPP14

Explicación de la solución

Usaremos la estructura de datos conocida como Segment Tree para modelar la solución, en este caso es una versión en un arreglo estático en lugar de la versión dinámica con apuntadores, esto nos permitirá un acceso más directo a los elementos de interés.

En primer lugar, vamos a leer los datos de entrada y colocarlos en un arreglo, posteriormente, ese arreglo lo colocaremos en otro junto a su índice de inicio en el arreglo original, éste será ordenado descendentemente con base en el número más alto que encontremos en el arreglo.

A continuación, iniciaremos el procedimiento con el Segment Tree que previamente habremos inicializado, éste consiste en realizar iteraciones sobre los segmentos, iniciaremos con el elemento más grande del arreglo, la primer query siempre nos regresará un cero ya que esos son los valores del arreglo al inicio, sin embargo, posteriormente procederemos a colocarlo en su respectiva posición del Segment Tree, en ella añadiremos una unidad para que las siguientes visitas lo contabilicen y vayan aumentando si es el caso.

Las siguientes iteraciones realizan el mismo procedimiento, las queries lo que realizarán es navegar hasta el respectivo segmento, y en él encontrará un valor que indica los números que son mayores que él, sin embargo, esto también dependerá de su índice original, ya que la navegación a través del Segment Tree se realiza con base a éste, que también es considerado a la hora de insertar, ya que según el índice iremos dividiendo las queries para contabilizar el segmento correspondiente del árbol y estos segmentos tendrán distintos valores según los que sean más grandes que el que es de nuestro interés.

A la hora de insertar un valor, se realiza un procedimiento similar a la query, con base en el índice original dividiremos el segmento en el árbol y aumentaremos una unidad en la posición final, que se sumarán recursivamente en la llamada original para fijar el valor resultante en lo que se considera como el nodo padre.

Todos los valores que nos regresan las queries se suman en una variable que será la que retornemos para ser impresa y otorgarnos el resultado final.

Código

```
#include < bits / stdc++.h >
using namespace std;
typedef long long int lli;
int n;

class SegmentTree {
    vector < int > segmentNode;
    public: void init(int n) {
        int N = 4 * n;
        segmentNode.resize(N, 0);
    }
    void insert(int node, int left, int right,
        const int indx) {
        if (indx < left or indx > right) {
            return;
        }
        if (left == right and indx == left) {
            segmentNode[node]++;
            return;
        }
        int leftNode = node << 1;
        int rightNode = leftNode | 1;
        int mid = left + (right - left) / 2;
        insert(leftNode, left, mid, indx);
```

```
25.         insert(rightNode, mid + 1, right, indx);
26.         segmentNode[node] = segmentNode[leftNode] + segmentNode[rightNode];
27.     }
28.     int query(int node, int left, int right,
29.         const int L,
30.             const int R) {
31.         if (left > R or right < L) {
32.             return 0;
33.         }
34.         if (left >= L and right <= R) {
35.             return segmentNode[node];
36.         }
37.         int leftNode = node << 1;
38.         int rightNode = leftNode | 1;
39.         int mid = left + (right - left) / 2;
40.         return query(leftNode, left, mid, L, R) + query(rightNode, mid + 1, right, L, R);
41.     }
42. };
43.
44. lli countGreater(vector < int > & v) {
45.     lli result = 0;
46.     vector < pair < int, int > > v(n);
47.     for (int i = 0; i < n; ++i) {
48.         v[i] = pair < int, int > (nums[i], i);
49.     }
50.     sort(data.begin(), data.end(), greater < pair < int, int > > ());
51.     SegmentTree segmentTree;
52.     segmentTree.init(n);
53.     for (int i = 0; i < n; ++i) {
54.         result += segmentTree.query(1, 0, n - 1, 0, v[i].second);
55.         segmentTree.insert(1, 0, n - 1, v[i].second);
56.     }
57.     return result;
58. }
59.
60. int main( ) {
61.     int t;
62.     cin >> t;
63.     for (int i = 0; i < t; ++i) {
64.         cin >> n;
65.         vector < int > data(n, 0);
66.         for (int j = 0; j < n; ++j) {
67.             cin >> data[j];
68.         }
69.         cout << countGreater(data) << endl;
70.     }
71.     return 0;
72. }
73.
```

Análisis de complejidad

Iniciaremos en el main, donde tendremos dos variables de las que dependerá el tamaño del problema, una es la *t* que es el número de queries a resolver, la segunda es la *n* para cada querie, que es el número de elementos que tendrá el arreglo en cada una de las queries. Por ello, nos enfocaremos el primer lugar en la complejidad para *n*, debido a que es la más significativa para el algoritmo.

Dentro del primer ciclo for, que resolverá cada una de las queries, nos encontramos que en primer lugar hacemos una lectura de todos los datos y los coloca en un arreglo, por ello la complejidad de ese ciclo es:

$$f_t(n) = n = O(n)$$

Posteriormente se manda llamar la función countGreater, que procederemos a analizar para determinar su complejidad y luego compararla con la obtenida anteriormente para dar un resultado final. Así, en primer lugar, la función vuelve a realizar una iteración sobre el arreglo y lo copia en un arreglo de pares, por lo que la complejidad es:

$$f_t(n) = n = O(n)$$

La función *sort* tiene un coste de O(n), tanto la declaración como la inicialización del Segment Tree es constante por las operaciones que realiza. Volvemos a encontrar un ciclo for que recorrerá todos los valores hasta n, sin embargo, dentro del mismo realiza operaciones de la clase Segment Tree, así que habrá que analizarlas en primer lugar para determinar el resultado.

La operación de query es un método recursivo que además tiene recurrencias no líneas al ir dividiendo la *n* en segmentos de mitades, sus operaciones quedarían distribuidas de la siguiente manera:

$$T(0) = 1$$

$$T(n) = 1 + T\left(\frac{n}{2}\right)$$

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)\ ;\ donde\ a = 1, b = 2\ y\ f(n) = 1$$

$$O\big(f(n)\big) = O(1)$$

$$Tomando\ el\ caso\ 2, tenemos\ que:$$

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$$

$$Esto\ es\ asintóticamente\ igual\ a\ O\big(f(n)\big)$$

$$\therefore T(n) = \theta\big(n^{\log_b a} \log n\big) = \theta(\log n)$$

La operación de insertar tiene una estructura similar, que podemos ver a continuación:

$$T(0) = 0$$

$$T(n) = 0 + T\left(\frac{n}{2}\right)$$

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)\ ;\ donde\ a = 1, b = 2\ y\ f(n) = 0$$

$$O\big(f(n)\big) = O(0) = O(1)$$

$$Tomando\ el\ caso\ 2, tenemos\ que:$$

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$$

$$Esto\ es\ asintóticamente\ igual\ a\ O\big(f(n)\big)$$

$$\therefore T(n) = \theta\big(n^{\log_b a} \log n\big) = \theta(\log n)$$

Tomando en cuenta las operaciones arriba mencionadas, tenemos que el ciclo for tiene una complejidad de:

$$f_t(n) = O\left(n\big(\theta(\log n) + \theta(\log n)\big)\right) = O\left(n\big(2\theta(\log n)\big)\right) = O(2n \log n) = O(n \log n)$$

Que, si lo comparamos con el primer ciclo for, es posible observar que nuestro último resultado tiene una mayor complejidad por lo que es el dominante en esta función, por lo tanto, la función countGreater tiene una complejidad de:

$$f_t(n) = O(n + n \log n) = O(n \log n)$$

Volviendo al ciclo inicial, encontramos que, para cada query, la complejidad será de:

$$f_t(n) = O(n + n \log n) = O(n \log n)$$