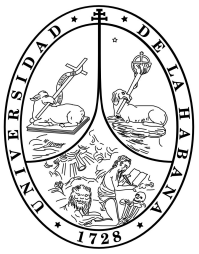


Proyecto de Diseño y Análisis de Algoritmos
"Meet in the middle"

Est. **Ariel Plasencia Díaz** A.PLASENCIA@ESTUDIANTES.MATCOM.UH.CU

C-312

Índice

1 Introducción

El objetivo de este trabajo es explicar y desarrollar la técnica **meet in the middle**, así como mostrar diversos ejercicios resueltos para una mejor comprensión del algoritmo. Dicha técnica fue introducida en 1974 por E. Horowitz y S. Sahni.

Meet in the middle es una técnica de búsqueda que se usa cuando la entrada es pequeña, pero no tan pequeña, de tal forma que la fuerza bruta no pueda usarse. Se procede como divide y vencerás, se divide el problema en dos subproblemas, se resuelven individualmente y se combinan para arribar a una solución. Lo explicado anteriormente no siempre se puede aplicar porque los subproblemas no tienen la misma estructura que el problema original.

2 Presentación de la técnica

Meet in the middle es una técnica donde el espacio de búsqueda es dividido en dos partes de igual tamaño. Se realizan búsquedas por separado para las dos partes, y finalmente se combinan los resultados de las búsquedas para llegar a la solución.

La técnica puede usarse si hay una manera eficaz de combinar los resultados de las búsquedas. En semejante situación, las dos búsquedas pueden requerir menos tiempo que una búsqueda grande. Típicamente, nosotros podemos convertir un factor de 2^N en un factor de $2^{\frac{N}{2}}$ usando la técnica **meet in the middle**. Como detalle nos percatamos que esta técnica se aplica para cuando N no excede de 40, es decir para $N \leq 40$, ya que 2^{40} nos tomaría un aproximado de 11 días en una computadora convencional, sin embargo usando **meet in the middle**, nos llevaría un poco menos de 1 segundo porque $2^{\frac{40}{2}} = 2^{20} = 1048576 \leq 2 * 10^6$ y 1048576 operaciones básicas cualquier microprocesador en nuestros días las realiza en un segundo.

2.1 Ejemplo de muestra

Consideremos el problema donde tenemos una lista L de N ($N \leq 40$) números y un entero X, y queremos averiguar si es posible escoger un subconjunto de la lista para que su suma sea X. Por ejemplo, dado $L = \{2, 4, 5, 9\}$ y $X = 15$, podemos escoger el subconjunto $C = \{2, 4, 9\}$ tal que $2 + 4 + 9 = 15$. Sin embargo, para $X = 10$ y la misma lista, no es posible formar la suma.

Una simple idea al problema es pasar por todos los subconjuntos de la lista L y chequear si la suma de cualquiera de los subconjuntos es X. La complejidad temporal a semejante idea es $O(2^N)$, porque existen 2^N subconjuntos. Sin embargo, usando la técnica **meet in the middle**, podemos lograr un costo más eficiente que el algoritmo anterior.

La solución por **meet in the middle** es dividir la lista L en dos listas A y B tal que ambas listas tengan la misma cantidad de números. La primera búsqueda genera todos los subconjuntos de A y en otra lista SA guarda las sumas de esos subconjuntos. Correspondientemente, la segunda búsqueda crea una lista SB con sus respectivas sumas pero en este caso de la lista B. Después de esto, basta verificar si es posible escoger un elemento de SA y otro elemento de SB tal que su suma sea X. Esto es exactamente posible cuando hay una manera de formar la suma X que usa los elementos de la lista original.

Por ejemplo, supongamos $L = \{2, 4, 5, 9\}$ y $X = 15$. Primero, dividimos la lista L en $A = \{2, 4\}$ y $B = \{5, 9\}$. Después de esto, creamos la lista $SA = \{0, 2, 4, 6\}$ y $SB = \{0, 5, 9, 14\}$ de forma tal que en estas dos últimas están las sumas de todos sus respectivos subconjuntos (incluyendo el conjunto vacío). En este caso, es posible formar la suma X porque SA contiene la suma 6, SB contiene la suma 9, y $6 + 9 = 15$. Esto corresponde a la solución $C = \{2, 4, 9\}$.

2.2 Complejidad temporal

La complejidad del algoritmo mostrado con anterioridad es $O(2^{\frac{N}{2}})$, porque ambas listas A y B contienen $\frac{N}{2}$ elementos que toma un tiempo de $2^{\frac{N}{2}}$ para calcular las sumas de sus respectivos subconjuntos a las listas SA y SB.

Luego ordenamos las lista SB en $Nlog(N)$. Después de esto, es posible encontrar dicha suma pasando por cada elemento de SA y buscando su complemento, es decir $X - SA[elemento]$, en SB, que se puede hacer con búsqueda binaria, por lo que $T(N)$ nos queda:

$T(N) = O(2^{\frac{N}{2}} + 2^{\frac{N}{2}} + N * log(N) + (\frac{N}{2}) * log(\frac{N}{2}))$

$T(N) = O(2 * 2^{\frac{N}{2}} + N * log(N) + (\frac{N}{2}) * log(\frac{N}{2}))$

$T(N) = O(2 * 2^{\frac{N}{2}})$

$T(N) = O(2^{\frac{N}{2}})$

En fin, la complejidad temporal del algoritmo **meet in the middle** es $O(2^{\frac{N}{2}})$.

3 Implementación

Para un mejor entendimiento de esta técnica nos basaremos en el problema explicado con anterioridad.

3.1 Problema

Dada una secuencia de N ($1 \leq N \leq 34$) números $S_1, S_2, ..., S_N$ ($-20000000 \leq S_i \leq 20000000$) y un entero X ($-500000000 \leq X \leq 500000000$), determina cuántos subconjuntos de S (incluido el vacío) suman X.

ENTRADA: La primera línea de la entrada contiene dos enteros N y X. Las siguientes N líneas contienen desde S_1 hasta S_N en ese orden.

SALIDA: Imprime un único entero que represente la cantidad de subconjuntos que satisfacen la propiedad anterior.

3.2 Código

Solución en C++

```
#include<bits/stdc++.h>
#define endl '\n'

using namespace std;

int A[100];
int B[100];
int SA[300000];
int SB[300000];

// método que separa lista(vector) desde first_pos(int) hasta last_pos(int)
void separate_array(vector<int> lista, int arr[], int first_pos, int last_pos) {
    int pos = 0;
    for(int i = first_pos; i < last_pos; i++) {
        arr[pos] = lista[i];
        pos++;
    }
}

// método que calcula las sumas de todos los elementos de arr(int[]) y lo guarda
// en a(int[])
void calculate_subcjto(int arr[], int len, int a[]) {
    for(int i = 0; i < (1 << len); i++) {
        int sum = 0;
        for(int j = 0; j < len; j++) {
            if(i & (1 << j))
                sum += arr[j];
        }
        a[i] = sum;
    }
}

// método que devuelve cuántos subconjuntos hay en L(vector) tal que su suma es X(int)
int get_solution(vector<int> L, int N, int X) {
    int len_A = N / 2;
    int len_B = N - len_A;

    separate_array(L, A, 0, len_A);
    separate_array(L, B, len_B - 1, N);
```

```
    calculate_subcjto(A, len_A, SA);
    calculate_subcjto(B, len_B, SB);

    int len_SA = 1 << len_A;
    int len_SB = 1 << len_B;

    // ordenamos SB(int[]) para realizar búsqueda binaria
    sort(SB, SB + len_SB);

    // en sol(int) guardamos la respuesta, es decir, cuantos subconjuntos
    // hay tal que su suma es X(int)
    int sol = 0;

    // iteramos por SA(int[]) y buscamos binario en SB(int[]) su complemento
    for(int i = 0; i < len_SA; i++) {
        int a = X - SA[i];

        // devuelve la posición del mayor elemento mayor o igual que X - SA[i]
        int x = lower_bound(SB, SB + len_SB, a) - SB;

        // devuelve la posición del mayor elemento mayor que X - SA[i]
        int y = upper_bound(SB, SB + len_SB, a) - SB;

        if(y >= x)
            sol += (y - x);
    }
    return sol;
}

int main() {
    int N, X, a;
    cin >> N >> X;

    vector<int> L;

    for(int i = 0; i < N; i++) {
        cin >> a;
        L.push_back(a);
    }

    int sol = get_solution(L, N, X);
    cout << sol << endl;
}
```

4 Ejercicios

En esta sección se utilizará la técnica de **meet in the middle** para resolver problemas un poco más interesantes, mostrando la utilidad de la misma. En algunos casos se mostrará la solución en varios lenguajes pero siempre haciendo referencia a C++.

4.1 Subset sums

4.1.1 Problema

Ver problema

Dada una secuencia de N ($1 \leq N \leq 34$) números S_1, S_2, ..., S_N ($-20,000,000 \leq S_i \leq 20,000,000$), determina cuántos subconjuntos de S (incluido el vacío) tienen su suma entre A y B ($-500,000,000 \leq A \leq B \leq 500,000,000$).

ENTRADA: La primera línea de la entrada contiene tres enteros N, A y B. Las siguientes N líneas contienen desde S_1 hasta S_N en ese orden.

SALIDA: Imprime un único entero que represente la cantidad de subconjuntos que satisfacen la propiedad anterior.

4.1.2 Análisis

En el ejercicio anterior nos piden cuántos subconjuntos existen tal que suma está entre A y B. Cabe destacar que la solución de probar con todos los subconjuntos no cumple con las restricciones de tiempo que nos exige el problema, que sería.

Como la cantidad de elementos no es muy grande ($1 \leq N \leq 34$), calculamos la suma de todos los subconjuntos tanto de la primera mitad del array como de la segunda, pero esta última la guardamos ordenada. Revisamos en el array de las sumas de la segunda mitad si existe algún subconjunto tal que su suma (S) cumpla que $A \leq S + X[i] \leq B$, esto último se hace de forma eficiente con búsqueda binaria.

4.1.3 Implementación

Solución en C++
Solución en csharp
Solución en Python

```
#include<bits/stdc++.h>
#define endl '\n'

using namespace std;

int X[300000];
int Y[300000];

// metodo que calcula las sumas de todos los elementos de lista(vector) a partir de
// ind(int) y lo guarda en arr(int[])
void calculate_subcjto(vector<int> lista, int arr[], int len, int ind) {
    for(int i = 0; i < (1 << len); i++) {
        int sum = 0;
        for(int j = 0; j < len; j++) {
            if(i & (1 << j))
                sum += lista[j + ind];
        }
        arr[i] = sum;
    }
}

// metodo que devuelve cuantos subconjuntos hay en lista(vector) entre A(int) y B(int)
int get_solution(vector<int> lista, int N, int A, int B) {
    int len_X = N / 2;
    int len_Y = N - len_X;

    calculate_subcjto(lista, X, len_X, 0);
    calculate_subcjto(lista, Y, len_Y, N / 2);

    int size_X = 1 << (len_X);
    int size_Y = 1 << (len_Y);

    // ordenamos Y(int[]) para realizar busqueda binaria
    sort(Y, Y + size_Y);

    // en sol(int) guardamos la respuesta, es decir, cuantos subconjuntos hay
```

```
    // entre A(int) y B(int)
    int sol = 0;

    // iteramos por X(int[]) y buscamos binario en Y(int[]) su complemento
    for(int i = 0 ; i < size_X; i++) {
        int a = A - X[i];
        int b = B - X[i];

        // devuelve la posicion del mayor elemento mayor o igual que A - X[i]
        int x = lower_bound(Y, Y + size_Y, a) - Y;
        // devuelve la posicion del mayor elemento mayor que B - X[i]
        int y = upper_bound(Y, Y + size_Y, b) - Y;

        if(y >= x)
            sol += (y - x);
    }

    return sol;
}

int main() {
    int A, B, N, a;
    cin >> N >> A >> B;

    vector<int> lista;

    for(int i = 0; i < N; i++) {
        cin >> a;
        lista.push_back(a);
    }

    int sol = get_solution(lista, N, A, B);
    cout << sol << endl;
}
```

4.2 Knapsack problem

4.2.1 Problema

Ulises es un cleptómano, es decir una persona adicta al robo. El recibe todos los meses una lista de los pesos de la tienda que está en la esquina de su casa. Ulises tiene una mochila y quiere saber cuántos subconjuntos de objetos le caben en la mochila sin que esta se rompa (sobrepase su peso). El le ha pedido ayuda a su hijo pero este ha estado muy ocupado con sus proyectos de la universidad. Usted puede ayudar a Ulises.

ENTRADA: La primera línea de la entrada contiene dos enteros N ($N \leq 40$) y K ($K \leq 10^{18}$), donde N son los pesos de la tienda y K es el peso de la mochila. La segunda línea contiene todos los pesos.

SALIDA: La solución al problema de Ulises.

4.2.2 Análisis

Resumiendo el problema anterior, nos piden calcular dado una lista L de números, cuántos subconjuntos hay menores que K. Para ello, utilizaremos la idea vista hasta el momento para generar las sumas de todos los subconjuntos tanto de la primera como de la segunda mitad de L procesadas en X y Y respectivamente. Luego ordenamos Y.

Iteramos por cada elemento de X y buscando binariamente en Y su complemento, es decir, un número tal que su suma no sobrepase K. Lo anterior se puede realizar porque Y está ordenado.

4.2.3 Implementación

Solución en C++
Solución en csharp
Solución en Python

```
#include<bits/stdc++.h>
#define endl '\n'

using namespace std;

int X[300000];
int Y[300000];

void generate_sum_subcjto(vector<int> weigth, int arr[], int len, int ind) {
    for(int i = 0; i < (1 << len); i++) {
        int sum = 0;
        for(int j = 0; j < len; j++) {
            if(i & (1 << j))
                sum += weigth[j + ind];

        }
        arr[i] = sum;
    }
}

int my_binary_search(int arr[], int len, int x, int weigth_knapsack) {

    int ans = 0;
    int low = 0;
    int high = len - 1;

    while(low < high) {

        int m = (low + high + 1) / 2;

        if(arr[m] + x <= weigth_knapsack)
            low = m;
        else
            high = m - 1;
    }

    if(x <= weigth_knapsack)
        ans = low + 1;

    return ans;
}

int knapsack_problem(vector<int> weigth, int weigth_knapsack, int N) {

    int len_X = N / 2;
    int len_Y = N - len_X;

    generate_sum_subcjto(weigth, X, len_X, 0);
    generate_sum_subcjto(weigth, Y, len_Y, N / 2);

    int size_X = 1 << (len_X);
    int size_Y = 1 << (len_Y);
```

```
    sort(Y, Y + size_Y);

    int sol = 0;
    for(int i = 0 ; i < size_X; i++) {
        sol += my_binary_search(Y, size_Y, X[i], weigth_knapsack);
    }

    return sol;
}

int main() {

    int weigth_knapsack, N, a;
    cin >> N >> weigth_knapsack;

    vector<int> weigth;

    for(int i = 0; i < N; i++) {
        cin >> a;
        weigth.push_back(a);
    }

    int sol = knapsack_problem(weigth, weigth_knapsack, N);
    cout << sol << endl;
}
```

4.3 Maximum Subsequence

4.3.1 Problema

Ver problema

Dado una lista L que contiene N enteros y otro número M. Debes seleccionar alguna secuencia de índices $b_1, b_2, ..., b_k$ $(1 \leq b_1 \leq b_2 \leq ... \leq b_k)$ tal que $\sum_{i=1}^{k} L_{b_i} \bmod M$ is máximo. Dicha secuencia puede ser vacía.

ENTRADA: La primera línea contiene dos enteros N y M $(1 \leq N \leq 35, 1 \leq M \leq 10^9)$. La segunda línea contiene N enteros $a_1, a_2, ..., a_N$ $(1 \leq i \leq 10^9)$.

SALIDA: Imprime el máximo valor posible de $\sum_{i=1}^{k} L_{b_i} \bmod M$.

4.3.2 Análisis

En el problema nos dan una lista de números enteros y un número M y nos piden encontrar la suma máxima de un subconjunto módulo M. Primeramente procesaremos la primera mitad (X) de la lista de números y nos quedaremos con la suma de los subconjuntos que esta genera módulo M, luego hacemos lo mismo para la segunda mitad (Y) pero lo guardamos de forma ordenada.

Ahora recorremos la primera mitad como sigue: buscamos en Y el mayor elemento S tal que $X[i] + S$ sea máximo, lo anterior se efectúa con búsqueda binaria ya que el array está ordenado.

4.3.3 Implementación

Solución en C++
Solución en csharp
Solución en Python

```
#include<bits/stdc++.h>
#define endl '\n'
```

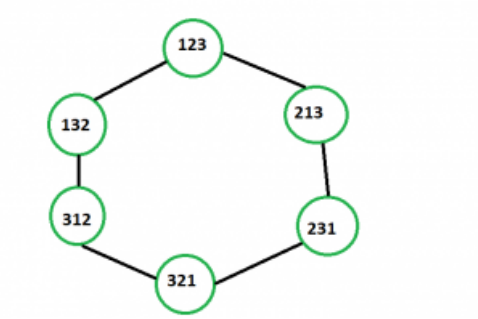
```
using namespace std;

vector<int> X, Y;

void generate_subcjto(vector<int> L, vector<int> &X, int len, int ind, int M) {
    for(int i = 0; i < (1 << len); i++) {
        int sum = 0;
        for(int j = 0; j < len; j++) {
            if(i & (1 << j))
                sum = (sum + L[j + ind]) % M;
        }
        X.push_back(sum);
    }
}

int get_solution(vector<int> L, int N, int M) {
    int len_X = N / 2;
    int len_Y = N - len_X;

    generate_subcjto(L, X, len_X, 0, M);
    generate_subcjto(L, Y, len_Y, N / 2, M);

    sort(Y.begin(), Y.end());
    int resp = 0;

    for(int i = 0; i < X.size(); i++) {
        int pos = upper_bound(Y.begin(), Y.end(), M - 1 - X[i]) - Y.begin() - 1;
        if(pos >= 0)
            resp = max(resp, X[i] + Y[pos]);
    }

    return resp;
}

int main() {

    int N, M, a;
    cin >> N >> M;

    vector<int> L;

    for(int i = 0; i < N; i++) {
        cin >> a;
        L.push_back(a);
    }

    int resp = get_solution(L, N, M);
    cout << resp << endl;
}
```

4.4 Shortest Path

4.4.1 Problema

Given a permutation $P = p_1, p_2, ..., p_N$ of first n natural numbers ($1 \leq n \leq 10$). One can swap any two consecutive elements p_i and p_{i+1} ($1 \leq i \leq N - 1$). The task is to find the minimum number of swaps to change P to another permutation $P' = p'_1, p'_2, ..., p'_n$.

4.4.2 Análisis



Este problema puede ser resuelto usando un simple BFS (breadth first search). Para ello convertiremos el problema en un grafo. Las permutaciones son los nodos y las aristas son los intercambios entre las permutaciones. Visto de esta manera, el problema se reduce a encontrar el menor camino simple entre los dos vértices principales. Tenemos $N!$ nodos, donde cada nodo tiene $N-1$ aristas, por lo que nuestra complejidad temporal es $O(N * log(N!) * N!)$. A continuación, usando **meet in the middle**, propondremos una solución más rápida al algoritmo anterior con algunas modificaciones.

Llamémosle a P vértice *start* y a P' vértice $finish$. Sean P y P' las raíces de nuestro grafo. Empezaremos el BFS al mismo tiempo por ambas raíces utilizando una misma cola. Primeramente en la cola están P y P', por lo que $visited_{start} = visited_{finish} = true$. Además en src_u está la raíz del vértice u, entonces $src_{start} = start$ y $src_{finish} = finish$, también en d_u se almacena la distancia más corta del vértice u hasta la raíz, o sea, $d_{start} = d_{finish} = 0$, tal que u pertenece $V(G)$.

Mientras la cola no esté vacía, sacamos el elemento que se encuentre en el tope de la cola, al cual llamaremos u, y meteremos todos los vértices v que son adyacentes con u y no se han visitado todavía ($visited_v = false$), entonces haremos $d_v = d_u + 1$, $src_v = src_u$ y $visited_v = true$. Sobre todo, si v fuera visitado y $srcv \neq srcu$, entonces la solución es $Du + Du + 1$.

4.4.3 Implementación

Solución en C++
Solución en csharp
Solución en Python

```
#include <bits/stdc++.h>
#define endl "\n"

using namespace std;

// function to find minimum number of swaps to make another permutation
int shortest_path(string start, string finish, int n) {
    queue<string> _queue;
    map<string, bool> visited;
    map<string, int> d;
    map<string, string> src;

    visited[start] = visited[finish] = true;
    d[start] = d[finish] = 0;
    src[start] = start;
    src[finish] = finish;
    _queue.push(start);
    _queue.push(finish);

    while (!_queue.empty()) {
        // take top vertex of the queue
        string element = _queue.front();
        _queue.pop();
```

```
        // generate n - 1 of it is permutations
        for (int i = 1; i < n; i++) {
            // generate next permutation
            string temp = element;
            swap(temp[i], temp[i - 1]);

            // ff temp is not visited
            if (!visited[temp]) {
                visited[temp] = true; // set it visited
                src[temp] = src[element]; // make root of u and root of v equal
                d[temp] = d[element] + 1; // increment it is distance by 1
                _queue.push(temp); // push this vertex into queue
            }
            // if it is already visited and roots are different then answer is found
            else if (src[element] != src[temp])
                return d[element] + d[temp] + 1;
        }
    }
}

int main() {
    string A, B;
    cin >> A >> B;

    int N = A.length();

    cout << shortest_path(A, B, N) << endl;
}
```

4.5 Discrete logarithm

4.5.1 Problema

Given three integers A, B and M. Find an integer K such that $A^K \equiv B \pmod{M}$ where A and M are relatively prime. If it is not possible for any K to satisfy this relation, print -1.

Input: 2 3 5
Output: 3
Explanation: $A = 2, B = 3, M = 5$. The value which satisfies the above equation is 3, because $2^3 = 2 * 2 * 2 = 8 \Rightarrow 2^3 \pmod 5 = 8 \pmod 5 \Rightarrow 3$.
Input: 3 7 11
Output: -1

4.5.2 Análisis

Una posible solución, en este caso por fuerza bruta, sería iterar desde 0 hasta M para probar con todos los valores de K y quedarnos con la satisfaga la propiedad. Si se llega al final y no se ha encontrado solución entonces la respuesta es -1. Esta idea es $O(M)$.

Usando la técnica **meet in the middle**, dividimos el problema en dos partes de $\sqrt{M}$, las resolvemos individualmente y encontramos las colisiones.

Primero que todo escribimos $K = i \cdot N - j$, donde $N = \left\lceil \sqrt{M} \right\rceil$. Nos percatamos que ningun valor de K en el intervalo $[0, M)$ puede ser representado de esta forma, donde $i \in [1, N)$ y $j \in [0, N)$.

$$\Longrightarrow A^K \equiv B \pmod{M}$$

$$\Longrightarrow A^{i \cdot N - j} \equiv B \pmod{M}$$

$$\Longrightarrow A^{i \cdot N} \equiv A^j \cdot B \pmod{M}$$

Ambos miembros pueden tomar solo valores distintos a N ya que $i, j \in [0, N)$. Generamos todos los posibles números para cualquiera de los dos miembros y lo guardamos en una estructura de datos, en este caso un *map*. Supongamos que tenemos todos los valores del miembro izquierdo, ahora iteramos por todos los términos del miembro derecho y buscamos el j que satisface la última ecuación. En caso que no haya solución imprimimos -1.

4.5.3 Implementación

Solución en C++
Solución en csharp
Solución en Python

```
#include<bits/stdc++.h>
#define endl "\n"

using namespace std;

int pow(int a, int n, int m) {
    if(n == 0)
        return 1;
    else if(n % 2 == 0) {
        int pot = pow(a, n / 2, m);
        return (pot % m * pot % m) % m;
    }
    else {
        int pot = pow(a, n - 1, m);
        return (pot % m * a % m) % m;
    }
}

int discrete_logarithm(int a, int b, int m) {

    int n = sqrt(m) + 1;

    // calculate (a ^ n) % m
    int an = pow(a, n, m);

    map<int, int> value;

    for(int i = 1, current = an; i <= n; i++) {
        if(!value[current])
            value[current] = i;
        current = (current * an) % m;
    }

    for(int i = 0, current = b; i <= n; i++) {
        // calculate (a ^ j) * b and check for collision
        if (value[current]) {
            int ans = value[current] * n - i;
            if (ans < m)
                return ans;
        }
        current = (current * a) % m;
    }

    return -1;
}

int main() {
```

```
    int A, B, M;
    cin >> A >> B >> M;

    cout << discrete_logarithm(A, B, M) << endl;
}
```

4.6 ABCDEF

4.6.1 Problem

Ver problema

You are given a set S of integers between -30000 and 30000 (inclusive). Find the total number of sextuples $(a, b, c, d, e, f) : a, b, c, d, e, f \in S$ y $d \neq 0$ that satisfy:

$$\frac{a \cdot b + c}{d} - e = f$$

INPUT: The first line contains integer N ($1 \leq N \leq 100$), the size of a set S. Elements of S are given in the next N lines, one integer per line. Given numbers will be distinct.

OUTPUT: Output the total number of plausible sextuples.

4.6.2 Análisis

Podemos escribir la ecuación de la siguiente manera usando transformaciones elementales:

$\frac{a \cdot b + c}{d} - e = f$

$\Rightarrow \frac{a \cdot b + c}{d} = f + e$

$\Rightarrow a \cdot b + c = d \cdot (f + e)$

$\Rightarrow a \cdot b + c = d \cdot f + d \cdot e$

Ahora podemos calcular todos los posibles conjuntos de valores del miembro izquierdo ($a \cdot b + c$) y del miembro derecho ($d \cdot (f + e)$) en $O(N^3)$, teniendo en cuenta que $N \leq 100$. Notar que d no puede ser 0, porque la división por 0 no está definida. Sea L y R, dos arrays tal que se encuentran todos los valores antes mencionado tanto del miembro izquierdo como del derecho respectivamente. Ordenamos estos arrays.

Calculamos la respuesta comparando L con R y verificando los valores repetidos, para ello contamos la frecuencia de todos los valores y lo guardamos en arrays diferentes, por ejemplo $cntL$ para el miembro izquierdo y $cntR$ para el derecho. Luego, con tan solo una iteración por los arrays L y R podemos saber cuántas séxtuplas existen tal que $\frac{a \cdot b + c}{d} - e = f$.

La complejidad temporal quedaría:

$T(N) = O(N^3) + O(N^3) + O(N \cdot log(N)) + O(N \cdot log(N)) + O(N^3) + O(N^3) + O(N^3)$

$T(N) = 5 \cdot O(N^3) + 2 \cdot O(N \cdot log(N))$

Por tanto, $T(N) = O(N^3)$, donde N es la cantidad de elementos del array original.

4.6.3 Implementación

Solución en C++

```
#include<bits/stdc++.h>
#define endl '\n'

using namespace std;

int L[1000000], R[1000000], cntL[1000000], cntR[1000000];

int generate_left(int arr[], int N) {
    int sizeL = 0;
    for(int i = 0; i < N; i++) {
```

```
        for(int j = 0; j < N; j++) {
            for(int k = 0; k < N; k++) {
                L[sizeL] = (arr[i] * arr[j]) + arr[k];
                sizeL++;
            }
        }
    }
    return sizeL;
}

int generate_rigth(int arr[], int N) {
    int sizeR = 0;
    for(int i = 0; i < N; i++) {
        if(arr[i] == 0)
            continue;

        for(int j = 0; j < N; j++) {
            for(int k = 0; k < N; k++) {
                R[sizeR] = arr[i] * (arr[j] + arr[k]);
                sizeR++;
            }
        }
    }
    return sizeR;
}

int get_pos(int L[], int cntL[], int sizeL) {
    int pos = 1;
    cntL[0] = 1;

    for(int i = 1; i < sizeL; i++) {
        if(L[i] != L[pos - 1]) {
            L[pos] = L[i];
            cntL[pos] = 1;
            pos++;
        }
        else
            cntL[pos - 1]++;
    }

    return pos;
}

long long get_answer(int pos1, int pos2) {
    long long ans = 0;
    for(int i = 0, j = 0; i < pos1 && j < pos2; ) {
        if(R[j] == L[i])
            ans += (long long)cntL[i] * cntR[j];
        if(L[i] < R[j])
            i++;
        else if(L[i] > R[j])
            j++;
        else {
            i++;
            j++;
        }
    }
    return ans;
```

```
}

long long get_solution(int arr[], int N) {

    int sizeL = generate_left(arr, N);
    int sizeR = generate_rigth(arr, N);

    sort(L, L + sizeL);
    sort(R, R + sizeR);

    int pos1 = get_pos(L, cntL, sizeL);
    int pos2 = get_pos(R, cntR, sizeR);

    long long ans = get_answer(pos1, pos2);

    return ans;
}

int main() {

    int N, x;
    cin >> N;

    int arr[N];

    for(int i = 0; i < N; i++) {
        cin >> x;
        arr[i] = x;
    }

    int ans = get_solution(arr, N);
    cout << ans << endl;
}
```

4.7 4 values whose sum is 0

4.7.1 Problem

Ver problema

The SUM problem can be formulated as follows: given four lists A, B, C, D of integer values, compute how many quadruplet (a, b, c, d) belongs to $A \times B \times C \times D$ are such that $a + b + c + d = 0$. In the following, we assume that all lists have the same size N.

INPUT: The first line of the input file contains the size of the lists N (this value can be as large as 2500). We then have N lines containing four integer values (with absolute value as large as 2^{28}) that belong respectively to A, B, C and D.

OUTPUT: Output should be printed on a single line.

4.7.2 Implementación

Solución en C++
Solución en csharp
Solución en Python

```
#include<bits/stdc++.h>
#define endl '\n'

using namespace std;

vector<int> X, Y;
```

```
void generate_sum(vector<int> first, vector<int> second, int flag) {

    for(int i = 0; i < first.size(); i++) {
        for(int j = 0; j < second.size(); j++) {
            int sum = first[i] + second[j];
            if(flag)
                X.push_back(sum);
            else
                Y.push_back(sum);
        }
    }
}

int get_solution(vector<int> A, vector<int> B, vector<int> C, vector<int> D) {

    generate_sum(A, B, 1);
    generate_sum(C, D, 0);

    sort(Y.begin(), Y.end());
    int sol = 0;

    for(int i = 0; i < X.size(); i++) {
        int a = 0 - X[i];

        int x = lower_bound(Y.begin(), Y.end(), a) - Y.begin();
        int y = upper_bound(Y.begin(), Y.end(), a) - Y.begin();

        if(y >= x)
            sol += (y - x);
    }

    return sol;
}

int main() {

    int N, a;
    cin >> N;

    vector<int> A, B, C, D;

    for(int i = 0; i < N; i++) {
        for(int j = 0; j < 4; j++) {
            cin >> a;

            if(j == 0)
                A.push_back(a);
            else if(j == 1)
                B.push_back(a);
            else if(j == 2)
                C.push_back(a);
            else
                D.push_back(a);
        }
    }

    int sol = get_solution(A, B, C, D);
```

```
    cout << sol << endl;
}
```