Computer Society
Uniandes

Programación Competitiva

Computer Society

Daniel Diaz

Tabla de Contenido

Solucionario

Introducción

01

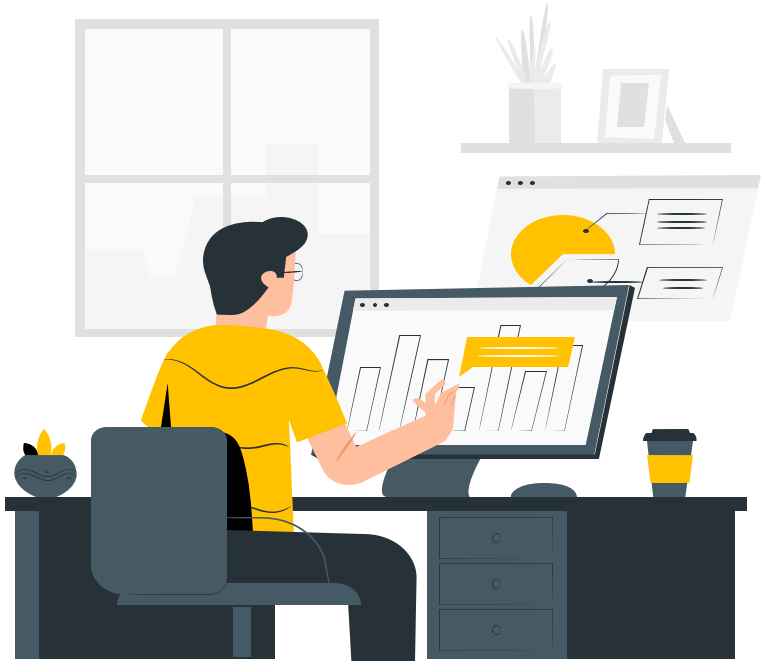
Presentación

Presentación de CS, presentación del Problemsetter

02

Revisión de problemas

Soluciones a los problemas introductorios



01

Presentación

¿Quién selecciona los problemas del curso?

El Problemsetter

Daniel Diaz

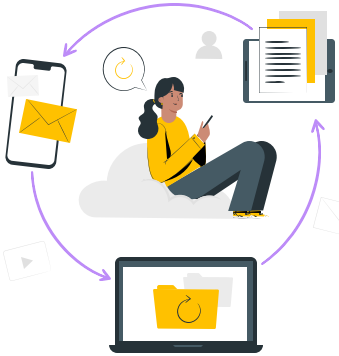
Problemsetter del equipo de Programación Competitiva

FML

Fundamentos matemáticos para la computación

EDA

Estructuras de Datos y Algoritmos



DSW

Desarrollo de Software en Equipos

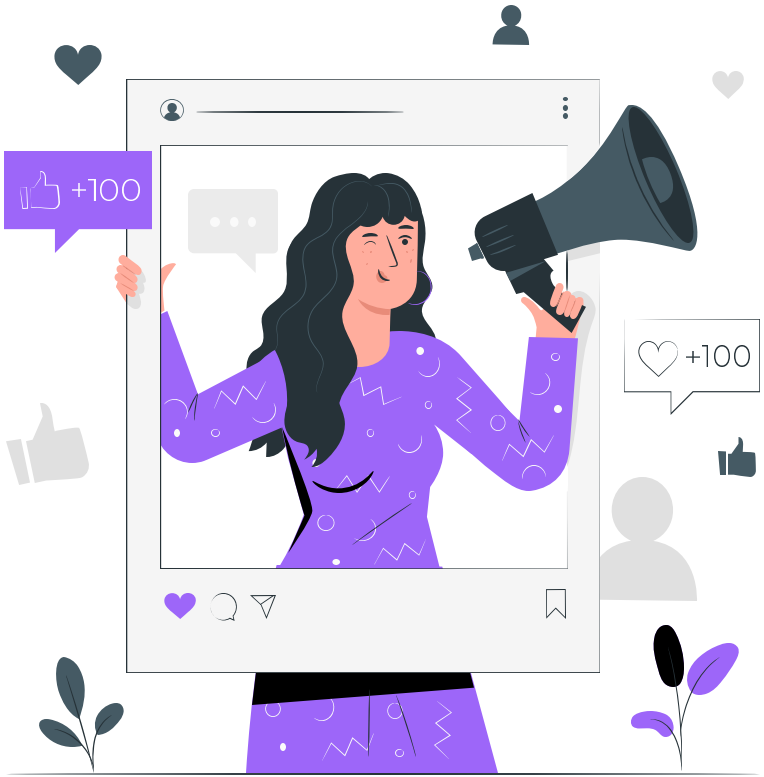
INFRACOMP

Infraestructura computacional

Competitiva

Regionalista ICPC 2024
Rango mundial < 65.000 en Leetcode (650+ problemas)

Experiencia



02

Revisión de problemas

¡Hagamos Upsolving!

ACM ICPC

Dados 6 números enteros, tu tarea es determinar si es posible dividirlos en dos grupos de tal manera que la suma de los números de cada grupo sea igual.

- Consideraciones

Resolver este problema con 3 ciclos anidados es sencillo, pero ineficiente. ¿Puedes resolverlo solo con dos ciclos anidados?

3	1	2	1	2	1

Triple Ciclo

Si la suma de los números es impar, es imposible dividirlos en dos grupos de igual suma. Esto se debe a que un número multiplicado por dos siempre será par.

De lo contrario, podemos explorar todas las combinaciones posibles utilizando tres bucles anidados (triple for). Esto nos permitirá verificar si existe una combinación de tres números cuya suma sea igual a la mitad de la suma total. Si encontramos tres números que cumplan esta condición, automáticamente los otros tres números también formarán un grupo con la misma suma.

```python
def solve(nums, target):
    for i in range(len(nums)):
        for j in range(i+1, len(nums)):
            for k in range(j+1, len(nums)):
                if (nums[i] + nums[j] + nums[k] == target):
                    return True
    return False

nums = [int(num) for num in input().split()]

sumOfNums = sum(nums)
if (sumOfNums % 2 == 1):
    print("No")
else:
    target = sumOfNums // 2
    if solve(nums, target):
        print("YES")
    else:
        print("NO")
```

Two Sum (con set)

Si la suma de los números es impar, es imposible dividirlos en dos grupos de igual suma. Esto se debe a que un número multiplicado por dos siempre será par.

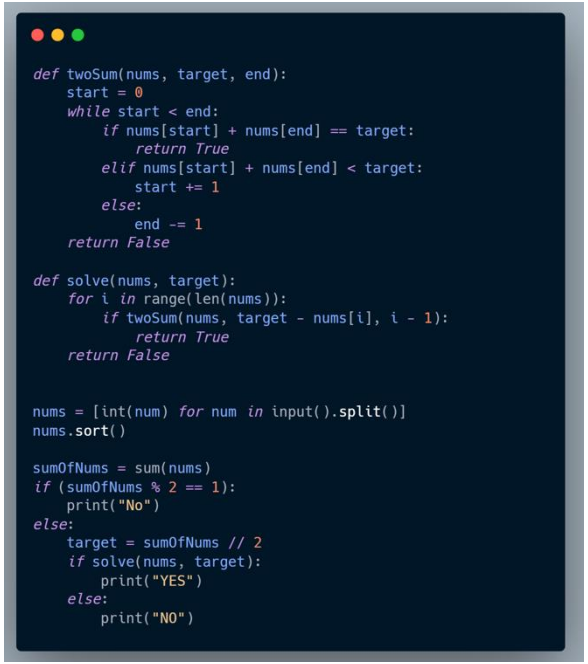
De lo contrario, utilizamos un bucle para fijar el i-ésimo número y luego buscamos los dos números restantes que, junto con el número fijado, sumen la mitad del total. Esto lo hacemos aplicando el algoritmo de Two Sum (visto en clase), que nos permite encontrar eficientemente dos números que sumen un valor específico.

```python
def twoSum(nums, target, end):
    visited = set()
    for i in range(end):
        if target - nums[i] in visited:
            return True
        visited.add(nums[i])
    return False

def solve(nums, target):
    for i in range(len(nums)):
        if twoSum(nums, target - nums[i], i):
            return True
    return False


nums = [int(num) for num in input().split()]

sumOfNums = sum(nums)
if (sumOfNums % 2 == 1):
    print("No")
else:
    target = sumOfNums // 2
    if solve(nums, target):
        print("YES")
    else:
        print("NO")
```

Two Sum (con Two Pointers)

Si la suma de los números es impar, es imposible dividirlos en dos grupos de igual suma. Esto se debe a que un número multiplicado por dos siempre será par.

De lo contrario, utilizamos un bucle para fijar el i-ésimo número y luego buscamos los dos números restantes que, junto con el número fijado, sumen la mitad del total. Esto lo hacemos aplicando el algoritmo de Two Sum, que nos permite encontrar eficientemente dos números que sumen un valor específico.



```python
def twoSum(nums, target, end):
    start = 0
    while start < end:
        if nums[start] + nums[end] == target:
            return True
        elif nums[start] + nums[end] < target:
            start += 1
        else:
            end -= 1
    return False

def solve(nums, target):
    for i in range(len(nums)):
        if twoSum(nums, target - nums[i], i - 1):
            return True
    return False


nums = [int(num) for num in input().split()]
nums.sort()

sumOfNums = sum(nums)
if (sumOfNums % 2 == 1):
    print("No")
else:
    target = sumOfNums // 2
    if solve(nums, target):
        print("YES")
    else:
        print("NO")
```

Comparación de Implementaciones

Fuerza bruta

Tiene una complejidad temporal de O(n^3) y una espacial de O(1). Es muy fácil de implementar (ideal si sabemos que el problema no tiene muchas restricciones de tiempo

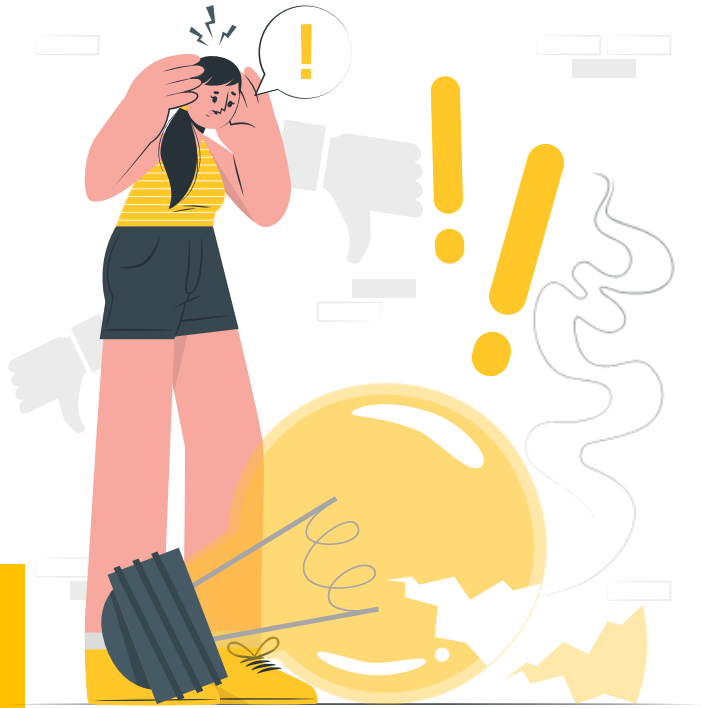
Set

Tiene una complejidad temporal de O(n^2) en promedio y de O(n^3) en el peor caso. Tiene complejidad espacial de O(n). Es poco recomendable usar Sets de hash en competencias debido a que suelen haber casos de prueba que ocasionen el peor caso

Two Pointers

Tiene una complejidad temporal de O(n^2) y complejidad espacial de O(n). Es más difícil de implementar pero es mucho más rápida, consume más memoria, pero esto no suele ser problema en competencias

Polygon



Se tiene una matriz n × n con cañones en el borde izquierdo y superior, que disparan proyectiles en línea recta hasta chocar con otro proyectil o un borde.
Dada una matriz de 0s y 1s (donde 1 indica un proyectil), se debe verificar si pudo haber sido generada por los disparos o si es inválida.

0	1	0
0	1	1
0	0	0

Recorrer la matriz

En lugar de analizar qué hace que una matriz sea válida, podemos ver qué la hace inválida. Dado que los cañones están ubicados a la izquierda y arriba de la matriz, un '1' en una posición específica será inválido si no tiene otro '1' a su derecha o abajo. Esto se debe a que un '1' no puede aparecer de manera aislada: no puede ser generado por un '1' arriba (ya que no hay cañones apuntando desde abajo) ni por un '1' a la izquierda (ya que no hay cañones apuntando desde la derecha).
De esta manera, el problema se reduce a verificar si existe al menos un punto inválido en la matriz. Si encontramos un '1' que no cumple con las condiciones mencionadas, la matriz completa se considera inválida.
Temporal: O(n*n) Espacial: O(1)

```
/**
    @author: Daniel Diaz
*/
#include <bits/stdc++.h>
using namespace std;


int main() {
    int t, n;
    vector<string> matriz;
    string s, respuesta;
    scanf("%d", &t);
    while (t--) {
        scanf("%d", &n);
        matriz.assign(n, "");
        for (int i = 0; i < n; ++i) {
            cin >> s;
            matriz[i] = s;
        }
        respuesta = "YES";
        for (int i = 0; i < n - 1; ++i) {
            if (respuesta == "NO") break;
            for (int j = 0; j < n - 1; ++j) {
                if (matriz[i][j] == '1' && matriz[i + 1][j] ==
'0' && matriz[i][j + 1] == '0') {
                    respuesta = "NO";
                    break;
                }
            }
        }
        printf("%s\n", respuesta.c_str());

}
```

Stalin Sort

El Stalin Sort es un algoritmo humorístico que ordena eliminando los elementos fuera de lugar en lugar de reordenarlos. Un arreglo se considera vulnerable si puede ordenarse en orden no creciente aplicando Stalin Sort repetidamente a sus subarreglos. Dado un arreglo de n enteros, el objetivo es determinar el mínimo número de eliminaciones necesarias para hacerlo vulnerable.

3	6	4	9	2	5	2

Doble ciclo

Debemos tener en cuenta que un arreglo con un solo elemento es vulnerable.
La solución propuesta consiste en explorar todos los posibles puntos de inicio para el arreglo vulnerable. Supongamos que el arreglo vulnerable comienza en la posición i. En este caso, debemos eliminar manualmente todos los elementos desde la posición 0 hasta i-1. Además, debemos eliminar todos los elementos desde i+1 hasta n que sean mayores que el valor del elemento en la posición i. Esto asegura que eliminamos tanto los elementos anteriores al inicio del arreglo vulnerable como aquellos posteriores que no cumplen con la condición de ser menores o iguales al elemento en la posición i.
Los elementos restantes, que no requieren una eliminación manual, pueden ser eliminados sin costo adicional utilizando el algoritmo conocido como Stalin Sort, que elimina todos los elementos que no están en orden creciente.

```cpp
/**
    @author: Daniel Diaz
*/
#include <bits/stdc++.h>
using namespace std;


int main() {
    int t, n, respuesta, actual;
    vector<int> numeros;
    scanf("%d", &t);
    while (t--) {
        scanf("%d", &n);
        respuesta = n - 1;
        numeros.assign(n, 0);
        for (int i = 0; i < n; ++i) {
            scanf("%d", &numeros[i]);
        }
        for (int i = 0; i < n; ++i) {
            actual = i;
            for (int j = i + 1; j < n; ++j) {
                if (numeros[j] > numeros[i]) {
                    actual += 1;
                }
            }
            respuesta = min(respuesta, actual);
        }
        printf("%d\n", respuesta);
    }
}
```

Perfect Standing

Una competencia de programación tiene 5 problemas (A, B, C, D y E), cada uno con un puntaje específico. Hay 31 combinaciones posibles de soluciones, desde resolver un solo problema hasta todos. El objetivo es listar todas las combinaciones ordenadas de mayor a menor puntaje según los valores dados para cada problema.

A	B	C	D	E
400	500	600	700	800

ABCDE	...	A

Generar todas las combinaciones (recursión)

Podemos utilizar recursión para generar todas las combinaciones posibles. El caso base de la recursión ocurre cuando ya hemos considerado todas las letras (problemas) para una combinación. En este punto, simplemente agregamos la combinación generada a la lista de resultados. En cada paso de la recursión, tenemos dos opciones:
- Incluir la letra actual en la combinación.
- Excluir la letra actual de la combinación.

Estas dos decisiones se traducen en dos llamadas recursivas, lo que nos permite explorar todas las combinaciones posibles de manera sistemática y eficiente.
De esta manera, podemos generar las 31 combinaciones posibles. Una vez generadas todas las combinaciones, solo nos queda ordenarlas e imprimir el resultado en el orden requerido.

Dado que siempre hay 31 combinaciones y cada una tiene como máximo 5 letras, la complejidad temporal y espacial del algoritmo es O(31 * 5) = O(1), lo que lo hace muy eficiente. Sin embargo, su desventaja es que puede ser muy largo y complejo de implementar.

```cpp
/**
    @author: Daniel Diaz
*/
#include <bits/stdc++.h>
using namespace std;


void generateCombinations(
    int index,
    int currentSum,
    string currentCombination,
    vector<int>& values,
    vector<string>& letters,
    vector<pair<int, string>>& combinations
  ) {
    if (index == 5) {
        if (!currentCombination.empty()) {
            combinations.push_back({-currentSum, currentCombination});
        }
        return;
    }
    generateCombinations(index + 1, currentSum + values[index], currentCombination + letters[index], values, letters, combinations);
    generateCombinations(index + 1, currentSum, currentCombination, values, letters, combinations);
}

int main() {
    int a, b, c, d, e;
    scanf("%d %d %d %d %d", &a, &b, &c, &d, &e);

    vector<int> values = {a, b, c, d, e};
    vector<string> letters = {"A", "B", "C", "D", "E"};
    vector<pair<int, string>> combinations;

    generateCombinations(0, 0, "", values, letters, combinations);

    sort(combinations.begin(), combinations.end());

    for (const auto& comb : combinations) {
        printf("%s\n", comb.second.c_str());
    }

    return 0;
}
```

Generar todas las combinaciones (Bitmask)

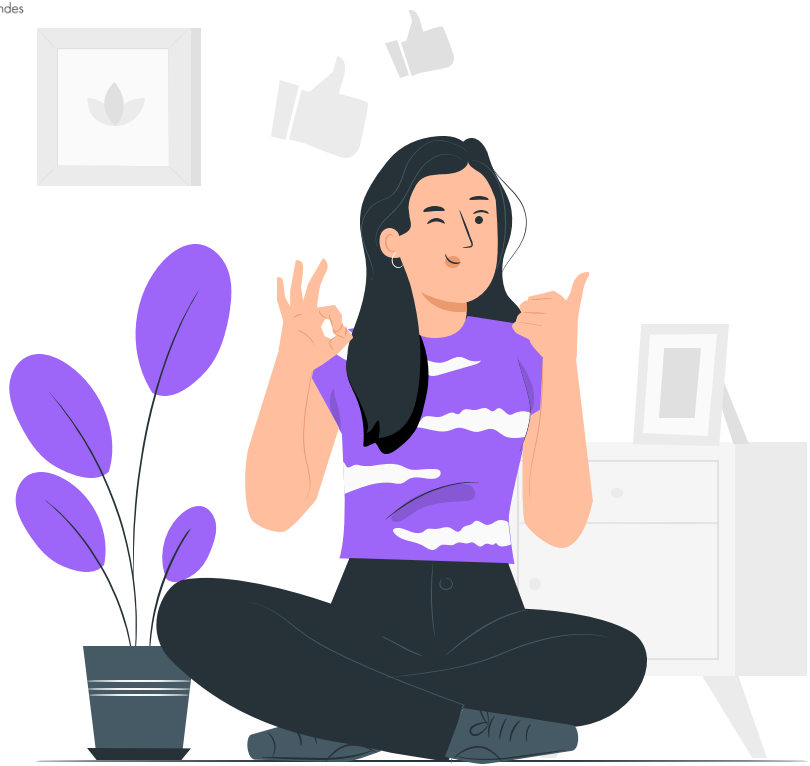
Podemos utilizar la representación en bits de los números del 1 al 31 (00001, 00010, 00011, ..., 11110, 11111). Cada uno de estos números representa una combinación única de los problemas A, B, C, D y E, donde cada bit activo (1) indica que el problema correspondiente está incluido en la combinación. Por ejemplo:
- 00001 representa solo el problema A.
- 01111 representa los problemas D, C, B y A.

De esta manera, podemos generar rápidamente las 31 combinaciones posibles. Una vez generadas todas las combinaciones, solo nos queda ordenarlas e imprimir el resultado en el orden requerido.

Tiene la misma complejidad que el enfoque anterior.

```cpp
/**
    @author: Daniel Diaz
*/
#include <bits/stdc++.h>
using namespace std;

int main(){
    int a,b,c,d,e, curr, idx, copyI;
    vector<pair<int, string>> v;
    string posibility;
    scanf("%d %d %d %d %d", &a, &b, &c, &d, &e);
    vector<int> values = {a,b,c,d,e};
    vector<string> valuesLetters = {"A", "B", "C", "D", "E"};
    for (int i = 1; i < 32; ++i) {
        posibility = "";
        curr = 0;
        copyI = i;
        idx = 0;
        while (copyI > 0) {
            if (copyI % 2 == 1) {
                posibility += valuesLetters[idx];
                curr += values[idx];
            }
            copyI /= 2;
            idx++;
        }
        v.push_back({-curr, posibility});

    }
    sort(v.begin(), v.end());
    for (int i = 0; i < 31; ++i) {
        printf("%s\n", v[i].second.c_str());
    }
}
```

Computer Society
Uniandes



¡Gracias!

Computer Society

Daniel Diaz