Ayudantía 1

Carlos Lagos - carlos.lagosc@usm.cl

Motivación

¿Que es programación competitiva?

La Programación Competitiva es un deporte mental, en el que los participantes se sumergen en la emoción de resolver problemas desafiantes con especificaciones concretas en un tiempo limitado.

¿De qué me sirve la programación competitiva?

La programación competitiva te ayuda a profundizar en temas avanzados de matemáticas y algoritmos, te brinda la habilidad de implementar soluciones complejas y entrena tu pensamiento lógico.

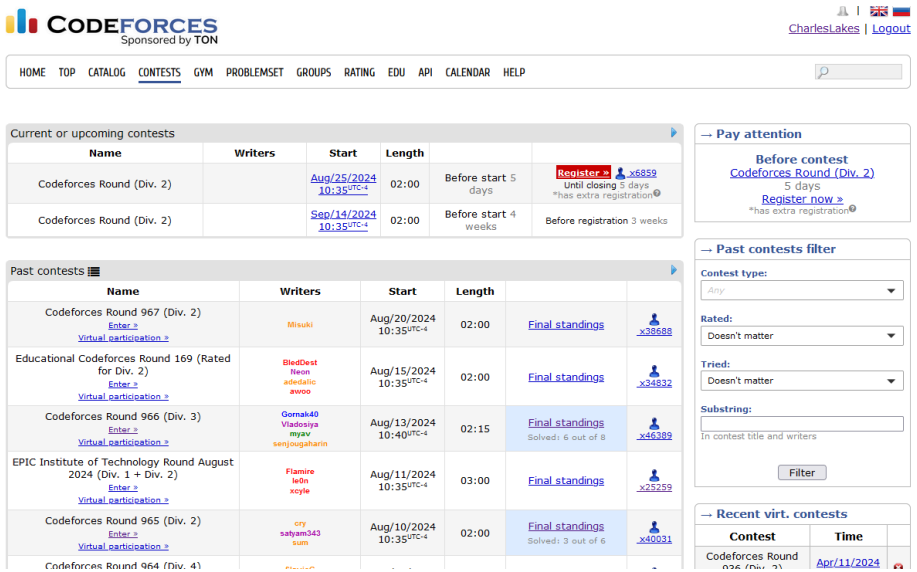
¿De qué me sirve en el futuro profesional?

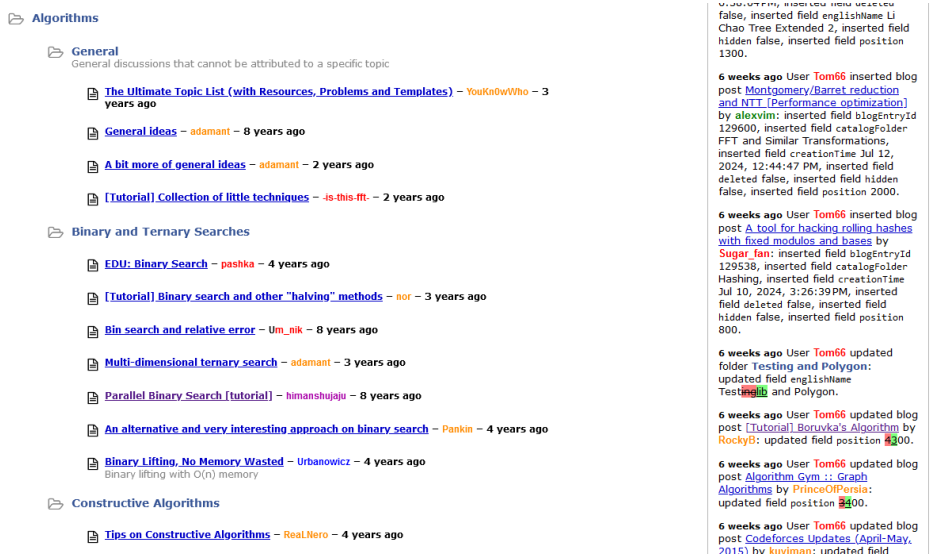
Muchas grandes empresas requieren superar pruebas técnicas, que a menudo incluyen la resolución e implementación de algoritmos.

Además, las empresas buscan personas capaces de crear e implementar soluciones a problemas complejos, por lo que esta habilidad es fundamental.

¿Dónde puedo aprender más sobre programación

La página más activa y con una amplia cantidad de problemas, recursos y tutoriales es Codeforces. En esta plataforma se realizan competencias con frecuencia, y la comunidad crea tutoriales para aprender temas avanzados.





¿Que competencias existen?

¿Que lenguajes de programación se usan?

Top

1. C++
2. Python
3. Java

¿Por qué C++?

- Es un lenguaje rápido en comparación con Python u otros de alto nivel.
- La biblioteca estándar de C++ (STL) ofrece una amplia gama de funcionalidades y estructuras de datos.
- Permite programar soluciones eficientes y versátiles gracias a la STL.
- En mi experiencia con programación competitiva, nunca he necesitado usar punteros.

C++ y STL

Estructura básica de un programa

```cpp
// Esta directiva incluye todas las bibliotecas estándar del lenguaje.
// Aunque no es óptima para desarrollar software complejo,
// en programación competitiva nos ahorra mucho tiempo.
#include <bits/stdc++.h>

using namespace std;

int main() {
    cout << "Hola mundo" << endl;

    return 0;
}
```

Estructuras de Datos - Vector

¿Cómo usar arreglos dinámicos sin punteros?

Podemos utilizar el contenedor `vector`, implementado en la STL de C++.

Se define de la siguiente manera:

```
vector<tipo_de_dato> nombre_de_variable;
```

Estructuras de Datos - Vector

```
vector<int> arr1; // Arreglo vacío de ints
vector<int> arr2(10); // Arreglo inicializado con 10 elementos
vector<int> arr3(8, -1); // Arreglo inicializado con 8 elementos, asignando -1 a todas las posiciones
vector<vector<string>> arr5; // Vector que almacena vectores de string

arr1.push_back(10); // Inserta 10 al final en O(1)
arr2.pop_back(); // Elimina el último elemento en O(1)
arr2[3] = 15; // Asigna 15 en la posición 3
arr2.clear(); // Elimina todos los elementos del vector
cout << arr3.size(); // Devuelve el tamaño del vector
```

Estructuras de Datos - Vector

```cpp
// Recorrer el vector usando un bucle for tradicional
for(int i = 0; i < arr1.size(); i++){
    cout << arr1[i] << endl;
}
// Recorrer el vector usando un foreach
for(int numero : arr1){
    cout << numero << endl;
}
```

Estructuras de Datos - Queue

Una `queue` (cola) es un contenedor que sigue el principio FIFO (First In, First Out).

Se define de la siguiente manera:

```
queue<tipo_de_dato> nombre_de_variable;
```

Estructuras de Datos - Queue

```
queue<int> cola;

cola.push(20); // Inserta el valor 20 al final de la cola
cout << cola.front() << endl; // Imprime el valor en el frente de la cola
cola.pop(); // Elimina el elemento al frente de la cola
cout << cola.size() << endl; // Devuelve la cantidad de elementos en la cola
cout << (cola.empty() ? "Cola vacía" : "Cola no vacía") << endl; // Verifica si la cola está vacía
```

Estructuras de Datos - Stack

Una `stack` (pila) es un contenedor que sigue el principio LIFO (Last In, First Out).

Se define de la siguiente manera:

```
stack<tipo_de_dato> nombre_de_variable;
```

Estructuras de Datos - Stack

```
stack<int> pila;

pila.push(30); // Inserta el valor 30 en la pila
cout << pila.top() << endl; // Imprime el valor en el tope de la pila
pila.pop(); // Elimina el elemento en el tope de la pila

cout << pila.size() << endl; // Devuelve la cantidad de elementos en la pila
cout << (pila.empty() ? "Pila vacía" : "Pila no vacía") << endl; // Verifica si la pila está vacía
```

Estructuras de Datos - Set

Un `set` es un contenedor que almacena elementos únicos en orden específico.

Se define de la siguiente manera:

```
set<tipo_de_dato> nombre_de_variable;
```

Estructuras de Datos - Set

```
set<int> conjunto; // Orden de menor a mayor
set<int,greater<int>> alreves; // Orden de menor a mayor

conjunto.insert(15); // Inserta el valor 15 en el set
conjunto.insert(20); // Inserta el valor 20 en el set
conjunto.insert(10); // Inserta el valor 10 en el set
cout << conjunto.size() << endl; // Devuelve la cantidad de elementos en el set
cout << (conjunto.empty() ? "Set vacío" : "Set no vacío") << endl; // Verifica si el set está vacío
// Imprimir todos los elementos del set
for (int elemento : conjunto) {
    cout << elemento << " "; // Imprime cada elemento
}
cout << endl;
// Eliminar un elemento
conjunto.erase(20); // Elimina el elemento 20 del set
```

Estructuras de Datos - Set

```cpp
// Uso de lower_bound
auto it_lower = conjunto.lower_bound(12); // Encuentra el primer elemento no menor que 12
if (it_lower != conjunto.end()) {
    cout << "El primer elemento no menor que 12 es: " << *it_lower << endl;
} else {
    cout << "No hay elementos no menores que 12 en el set" << endl;
}
// Uso de upper_bound
auto it_upper = conjunto.upper_bound(12); // Encuentra el primer elemento mayor que 12
if (it_upper != conjunto.end()) {
    cout << "El primer elemento mayor que 12 es: " << *it_upper << endl;
} else {
    cout << "No hay elementos mayores que 12 en el set" << endl;
}
```

Estructuras de Datos - Map

Un `map` es un contenedor que almacena pares clave-valor ordenados por las claves.

Se define de la siguiente manera:

```
map<tipo_de_clave, tipo_de_valor> nombre_de_variable;
```

Estructuras de Datos - Map

```cpp
map<string, int> mapa;

mapa["manzanas"] = 10; // Asigna el valor 10 a la clave "manzanas"
mapa["naranjas"] = 5; // Asigna el valor 5 a la clave "naranjas"

cout << mapa["manzanas"] << endl; // Imprime el valor asociado a la clave "manzanas"
mapa.erase("naranjas"); // Elimina el par clave-valor con la clave "naranjas"

cout << mapa.size() << endl; // Devuelve la cantidad de elementos en el map
cout << (mapa.empty() ? "Mapa vacío" : "Mapa no vacío") << endl; // Verifica si el map está vacío

// Iteración sobre el map usando un foreach
for (auto par : mapa) {
    cout << "Clave: " << par.first << ", Valor: " << par.second << endl; // Imprime cada clave y su valor asociado
}
```

Otros Tipos de Datos - Pair

Un `pair` es una estructura que puede contener dos valores de diferentes tipos. Se usa comúnmente para almacenar pares de datos relacionados, como coordenadas o claves y valores.

Definición y uso:

```
pair<tipo_de_dato1, tipo_de_dato2> nombre_de_variable;
pair<int, string> par;

// Inicializar un par
par = make_pair(1, "uno");

// Acceso a elementos
cout << par.first << endl; // Imprime el primer elemento (1)
cout << par.second << endl; // Imprime el segundo elemento ("uno")
```

Otros Tipos de Datos - Tuple

Una `tuple` es similar a un `pair`, pero puede contener múltiples valores de diferentes tipos. Esto es útil cuando necesitas agrupar más de dos valores.

Definición y uso:

```
#include <tuple>
tuple<tipo_de_dato1, tipo_de_dato2, tipo_de_dato3> nombre_de_variable;
tuple<int, string, double> tupla;

// Inicializar una tupla
tupla = make_tuple(1, "uno", 3.14);

// Acceso a elementos
cout << get<0>(tupla) << endl; // Imprime el primer elemento (1)
cout << get<1>(tupla) << endl; // Imprime el segundo elemento ("uno")
cout << get<2>(tupla) << endl; // Imprime el tercer elemento (3.14)
```

Algoritmos - Sort

La función `sort` es utilizada para ordenar elementos en un contenedor. Esta función es parte de la librería `<algorithm>` y proporciona ordenamientos rápidos y eficientes $O(nlog(n))$.

Algoritmos - Sort

Ordenar en Orden Ascendente

```
vector<int> numeros = {5, 2, 9, 1, 5, 6};

// Ordenar el vector en orden ascendente
sort(numeros.begin(), numeros.end());

// Imprimir el vector ordenado
cout << "Vector ordenado: ";
for (int num : numeros) {
    cout << num << " ";
}
cout << endl;
```

Algoritmos - Sort

Ordenar en Orden Descendente

Para ordenar en orden descendente, puedes utilizar la función `greater<int>()` como tercer argumento:

```
sort(numeros.begin(), numeros.end(), greater<int>());

cout << "Vector ordenado en orden descendente: ";
for (int num : numeros) {
    cout << num << " ";
}
cout << endl;
```

Algoritmos - Sort

Ordenar con una Función de Comparación Personalizada

Puedes definir una función de comparación personalizada para ordenar elementos de acuerdo con un criterio específico. A continuación, se muestra un ejemplo de cómo ordenar un vector de pares de enteros según el segundo elemento de cada par:

Algoritmos - Sort

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

bool compararPorSegundo(pair<int, int> a, pair<int, int> b) {
    return a.second < b.second;
}

int main() {
    vector<pair<int, int>> pares = {{1, 4}, {2, 2}, {3, 3}};
    sort(pares.begin(), pares.end(), compararPorSegundo);
    cout << "Vector de pares ordenado por el segundo elemento: ";
    for (pair<int,int> par : pares) {
        cout << "(" << par.first << ", " << par.second << ") ";
    }
    cout << endl;
    return 0;
}
```

Algoritmos - Reverse

La función `reverse` invierte el orden de los elementos en un contenedor.

```
vector<int> numeros = {1, 2, 3, 4, 5};

// Invertir el vector
reverse(numeros.begin(), numeros.end());

// Imprimir el vector invertido
cout << "Vector invertido: ";
for (int num : numeros) {
    cout << num << " ";
}
cout << endl;
```

Algoritmos - Lower Bound y Upper Bound

Las funciones `lower_bound` y `upper_bound` se utilizan para buscar elementos en contenedores ordenados. `lower_bound` devuelve un iterador al primer elemento que no es menor que el valor especificado, mientras que `upper_bound` devuelve un iterador al primer elemento mayor que el valor especificado.

Algoritmos - Lower Bound y Upper Bound

```
vector<int> numeros = {1, 2, 4, 4, 5, 6, 7};

auto it_lower = lower_bound(numeros.begin(), numeros.end(), 4);
auto it_upper = upper_bound(numeros.begin(), numeros.end(), 4);

cout << "Lower bound de 4: " << distance(numeros.begin(), it_lower) << endl; // Índice del primer 4
cout << "Upper bound de 4: " << distance(numeros.begin(), it_upper) << endl; // Índice del primer elemento mayor que 4
```

Algoritmos - Next Permutation

La función `next_permutation` transforma el contenedor en la siguiente permutación lexicográficamente mayor. Si el contenedor está en su última permutación posible, lo transforma a la primera permutación.

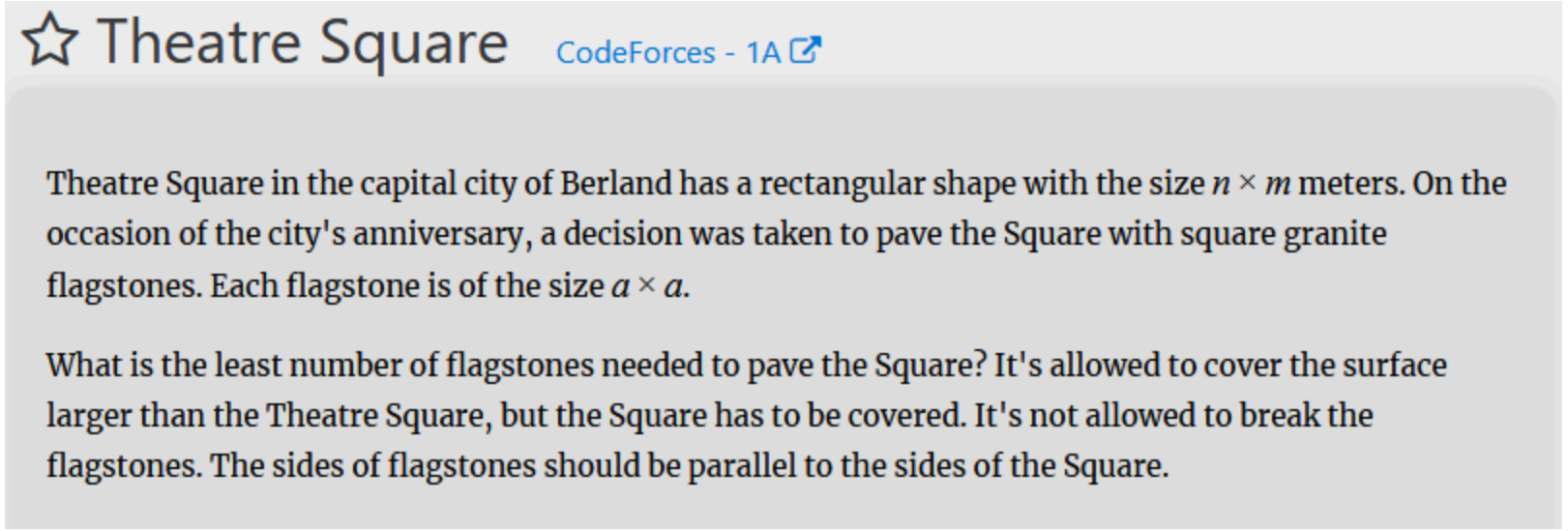
```
vector<int> numeros = {1, 2, 3};

do {
    // Imprimir la permutación actual
    for (int num : numeros) {
        cout << num << " ";
    }
    cout << endl;
} while (next_permutation(numeros.begin(), numeros.end()));
```

¿Cómo abordar un problema de programación competitiva?

Partes de un Problema

Enunciado

Proporciona una descripción detallada del problema a resolver.



☆ Theatre Square CodeForces - 1A

Theatre Square in the capital city of Berland has a rectangular shape with the size $n \times m$ meters. On the occasion of the city's anniversary, a decision was taken to pave the Square with square granite flagstones. Each flagstone is of the size $a \times a$.

What is the least number of flagstones needed to pave the Square? It's allowed to cover the surface larger than the Theatre Square, but the Square has to be covered. It's not allowed to break the flagstones. The sides of flagstones should be parallel to the sides of the Square.

Partes de un Problema

Entrada

Define el formato y las restricciones de la entrada. Puedes asumir que las restricciones siempre se cumplirán. La entrada debe ser leída desde la entrada estándar (por consola, usando `cin` o `input()`).

Input

The input contains three positive integer numbers in the first line: n, m and a ($1 \leq n, m, a \leq 10^9$).

Partes de un Problema

Salida

Especifica el formato de la salida que debe devolver el programa. La salida debe ser mostrada en la salida estándar (por consola, usando `cout` o `print()`).

Output

Write the needed number of flagstones.

Partes de un Problema

Ejemplos

Se proporcionan casos de prueba públicos para verificar tu código. Sin embargo, al enviar tu solución, el juez evaluará tu código también con casos de prueba ocultos.

Examples

Input	copy	Output	copy
6 6 4		4	

Partes de un Problema

Tiempo Límite y Límite de Memoria

Cada problema tiene un tiempo límite y un límite de memoria. Si la solución excede estas restricciones, se rechazará con un error de TLE (Time Limit Exceeded) o MLE (Memory Limit Exceeded). Aproximadamente, 10^8 operaciones toman un segundo.

Cómo abordar un ejercicio correctamente

- Comienza leyendo la sección de entrada y salida del problema.
- Luego, lee el enunciado completo (con el contexto de la entrada y salida, podrás identificar información irrelevante más rápidamente).
- Haz observaciones y deduce propiedades clave del problema.
- Diseña un algoritmo. Si realiza menos de 10^8 operaciones, procede a programarlo; si no, regresa al paso 3.

¿Cuándo pedir ayuda?

- Si has dedicado suficiente tiempo a pensar en la solución (más de 60 minutos).
  - Si la respuesta es sí, quizás sea un buen momento para pedir ayuda.
- Si no logras resolver el ejercicio completo, pero has encontrado propiedades y observaciones interesantes.
  - Sigue intentándolo por unos 30 minutos más; si no progresas, considera pedir ayuda.

FIN