

Estructuras de Datos I

Jose Ignacio Jaldin Janko

Taller de programación competitiva 2025
Universidad Mayor de San Simón

May 3, 2025

¿Qué es la STL y por qué usarla?

Es una colección de estructuras de datos y algoritmos listos para usar en C++

¿Qué es la STL y por qué usarla?

Es una colección de estructuras de datos y algoritmos listos para usar en C++

¿Cómo es que funciona por dentro?

Introducción a la STL

¿Qué es la STL y por qué usarla?

Es una colección de estructuras de datos y algoritmos listos para usar en C++

¿Cómo es que funciona por dentro?



El concepto de la caja negra



El concepto de la caja negra



No nos interesa el “CÓMO” lo hace, si no el “QUÉ” hace.

Beneficios:

- Eficiencia
- Ahorro de tiempo
- Seguridad

(Comparado con estructuras manuales).

La STL se puede “dividir” en 3 grandes grupos:

La STL se puede “dividir” en 3 grandes grupos:

- Contenedores

Son estructuras de datos ya implementadas que almacenan objetos de forma organizada.

Ejemplos: `vector`, `deque`, `stack`, `set`, `map`, etc.

No tienes que implementar desde cero un árbol, una lista enlazada o un heap: ya existen contenedores en la STL.

La STL se puede “dividir” en 3 grandes grupos:

- Algoritmos

Son funciones genéricas que operan sobre datos, como ordenar, buscar, modificar, etc.

Ejemplos: `sort`, `binary_search`, `lower_bound`, `max_element`, `reverse`, `next_permutation`, etc.

La STL ya trae algoritmos súper optimizados, que trabajan sobre contenedores o rango de iteradores.

La STL se puede “dividir” en 3 grandes grupos:

- **Funciones** Son herramientas adicionales que no son exactamente algoritmos o contenedores, pero ayudan a trabajar con ellos.

Ejemplos:

`accumulate`, `iota`, `fill`, `swap`, etc.

Simplifican operaciones comunes que de otra forma tendrías que programar a mano.

Notación general: Antes de poder utilizar toda la STL, tenemos que “importar” cada utilidad:

- `#include<vector>`
- `#include<stack>`
- `#include<set>`
- `#include<algorithm>`

Y podemos seguir así...

Notación general:

O como en muchos lenguajes podemos incluir TODA la STL en una sola línea:

```
#include<bits/stdc++.h>
```

Notación general:

O como en muchos lenguajes podemos incluir TODA la STL en una sola línea:

```
#include<bits/stdc++.h>
```

Notación general:

Y cada vez que queramos usar alguna utilidad debemos usar el prefijo `std::` en cada estructura o función.

Por ejemplo:

```
std::vector<int> vectorsito;  
std::sort(vectorsito.begin(), vectorsito.end());
```

Y así ...

Notación general:

O, podemos decirle al compilador que queremos usar el espacio de nombres (namespace) `std` de forma implícita, es decir, sin tener que escribir `std::` delante de cada cosa que pertenece a la biblioteca estándar.

La sintaxis es:

```
#include<bits/stdc++.h>
```

```
using namespace std;
```

Tu código ...

Problema motivacional — vector

Dado n números (notas entre 0 y 100), calcula el promedio y mostrar en la primera línea dicho promedio entero. En la segunda línea, imprimir todas las notas menores al promedio.

Entrada:

Un entero $1 \leq n \leq 10^6$, seguido de n enteros (las notas).

Salida:

Línea 1: El promedio (con decimales).

Línea 2: Las notas estrictamente menores al promedio.

Ejemplo — vector

Entrada:

- 5
- 60 70 90 80 100

Salida:

- 80
- 60 70

Nota: El promedio es 80. Las únicas notas menores a él son 60 y 70.

¿Qué son?

Son estructuras que almacenan elementos en un orden específico (normalmente el orden de inserción).

```
vector<T> vectorsito;
```

Donde T es el tipo de dato que quieres almacenar en ese vector

Por ejemplo, un vector de enteros, o un vector de cadenas:

```
vector<int> numeros;
```

```
vector<string> palabras;
```

El equivalente al `ArrayList<T>` en java

```
vector<int> vectorsito;
```

push_back(elemento)

- Inserta el “elemento” al final de la estructura.
- Costo: $O(1)$ (amortizado)

```
vector<int> vectorsito;
```

pop_back()

- Elimina el último elemento del vector.
- Costo: $O(1)$

Ojo, el vector debe tener al menos un elemento para usar este método, porque si no, nos salta error en tiempo de ejecución.

```
vector<int> vectorsito;
```

back()

- Devuelve el último elemento.
- Costo: $O(1)$

Ojo, el vector debe tener al menos un elemento para usar este método, porque si no, nos salta error en tiempo de ejecución.

```
vector<int> vectorsito;
```

Acceder a una posición $[i]$

- Acceder la posición i en el vector, exactamente igual como en los arreglos.
- Sirve tanto para acceder para recibir el valor que se encuentra en esa posición o para asignar algún valor en esa posición
- Costo: $O(1)$

Ojo, la posición i debe ser válida en el vector, de lo contrario nos salta un error en tiempo de ejecución

Métodos de vector

```
vector<int> vectorsito;
```

size()

- Nos da el tamaño del vector, este método es común para todas (o casi) las estructuras de datos.
- Nos devuelve un tipo de dato `size_t` NO ES UN ENTERO.
- Costo: $O(1)$

Ojo, siempre conviene castear el `size()` de cualquier estructura de datos, porque al no ser un entero a veces los programas suelen fallar.

Por ejemplo:

```
(int)vectorsito.size()
```

O mejor el macro:

```
#define sz(x) (int)(x).size()
```

Y en su código escriben únicamente `sz(vectorsito)`

Al momento de crear un vector podemos usar sus distintos “constructores”:

- `vector<int> numeros(10);`
- `vector<int> numeritos(7, 5);`
- `vector<vector<int>> numeros(5, vector<int>(7));`

Problema motivacional — vector

Dado n números (notas entre 0 y 100), calcula el promedio y mostrar en la primera línea dicho promedio entero. En la segunda línea, imprimir todas las notas menores al promedio.

Entrada:

Un entero $1 \leq n \leq 10^6$, seguido de n enteros (las notas).

Salida:

Línea 1: El promedio (con decimales).

Línea 2: Las notas estrictamente menores al promedio.

Ejemplo — vector

Entrada:

- 5
- 60 70 90 80 100

Salida:

- 80
- 60 70

Nota: El promedio es 80. Las únicas notas menores a él son 60 y 70.

Arreglos en C++

Son similares a los arreglos en java, y son similares a los vectores en acceder a elementos.

Sintaxis:

```
tipo nombre[tamaño];
```

Ejemplo:

```
int arr[50];
```

```
int arr[50];
```

A diferencia del vector, tenemos que limpiar explícitamente los valores de un arreglo, porque estos pueden contener valores que desconocemos.

```
memset(arr, 0, sizeof(arr));
```

Cuidado que `memset` utiliza operaciones binarias, solo sirve para llenar todos los elementos a $\{0, -1\}$ y nada más.

El mismo ejercicio pero con arreglos

Arreglo vs Vector

- **Arreglo:**

- Tiene un tamaño fijo, es decir, debe definirse en el momento de la declaración.
- La sintaxis es sencilla: `tipo nombre[tamaño];`
- Los elementos de un arreglo pueden ser manipulados directamente con el operador `[i]`.
- No se ajusta dinámicamente: no podemos cambiar su tamaño después de la creación.
- No ofrece métodos para manipular sus elementos como `push_back()` o `pop_back()`.
- Los arreglos pueden contener valores indeterminados si no se inicializan, por lo que es necesario limpiarlos explícitamente con `memset`.

- **Vector:**

- Es un contenedor dinámico que puede redimensionarse automáticamente.
- Permite agregar elementos al final con `push_back()` y eliminarlos con `pop_back()`.
- Tiene métodos como `size()` para obtener su tamaño y `back()` para acceder al último elemento.
- Acceso a los elementos de la misma manera que los arreglos con `[i]`, pero con mayor flexibilidad y seguridad.
- No es necesario limpiar manualmente los elementos, ya que el vector maneja su memoria internamente.

Problema motivacional — deque

Estás procesando una cola de atención en una oficina. Llegan personas con prioridad normal o urgente.

Las personas normales se agregan al final de la fila, mientras que las personas urgentes se agregan al inicio.

Entrada:

- Un entero n ($1 \leq n \leq 10^6$), el número de operaciones.
- n operaciones, donde cada una es:
 - normal x : agregar la persona con ID x al final de la fila. ($1 \leq x \leq 10^6$)
 - urgente x : agregar la persona con ID x al inicio de la fila. ($1 \leq x \leq 10^6$)

Salida:

- Imprimir los IDs de las personas en el orden en que serán atendidas, desde la primera en ser atendida hasta la última.

Entrada:

- 5
- normal 1
- urgente 2
- urgente 3
- normal 4
- urgente 5

Salida:

- 5 3 2 1 4

¿Qué es deque?

- deque (Double-Ended Queue) es un contenedor de la STL en C++.
- Permite agregar y eliminar elementos de manera eficiente tanto al principio como al final.
- Ideal para escenarios donde necesitas operaciones rápidas en ambos extremos.

Características principales de deque

- Acceso aleatorio ($O(1)$ para cualquier índice).
- Inserción y eliminación eficiente en ambos extremos ($O(1)$).
- Flexible y dinámico, con ajuste de tamaño automático.

Sintaxis básica y operaciones

- Declaración:
 - `deque<int> dq;`
- Operaciones:
 - `push_back()` - agregar al final $O(1)$
 - `push_front()` - agregar al principio $O(1)$
 - `pop_back()` - eliminar del final $O(1)$
 - `pop_front()` - eliminar del principio $O(1)$

Comparación con vector

- deque es más eficiente que vector cuando se insertan o eliminan elementos en el principio.
- En vector, las inserciones en el principio requieren mover elementos ($O(n)$).
- deque permite inserciones en ambos extremos en $O(1)$.

Problema motivacional — deque

Estás procesando una cola de atención en una oficina. Llegan personas con prioridad normal o urgente.

Las personas normales se agregan al final de la fila, mientras que las personas urgentes se agregan al inicio.

Entrada:

- Un entero n ($1 \leq n \leq 10^6$), el número de operaciones.
- n operaciones, donde cada una es:
 - normal x : agregar la persona con ID x al final de la fila. ($1 \leq x \leq 10^6$)
 - urgente x : agregar la persona con ID x al inicio de la fila. ($1 \leq x \leq 10^6$)

Salida:

- Imprimir los IDs de las personas en el orden en que serán atendidas, desde la primera en ser atendida hasta la última.

Entrada:

- 5
- normal 1
- urgente 2
- urgente 3
- normal 4
- urgente 5

Salida:

- 5 3 2 1 4

¿Qué son stack y queue?

- Contenedores adaptadores de la STL.
- No son estructuras en sí, sino envoltorios sobre otros contenedores (por defecto, deque).
- **stack**: modelo LIFO (Last In, First Out).
- **queue**: modelo FIFO (First In, First Out).

Entrada:

- `((()()))`

Salida:

- Es válida

Entrada:

- `((()())`

Salida:

- Es inválida

Uso básico de stack

- `push(x)`: inserta `x` al tope.
- `pop()`: elimina el elemento en el tope.
- `top()`: accede al elemento del tope.
- `empty()`, `size()`.

Dada una secuencia de paréntesis, verifica si es válida.

Una secuencia es válida si:

- Cada paréntesis que abre tiene uno que cierra.
- Los paréntesis están bien anidados.

Entrada:

- Una cadena s de longitud n ($1 \leq n \leq 10^6$) compuesta solo por los caracteres '(' y ')'.
')'.

Salida:

- Imprime "Es valida" si la secuencia es válida, o "No es valida" en caso contrario.

Entrada:

- `((()())`

Salida:

- Es válida

Entrada:

- `((()()`

Salida:

- Es inválida

Uso básico de queue

- `push(x)`: inserta `x` al final.
- `pop()`: elimina el elemento al frente.
- `front()`: accede al primero en entrar.
- `empty()`, `size()`.

Complejidad de operaciones

- Todas las operaciones de `stack` y `queue` tienen complejidad **$O(1)$** .
- Internamente usan deque, lo que permite eficiencia en ambos extremos.
- No permiten iteradores ni acceso aleatorio.

- Todas las operaciones de `stack` y `queue` tienen complejidad **$O(1)$** .
- Internamente usan deque, lo que permite eficiencia en ambos extremos.
- No permiten iteradores ni acceso aleatorio.

¿Cuándo usar?

- `stack`: recorridos DFS, matching de paréntesis, etc.
- `queue`: BFS, simulaciones, procesos en orden de llegada.

Resumen: stack vs queue

Operación	stack	queue
Insertar	push() al tope	push() al final
Eliminar	pop() del tope	pop() del frente
Acceso	top()	front()
Orden	Último entra, primero sale	Primero entra, primero sale

Contenedores asociativos

(Basados en estructuras como árboles)

Problema motivacional — set

Dada una secuencia de n números enteros, imprimí "SI" si el número es la primera vez que aparece, o "NO" si ya apareció antes.

Entrada:

- Un entero n ($1 \leq n \leq 10^6$).
- Una secuencia de n enteros a_1, a_2, \dots, a_n con $1 \leq a_i \leq 10^9$.

Salida:

- Por cada número de la secuencia, imprimí SI si es la primera vez que aparece, o NO si ya apareció antes.

Entrada:

- 6
- 5 1 2 1 2 3

Salida:

- SI
- SI
- SI
- NO
- NO
- SI

- Un set es un contenedor de la STL que almacena elementos únicos y ordenados automáticamente.
- Internamente está implementado como un **árbol balanceado** (Red-Black Tree).
- Cada operación básica (insertar, borrar, buscar, contar) tiene complejidad $\mathcal{O}(\log n)$.
- **No permite elementos duplicados.**

Operaciones básicas en set

- `insert(x)`: Inserta el elemento x si no existe.
- `erase(x)`: Borra el elemento x si está presente.
- `count(x)`: Devuelve 1 si x está en el set, 0 si no.
- `find(x)`: Devuelve un **iterador** al elemento x o `end()` si no está.
- `empty()`, `size()` también están disponibles.

- set mantiene sus elementos en orden creciente (por defecto con `operator<`).
- Puedes recorrerlo con iteradores o con range-based for loops.
- También puedes usar `auto it = s.lower_bound(x);` para obtener el primer elemento $\geq x$.

Nota: No puedes modificar directamente un valor a través del iterador (porque el orden debe mantenerse).

¿Cuándo usar un set?

- Cuando necesitas mantener un conjunto de elementos **únicos y ordenados**.
- Cuando necesitas inserciones, búsquedas y borrados en **tiempo logarítmico**.
- Muy útil para:
 - Encontrar rápidamente el menor o mayor elemento mayor/menor que cierto valor.
 - Simular estructuras como árboles o conjuntos dinámicos ordenados.

Problema motivacional — set

Dada una secuencia de n números enteros, imprimí "SI" si el número es la primera vez que aparece, o "NO" si ya apareció antes.

Entrada:

- Un entero n ($1 \leq n \leq 10^6$).
- Una secuencia de n enteros a_1, a_2, \dots, a_n con $1 \leq a_i \leq 10^9$.

Salida:

- Por cada número de la secuencia, imprimí SI si es la primera vez que aparece, o NO si ya apareció antes.

Entrada:

- 6
- 5 1 2 1 2 3

Salida:

- SI
- SI
- SI
- NO
- NO
- SI

Problema motivacional — multiset

Estás desarrollando un sistema de control de precios para una tienda. Cada vez que un producto se vende o se agrega nuevo stock, se actualiza el precio más bajo disponible.

Entrada:

- Un entero q ($1 \leq q \leq 10^5$), el número de operaciones.
- Luego q operaciones de dos tipos:
 - `add x` — se agrega un producto con precio x ($1 \leq x \leq 10^9$).
 - `sell` — se vende el producto con menor precio.

Salida:

- Por cada operación `sell`, imprimir el precio del producto vendido.

Ejemplo — multiset

Entrada:

- 7
- add 10
- add 5
- add 10
- sell
- sell
- add 3
- sell

Salida:

- 5
- 10
- 3

- Contenedor ordenado que permite almacenar elementos duplicados.
- Implementado como un árbol balanceado (Red-Black Tree).
- Elementos siempre ordenados (como en `set`).
- Las operaciones básicas (insertar, borrar, buscar) tienen complejidad $\mathcal{O}(\log n)$.

- `insert(x)`: Inserta el elemento x .
Complejidad: $\mathcal{O}(\log n)$
- `erase(x)`: Borra **todas** las apariciones de x .
Complejidad: $\mathcal{O}(\log n + k)$, donde k es la cantidad de elementos borrados
- `erase(it)`: Borra solo el elemento apuntado por el iterador.
Complejidad: $\mathcal{O}(1)$
- `count(x)`: Devuelve cuántas veces aparece x .
Complejidad: $\mathcal{O}(\log n)$
- `find(x)`: Devuelve un iterador a **una** ocurrencia de x .
Complejidad: $\mathcal{O}(\log n)$

Orden y duplicados en multiset

- Los elementos están siempre ordenados (por defecto de menor a mayor).
- Permite múltiples copias de un mismo valor.
- `lower_bound(x)` y `upper_bound(x)` permiten recorrer un rango de valores iguales.
- Útil para manejar estructuras con frecuencias donde el orden importa.

Problema motivacional — multiset

Estás desarrollando un sistema de control de precios para una tienda. Cada vez que un producto se vende o se agrega nuevo stock, se actualiza el precio más bajo disponible.

Entrada:

- Un entero q ($1 \leq q \leq 10^5$), el número de operaciones.
- Luego q operaciones de dos tipos:
 - `add x` — se agrega un producto con precio x ($1 \leq x \leq 10^9$).
 - `sell` — se vende el producto con menor precio.

Salida:

- Por cada operación `sell`, imprimir el precio del producto vendido.

Ejemplo — multiset

Entrada:

- 7
- add 10
- add 5
- add 10
- sell
- sell
- add 3
- sell


Salida:

- 5
- 10
- 3

Antes de ver el map, que es un pair?

- Es una estructura de la STL que almacena dos valores (posiblemente de distintos tipos).
- Se accede a sus elementos con `.first` y `.second`.
- Muy usado en contenedores como `map`, `set`, o en retorno de funciones.

Ejemplo:



```
pair<string, int> p = {"Juan", 42};  
cout << p.first << " " << p.second << "\n";
```

Estás procesando un texto y necesitas contar cuántas veces aparece cada palabra.

Entrada:

- Un entero n ($1 \leq n \leq 10^5$), el número de palabras.
- Luego n palabras, compuestas solo de letras minúsculas, sin espacios.

Salida:

- Para cada palabra distinta que aparece, imprimir una línea con la palabra seguida del número de veces que apareció.
- Las palabras deben imprimirse en orden alfabético.

Ejemplo

Entrada:

- 7
- apple banana apple orange banana apple grape

Salida:

- apple 3
- banana 2
- grape 1
- orange 1

Qué es map?

- Contenedor asociativo que almacena pares clave-valor ordenados por clave.
- Cada clave es única.
- Se implementa con árboles balanceados (normalmente Red-Black Tree).
- Todas las operaciones principales tienen complejidad:
- Se accede con: `map[key]`

map como arreglo con índices infinitos

- Un map se puede ver como un arreglo donde:
 - Los índices (claves) pueden ser cualquier tipo ordenable: `int`, `string`, `pair`, etc.
 - No necesita reservar espacio para índices no usados.
 - Cada clave puede aparecer una sola vez.
- Ideal cuando los índices posibles son grandes o arbitrarios.

Operaciones básicas en map

- `m[key] = value`: Inserta o actualiza.
- `m.insert({key, value})`: Inserta si la clave no existe.
- `m.erase(key)`: Elimina la clave si existe.
- `m.find(key)`: Devuelve un iterador al par clave-valor.

¿Cuándo usar map?

- Contador de frecuencias con claves ordenadas.
- Diccionarios de búsqueda rápida por clave.
- Simulación de arreglos dispersos: índices grandes con pocos datos.
- Búsquedas por piso/techo usando `lower_bound` y `upper_bound`.
- Estructuras auxiliares para programación dinámica con estados ordenados.

Estás procesando un texto y necesitas contar cuántas veces aparece cada palabra.

Entrada:

- Un entero n ($1 \leq n \leq 10^5$), el número de palabras.
- Luego n palabras, compuestas solo de letras minúsculas, sin espacios.

Salida:

- Para cada palabra distinta que aparece, imprimir una línea con la palabra seguida del número de veces que apareció.
- Las palabras deben imprimirse en orden alfabético.

Ejemplo

Entrada:

- 7
- apple banana apple orange banana apple grape

Salida:

- apple 3
- banana 2
- grape 1
- orange 1

`unordered_map` y `unordered_set`

- Tienen todas las operaciones de map y set, pero la complejidad de sus operaciones es $O(1)$
- A diferencia de set y map, sus claves no son ordenadas, es decir los elementos no tienen un orden específico, es “aleatorio”
- Por debajo están implementados sobre tablas hash, por eso el
- No vale la pena usarlos, siempre será mejor un map o un set.

priority_queue en STL

- Contenedor adaptador que funciona como una cola con prioridades.
- Por defecto, devuelve el **mayor** elemento primero (orden descendente).
- Internamente implementada como un **heap binario**.
- Soporta:
 - `push(x)`: inserta un elemento. Costo: $O(\log(n))$
 - `pop()`: elimina el de mayor prioridad. Costo: $O(\log(n))$
 - `top()`: accede al de mayor prioridad. Costo: $O(\log(n))$
 - `empty()`, `size()`.
- Complejidad de `push` y `pop`: $O(\log n)$

priority_queue de mínimos

- Por defecto, es de máximos. Para que sea de mínimos se usa:

```
priority_queue<int, vector<int>, greater<int>> pq;
```

- Cambia la comparación interna a `greater<T>`.
- Ahora el menor elemento será el primero en salir.

Y en si, para cualquier tipo de dato:

```
priority_queue<T, vector<T>, greater<T>> pq;
```

- Un iterador es un “puntero inteligente” que permite recorrer contenedores de la STL.
- Funciona parecido a un puntero: puedes avanzar (++), retroceder (--) y acceder al valor con *.
- Existen varios tipos según el contenedor (aleatorio, bidireccional, etc).

Recorrer contenedores con iteradores

- Todos los contenedores tienen métodos:
 - `begin()` → primer elemento.
 - `end()` → posición después del último.
- Se puede recorrer con bucles tradicionales o con `auto`.
- También se pueden modificar valores:

Ejemplo

Util para recorrer sets, maps y multimaps;

Búsquedas con `find` y uso del iterador

- En `set`, `map`, `multiset`, `find(x)` devuelve un iterador a `x` si es que existe.
- Si no existe, devuelve `end()`.

Tipos especiales de iteradores

- `const_iterator`: No permite modificar el valor apuntado. (sets y maps)
- `reverse_iterator`: Recorre en orden inverso (`rbegin()`, `rend()`).
- `auto`: Recomendado para no escribir tipos largos.
- **Tip**: Siempre verificar si el iterador es `!= container.end()` antes de usarlo.

- La STL ofrece una colección de algoritmos genéricos que trabajan con iteradores.
- Funcionan con cualquier contenedor que tenga `begin()` y `end()`.
- Incluyen ordenamientos, búsquedas, transformaciones, conteos, etc.

- `find(begin, end, x)`: Devuelve un iterador al elemento x (o `end`). Costo: $O(n)$
- `count(begin, end, x)`: Cuenta cuántas veces aparece x . Costo: $O(n)$
- `binary_search(begin, end, x)`: Retorna `true` si x está (requiere ordenado). Costo: $O(\log(n))$
- `lower_bound(begin, end, x)`: Retorna un iterador al primer elemento mayor o igual a x (requiere ordenado). Costo: $O(\log(n))$
- `upper_bound(begin, end, x)`: Retorna un iterador al primer elemento mayor a x (requiere ordenado). Costo: $O(\log(n))$

- `sort(begin, end)`: Ordena los elementos del contenedor. Costo: $O(n \log n)$
- `reverse(begin, end)`: Invierte el orden de los elementos. Costo: $O(n)$
- `next_permutation` / `prev_permutation`: Generan permutaciones lexicográficas. Costo: $O(n)$
- `is_sorted(begin, end)`: Verifica si está ordenado. Costo: $O(n)$

- `min, max, min_element, max_element`.
- `accumulate(begin, end, 0)`: Suma los elementos entre `begin` y `end`. Costo: $O(n)$
- `fill(begin, end, val)`: Asigna un valor a todo el rango. Costo: $O(n)$

Consejos al usar algoritmos STL

- Usa `auto` para manejar iteradores más fácilmente.
- Cuidado con los algoritmos que requieren contenedores ordenados.
- Aprovecha los algoritmos: te ahorran código, tiempo y errores.