La STL en C++

Sebastian Mestre

¿Qué es la STL?

- C++ trae varias **estructuras de datos** y **algoritmos** ya implementados.
- El objetivo es **simplificarnos la vida**: usar código probado en lugar de reinventar la rueda.
- Algunos contenedores importantes:
  - `std::vector<T>` — arreglo dinámico.
  - `std::set<T>` — conjunto ordenado.
  - `std::map<K,V>` — diccionario ordenado por clave.
  - `std::stack<T>` — pila (LIFO).
  - `std::queue<T>` — cola (FIFO).
  - `std::priority_queue<T>` — cola de prioridad (max heap).
- Referencia completa: cppreference: containers.

Interfaz de iteradores

- Muchas estructuras de la STL exponen una **interfaz de iteradores** para recorrer sus elementos.
- Funciones y operaciones básicas:
  - `begin(s)` — iterador al inicio de una secuencia `s`.
  - `end(s)` — iterador al final de una secuencia `s`.
  - `*it` — elemento apuntado por el iterador `it`.
- Conceptualmente, un iterador representa una **posición** en la secuencia.
- En una secuencia de N elementos hay $N + 1$ posiciones (antes del primero, entre cada par, y después del último).

```
s = | 2 | 3 | 5 | 8 | 13 |        (5 elementos, 6 posiciones)
    ^                    ^
    begin(s)             end(s)
```

Recorriendo con iteradores

- Podemos recorrer una secuencia usando un bucle `for` con iteradores.

Ejemplo

```
1 vector<int> s = {2, 3, 5, 8, 13};
2
3 for (auto it = begin(s); it != end(s); ++it) {
4   cout << *it << " ";
5 }
6 cout << "\n";
7
```

- Operaciones útiles sobre iteradores:
  - `it1 == it2` verifica si apuntan a la misma posición.
  - `++it` / `-it` avanzar / retroceder.
  - `prev(it)`, `next(it)` — iterador anterior / siguiente.

Problema de ejemplo

Enunciado

Nos dan un arreglo de N números distintos $a[1], a[2], \ldots, a[N]$.
Encontrar la **mínima diferencia** entre dos elementos de este arreglo.

- Restricciones:
  - $1 \leq N \leq 5 \cdot 10^5$
  - $1 \leq a[i] \leq 10^9$
- Entrada:
  - Primera línea: N.
  - Segunda línea: los N valores distintos $a[1], \ldots, a[N]$.
- Salida: la mínima diferencia entre dos elementos de a.

Ejemplo de entrada/salida

Ejemplo

Entrada:

```
6
9000 5004 1000 8900 5001 1100
```

Salida:

```
3
```

Solución naive $O(N^2)$

- Idea: probar **todos los pares** (i, j) y quedarnos con la mínima diferencia.
- Complejidad: $O(N^2)$, demasiado lenta para $N = 5 \cdot 10^5$.

Código

```
int a[500100];
int main() {
  int n; cin >> n;
  forn(i, n) cin >> a[i];

  int ans = abs(a[1] - a[0]);
  forn(i, n) forn(j, n)
      if (i != j) ans = min(ans, abs(a[i] - a[j]));
  cout << ans << "\n";
}
```

`std::vector`: arreglo dinámico

- `std::vector<T>` es la estructura más usada en programación competitiva.
- Se comporta como un **arreglo dinámico**:
  - `vector<T>(n)` crea un vector de n elementos.
  - `a[i]` acceso al elemento en la posición i.
  - `a.size()` tamaño del vector. (todas las estructuras de la STL tienen esto)
  - `a.push_back(x)` agrega un elemento al final.
  - `a.pop_back()` elimina el último elemento.
  - `a.back()` último elemento.
  - `a.clear()` vacia un vector
  - `a.empty()` responde si esta vacio
  - `begin(a)`, `end(a)` iteradores.
- Referencia: cppreference: vector.

Leyendo con `vector`

- Con `vector` no necesitamos fijar un N máximo a mano.

Ejemplo

```
1 int n; cin >> n;
2
3 vector<int> a(n);
4 forn(i, n) cin >> a[i];
```

- El resto del código puede ser igual a la versión con arreglo estático.

Solución con `std::set`

- `std::set<int>` mantiene los elementos **ordenados y sin duplicados**.
- Podemos ir insertando elementos y comparar con sus **vecinos** en el conjunto.

Idea

Para cada x en a:

- Buscamos el primer elemento $\geq x$ y el anterior a él.
- Actualizamos la mejor diferencia.

std::set

- `std::set<T>` es un conjunto ordenado de elementos únicos.
- Métodos principales:
  - `s.size()` cantidad de elementos
  - `s.empty()` ¿está vacío?
  - `s.clear()` elimina todos los elementos
  - `s.insert(x)` inserta el elemento x
  - `s.erase(x)` elimina el elemento x
  - `s.count(x)` indica si x está presente (1 si está, 0 si no)
  - `s.find(x)` devuelve un iterador a la izquierda de x (o `end(s)` si no está)
  - `s.lower_bound(x)` iterador que separa los elementos $< x$ de los $\geq x$
  - `s.upper_bound(x)` iterador que separa los elementos $\leq x$ de los $> x$
  - `begin(s)`, `end(s)` iteradores al inicio y final
- Todos los métodos tienen costo $O(\log N)$.
- Referencia: cppreference: set.

Código con set ($O(N \log N)$)

Implementación

```
int ans = abs(a[1] - a[0]);

set<int> s;
for (int x : a) {                          // O(N) iteraciones
  if (auto it = s.upper_bound(x); it != end(s)) { // O(log N)
    ans = min(ans, *it - x);               // O(1)
  }
  if (auto it = s.lower_bound(x); it != begin(s)) { // O(log N)
    ans = min(ans, x - *prev(it));         // O(1)
  }
  s.insert(x);                             // O(log N)
}
// total: O(N log N)

```

lower_boundy

```
s = | 2 | 3 | 5 | 8 | 13 |
              ^   ^
              |   upper_bound(5)
              lower_bound(5)
```

Variación con `set`

- También podemos construir el `set` a partir del `vector`.
- **Observación**: todo elemento es vecino derecho de su vecino izquierdo

Ejemplo

```
int ans = abs(a[1] - a[0]);

set<int> s(begin(a), end(a)); // O(N log N)
for (int x : a) {
  if (auto it = s.upper_bound(x); it != end(s)) {
    ans = min(ans, *it - x);
  }
}
// total: O(N log N)

```

La cabecera <algorithm>

- Además de contenedores, C++ trae una **biblioteca de algoritmos**.
- Algunos muy usados:
  - `sort(begin(a), end(a))` ordena el vector a.
  - `reverse(begin(a), end(a))` invierte el orden.
  - `rotate(begin(a), begin(a)+k, end(a))` rota k posiciones.
  - `unique(begin(a), end(a))` compacta elementos únicos consecutivos.
  - `fill(begin(a), end(a), x)` llena con un valor.
- Referencia: cppreference: algorithm.

Solución óptima con sort

- **Observación:** Si ordenamos el arreglo, los vecinos serán elementos consecutivos.

Implementación

```
1 sort(begin(a), end(a)); // O(N log N)
2 int ans = a[1] - a[0];
3 forn(i, n-1) ans = min(ans, a[i+1] - a[i]);
4 cout << ans << "\n";
5 // total: O(N log N)
```

Problemas para practicar

- Problemas de CSES donde la STL y `<algorithm>` son especialmente útiles:
  - Missing Number.
  - Palindrome Reorder.
  - Gray Code.
  - Distinct Numbers.
  - Concert Tickets.
  - Restaurant Customers.
- Recomendación: implementar varias soluciones usando `vector`, `set`, `map` y algoritmos de la STL.

Preguntas

Q&A