Apuntes de C++ para Algoritmos y Complejidad

Cristian Quintanilla Ponce | José Alvarado Toro

12 de noviembre de 2025

Índice

1. Contenedor `vector`

1.1. Introducción

`vector` es un contenedor secuencial de la STL que ofrece un **arreglo dinamico**: los elementos se almacenan en memoria contigua y el vector puede crecer o reducir su tamano automaticamente. Es ideal cuando se necesita **acceso aleatorio por indice** y crecimiento dinamico. Las operaciones de insercion o eliminacion al final son rapidas (amortizadas O(1)), pero insertar o borrar en el medio o al inicio puede ser costoso (O(n)).

1.2. Operaciones importantes

1.2.1. Declaracion e inicializacion

```
#include <vector>
#include <string>

// Declaraciones
vector<int> v1;                  // vacio
vector<int> v2(5);               // 5 elementos
    inicializados a 0
vector<int> v3(5, 42);           // 5 elementos
    con valor 42
vector<int> v4 = {1, 2, 3, 4};   //
    inicializacion con lista
vector<string> names = {"Ana","Luis"};
```

1.2.2. Agregar elementos: `push_back()`

Anade un elemento al final del vector. Complejidad amortizada O(1).

```
vector<int> v;
v.push_back(10);
v.push_back(20); // v = {10, 20}
```

1.2.3. Eliminar último elemento: `pop_back()`

Elimina el último elemento (no devuelve su valor).

```
if (!v.empty()) {
    v.pop_back(); // elimina 20
}
```

1.2.4. Acceso a elementos: `operator[]` y `at()`

`operator[]` no comprueba limites; `at()` lanza `out_of_range` si el indice no es valido.

```
v[0] = 5;           // rapido, sin comprobacion
try {
    v.at(2) = 7;    // seguro
} catch (const out_of_range& e) {
    // manejo de error
}
int x = v[0];       // lectura
```

1.2.5. Tamano y capacidad

```
cout << "size: " << v.size() << "\n";
cout << "capacity: " << v.capacity() << "\n";
if (v.empty()) cout << "vacio\n";
```

1.2.6. Reservar espacio y cambiar tamano

`reserve(n)` reserva capacidad; `resize(n)` cambia el tamano logico.

```
v.reserve(100);   // reserva espacio para 100
    elementos
v.resize(8);      // cambia tamano a 8 (anade
    ceros)
v.resize(3);      // reduce tamano a 3
```

1.2.7. Insertar y borrar en posiciones arbitrarias

Inserta o elimina en el medio (coste O(n)).

```
vector<int> a = {1,2,4,5};
auto it = a.begin() + 2;   // apunta al 4
a.insert(it, 3);           // a = {1,2,3,4,5}
a.erase(a.begin() + 1);    // a = {1,3,4,5}
```

1.2.8. Recorrer el vector

```
// range-based for
for (int x : a) cout << x << ' ';

// iteradores
for (auto it = a.begin(); it != a.end(); ++it)
    *it += 1;

// por indice
for (size_t i = 0; i < a.size(); ++i)
    cout << a[i] << ' ';
```

1.2.9. Primer y último elemento

```
if (!a.empty()) {
    cout << "first: " << a.front()
              << ", last: " << a.back();
}
```

1.2.10. Eliminar todos los elementos

```
a.clear(); // tamano 0, mantiene capacidad
// liberar memoria
a.shrink_to_fit();
```

1.2.11. Intercambiar contenidos

```
vector<int> x = {1,2,3}, y = {9,8};
x.swap(y); // x = {9,8}, y = {1,2,3}
```

1.3. Complejidad tipica

- Acceso por indice: O(1)
- `push_back`: amortizada O(1)
- `insert`/`erase` en medio: O(n)
- `clear`: O(n)

2. Contenedor `list`

2.1. Introducción

`list` es un contenedor secuencial que implementa una **lista doblemente enlazada**. Cada elemento contiene punteros al anterior y al siguiente, lo que permite insertar o eliminar elementos en cualquier posicion en tiempo constante O(1), siempre que se tenga un iterador a esa posicion. A diferencia de `vector`, no ofrece acceso aleatorio rapido (no se puede usar el operador `[]`).

2.2. Operaciones importantes

2.2.1. Declaracion e inicializacion

```
#include <list>
#include <string>

list<int> l1;                  // vacia
list<int> l2(5, 100);          // 5 elementos
    con valor 100
list<int> l3 = {1, 2, 3, 4};   // inicializacion
     con lista
list<string> names = {"Ana", "Luis"};
```

2.2.2. Agregar y eliminar elementos al inicio o final

`push_front()` y `push_back()` permiten insertar rapido al inicio o final. `pop_front()` y `pop_back()` eliminan esos elementos.

```
list<int> l = {2, 3};
l.push_front(1);    // {1,2,3}
l.push_back(4);     // {1,2,3,4}
l.pop_front();      // {2,3,4}
l.pop_back();       // {2,3}
```

2.2.3. Acceder al primer y ultimo elemento

```
if (!l.empty()) {
    cout << "first: " << l.front()
              << ", last: " << l.back();
}
```

2.2.4. Insertar y borrar en posiciones arbitrarias

Usando iteradores, se puede insertar o eliminar en cualquier parte en O(1).

```
list<int> a = {1,2,4};
auto it = next(a.begin(), 2); // apunta al 4
a.insert(it, 3);                    // {1,2,3,4}
a.erase(next(a.begin()));     // elimina el 2
    -> {1,3,4}
```

2.2.5. Recorrer la lista

```
// range-based for
for (int x : a) cout << x << ' ';

// iteradores
for (auto it = a.begin(); it != a.end(); ++it)
    *it *= 2;
```

2.2.6. Tamaño y estado

```
cout << "size: " << a.size() << "\n";
if (a.empty()) cout << "lista vacia\n";
```

2.2.7. Eliminar elementos especificos

```
list<int> b = {1,2,3,2,4};
b.remove(2); // elimina todos los valores
    iguales a 2 -> {1,3,4}
```

2.2.8. Ordenar, invertir y eliminar duplicados

`sort()`, `reverse()` y `unique()` operan en la propia lista.

```
list<int> nums = {4,1,3,2,3};
nums.sort();     // {1,2,3,3,4}
nums.unique();   // {1,2,3,4}
nums.reverse();  // {4,3,2,1}
```

2.2.9. Fusionar dos listas ordenadas

```
list<int> l1 = {1,3,5};
list<int> l2 = {2,4,6};
l1.merge(l2); // l1 = {1,2,3,4,5,6}, l2 queda
    vacia
```

2.2.10. Intercambiar contenidos

```
list<int> x = {1,2}, y = {9,8};
x.swap(y); // x = {9,8}, y = {1,2}
```

2.3. Complejidad tipica

- Acceso secuencial: O(n)
- Insercion/eliminacion con iterador: O(1)
- Borrado por valor (`remove`): O(n)
- Ordenar (`sort`): O(n log n)

3. Adaptador `stack`

3.1. Introducción

`stack` es un adaptador de contenedor que implementa una **pila (LIFO - Last In, First Out)**. Utiliza otro contenedor subyacente (por defecto `deque`, pero puede ser `vector` o `list`) y restringe el acceso únicamente al elemento superior. Es ideal cuando se necesita procesar elementos en orden inverso al de inserción.

3.2. Operaciones importantes

3.2.1. Declaración e inicialización

```
#include <stack>

stack<int> s1;                          // pila
    vacia (usa deque)
stack<int, vector<int>> s2;             // pila con
    vector subyacente
stack<int, list<int>> s3;               // pila con
    list subyacente
```

3.2.2. Agregar elemento: `push()`

Añade un elemento al tope de la pila. Complejidad O(1).

```
stack<int> s;
s.push(10);
s.push(20);
s.push(30); // tope: 30
```

3.2.3. Acceder al elemento superior: `top()`

Devuelve una referencia al elemento en el tope (no lo elimina).

```
if (!s.empty()) {
    cout << "Tope: " << s.top() << "\n"; // 30
    s.top() = 99; // se puede modificar el tope
}
```

3.2.4. Eliminar elemento superior: `pop()`

Elimina el elemento del tope. No devuelve su valor.

```
if (!s.empty()) {
    int valor = s.top(); // primero obtener el
        valor
    s.pop();             // luego eliminar
}
```

3.2.5. Tamaño y estado

```
cout << "size: " << s.size() << "\n";
if (s.empty()) {
    cout << "pila vacia\n";
}
```

3.2.6. Patrón típico: procesar todos los elementos

```
while (!s.empty()) {
    int valor = s.top();
    s.pop();
    cout << valor << " "; // procesar en orden
        LIFO
}
```

3.2.7. Intercambiar contenidos

```
stack<int> x, y;
x.push(1); x.push(2);
y.push(9);
x.swap(y); // x tiene {9}, y tiene {1,2}
```

3.3. Complejidad típica

- `push`: O(1)
- `pop`: O(1)
- `top`: O(1)
- No permite iteración directa ni acceso aleatorio

4. Adaptador `queue`

4.1. Introducción

`queue` es un adaptador de contenedor que implementa una **cola (FIFO - First In, First Out)**. Utiliza otro contenedor subyacente (por defecto `deque`, pero puede ser `list`) y restringe el acceso a los extremos: inserción por el final y extracción por el frente. Es ideal cuando se necesita procesar elementos en el mismo orden en que fueron añadidos.

4.2. Operaciones importantes

4.2.1. Declaración e inicialización

```
#include <queue>

queue<int> q1;                      // cola vacia (
    usa deque)
queue<int, list<int>> q2;           // cola con
    list subyacente
```

4.2.2. Agregar elemento: `push()`

Añade un elemento al final de la cola. Complejidad O(1).

```
queue<int> q;
q.push(10);
q.push(20);
q.push(30); // q: frente[10, 20, 30]final
```

4.2.3. Acceder al primer elemento: `front()`

Devuelve una referencia al elemento en el frente (no lo elimina).

```
if (!q.empty()) {
    cout << "Frente: " << q.front() << "\n"; //
        10
    q.front() = 99; // se puede modificar
}
```

4.2.4. Acceder al último elemento: `back()`

Devuelve una referencia al elemento al final de la cola.

```
if (!q.empty()) {
    cout << "Final: " << q.back() << "\n"; //
        30
    q.back() = 88; // se puede modificar
}
```

4.2.5. Eliminar primer elemento: pop()

Elimina el elemento del frente. No devuelve su valor.

```
if (!q.empty()) {
    int valor = q.front(); // primero obtener
        el valor
    q.pop();               // luego eliminar
}
```

4.2.6. Tamaño y estado

```
cout << "size: " << q.size() << "\n";
if (q.empty()) {
    cout << "cola vacia\n";
}
```

4.2.7. Patrón típico: procesar todos los elementos

```
while (!q.empty()) {
    int valor = q.front();
    q.pop();
    cout << valor << " "; // procesar en orden
        FIFO
}
```

4.2.8. Intercambiar contenidos

```
queue<int> x, y;
x.push(1); x.push(2);
y.push(9);
x.swap(y); // x tiene {9}, y tiene {1,2}
```

4.3. Complejidad típica

- push: O(1)
- pop: O(1)
- front, back: O(1)
- No permite iteración directa ni acceso aleatorio

5. Adaptador priority_queue

5.1. Introducción

priority_queue es un adaptador de contenedor que implementa una **cola de prioridad**. Los elementos se organizan en un heap (por defecto max-heap), donde el elemento de mayor prioridad siempre está en el tope. Utiliza vector como contenedor subyacente por defecto. Es ideal cuando se necesita acceso rápido al elemento de mayor (o menor) prioridad.

5.2. Operaciones importantes

5.2.1. Declaración e inicialización

```
#include <queue>
#include <vector>
#include <functional>

// max-heap (mayor prioridad = mayor valor)
priority_queue<int> pq1;

// min-heap (mayor prioridad = menor valor)
priority_queue<int, vector<int>, greater<int>>
    pq2;

// con comparador personalizado
priority_queue<int, vector<int>, less<int>> pq3
    ;

// inicializar desde un rango
vector<int> v = {3, 1, 4, 1, 5};
priority_queue<int> pq4(v.begin(), v.end());
```

5.2.2. Agregar elemento: push()

Añade un elemento y mantiene la propiedad del heap. Complejidad O(log n).

```
priority_queue<int> pq;
pq.push(10);
pq.push(30);
pq.push(20); // tope: 30 (el mayor)
```

5.2.3. Acceder al elemento de mayor prioridad: top()

Devuelve una referencia constante al elemento en el tope (no lo elimina).

```
if (!pq.empty()) {
    cout << "Mayor: " << pq.top() << "\n"; //
        30
    // NO se puede modificar: pq.top() = 99; //
        ERROR
}
```

5.2.4. Eliminar elemento de mayor prioridad: pop()

Elimina el elemento del tope. No devuelve su valor. Complejidad O(log n).

```
if (!pq.empty()) {
    int valor = pq.top(); // primero obtener el
        valor
    pq.pop();             // luego eliminar
}
```

5.2.5. Tamaño y estado

```
cout << "size: " << pq.size() << "\n";
if (pq.empty()) {
    cout << "cola vacia\n";
}
```

5.2.6. Patrón típico: procesar por prioridad

```
while (!pq.empty()) {
    int valor = pq.top();
    pq.pop();
    cout << valor << " "; // procesar en orden
        de prioridad
}
```

5.2.7. Min-heap: menor elemento primero

```
priority_queue<int, vector<int>, greater<int>>
    minHeap;
minHeap.push(30);
minHeap.push(10);
minHeap.push(20);
cout << minHeap.top(); // 10 (el menor)
```

5.2.8. Intercambiar contenidos

```
priority_queue<int> x, y;
x.push(1); x.push(2);
y.push(9);
x.swap(y); // x tiene {9}, y tiene {2,1}
```

5.3. Complejidad típica

- `push`: O(log n)
- `pop`: O(log n)
- `top`: O(1)
- Construcción desde rango: O(n)
- No permite iteración directa ni acceso aleatorio

6. Contenedor `deque`

6.1. Introducción

`deque` (double-ended queue) es un contenedor secuencial que permite inserción y eliminación eficiente en **ambos extremos**. A diferencia de `vector`, no garantiza almacenamiento contiguo, sino que usa múltiples bloques de memoria. Ofrece acceso aleatorio por índice similar a `vector`, pero con mejor rendimiento en inserciones/eliminaciones al inicio. Es el contenedor por defecto usado por `stack` y `queue`.

6.2. Operaciones importantes

6.2.1. Declaración e inicialización

```
#include <deque>
#include <string>

deque<int> d1;                  // vacio
deque<int> d2(5);               // 5 elementos
    inicializados a 0
deque<int> d3(5, 42);           // 5 elementos
    con valor 42
deque<int> d4 = {1, 2, 3, 4};   //
    inicializacion con lista
deque<string> names = {"Ana", "Luis"};
```

6.2.2. Agregar elementos al final: `push_back()`

Añade un elemento al final. Complejidad O(1).

```
deque<int> d;
d.push_back(10);
d.push_back(20); // d = {10, 20}
```

6.2.3. Agregar elementos al inicio: `push_front()`

Añade un elemento al inicio. Complejidad O(1).

```
d.push_front(5); // d = {5, 10, 20}
d.push_front(1); // d = {1, 5, 10, 20}
```

6.2.4. Eliminar último elemento: `pop_back()`

Elimina el último elemento (no devuelve su valor).

```
if (!d.empty()) {
    d.pop_back(); // elimina 20
}
```

6.2.5. Eliminar primer elemento: `pop_front()`

Elimina el primer elemento (no devuelve su valor).

```
if (!d.empty()) {
    d.pop_front(); // elimina 1
}
```

6.2.6. Acceso a elementos: `operator[]` y `at()`

`operator[]` no comprueba límites; `at()` lanza `out_of_range` si el índice no es válido.

```
d[0] = 99;          // rapido, sin comprobacion
try {
    d.at(1) = 77;   // seguro
} catch (const out_of_range& e) {
    // manejo de error
}
int x = d[0];       // lectura
```

6.2.7. Primer y último elemento

```
if (!d.empty()) {
    cout << "first: " << d.front()
         << ", last: " << d.back();
    d.front() = 100; // modificar
    d.back() = 200;
}
```

6.2.8. Tamaño y estado

```
cout << "size: " << d.size() << "\n";
if (d.empty()) cout << "vacio\n";
```

6.2.9. Cambiar tamaño

`resize(n)` cambia el tamaño lógico del deque.

```
d.resize(8);      // cambia tamano a 8 (anade
    ceros)
d.resize(3);      // reduce tamano a 3
```

6.2.10. Insertar y borrar en posiciones arbitrarias

Inserta o elimina en el medio (coste O(n), pero más eficiente que vector cerca de los extremos).

```
deque<int> a = {1, 2, 4, 5};
auto it = a.begin() + 2;    // apunta al 4
a.insert(it, 3);            // a = {1, 2, 3, 4,
    5}
a.erase(a.begin() + 1);     // a = {1, 3, 4, 5}
```

6.2.11. Recorrer el deque

```
// range-based for
for (int x : a) cout << x << ' ';

// iteradores
for (auto it = a.begin(); it != a.end(); ++it)
    *it += 1;

// por indice
for (size_t i = 0; i < a.size(); ++i)
    cout << a[i] << ' ';
```

6.2.12. Eliminar todos los elementos

```
a.clear(); // tamano 0
```

6.2.13. Intercambiar contenidos

```
deque<int> x = {1, 2, 3}, y = {9, 8};
x.swap(y); // x = {9, 8}, y = {1, 2, 3}
```

6.3. Complejidad típica

- Acceso por índice: O(1)
- `push_front`, `push_back`: O(1)
- `pop_front`, `pop_back`: O(1)
- `insert`/`erase` en medio: O(n)
- `clear`: O(n)

6.4. Diferencias con `vector`

- Inserción/eliminación O(1) al inicio (vs O(n) en vector)
- No garantiza memoria contigua
- Ligeramente más lento en acceso aleatorio
- No tiene `capacity()` ni `reserve()`

7. Contenedor `set`

7.1. Introducción

`set` es un contenedor asociativo que almacena elementos **únicos** en **orden ordenado**. Internamente usa un árbol binario balanceado (típicamente red-black tree), lo que garantiza ordenamiento automático y búsquedas eficientes. No permite elementos duplicados. Si se intenta insertar un elemento que ya existe, la operación no tiene efecto. Es ideal cuando se necesita mantener elementos únicos ordenados con búsquedas rápidas.

7.2. Operaciones importantes

7.2.1. Declaración e inicialización

```
#include <set>
#include <string>

set<int> s1;                          // vacio
set<int> s2 = {3, 1, 4, 1, 5};        // {1, 3, 4,
    5} ordenado, sin duplicados
set<string> nombres = {"Ana", "Luis", "Ana"};
    // {"Ana", "Luis"}

// Con comparador personalizado (orden
    descendente)
set<int, greater<int>> s3 = {1, 3, 2}; // {3,
    2, 1}
```

7.2.2. Insertar elementos: `insert()`

Inserta un elemento si no existe. Retorna un par: iterador y bool indicando si se insertó. Complejidad O(log n).

```
set<int> s;
auto result = s.insert(10);
if (result.second) {
    cout << "insertado\n";
}
s.insert(20);
s.insert(10); // no tiene efecto, 10 ya existe

// Insertar multiples
s.insert({30, 40, 50});
```

7.2.3. Buscar elementos: `find()`

Busca un elemento y retorna un iterador. Si no existe, retorna `end()`. Complejidad O(log n).

```
auto it = s.find(20);
if (it != s.end()) {
    cout << "encontrado: " << *it << "\n";
} else {
    cout << "no encontrado\n";
}
```

7.2.4. Contar elementos: `count()`

Retorna 1 si el elemento existe, 0 si no. Complejidad O(log n).

```
if (s.count(20)) {
    cout << "20 existe\n";
}
```

7.2.5. Verificar existencia: `contains()` (C++20)

Verifica si un elemento existe (más legible que `count()`).

```
if (s.contains(20)) {
    cout << "20 existe\n";
}
```

7.2.6. Eliminar elementos: `erase()`

Elimina un elemento por valor o por iterador. Complejidad O(log n).

```
s.erase(20);         // elimina 20
s.erase(s.begin()); // elimina primer elemento

auto it = s.find(30);
if (it != s.end()) {
    s.erase(it);     // elimina usando iterador
}
```

7.2.7. Tamaño y estado

```
cout << "size: " << s.size() << "\n";
if (s.empty()) {
    cout << "set vacio\n";
}
```

7.2.8. Recorrer el set (siempre ordenado)

```
// range-based for (en orden)
for (int x : s) {
    cout << x << ' ';
}

// iteradores
for (auto it = s.begin(); it != s.end(); ++it)
    {
    cout << *it << ' ';
}

// iteradores reversos
for (auto it = s.rbegin(); it != s.rend(); ++it
    ) {
    cout << *it << ' ';
}
```

7.2.9. Primer y último elemento

```
if (!s.empty()) {
    cout << "min: " << *s.begin() << "\n";
    cout << "max: " << *s.rbegin() << "\n";
}
```

7.2.10. Límites: `lower_bound()` y `upper_bound()`

`lower_bound(x)` retorna iterador al primer elemento $>= x$. `upper_bound(x)` retorna iterador al primer elemento $>x$.

```
set<int> s = {1, 3, 5, 7, 9};
auto it1 = s.lower_bound(5);  // apunta a 5
auto it2 = s.upper_bound(5);  // apunta a 7
auto it3 = s.lower_bound(4);  // apunta a 5 (>=
    4)
```

7.2.11. Eliminar todos los elementos

```
s.clear(); // elimina todos
```

7.2.12. Intercambiar contenidos

```
set<int> x = {1, 2, 3}, y = {9, 8};
x.swap(y); // x = {8, 9}, y = {1, 2, 3}
```

7.3. Complejidad típica

- `insert`, `erase`, `find`, `count`: O(log n)
- `lower_bound`, `upper_bound`: O(log n)
- Iteración completa: O(n)
- No permite acceso por índice

8. Contenedor `multiset`

8.1. Introducción

`multiset` es similar a `set`, pero **permite elementos duplicados**. Mantiene los elementos ordenados automáticamente usando un árbol binario balanceado. Los elementos duplicados se almacenan juntos en orden. Es ideal cuando se necesita mantener elementos ordenados con posibles repeticiones.

8.2. Operaciones importantes

8.2.1. Declaración e inicialización

```
#include <set>

multiset<int> ms1;                      // vacio
multiset<int> ms2 = {3, 1, 4, 1, 5};  // {1, 1,
    3, 4, 5} permite duplicados
multiset<int, greater<int>> ms3;       // orden
    descendente
```

8.2.2. Insertar elementos: `insert()`

Siempre inserta el elemento (permite duplicados). Retorna iterador al elemento insertado. Complejidad O(log n).

```
multiset<int> ms;
ms.insert(10);
ms.insert(20);
ms.insert(10); // se inserta, ahora hay dos 10
ms.insert(10); // {10, 10, 10, 20}

// Insertar multiples
ms.insert({30, 20, 30}); // {10, 10, 10, 20,
    20, 30, 30}
```

8.2.3. Buscar elementos: `find()`

Busca un elemento y retorna iterador a una de las ocurrencias. Complejidad O(log n).

```
auto it = ms.find(10);
if (it != ms.end()) {
    cout << "encontrado: " << *it << "\n";
}
```

8.2.4. Contar elementos: `count()`

Retorna el número de ocurrencias del elemento. Complejidad O(log n + k), donde k es el número de ocurrencias.

```
cout << "10 aparece " << ms.count(10) << "
    veces\n"; // 3
cout << "50 aparece " << ms.count(50) << "
    veces\n"; // 0
```

8.2.5. Eliminar elementos: `erase()`

`erase(valor)` elimina **todas** las ocurrencias. `erase(iterador)` elimina solo una ocurrencia.

```
multiset<int> ms = {1, 2, 2, 2, 3};

// Eliminar todas las ocurrencias de 2
ms.erase(2);            // {1, 3}

multiset<int> ms2 = {1, 2, 2, 2, 3};
// Eliminar solo una ocurrencia de 2
auto it = ms2.find(2);
if (it != ms2.end()) {
    ms2.erase(it);      // {1, 2, 2, 3}
}
```

8.2.6. Tamaño y estado

```
cout << "size: " << ms.size() << "\n";
if (ms.empty()) {
    cout << "multiset vacio\n";
}
```

8.2.7. Recorrer el multiset (incluye duplicados)

```
multiset<int> ms = {3, 1, 4, 1, 5, 9, 2, 6, 5};
// Salida: 1 1 2 3 4 5 5 6 9
for (int x : ms) {
    cout << x << ' ';
}
```

8.2.8. Primer y último elemento

```
if (!ms.empty()) {
    cout << "min: " << *ms.begin() << "\n";
    cout << "max: " << *ms.rbegin() << "\n";
}
```

8.2.9. Rango de elementos iguales: `equal_range()`

Retorna un par de iteradores delimitando todas las ocurrencias de un valor.

```
multiset<int> ms = {1, 2, 2, 2, 3};
auto range = ms.equal_range(2);
cout << "Ocurrencias de 2: ";
for (auto it = range.first; it != range.second;
     ++it) {
    cout << *it << ' '; // 2 2 2
}
```

8.2.10. Límites: `lower_bound()` y `upper_bound()`

```
multiset<int> ms = {1, 3, 3, 3, 5, 7};
auto it1 = ms.lower_bound(3);  // apunta al
    primer 3
auto it2 = ms.upper_bound(3);  // apunta a 5 (
    despues de los 3)
```

8.2.11. Eliminar todos los elementos

```
ms.clear();
```

8.2.12. Intercambiar contenidos

```
multiset<int> x = {1, 1, 2}, y = {9, 9};
x.swap(y); // x = {9, 9}, y = {1, 1, 2}
```

8.3. Complejidad típica

- `insert`, `find`: O(log n)
- `erase(valor)`: O(log n + k), donde k es el número de elementos eliminados
- `erase(iterador)`: O(1) amortizado
- `count`: O(log n + k), donde k es el número de ocurrencias
- No permite acceso por índice

8.4. Diferencias con `set`

- Permite elementos duplicados
- `insert()` siempre inserta (no retorna pair con bool)
- `count()` puede retornar valores >1
- `erase(valor)` elimina todas las ocurrencias
- `equal_range()` es útil para procesar duplicados

9. Contenedor `string`

9.1. Introducción

`string` es un contenedor especializado para manejar cadenas de caracteres en C++. Internamente es similar a `vector<char>` pero con operaciones específicas para texto. Permite manipulación eficiente de cadenas, crecimiento dinámico y ofrece numerosas funciones para búsqueda, comparación y transformación. Es la forma recomendada de trabajar con texto en C++ moderno, reemplazando los arreglos de caracteres estilo C.

9.2. Operaciones importantes

9.2.1. Declaración e inicialización

```cpp
#include <string>
#include <iostream>
using namespace std;

string s1;                              // cadena
    vacia
string s2 = "Hola";                     // desde
    literal
string s3("Mundo");                     //
    constructor
string s4(5, 'a');                      // "aaaaa"
    (5 caracteres 'a')
string s5 = s2;                         // copia
string s6(s2, 1, 2);                    //
    subcadena: "ol" (desde pos 1, 2 chars)
```

9.2.2. Concatenación: operator+ y append()

```cpp
string a = "Hola";
string b = " Mundo";
string c = a + b;                       // "Hola
    Mundo"
c = c + "!";                            // "Hola
    Mundo!"
c += " 2024";                           // "Hola
    Mundo! 2024"

a.append(b);                            // a = "
    Hola Mundo"
a.append(3, '.');                       // a = "
    Hola Mundo..."
```

9.2.3. Acceso a caracteres: operator[] y at()

```cpp
string s = "Hola";
char c1 = s[0];                         // 'H' (sin
     verificacion)
char c2 = s.at(1);                      // 'o' (con
     verificacion)
s[0] = 'h';                             // s = "
    hola"

try {
    char c = s.at(10);                  // lanza
        out_of_range
} catch (const out_of_range& e) {
    cout << "indice invalido\n";
}
```

9.2.4. Primer y último carácter: front() y back()

```cpp
string s = "Texto";
char primero = s.front();               // 'T'
char ultimo = s.back();                 // 'o'
s.front() = 't';                        // s = "
    texto"
s.back() = 's';                         // s = "
    textos"
```

9.2.5. Tamaño: size(), length() y empty()

```cpp
string s = "Hola";
cout << s.size() << "\n";               // 4
cout << s.length() << "\n";             // 4 (
    equivalente a size)
if (s.empty()) {
    cout << "cadena vacia\n";
}
```

9.2.6. Modificar tamaño: resize() y clear()

```cpp
string s = "Hola";
s.resize(7, 'x');                       // s = "
    Holaxx"
s.resize(3);                            // s = "Hol
    "
s.clear();                              // s = "" (
    vacia)
```

9.2.7. Insertar texto: insert()

```cpp
string s = "Hola Mundo";
s.insert(5, "Bello ");                  // "Hola
    Bello Mundo"
s.insert(0, ">> ");                     // ">> Hola
     Bello Mundo"
s.insert(s.length(), " <<");            // agregar
    al final
```

9.2.8. Eliminar texto: erase()

```cpp
string s = "Hola Mundo";
s.erase(5, 6);                          // elimina
    6 chars desde pos 5: "Hola"
s.erase(2);                             // elimina
    desde pos 2 al final: "Ho"

string s2 = "ABCDEF";
s2.erase(s2.begin() + 2);               // elimina
    'C': "ABDEF"
```

9.2.9. Reemplazar texto: replace()

```cpp
string s = "Hola Mundo";
s.replace(5, 5, "Amigo");               // "Hola
    Amigo" (desde pos 5, 5 chars)
s.replace(0, 4, "Adios");               // "Adios
    Amigo"
```

9.2.10. Buscar texto: find()

```cpp
string s = "Hola Mundo Hola";
size_t pos = s.find("Mundo");           // pos = 5
if (pos != string::npos) {
    cout << "encontrado en pos " << pos << "\n"
        ;
}

pos = s.find("Hola", 1);                // busca
    desde pos 1: encuentra 11
pos = s.find('M');                      // busca
    caracter: pos = 5
pos = s.find("XYZ");                    // no
    existe: string::npos
```

9.2.11. Buscar desde el final: rfind()

```cpp
string s = "Hola Mundo Hola";
size_t pos = s.rfind("Hola");           // ultima
    ocurrencia: pos = 11
pos = s.rfind('o');                     // ultima '
    o': pos = 13
```

9.2.12. Buscar primer/último de un conjunto: find_first_of() y find_last_of()

```
string s = "Hola Mundo 123";
// Busca primer vocal
size_t pos = s.find_first_of("aeiou");  // pos
    = 1 ('o')
// Busca primer digito
pos = s.find_first_of("0123456789");    // pos
    = 11 ('1')
// Busca ultimo digito
pos = s.find_last_of("0123456789");     // pos
    = 13 ('3')
```

9.2.13. Buscar primer carácter que NO esté en conjunto: find_first_not_of()

```
string s = "   Hola";
size_t pos = s.find_first_not_of(" "); // pos =
    3 (primer no-espacio)
```

9.2.14. Subcadenas: substr()

```
string s = "Hola Mundo";
string sub1 = s.substr(5);          // "Mundo"
    (desde pos 5 hasta el final)
string sub2 = s.substr(0, 4);       // "Hola" (
    desde pos 0, 4 chars)
string sub3 = s.substr(5, 3);       // "Mun"
```

9.2.15. Comparación: compare() y operadores

```
string s1 = "abc";
string s2 = "abd";

// Usando operadores
if (s1 == s2) cout << "iguales\n";
if (s1 < s2) cout << "s1 es menor\n";    //
    true (orden lexicografico)
if (s1 != s2) cout << "diferentes\n";

// Usando compare()
int result = s1.compare(s2);
// result < 0 si s1 < s2
// result > 0 si s1 > s2
// result == 0 si s1 == s2
```

9.2.16. Conversión a C-string: c_str() y data()

```
string s = "Hola";
const char* cstr = s.c_str();       // puntero
    a arreglo terminado en '\0'
printf("%s\n", s.c_str());          // para
    funciones estilo C
```

9.2.17. Conversión de números a string

```
#include <string>

int num = 42;
double pi = 3.14159;

string s1 = to_string(num);         // "42"
string s2 = to_string(pi);          //
    "3.141590"
string s3 = to_string(true);        // "1"
```

9.2.18. Conversión de string a números

```
string s1 = "42";
string s2 = "3.14";
string s3 = "123abc";

int num = stoi(s1);                 // 42
double d = stod(s2);                // 3.14
long l = stol(s1);                  // 42L

// stoi ignora caracteres no numericos al final
int n = stoi(s3);                   // 123

// Manejo de errores
try {
    int x = stoi("abc");            // lanza
        invalid_argument
} catch (const invalid_argument& e) {
    cout << "conversion invalida\n";
}
```

9.2.19. Recorrer string

```
string s = "Hola";

// range-based for
for (char c : s) {
    cout << c << ' ';               // H o l a
}

// por indice
for (size_t i = 0; i < s.size(); ++i) {
    cout << s[i] << ' ';
}

// iteradores
for (auto it = s.begin(); it != s.end(); ++it)
    {
    *it = toupper(*it);             //
        convertir a mayusculas
}
```

9.2.20. Transformaciones comunes

```
#include <algorithm>
#include <cctype>

string s = "Hola Mundo";

// A mayusculas
transform(s.begin(), s.end(), s.begin(), ::
    toupper);
// s = "HOLA MUNDO"

// A minusculas
transform(s.begin(), s.end(), s.begin(), ::
    tolower);
// s = "hola mundo"

// Invertir
reverse(s.begin(), s.end());
// s = "odnum aloh"
```

9.2.21. Eliminar espacios al inicio/final

```
string s = "   Hola Mundo   ";

// Eliminar espacios al inicio
s.erase(s.begin(), find_if(s.begin(), s.end(),
    [](unsigned char c) {
    return !isspace(c);
```

```
}));

// Eliminar espacios al final
s.erase(find_if(s.rbegin(), s.rend(), [](
    unsigned char c) {
    return !isspace(c);
}).base(), s.end());
// s = "Hola Mundo"
```

9.2.22. Dividir string (split)

```
string s = "uno,dos,tres,cuatro";
vector<string> tokens;
stringstream ss(s);
string token;

while (getline(ss, token, ',')) {
    tokens.push_back(token);
}
// tokens = {"uno", "dos", "tres", "cuatro"}
```

9.2.23. Extraer números de una línea con stringstream

```
// Extraer numeros separados por espacios
string linea = "5 10 100 1000";
stringstream ss(linea);
vector<int> numeros;
int num;

while (ss >> num) {
    numeros.push_back(num);
}
// numeros = {5, 10, 100, 1000}

// Alternativa: extraer numero por numero
stringstream ss2("42 73 99");
int a, b, c;
ss2 >> a >> b >> c;  // a=42, b=73, c=99

// Con diferentes tipos
string datos = "3.14 100 texto";
stringstream ss3(datos);
double pi;
int entero;
string palabra;
ss3 >> pi >> entero >> palabra;  // pi=3.14,
    entero=100, palabra="texto"
```

9.2.24. Intercambiar contenidos

```
string s1 = "Hola";
string s2 = "Mundo";
s1.swap(s2);                        // s1 = "
    Mundo", s2 = "Hola"
```

9.3. Complejidad típica

- Acceso por índice: O(1)
- `find`, `rfind`: O(n*m) donde n es el tamaño del string y m del patrón
- `insert`, `erase`, `replace`: O(n)
- `append`, `+=`: amortizada O(1)
- `substr`: O(m) donde m es el tamaño de la subcadena
- Conversiones `stoi`, `stod`: O(n)

10. Contenedor `map`

10.1. Introducción

`map` es un contenedor asociativo que almacena pares **clave-valor** únicos, ordenados por clave. Internamente usa un árbol binario balanceado (típicamente red-black tree). Cada clave es única y está asociada a un valor. Las claves están ordenadas automáticamente. Es ideal para búsquedas rápidas, asociaciones clave-valor y cuando se necesita mantener las claves ordenadas.

10.2. Operaciones importantes

10.2.1. Declaración e inicialización

```
#include <map>
#include <string>

map<string, int> m1;    // vacio
map<string, int> edades = {
    {"Ana", 25},
    {"Luis", 30},
    {"Pedro", 28}
}; // ordenado por clave alfabeticamente

// Con comparador personalizado (orden
    descendente)
map<int, string, greater<int>> m2;
```

10.2.2. Insertar elementos: `insert()` y `operator[]`

`operator[]` crea la entrada si no existe e inicializa el valor. `insert()` solo inserta si la clave no existe. Complejidad O(log n).

```
map<string, int> m;

// Usando operator[] (crea si no existe)
m["Ana"] = 25;
m["Luis"] = 30;
m["Ana"] = 26;  // actualiza valor existente

// Usando insert (no sobrescribe)
m.insert({"Pedro", 28});
auto result = m.insert({"Ana", 99}); // no
    inserta, Ana existe
if (!result.second) {
    cout << "Ana ya existe\n";
}

// Insert con make_pair
m.insert(make_pair("Maria", 27));
```

10.2.3. Acceder a valores: `operator[]` y `at()`

`operator[]` crea la entrada si no existe. `at()` lanza excepción si la clave no existe.

```
cout << m["Ana"] << "\n";        // 26
cout << m["Carlos"] << "\n";     // crea Carlos
    con valor 0

try {
    cout << m.at("Luis") << "\n";  // 30
```

```
    cout << m.at("Inexistente");   // lanza
        out_of_range
} catch (const out_of_range& e) {
    cout << "clave no encontrada\n";
}
```

10.2.4. Buscar elementos: `find()`

Busca una clave y retorna iterador. Si no existe, retorna `end()`. Complejidad O(log n).

```
auto it = m.find("Ana");
if (it != m.end()) {
    cout << it->first << ": " << it->second <<
        "\n";
} else {
    cout << "no encontrado\n";
}
```

10.2.5. Verificar existencia: `count()` y `contains()`

`count()` retorna 1 si existe, 0 si no. `contains()` (C++20) es más legible.

```
if (m.count("Ana")) {
    cout << "Ana existe\n";
}

if (m.contains("Luis")) {
    cout << "Luis existe\n";
}
```

10.2.6. Eliminar elementos: `erase()`

Elimina por clave o por iterador. Complejidad O(log n).

```
m.erase("Pedro");       // elimina por clave
m.erase(m.begin());     // elimina primer
    elemento

auto it = m.find("Ana");
if (it != m.end()) {
    m.erase(it);        // elimina usando
        iterador
}
```

10.2.7. Tamaño y estado

```
cout << "size: " << m.size() << "\n";
if (m.empty()) {
    cout << "map vacio\n";
}
```

10.2.8. Recorrer el map (ordenado por clave)

```
// range-based for con structured binding (C
    ++17)
for (const auto& [clave, valor] : m) {
    cout << clave << ": " << valor << "\n";
}

// range-based for tradicional
for (const auto& par : m) {
    cout << par.first << ": " << par.second <<
        "\n";
}

// iteradores
for (auto it = m.begin(); it != m.end(); ++it)
    {
    cout << it->first << ": " << it->second <<
        "\n";
}
```

10.2.9. Primera y última entrada

```
if (!m.empty()) {
    auto primero = m.begin();
    auto ultimo = m.rbegin();
    cout << "min clave: " << primero->first <<
        "\n";
    cout << "max clave: " << ultimo->first << "
        \n";
}
```

10.2.10. Límites: `lower_bound()` y `upper_bound()`

```
map<int, string> m = {{1,"a"}, {3,"c"}, {5,"e"
    }};
auto it1 = m.lower_bound(3);  // apunta a {3,"c
    "}
auto it2 = m.upper_bound(3);  // apunta a {5,"e
    "}
auto it3 = m.lower_bound(2);  // apunta a {3,"c
    "}
```

10.2.11. Modificar valores existentes

```
if (m.count("Ana")) {
    m["Ana"] += 1;  // incrementar
}

auto it = m.find("Luis");
if (it != m.end()) {
    it->second = 35;  // modificar via iterador
}
```

10.2.12. Eliminar todos los elementos

```
m.clear();
```

10.2.13. Intercambiar contenidos

```
map<string, int> x = {{"a",1}, {"b",2}};
map<string, int> y = {{"z",9}};
x.swap(y);
```

10.3. Complejidad típica

- `insert`, `erase`, `find`, `operator[]`, `at`: O(log n)
- `lower_bound`, `upper_bound`: O(log n)
- Iteración completa: O(n)
- No permite acceso por índice numérico

11. Algoritmos de la STL

11.1. Introducción

La STL proporciona una amplia colección de algoritmos genéricos en `<algorithm>` y `<numeric>`. Estos algoritmos trabajan con iteradores, permitiendo su uso con cualquier contenedor. La mayoría no modifican el tamaño del contenedor, solo sus elementos.

11.2. Algoritmos de ordenamiento

11.2.1. Ordenar: `sort()`

Ordena elementos en orden ascendente. Complejidad O(n log n).

```
#include <algorithm>
#include <vector>

vector<int> v = {4, 2, 5, 1, 3};
sort(v.begin(), v.end());  // {1, 2, 3, 4, 5}

// Orden descendente
sort(v.begin(), v.end(), greater<int>());

// Comparador personalizado
sort(v.begin(), v.end(), [](int a, int b) {
    return a > b;  // descendente
});
```

11.2.2. Ordenar parcialmente: `partial_sort()`

Ordena los primeros n elementos. Complejidad O(n log k).

```
vector<int> v = {5, 2, 8, 1, 9, 3};
// Ordenar solo los 3 primeros
partial_sort(v.begin(), v.begin() + 3, v.end())
    ;
// v = {1, 2, 3, ...} (resto sin orden
    garantizado)
```

11.2.3. Verificar si está ordenado: `is_sorted()`

Verifica si un rango está ordenado. Complejidad O(n).

```
vector<int> v = {1, 2, 3, 4, 5};
if (is_sorted(v.begin(), v.end())) {
    cout << "ordenado\n";
}
```

11.3. Algoritmos de búsqueda

11.3.1. Búsqueda binaria: `binary_search()`

Verifica si un elemento existe en un rango **ordenado**. Complejidad O(log n).

```
vector<int> v = {1, 2, 3, 4, 5};
if (binary_search(v.begin(), v.end(), 3)) {
    cout << "encontrado\n";
}
```

11.3.2. Límites: `lower_bound()` y `upper_bound()`

Búsqueda en rango ordenado. Complejidad O(log n).

```
vector<int> v = {1, 2, 4, 4, 4, 5, 7};
auto it1 = lower_bound(v.begin(), v.end(), 4);
    // primer 4
auto it2 = upper_bound(v.begin(), v.end(), 4);
    // despues del ultimo 4
int count = it2 - it1;  // cantidad de 4s
```

11.3.3. Buscar elemento: `find()`

Busca la primera ocurrencia de un valor. Complejidad O(n).

```
vector<int> v = {1, 2, 3, 4, 5};
auto it = find(v.begin(), v.end(), 3);
if (it != v.end()) {
    cout << "encontrado en posicion " << (it -
        v.begin()) << "\n";
}
```

11.3.4. Buscar con condición: `find_if()`

Busca el primer elemento que cumple una condición. Complejidad O(n).

```
vector<int> v = {1, 2, 3, 4, 5};
auto it = find_if(v.begin(), v.end(), [](int x)
    {
    return x > 3;
});
if (it != v.end()) {
    cout << "primer elemento > 3: " << *it << "
        \n";  // 4
}
```

11.4. Algoritmos de modificación

11.4.1. Invertir: `reverse()`

Invierte el orden de los elementos. Complejidad O(n).

```
vector<int> v = {1, 2, 3, 4, 5};
reverse(v.begin(), v.end());  // {5, 4, 3, 2,
    1}
```

11.4.2. Copiar: `copy()`

Copia elementos de un rango a otro. Complejidad O(n).

```
vector<int> src = {1, 2, 3};
vector<int> dst(3);
copy(src.begin(), src.end(), dst.begin());
```

11.4.3. Llenar: `fill()`

Asigna un valor a todos los elementos. Complejidad O(n).

```
vector<int> v(5);
fill(v.begin(), v.end(), 42);  // {42, 42, 42,
    42, 42}
```

11.4.4. Transformar: `transform()`

Aplica una función a cada elemento. Complejidad O(n).

```
vector<int> v = {1, 2, 3, 4, 5};
transform(v.begin(), v.end(), v.begin(), [](int
    x) {
    return x * 2;
});  // {2, 4, 6, 8, 10}
```

11.4.5. Reemplazar: `replace()`

Reemplaza todas las ocurrencias de un valor. Complejidad O(n).

```
vector<int> v = {1, 2, 3, 2, 5};
replace(v.begin(), v.end(), 2, 99);  // {1, 99,
    3, 99, 5}
```

11.4.6. Eliminar duplicados: `unique()`

Elimina duplicados **consecutivos** (requiere ordenar primero). Complejidad O(n).

```
vector<int> v = {1, 2, 2, 3, 3, 3, 4};
auto it = unique(v.begin(), v.end());
v.erase(it, v.end());  // {1, 2, 3, 4}

// Patron completo con sort
vector<int> v2 = {3, 1, 3, 2, 1};
sort(v2.begin(), v2.end());
v2.erase(unique(v2.begin(), v2.end()), v2.end()
    );
```

11.4.7. Rotar: `rotate()`

Rota elementos en un rango. Complejidad O(n).

```
vector<int> v = {1, 2, 3, 4, 5};
rotate(v.begin(), v.begin() + 2, v.end());
// {3, 4, 5, 1, 2}
```

11.5. Algoritmos de permutación

11.5.1. Siguiente permutación: `next_permutation()`

Genera la siguiente permutación lexicográfica. Complejidad O(n).

```
vector<int> v = {1, 2, 3};
do {
    for (int x : v) cout << x << " ";
    cout << "\n";
} while (next_permutation(v.begin(), v.end()));
// Genera: 123, 132, 213, 231, 312, 321
```

11.5.2. Permutación anterior: `prev_permutation()`

Genera la permutación lexicográfica anterior.

```
vector<int> v = {3, 2, 1};
prev_permutation(v.begin(), v.end());  // {3,
    1, 2}
```

11.6. Algoritmos de conjunto

11.6.1. Unión: `set_union()`

Une dos rangos ordenados. Complejidad O(n + m).

```
vector<int> a = {1, 2, 3, 4};
vector<int> b = {3, 4, 5, 6};
vector<int> result;
set_union(a.begin(), a.end(),
          b.begin(), b.end(),
          back_inserter(result));
// result = {1, 2, 3, 4, 5, 6}
```

11.6.2. Intersección: `set_intersection()`

Encuentra elementos comunes en dos rangos ordenados. Complejidad O(n + m).

```
vector<int> a = {1, 2, 3, 4};
vector<int> b = {3, 4, 5, 6};
vector<int> result;
set_intersection(a.begin(), a.end(),
                 b.begin(), b.end(),
                 back_inserter(result));
// result = {3, 4}
```

11.6.3. Diferencia: `set_difference()`

Elementos en el primer rango pero no en el segundo. Complejidad O(n + m).

```
vector<int> a = {1, 2, 3, 4};
vector<int> b = {3, 4, 5, 6};
vector<int> result;
set_difference(a.begin(), a.end(),
               b.begin(), b.end(),
               back_inserter(result));
// result = {1, 2}
```

11.7. Algoritmos numéricos

11.7.1. Sumar elementos: `accumulate()`

Suma todos los elementos (requiere `<numeric>`). Complejidad O(n).

```
#include <numeric>

vector<int> v = {1, 2, 3, 4, 5};
int suma = accumulate(v.begin(), v.end(), 0);
    // 15

// Con operacion personalizada
int producto = accumulate(v.begin(), v.end(),
    1,
                          [](int a, int b) {
                              return a * b; });
```

11.7.2. Mínimo y máximo: `min_element()` y `max_element()`

Encuentra el elemento mínimo o máximo. Complejidad O(n).

```
vector<int> v = {3, 1, 4, 1, 5};
auto min_it = min_element(v.begin(), v.end());
auto max_it = max_element(v.begin(), v.end());
cout << "min: " << *min_it << ", max: " << *
    max_it << "\n";
```

11.7.3. Contar elementos: `count()` y `count_if()`

Cuenta ocurrencias o elementos que cumplen condición. Complejidad O(n).

```cpp
vector<int> v = {1, 2, 3, 2, 5, 2};
int n = count(v.begin(), v.end(), 2);  // 3

int pares = count_if(v.begin(), v.end(), [](int
     x) {
    return x % 2 == 0;
});  // 3
```

11.8. Algoritmos de verificación

11.8.1. Verificar condición: `all_of()`, `any_of()`, `none_of()`

Verifican si todos, alguno o ninguno cumple una condición. Complejidad O(n).

```cpp
vector<int> v = {2, 4, 6, 8};

bool todos_pares = all_of(v.begin(), v.end(),
    [](int x) {
    return x % 2 == 0;
});  // true

bool hay_impar = any_of(v.begin(), v.end(), [](
    int x) {
    return x % 2 != 0;
});  // false

bool ninguno_negativo = none_of(v.begin(), v.
    end(), [](int x) {
    return x < 0;
});  // true
```

12. Funciones Útiles y Patrones Comunes

12.1. Generación de Permutaciones

12.1.1. Función para generar todas las permutaciones

Utiliza `next_permutation()` para generar todas las permutaciones de un vector. El vector debe estar inicialmente ordenado. Complejidad: O(n! * n).

```cpp
void perm(vector<char> &arr){
    do{
        for(auto c : arr){
            cout << c << " ";
        }
        cout << endl;
    } while(next_permutation(arr.begin(), arr.
        end()));
}
```

12.2. Generación de Subconjuntos (Bitmask)

12.2.1. Función para generar todos los subconjuntos

Utiliza máscaras de bits para generar todos los subconjuntos posibles (conjunto potencia). Para un conjunto de tamaño n, genera 2^n subconjuntos. Complejidad: O(2^n * n).

```cpp
void mask(vector<char> &arr){
    int n = arr.size();
    int total_subset = 1 << n;  // 2^n
        subconjuntos

    for(int i = 0; i < total_subset; i++){
        for(int j = 0; j < n; j++){
            if(i & (1 << j)){
                cout << arr[j];
                if(j < n - 1) cout << " ";
            }
        }
        cout << endl;
    }
}
```

13. Backtracking

13.1. Introduccion

El backtracking es una técnica de diseño de algoritmos para resolver problemas combinatorios. Consiste en construir soluciones parciales y retroceder cuando se determina que una solución parcial no puede llevar a una solución completa válida.

13.2. Fuerza Bruta (Brute Force)