# Manual 2 - 2do Torneo de Programación Competitiva

Lions R.C.

Junio 2019

## Contents



# 1 Estructuras de datos sencillos

Muchas veces, nos encontramos en medio de un problema que ocupa el uso de algo más flexible que un arreglo. Si queremos borrar datos de un arreglo, tendriamos que desplazar todos los demás datos que estan enfrente hacia atras, y nuestros programas se vuelven más ineficientes y más complicados de mantener.

Una solución es utilizar otras estructuras de datos, que almacenan y manejan datos de distintas maneras. En esta sección, solo se explicaran algunas de las muchas estructuras de datos, y la siguiente semana se darán a conocer los demás.

## 1.1 Vectores

Un vector no es nada más que un arreglo dinámico. Esto significa que el vector no tiene un tamaño fijo y puedes agregar y quitar elementos sin problema.

Para crear un vector es necesario agregar una librería especifica de esta estructura llamada **vector**.

Para incluir esta librería se debe de escribir **#include <vector>** en las primeras líneas de tu programa.

Listing 1: Vectores

```
#include <iostream>
#include <vector>

using namespace std;

int main() {

}
```

Como un vector guarda datos al igual que un arreglo, debes definir el tipo de dato que se almacenará a la hora de declararlo. Se puede declarar el vector con el sintaxis **vector<tipo> nombre;** Como se puede observar, el vector no requiere tener un tamaño predeterminado.

Listing 2: Vectores

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> enteros;
}
```

Aqui se ha declarado un vector de enteros, para agregarle elementos a este vector se debe escribir el nombre del vector seguido por un punto y la función **push_back()** con el valor del elemento entre los paréntesis.

Listing 3: Agregando valores

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> enteros;
    enteros.push_back(1);
    enteros.push_back(4);
    enteros.push_back(9);
}
```

Aqui, nuestro vector guardará los valores de 1, 4 y 9.

Para ver o modificar el valor en algún índice se puede utilizar el mismo sintaxis de un arreglo. Si en el ejemplo queremos cambiar el 9 a un 7, podemos modificarlo y ver sus cambios con el siguiente código:

Listing 4: Modificando valores

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> enteros;
    enteros.push_back(1);
    enteros.push_back(4);
    enteros.push_back(9);
    for(int i = 0; i < 3; i++) {
        cout << enteros[i] << endl;
    }
    enteros[2] = 7;
    for(int i = 0; i < 3; i++) {
        cout << enteros[i] << endl;
    }
}
```

Primero se imprimirán los valores 1, 4 y 9, luego se verán los valores 1, 4 y 7 en la consola.

Existen más funciones que modifican vectores como **insert()** que inserta elementos en ciertos índices, **erase** que elimina ciertos elementos, **clear()** que borra todos los datos en un vector, **pop_back()** que elimina el último valor y finalmente **size()**, que indica el tamaño de un vector.

Listing 5: Jugando con vectores

```
#include <iostream>
```

```cpp
#include <vector>

using namespace std;

int main() {
    vector<int> enteros;
    for(int i = 0; i < 10; i++) {
        enteros.push_back(i);
    }
    enteros.insert(enteros.begin() + 5, 25);
    enteros.erase(enteros.begin() + 7);
    cout << "Cantidad: " << enteros.size() << endl;
    for(int i = 0; i < enteros.size(); i++) {
        cout << enteros[i] << endl;
    }
    enteros.clear();
    cout << "Cantidad: " << enteros.size() << endl;
}
```

Liga al código

Como se puede ver en el código de arriba, se crea un vector de enteros y se llena de 0 a 9, luego se inserta el valor 25 en el índice 5 y se elimina el valor en el índice 7. Luego se imprime el tamaño del arreglo (12 en ese momento), se imprimen todos los valores, se limpia el vector y finalmente se imprime el tamaño final (cero porque se limpió).

Se debe notar que para las funciones **insert** y **erase**, se ocupa llamar a la función begin para ese vector y luego se debe sumar el índice deseado a ese valor. Esta suma determina el lugar en la memoria donde esta guardado el valor con ese índice.

## 1.2 Pares

A veces es conveniente guardar cosas en pares, por ejemplo se pueden guardar dos enteros en un par para representar las coordenadas **x** y **y** de un plano, o se puede guardar un caracter con un booleano indicando si es vocal. Se puede crear pares de cualquier tipo de dato y los dos tipos no tienen que ser iguales.

Para crear un par, se debe incluir la librería **utility** y para declararlo se tiene que escribir **pair<tipo, tipo> nombre;**

Listing 6: Declarando pares

```cpp
#include <iostream>
#include <utility>

using namespace std;
```

```
int main() {
    pair<int, float> miPar;
}
```

Para declarar un par, se puede utilizar la función **make_pair** y para ver los datos de este par, se puede llamar el elemento **first** para el primer dato (en este caso el entero) y **second** para el segundo dato (en este caso el número flotante).

Listing 7: Declarando pares

```
#include <iostream>
#include <utility>

using namespace std;

int main() {
    pair<int, float> miPar;
    miPar = make_pair(20, 4.472136);
    cout << miPar.first << " " << miPar.second << endl;
}
```

Combinado con arreglos y vectores, pueden servir para una multitud de aplicaciones. Digamos que queremos guardar todos los puntos de un triángulo para verlos después. Podemos guardar un arreglo de tres pares correspondientes a los tres puntos de ese triángulo:

Listing 8: Arreglo de pares

```
#include <iostream>
#include <utility>

using namespace std;

int main() {
    pair<int, int> triangulo[3];
    for(int i = 0; i < 3; i++) {
        int x, y;
        cin >> x >> y; //Lee las coordenadas
        pair<int, int> punto = make_pair(x, y);
        triangulo[i] = punto;
    }
}
```

Como se puede ver en el código, primero leemos dos números de la consola tres veces, correspondiendo a las X y las Y de los puntos del triángulo. Luego guardamos estos valores en el par **punto**, donde el primer valor es la X y el segundo valor es la Y. Finalmente, guardamos ese par en el arreglo triángulo.

## 1.3 Mapas

El mapa es una estructura de datos bastante útil debido a que mapea un valor a otro, como indicaría el nombre. Esto te permite asociar algún valor con otro para su busqueda rápida. Puedes declarar un mapa incluyendo la librería **map** y escribiendo **map<tipo, tipo> nombre;**

Digamos que quieres guardar las edades de cada uno de tus amigos. Una manera de hacer esto es tener un arreglo de pares con el nombre de tu amigo y su edad, y luego para obtener la edad de alguno de ellos tendrías que iterar sobre todos tus amigos hasta encontrar el que quieres. Una alternativa más fácil es usar un mapa que simplemente recibe el nombre de tu amigo y que te de su edad.

Les mostraremos las dos opciones, empezando con la implementación de este problema con un arreglo de pares:

Listing 9: Implementación con pares

```
#include <iostream>
#include <utility>

using namespace std;

int main() {
    int n;
    cin >> n; //Lee el numero de amigos
    pair<string, int> edades[n];
    //Guardar todos los amigos y sus edades
    for(int i = 0; i < n; i++) {
        string nombre;
        int edad;
        cin >> nombre >> edad;
        pair<string, int> amigo = make_pair(nombre, edad);
        edades[i] = amigo;
    }
    //Buscar el amigo "Juan" y desplegar su edad
    for(int i = 0; i < n; i++) {
        if(edades[i].first == "Juan") {
            cout << "Juan tiene " << edades[i].second << endl;
            break;
        }
    }
}
```

Como se puede ver, se guardaron todos los valores en un arreglo de pares y luego se tuvo que hacer una busqueda de todos los amigos hasta encontrar a Juan. Esta es la simplificación con map:

Listing 10: Implementación con map

```
#include <iostream>
#include <map>

using namespace std;

int main() {
    int n;
    cin >> n; //Lee el numero de amigos
    map<string, int> edades;
    for(int i = 0; i < n; i++) {
        string nombre;
        int edad;
        cin >> nombre >> edad;
        edades[nombre] = edad;
    }
    //Buscar el amigo "Juan" y desplegar su edad
    cout << "Juan tiene " << edades["Juan"] << endl;
}
```

El primer valor funciona como un estilo de índice que guarda el segundo valor, y se puede utilizar cualquier tipo de variable como este índice.

Pero también se debe aclarar que hay dos tipos de mapa, el mapa ordenado y el mapa desordenado. Hasta ahorita, hemos estado utilizando el mapa ordenado o **map**. El mapa desordenado se llama **unordered_map** y para utilizarlo debes incluir la librería con este mismo nombre.

Existen varias ventajas de usar un mapa desordenado sobre uno ordenado, el aceso de datos es más rápido en un mapa desordenado, pero a la vez ocupa más memoria que un mapa ordenado. También tiene el mismo sintaxis que el mapa ordenado.

Listing 11: Mapa desordenado

```
#include <iostream>
#include <unordered_map>

using namespace std;

int main() {
    int n;
    cin >> n; //Lee el numero de amigos
    unordered_map<string, int> edades;
    for(int i = 0; i < n; i++) {
        string nombre;
        int edad;
        cin >> nombre >> edad;
        edades[nombre] = edad;
```

```
    }
    //Buscar el amigo "Juan" y desplegar su edad
    cout << "Juan tiene " << edades["Juan"] << endl;
}
```

## 1.4 Sets

Sets son como vectores pero con condiciones más estrictas; Un set no puede tener valores repetidos y no permite la modificación de estos valores. Para utilizar un set, se debe incluir la librería **set** y se debe declarar con **set<tipo> nombre;**

Para insertar valores en un set, es necesario utilizar la función **insert()**. Si se llama insert para un valor que ya existe dentro del set no pasa nada, simplemente es ignorado. Para checar si algun valor es miembro de un set, se puede llamar **count()** sobre ese elemento y retornará 0 si no existe o 1 si sí existe.

El siguiente código lee varios números de la consola y luego checa si 25 es uno de ellos.

Listing 12: Sets

```
#include <iostream>
#include <set>

using namespace std;

int main() {
    int n;
    cin >> n;
    set<int> entradas;
    for(int i = 0; i < n; i++) {
        int x;
        cin >> x;
        entradas.insert(x);
    }
    if(entradas.count(25) == 0) {
        cout << "25 no es miembro de este set" << endl;
    } else {
        cout << "25 si es miembro de este set" << endl;
    }
}
```

Igual que map, el set tiene una version desordenada llamada **unordered_set** que tiene accesos más rápidos a costo de más memoria.

# 2 Complejidad de tiempo

Ningun curso de programación con algoritmos esta completo sin el tema de complejidad de tiempo. Este concepto es fundamental para poder hacer buenos programas y es lo que permite a un programador resolver problemas de la manera más optima.

Vamos a analizar un programa que obtiene todos los números primos menores a N. Se sabe que un número M es primo siempre y cuando solo es divisible entre 1 y M. Para comprobar que M es primo, podemos hacer un ciclo for que itera de 2 a M - 1 y checa si M es divisible entre cualquiera de estos números. En caso que sí es divisible, sabemos que el número no puede ser primo y lo marcamos como un número compuesto.

Esta sería la implementación:

Listing 13: Probando si M es primo

```
#include <iostream>

using namespace std;

int main() {
    int m;
    cin >> m;
    bool primo = true; //Asumimos que es primo hasta encontrar un divisor
    for(int i = 2; i < m; i++) {
        if(m % i == 0) {
            primo = false; //M no es primo
        }
    }
    if(primo) {
        cout << m << " es primo" << endl;
    } else {
        cout << m << " no es primo" << endl;
    }
}
```

Si pruebas este código con números mayores o iguales a dos indicará si es o no es primo. Ahora queremos probar si todos los números de 2 a N son primos, podemos encerrar este código en otro ciclo for:

Listing 14: Encontrando primos menores que N

```
#include <iostream>

using namespace std;

int main() {
    int n;
    cin >> n;
```

```
    for(int m = 2; m < n; m++) {
        bool primo = true; //Asumimos que es primo hasta encontrar un divisor
        for(int i = 2; i < m; i++) {
            if(m % i == 0) {
                primo = false; //M no es primo
            }
        }
        if(primo) {
            cout << m << endl;
        }
    }
}
```

Al correr este código se debera de imprimir una lista de primos después de darle el número N. Pero si se usa N = 100,000 o 1,000,000 se puede notar que el código se tarda algunos segundos en completar, y entre más grande es N, más se tarda.

Nosotros queremos optimizar este código para que pueda encontrar todos los números primos menores a 1,000,000 en muy poco tiempo. Analizando el problema, se te ocurre imprimir 2, iniciar M en 3 en lugar de 2 y saltar en incrementos de 2 para asegurar que M siempre es impar, ya que el único primo par es 2. Al saltar en incrementos de dos, solo checaras si la mitad de los números son primos y razonas que esto debe ser dos veces más rápido porque eliminas la mitad de los pruebas. Asi queda la implementación:

Listing 15: ¿Optimizando?

```
#include <iostream>

using namespace std;

int main() {
    int n;
    cin >> n;
    cout << 2 << endl;
    for(int m = 3; m < n; m += 2) {
        bool primo = true; //Asumimos que es primo hasta encontrar un divisor
        for(int i = 2; i < m; i++) {
            if(m % i == 0) {
                primo = false; //M no es primo
            }
        }
        if(primo) {
            cout << m << endl;
        }
    }
}
```

Liga al código

Corres este programa y notas que imprime lo mismo que el programa anterior, pero también notas que dura practicamente lo mismo que el programa anterior también. ¿Por qué sera que al no probar la mitad de los números, el tiempo no se reduce a la mitad, si no que dura practicamente lo mismo?

Esto puede ser explicado con complejidades de tiempo.

## 2.1 Número de operaciones

Vamos a analizar el código anterior de los números primos. Imaginemonos que cada operación se tarda un nanosegundo en hacerse y que cada línea de código (sin incluir ciclos y funciones) cuenta como una operación. Si queremos contar el número de operaciones que nuestro programa hace cada vez que se ejecuta, podemos analizarlo línea por línea y obtener una expresión matemática.

Vamos a considerar que el programa inicia justo después de leer la entrada de N de la consóla y vamos a denotar X como el número de operaciones. Podemos ver que en la siguiente línea se llama cout, asi que podemos decir que X = 1 hasta ahorita. Luego, se ingresa en el primer ciclo, que corre (N - 3) / 2 veces. En cada uno de estos ciclos, se hace una operación que declara primo como cierto, luego se corre otro ciclo for. Sabemos que el segundo ciclo for corre M - 2 veces, y que consiste de 2 lineas de código, una comparación y una igualación. Luego, se sale de ese ciclo y se hacen dos operaciónes más, otra comparación y una impresión a la consola.

Fuera del primer ciclo, determinamos que hay una sola operación y el primer ciclo. Dentro del primer ciclo, se corren 3 líneas de código (la declaración de primo y su comparación para luego imprimir a la consola) y el segundo ciclo. Dentro del segundo ciclo, Se corren 2 líneas de código.

Podemos ir elaborando una formula matemática con estos conocimientos. Si consideramos A como el número de operaciones fuera del primer ciclo, B como el número de operaciones dentro del primer ciclo y C como el número de operaciones dentro del segundo ciclo, obtenemos este sistema de ecuaciones:

$$A = 1 + \frac{N-3}{2} * B \tag{1}$$

$$B = 3 + (M-2) * C \tag{2}$$

$$C = 2 \tag{3}$$

Juntando estas tres ecuaciones, se puede obtener X, el número de operaciones totales.

$$X = 1 + \frac{N-3}{2} * (3 + (M-2) * 2) \tag{4}$$

Esta ecuación no esta muy bonita y puede ser confusa. Como no nos interesa saber la cantidad exacta de operaciones, sino que un aproximado, podemos simplificar esta formula quitando algunas partes. Sabemos que este número será

muy grande si N es muy grande, asi que podemos ignorar las partes insignificantes.

Primero, se puede quitar el 1 que suma $\frac{N-3}{2} * (3 + (M-2) * 2)$ debido a que cuando N es muy grande, no nos va a importar. Luego, se puede quitar el 3 que suma $(M-2) * 2$ porque el mismo razón. También quitaremos el 3 que resta N y el 2 que resta M para obtener la siguiente expresión:

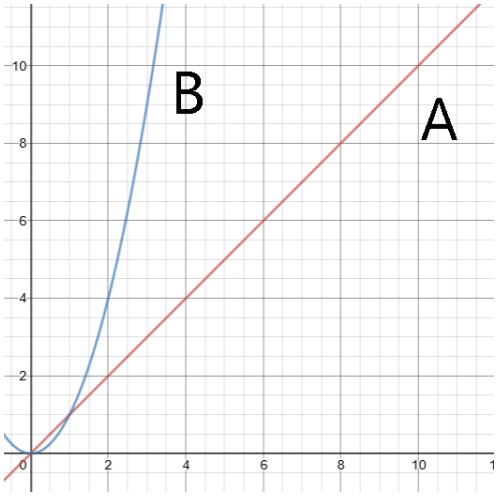
$$X \approx \frac{N}{2} * (M * 2) \tag{5}$$

Esta formula es muchisimo más practico, y puede simplificado aún más para darnos $X \approx N * M$. Si volvemos a repetir todo el proceso con nuestro programa anterior que sumaba de uno en uno, llegaremos al mismo resultado.

$$X \approx N * M \tag{6}$$

## 2.2 Límites

Nos conviene expresar el tiempo de ejecución de nuestros programas como una especie de función que depende de las entradas, porque asi podemos darnos una idea de como va a comportar a la hora de recibir entradas grandes. Digamos que comparamos dos programas A y B que calculan lo mismo, y el primero tiene ejecuta un número de operaciones $X = N$ y el segundo ejecuta $X = N^2$ instrucciones. Si comparamos las gráficas de ambas funciones, nos damos cuenta que el tiempo de ejecución de B aumenta a una velocidad más rapida que el tiempo de ejecución de A.




Esto significa que si le damos una entrada de aproximadamente 1,000,000 a A y a B, ¡A podría tardar un segundo mientras que B podría tardar 12 dias! Es muchisimo más conveniente tener la respuesta de un problema en un segundo que tener que esperar varios dias.

Pero esto es en el peor de los casos, en el mejor de los casos, se les puede dar una entrada de 1 y ambos programas terminanarán instantaneamente. Por esa rázon, se tiende a pensar en el peor de los casos.

Estas expresiones que hemos estado obteniendo se llaman las complejidades de tiempo de A y B, y existen maneras fáciles de determinar cual es la complejidad de tiempo de un algoritmo. Se tiende a referir al peor de los casos como O de cierta función, y se denota encerrando esa expresión entre parentesis. Por ejemplo, podemos decir que A tiene complejidad **O(N)** mientras que B tiene **O($N^2$)**.

La manera más fácil de determinar la complejidad de tiempo de un algoritmo es dividirlo en secciones de ciclos y sumar estos términos, y descartar todo lo que no sea el término más grande. Si tenemos un programa con un solo ciclo que se repite N veces, se puede decir que tiene complejidad **O(N)** porque lo que esta dentro de ese ciclo se repite N veces, y si hay dos ciclos anidados que se repiten N veces, tiene complejidad **O($N^2$)** porque se multiplican las N iteraciones del primer ciclo con las N iteraciones del segundo. En caso de tener un programa sin ciclos, se dice que tiene complejidad constante o **O(1)**, lo que significa que siempre se tardará el mismo tiempo en correr sin importar que tan grande es su entrada. Ahora se mostrará el ejemplo de dos ciclos separados que se repiten N veces:

Listing 16: Encontrando primos menores que N

```
#include <iostream>

using namespace std;

int main() {
    int n;
    cin >> n;
    cout << "Los numeros de 0 a N - 1:" << endl;
    for(int i = 0; i < n; i++) {
        cout << i << endl;
    }
    cout << "Los numeros pero ahora al reves:" << endl;
    for(int i = n - 1; i >= 0; i--) {
        cout << i << endl;
    }
}
```

La complejidad de ambos ciclos es O(N) y si se deciden sumar estas complejidades resultarían en la expresión O(2N), pero como la complejidad es una representación de la escala de tiempo que se abarca en un programa, el dos es un constante innecesario que se debería de quitar para dar O(N) denuevo. Esto significa que si se tienen 2, 3 o incluso más ciclos for que se estan sumando aqui y que no estan anidados, no tendrán un impacto grande en el tiempo de ejecución.

Si se decide combinar un ciclo de O($N$) con un ciclo de O($N^2$), al sumar los términos se obtendría O($N^2 + N$). Como $N^2$ es el término más significante, cuando N se vuelve muy grande $N$ se hará insignificante comparado con $N^2$ y no tendrá un gran impacto en el programa. Por esa razón, se debe quitar la $N$ de esa suma para dar O($N^2$).

Esto significa que sin importar que existen otros ciclos en el programa, el tiempo de ejecución siempre será proporcional a la sección más lenta del programa.

Con esto, podemos explicar la razón por la que no funcionaba nuestro programa que calculaba números primos y podemos elaborar una manera más eficiente de hacerlo.

Listing 17: ¿Optimizando?

```
#include <iostream>

using namespace std;

int main() {
    int n;
    cin >> n;
    cout << 2 << endl;
    for(int m = 3; m < n; m += 2) {
        bool primo = true; //Asumimos que es primo hasta encontrar un divisor
        for(int i = 2; i < m; i++) {
            if(m % i == 0) {
                primo = false; //M no es primo
            }
        }
        if(primo) {
            cout << m << endl;
        }
    }
}
```

Como podemos ver, hay un ciclo con complejidad O(N/2) y hay otro ciclo adentro del primero con complejidad O(M). Como se deben quitar las coeficientes a las complejidades, el primer ciclo tendrá complejidad O(N). Al multiplicar las dos complejidades, se obtiene que el programa es de complejidad O($N * M$). En el peor de los casos, $M = N$ asi que podemos sustituir M por N para obtener que nuestro programa tiene O($N^2$). Si queremos mejorar nuestro programa, debemos cambiar las Ns por términos más pequeños. No podemos quitar la primera N o el primer ciclo porque es la que imprimirá que números son primos, asi que debemos buscar modificar el segundo ciclo. En este, estamos iterando de 2 a M - 1 para ver si un valor es divisible entre M.

Una propiedad de los números que no son primos es que todos sus divisores estan en pares excepto la raíz cuadrada en caso de que sea un número cuadrático.

Asi que si no se han encontrado pares cuando se llega a la raíz de un número, se puede determinar que ese número no tiene divisores después de su raíz cuadrada y entonces debe ser primo. Esto nos dice que podemos reducir nuestra busqueda de divisores de 2 hasta la raíz cuadrada de M.

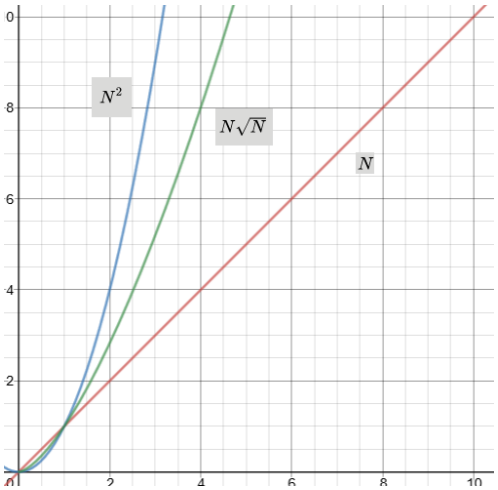
Listing 18: ¿Optimizando?

```
#include <iostream>
#include <cmath>

using namespace std;

int main() {
    int n;
    cin >> n;
    cout << 2 << endl;
    for(int m = 3; m < n; m += 2) {
        bool primo = true; //Asumimos que es primo hasta encontrar un divisor
        for(int i = 2; i <= sqrt(m); i++) {
            if(m % i == 0) {
                primo = false; //M no es primo
                break;
            }
        }
        if(primo) {
            cout << m << endl;
        }
    }
}
```

Con esta optimización, podemos determinar que nuestro programa tendra complejidad $O(N * \sqrt{M})$. Como $M = N$ en el peor de los casos, nuestra complejidad termina siendo $O(N\sqrt{N})$. Si comparamos las gráficas de $O(N^2)$, $O(N\sqrt{N})$ y $O(N)$, podemos ver que $O(N\sqrt{N})$ queda justo en medio, y esto significa que es más rápido que nuestra ultima solución $O(N^2)$ pero no tán rápido como $O(N)$.




Si corremos el código, podemos ver una mejora enorme.

### 2.3 Complejidad de memoria

Además de la complejidad de tiempo, existe la complejidad de memoria. Esta no es tan popular como la complejidad de tiempo debido a que la memoria tiende a ser abundante mientras que se prefiere minimizar el tiempo.

Muchas veces, un programa se puede hacer más eficiente utilizando más memoria o menos eficiente utilizando menos memoria, asi que es bueno saber que esto existe.

Si se tiene un valor N de entrada y se guardan N valores en un vector o en un arreglo, se puede decir que la complejidad de memoria de ese programa es **O(N)**, mientras que si el número de variables no cambia en un programa, se dice que su complejidad de memoria es **O(1)**.

## 3 Busquedas

Es conveniente realizar busquedas en arreglos para poder encontrar si ciertos datos estan guardados en ese arreglo y en que lugares. Pero tambien existen varios algoritmos o métodos que son mas rápidos que otros a costo de más trabajo.

### 3.1 Busqueda lineal

Esta es la busqueda más sencilla y común y consiste en tener un arreglo cualquiera e iterar sobre todos los valores hasta encontrar el deseado. Digamos que tenemos un arreglo con los valores 5, 7, 9, 1, 2 y queremos encontrar donde esta el 1.

Para hacer esto, podemos hacer un ciclo for desde el primer valor hasta el último valor y podemos parar el ciclo cuando ya encontramos el valor. En este caso, compararíamos 5, 7 y 9 hasta finalmente llegar al 1. Su implementación es muy sencilla:

Listing 19: ¿Optimizando?

```
#include <iostream>

using namespace std;

int main() {
    int valores[] = {5, 7, 9, 1, 2};
    for(int i = 0; i < 5; i++) {
        if(valores[i] == 1) {
            cout << "1 fue encontrado en el lugar " << i << endl;
            break;
        }
    }
}
```

Este algoritmo es la más sencilla pero a la vez puede ser la más ineficiente en ciertos casos. Por ejemplo, si queremos encontrar la posición de un 3 en el arreglo, tendriamos que recorrer todo el arreglo hasta determinar que este valor ni si quiera esta en este arreglo. Por esta rázon, este algoritmo tiene un peor caso de O(N), que es cuando se tiene que recorrer todo el arreglo.

## 3.2 Busqueda binaria

Este algoritmo es muchisimo más rápida que la busqueda lineal debido a que tiene una mejor complejidad de tiempo, pero requiere que el arreglo este ordenado primero. Si queremos encontrar donde esta el valor 8 en un arreglo 1, 3, 4, 5, 6, 7, 8, 9, 10 y sabemos que esta ordenado, podemos hacer una adivinación y probar el valor que esta justo en medio.

Si este es menor que nuestro valor, sabemos que 8 debe estar en la segunda mitad del arreglo, pero si es mayor sabemos que 8 esta en la primera mitad. Tambien puede ser que 8 es el valor que esta justo en medio, en ese caso no se tendría que hacer nada más porque ya habríamos encontrado nuestro valor.

Vamos a probar este algoritmo con nuestro arreglo. Primero comparamos 8 con 6 y nos damos cuenta que 8 es mayor a 6, asi que podemos descartar todos los valores de la izquierda e incluyendo a 6. 1, 3, 4, 5, 6 se quedan descartados y nos sobra 7, 8, 9, 0. Cuando el número de elementos es par, se puede escoger el número a la derecha o el número a la izquierda dependiendo de la preferencia de tu algoritmo. En este caso, escogeré el número de la derecha 9 y lo compararé con 8, y puedo determinar que todo lo que esta a la derecha de ese 9 debe ser descartado junto con el 9. 9, 10 quedan descartados y nos queda 7, 8. Probamos el valor de la derecha y obtenemos 8, el valor que estabamos buscando en solo 3 busquedas.

Si comparamos la busqueda lineal con la busqueda binaria, la lineal hubiera encontrado el 8 después de 7 operaciones mientras que la binaria lo encontró después de 3 operaciones.

Si se analiza la complejidad de tiempo de este algoritmo, obtenemos que es O(log N) o logaritmo base 2 de N, que es mucho mejor que N. Por ejemplo, si escogemos N = 1,000,000 podemos ver que log N es 9. Esto significa que en un arreglo con un millón de valores una busqueda lineal hará un millón de operaciones en el peor de los casos mientras que en una busqueda binaria se harán solo 9 operaciones.

## 3.3 Busqueda alfabetica

Si se estan guardando strings y se quiere buscar una palabra, se pueden crear 26 arreglos de strings para cada letra, es decir un arreglo que guarda todas las palabras empezando con 'A', otro arreglo que guarda las palabras empezando con 'B', etc. y esto será muchisimo más rapido que una busqueda binaria si se mantienen ordenadas.