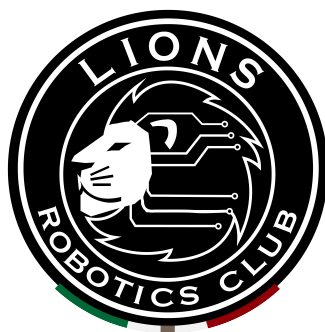


Manual Completo de Programación Competitiva

Lions R.C.

Agosto 2019



Contents

| | | |
|----------|--|-----------|
| 1 | Instalación de IDE para editar y compilar C++ | 3 |
| 1.1 | Code::Blocks [Windows] | 3 |
| 1.2 | Repl.it [Página web] | 3 |
| 1.3 | Visual Studio Community [Windows y Mac] | 4 |
| 1.4 | Visual Studio Code + Terminal [Mac y Linux] | 4 |
| 2 | ¿Mi primer programa? | 5 |
| 2.1 | Donde empezar | 5 |
| 2.2 | Comentarios | 8 |
| 3 | Leyendo y escribiendo | 9 |
| 4 | Operaciones básicas | 10 |
| 4.1 | Suma, resta, multiplicación, división | 10 |
| 4.2 | Modulación | 11 |
| 5 | Tipos de datos | 11 |
| 5.1 | Númericos | 11 |
| 5.2 | Signed y unsigned | 11 |
| 5.3 | Caracteres y cadenas | 12 |

| | | |
|-----------|---------------------------------------|-----------|
| 5.4 | Booleanos | 12 |
| 5.5 | Arreglos | 12 |
| 6 | Funciones | 13 |
| 7 | Flujo de datos | 14 |
| 7.1 | Condiciona1 if | 15 |
| 7.2 | Condiciones | 17 |
| 7.3 | Ciclo for | 18 |
| 7.4 | Ciclo while | 19 |
| 7.5 | Break | 20 |
| 7.6 | Continue | 21 |
| 8 | Errores comunes | 22 |
| 8.1 | Falta de inicialización | 22 |
| 8.2 | Ciclos con otras variables | 23 |
| 8.3 | Error por uno | 24 |
| 8.4 | Segmentation fault | 25 |
| 8.5 | Sobreflujos | 25 |
| 8.6 | Errores de redondeo | 26 |
| 8.7 | Faltas de memoria | 27 |
| 9 | Estructuras de datos sencillos | 28 |
| 9.1 | Vectores | 28 |
| 9.2 | Pares | 31 |
| 9.3 | Mapas | 32 |
| 9.4 | Sets | 34 |
| 10 | Complejidad de tiempo | 35 |
| 10.1 | Número de operaciones | 37 |
| 10.2 | Límites | 38 |
| 10.3 | Complejidad de memoria | 42 |
| 11 | Busquedas | 42 |
| 11.1 | Busqueda lineal | 43 |
| 11.2 | Busqueda binaria | 43 |
| 11.3 | Busqueda alfabetica | 44 |
| 12 | Estructuras de datos avanzadas | 44 |
| 12.1 | Stack | 44 |
| 12.2 | Queue | 46 |
| 12.3 | Deque | 47 |
| 12.4 | Priority queue | 49 |
| 13 | Recursividad | 49 |

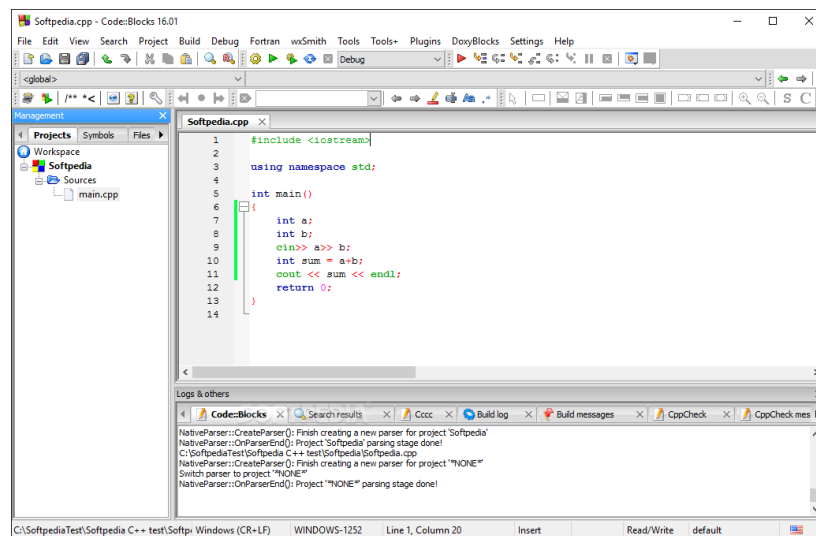
| | |
|---|-----------|
| 14 Ordenamientos | 52 |
| 14.1 Ordenamiento por selección | 52 |
| 14.2 Ordenamiento por mezcla | 54 |
| 14.3 Ordenamiento estandar de C++ | 56 |
| 15 Programación dinámica | 58 |

1 Instalación de IDE para editar y compilar C++

1.1 Code::Blocks [Windows]

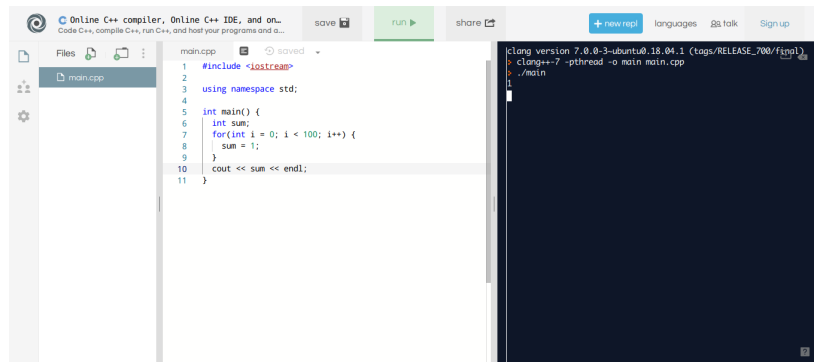
Si desea instalar un programa para compilar código facilmente, puede descargar Code::Blocks de la página oficial: <http://www.codeblocks.org/>

Este programa permite al usuario escribir y compilar código en Windows sin tener que instalar las diversas herramientas de compilación a mano. Para descargarlo, debe ir a la sección de Downloads y escoger Binaries, y de ahí descargar la primera opción (codeblocks-17.12-setup.exe).



1.2 Repl.it [Página web]

Repl.it es una página web que te permite probar código desde cualquier plataforma, la unica desventaja puede ser que ocupes una cuenta para guardar tu código. Para usarlo, debe entrar a la página <https://repl.it/languages/cpp>

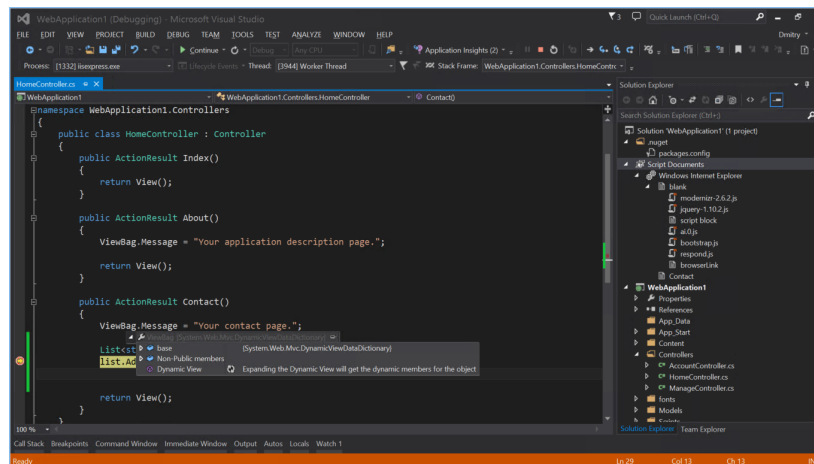


1.3 Visual Studio Community [Windows y Mac]

Esta es la versión gratis del software famoso de Microsoft llamado Visual Studio, te permite compilar código en muchos lenguajes y ofrece herramientas avanzadas de análisis de código.

Se puede descargar en la página oficial <https://visualstudio.microsoft.com/vs/community/> y te permite usarlo siempre y cuando estes ingresado con una cuenta de Microsoft. Si no tienes una cuenta de Microsoft y quisiera utilizar este programa, puedes registrarte en <https://account.microsoft.com/account?lang=es-MX>

A pesar de que esta instalación es más tardado, es una opción bastante profesional y buena si estas interesado en programar cosas más adelante.

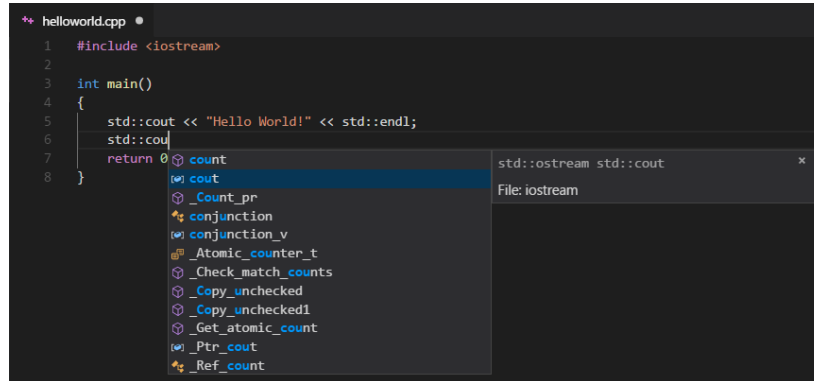


1.4 Visual Studio Code + Terminal [Mac y Linux]

El programa de Visual Studio Code es muy popular para la programación, tiene soporte para todos los lenguajes y es completamente gratis, con la desventaja de que no puede compilar código.

Si estas en Windows, puedes utilizar este programa pero debes instalar algún emulador de terminal de sistemas Unix como mingw para poder compilar tu código.

Si estas usando Mac o Linux, puedes compilar un programa de C++ desde la terminal. Para hacer esto, debes asegurarte de tener instalado el paquete *gcc* y debes generar un archivo ejecutable a partir del código con el comando *c++*



```
helloworld.cpp
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Hello World!" << std::endl;
6     std::cout
7     return 0;
8 }
```

The screenshot shows a code editor with a file named 'helloworld.cpp'. The code contains a C++ program that prints 'Hello World!'. On line 6, 'std::cout' is followed by an opening curly brace '{' and the cursor is positioned before the closing curly brace '}'. An autocomplete menu is open, showing various options including 'cout', '_Count_pr', 'conjunction', 'conjunction_v', '_Atomic_counter_t', '_Check_match_counts', '_Copy_unchecked', '_Copy_unchecked1', '_Get_atomic_count', '_Ptr_count', and '_Ref_count'. The 'cout' option is highlighted. To the right of the code editor, there is a small panel showing 'std::ostream std::cout' and 'File: iostream'.

2 ¿Mi primer programa?

2.1 Donde empezar

Antes de crear un programa, debes asegurarte de tener en mente lo que quieres que haga. Pero esto es algo muy difícil si vas empezando, ¡porque no tienes idea de lo que puedes hacer con un programa!

Así que durante estas próximas seis semanas, les daremos problemas de programación similares a los que aparecen en competencias más difíciles para que puedan aprender como crear sus propios programas y resolver problemas reales.

El programa más sencillo es uno que no hace nada; se abre y se cierra instantáneamente. Pero eso es aburrido y no tiene sentido. ¿Para qué quiero un programa que no hace nada?

Tú tienes que decirle al programa que quieras que haga, y el programa seguirá tus instrucciones sin cuestionarlas, de tal grado que a veces cometerá errores porque tú no esperabas que fuera a hacer todo al pie de la letra de como le dijiste.

Vamos a intentar construir un programa que sume dos números. Antes de empezar tu programa tendrás que escribir cuatro líneas muy importantes. Si quitas alguno de estos cuatro líneas, tu programa dejará de funcionar y te arrojará un error, y ahorita es muy temprano para poder decirles exactamente lo que hacen estas líneas. Cuando tengan más experiencia, podrán cambiarlos a su gusto.

Listing 1: Las cuatro líneas esenciales

```
#include <iostream>

using namespace std;

int main() {

}
```

Ahí esta su primer programa, él que les dije que se abre y se cierra sin hacer nada. Por ahora, todo su código debe ir entre los dos llaves.

Tenemos que decirle a nuestro programa que queremos dos números enteros, los cuales serán sumados después. Para cada número entero que queremos, tenemos que asignarle un nombre.

Para hacer esto, escribimos una línea para cada variable con `int` seguido por el nombre de la variable después (sin espacios y sin caracteres especiales). Finalmente, le ponemos un punto y coma después de cada línea para decirle que eso es lo único que queremos hacer en esa línea.

Listing 2: Dos variables

```
#include <iostream>

using namespace std;

int main() {
    int a;
    int b;
}
```

Y así tenemos dos números. Podemos nombrarlos como queramos, por ejemplo: `numeroA` y `numeroB`, `alfa` y `beta`, `n1` y `n2`, etc. Si corremos el código, no pasa nada. Esto es porque no le hemos dicho al programa que queremos que haga con estos dos números, ni le hemos dicho que valores tienen.

En una línea, podemos escribir `a = 5;` para decirle que queremos que la variable `a` valga 5. También es válido modificar la línea `int b;` para que sea `int b = 2;`

Listing 3: Guardando valores

```
#include <iostream>

using namespace std;

int main() {
    int a;
    a = 5;
    int b = 2;
}
```

Es de gran importancia saber que el programa lee el código desde arriba hacia abajo, así que no podemos poner la línea que dice `a = 5;` antes de `int a;` porque no sabrá que `a` es una variable que definimos.

Ahora solo falta decirle que queremos que sume `a` y `b`.

Listing 4: Guardando valores

```
#include <iostream>

using namespace std;

int main() {
    int a;
    a = 5;
    int b = 2;
    a + b;
}
```

Este nuevo programa parece sumarlos, pero si se corre tampoco pasa nada. Esto es porque tenemos que decirle al programa que además de sumarlos, queremos ver esta suma. Si no, calculará `a + b`, no sabrá que hacer con esa suma y lo descartará.

Para hacer que nos muestre este dato, tenemos que utilizar algo llamado `cout`, que no es nada más que un comando que nos muestre datos.

Para usar `cout`, se debe escribir `cout << valor;` donde `valor` es lo que quieres ver.

Listing 5: Tu primer programa

```
#include <iostream>

using namespace std;

int main() {
    int a;
    a = 5;
    int b = 2;
    cout << a + b;
}
```

[Liga al código](#)

Y de esa manera, tenemos nuestro primer programa. Hay miles de maneras de hacer este mismo código, y todos son iguales de válidos. Aquí mostraremos algunos ejemplos:

Listing 6: ¿El mismo programa?

```
#include <iostream>
```

```
using namespace std;

int main() {
    int a = 5, b = 2;
    cout << a + b;
}
```

Listing 7: ¿El mismo programa?

```
#include <iostream>

using namespace std;

int main() {
    int a;
    int b;
    int c;
    a = 5;
    b = 2;
    c = a + b;
    cout << c;
}
```

Listing 8: ¿El mismo programa?

```
#include <iostream>

using namespace std;

int main() {
    cout << 2 + 5;
}
```

2.2 Comentarios

C++ te permite escribir comentarios en medio de tu código y estos pueden servir para describir lo que hace cierto código, para recordarte de hacer algo en cierta parte o para mantener tu código ordenado.

El tipo de comentario más popular es el de una sola línea, Estos se inician con dos // seguidos por tu nota.

Listing 9: Comentario

```
#include <iostream>

using namespace std;

//Los comentarios pueden ir en lineas separadas
```



```

int main() {
    //En cualquier parte del codigo
    int a; //Incluso aqui!
    int b;
    int c; //Esta variable es la suma de a y b
    a = 5;
    b = 2;
    c = a + b; //Aqui se hace la suma
    cout << c; //Aqui se imprime el resultado
}

```

Tambien existen los comentarios de multiples líneas, estos se deben iniciar con `/*` y se deben terminar con `*/`

Listing 10: Comentario

```

#include <iostream>

using namespace std;

/* Esto es un comentario de multiples lineas
Puedes escribir cuantas lineas quieras aqui
Tambien sirve para temporalmente deshabilitar codigo
*/
int main() {
    int a; /* Tambien debes saber que pueden ocupar una sola linea */
    int b;
    int c;
    a = 5;
    b = 2;
    c = a + b;
    cout << c;
}

```

3 Leyendo y escribiendo

A la hora de compilar y ejecutar tu programa, seguramente notaste que se abre una consola con todo lo que decidiste ver. Esto es porque C++ corre dentro de esa consola (Tu compilador ejecuta comandos para abrir el programa sin que tú veas).

Esto te permite leer y escribir datos mientras que tu programa corre. Ya se ha visto que para escribir datos, se puede utilizar `cout` (que significa `c + out` o output de C) seguido por la variable que quieres "imprimir" a la consola.

Para leer datos, se ocupa tener una variable para almacenar ese dato de entrada, luego se debe utilizar `cin` (que significa `c + in` o input de C) seguido por esa variable.

Estas dos herramientas te permiten crear programas generales que resuelven problemas específicos.

Digamos que queremos crear un programa que lee dos números enteros (sin importar cuales sean) y que imprime su suma. Podemos crear este programa utilizando `cin` y `cout` para que el usuario diga cuales dos números quiere que sume.

Listing 11: Suma

```
#include <iostream>

using namespace std;

int main() {
    int a, b;
    cin >> a >> b;
    cout << a + b << endl;
}
```

[Liga al código](#)

Cuando ejecutas tu código, verás que la consola esta esperando una entrada. Puedes ingresar dos números separados por espacios y picarle enter o poner el primer número, picar enter, poner el segundo y picarle enter denuevo. Al terminar de hacer eso, verás que tu programa hace la suma instantaneamente y que lo despliega debajo de los números que le diste.

4 Operaciones básicas

4.1 Suma, resta, multiplicación, división

Como se sabe, puedes sumar, restar, multiplicar y dividir con los símbolos `+`, `-`, `*` y `/`.

Además de estos, si quieres guardar el resultado de una operación con una variable en esa misma variable, puedes simplificar la notación utilizando los operadores `+=`, `-=`, `*=` y `/=`.

```
int x += 5;
int x -= 8;
int x *= 2;
int x /= 6;
```

Para sumar y restar solo uno, la notación es aun mas sencilla:

```
x++;
x--;
```

4.2 Modulación

Una operación muy útil e importante es la que se llama modulación, este se utiliza para sacar el residuo de una división con el símbolo %.

Digamos que queremos el residuo de 13 dividido por 5, esto lo sacamos simplemente con **13 % 5;**

5 Tipos de datos

A comparación de otros lenguajes de programación, en C++ tú tienes que definir que tipo de dato quieres guardar en una variable.

5.1 Numéricos

En C++, los números enteros y los números decimales se guardan en dos tipos de variables distintos. Esto es porque las variables que manejan números decimales no son completamente precisos (tienen un margen de error), mientras que los datos enteros sí lo son. Además, los números decimales están representados de una manera completamente distinta en binario que los enteros, y esto tiene sus ventajas y desventajas.

Los números enteros se conocen como **int** o integers, y son capaces de guardar cualquier número desde -2,147,483,648 hasta 2,147,483,647.

Los números decimales se conocen como **float** o números flotantes, en teoría son capaces de guardar números hasta 3.4028235×10^{38} , con la única desventaja que el error incrementa dependiendo del tamaño del número.

También existe una variante de **float** conocido como **double**, que tiene el doble de precisión y es capaz de guardar números hasta 10^{308} .

Estos son los tipos de datos numéricos más comunes, pero existen más con distintos propósitos. También existe **long long int**, una variante de **int** que es capaz de guardar números muchísimo más grandes (desde -9,223,372,036,854,775,808 hasta 9,223,372,036,854,775,807), a cambio que ocupa el doble de memoria.

También existe **long long double**, con más precisión que un **double**.

Raramente, tendrás que hacer cálculos con números más grandes que los permitidos por estos datos, o con mayor precisión. Existen librerías externas que pueden ser utilizados para manejar números de escala inmensa.

5.2 Signed y unsigned

Existe una posibilidad que trabajarás con tipos de datos que son signed y unsigned, y estos definen si una variable puede guardar números negativos o solo positivos.

Todos los tipos de datos numéricos empiezan como signed a menos que defines lo contrario. Esto significa que puede guardar números negativos y números positivos. Si quisieras convertir un tipo de dato en unsigned para que solo maneje números positivos, entonces puedes escribir unsigned antes del tipo de dato. Por ejemplo: **unsigned int a = 5;**

A la hora de hacer una variable unsigned, el máximo se convierte en el doble de lo que era antes. Esto es porque la mitad de los números fueron utilizados para representar los números negativos en la representación signed, y esa mitad se junta con la otra mitad para hacer el doble. Es decir, si el mayor número que se puede guardar en un **int** es 2,147,483,647, el mayor número que se puede guardar en un **unsigned int** es 4,294,967,295.

Esto se tiende a usar para manejar datos que se saben que nunca serán negativos, como temperaturas.

5.3 Caracteres y cadenas

C++ es capaz de guardar texto letra por letra en variables que se llaman caracteres y cuerdas. Un carácter o **char** es una variable que contiene una sola letra o símbolo conforme al estándar ASCII, mientras que una cuerda o un **string** es una variable que guarda mensajes completos.

Para asignar un valor a un carácter, se debe colocar una sola comilla antes y después de la letra: **char estrella = '*';**

Para asignar un valor a un string, se debe utilizar doble comilla antes y después del texto: **string mensaje = "Hola mundo!";**

5.4 Booleanos

Estos tipos de datos son muy útiles porque solo guardan dos valores: verdadero o **true** y falso o **false**.

Para asignar un booleano, se utiliza el término **bool**. Por ejemplo: **bool decirMentira = false;**

5.5 Arreglos

Cuando estamos trabajando con muchas variables a la vez, las cosas se pueden complicar y se puede requerir guardar un número grande de cosas. Para no tener que escribir cientos de variables para una sola tarea, se pueden crear arreglos, que guardan una cantidad predeterminada de datos.

Se pueden hacer arreglos de cualquier tipo de dato y su nomenclatura consiste en poner dos corchetes [después del nombre con el número de elementos que se desea tener en ese arreglo.

Digamos que queremos un arreglo de 100 enteros, eso se podría definir como **int valores[100];**

Luego, para modificar o leer uno de estos valores, se puede llamar **valores[N]** donde N es el índice del elemento.

Es muy importante mencionar que el primer elemento tiene índice 0 y el último elemento tendría índice 99. Los arreglos siempre empiezan desde cero y incrementan hasta el valor del tamaño menos uno. Si se intenta guardar o ver un valor que no está en este rango, podrá causar errores en su programa.

Las cuerdas funcionan como arreglos, y se puede ver el carácter en la posición N utilizando el mismo sintaxis.

6 Funciones

Las funciones son muy utiles para cuando tienes que repetir código similar muchas veces. Estos son similares a funciones matemáticas, reciben cuantas variables de entrada quieres que tengan, haran una operación con estas entradas y regresará una sola variable de salida.

Para una función, se dice que se "regresa" su salida porque para indicar que se quiere terminar la función, se utiliza el operador *return* seguido por el valor de la variable de salida.

Un ejemplo sencillo es una función que suma dos números enteros A y B y que regresa otro entero C, la suma. Esto lo podemos implementar de la siguiente manera:

Listing 12: Función de suma

```
int suma(int a, int b) {  
    int c = a + b;  
    return c;  
}
```

Como se puede ver, se inicia una función con un tipo de dato que corresponde al tipo de dato de la salida. Como `int` va antes que el nombre de la función (`suma`), nos dice que queremos que regrese un entero.

Luego, sigue el nombre de la función y dos paréntesis. Dentro de los paréntesis van las variables de entrada separadas por comas. Como tenemos un entero `a` y otro entero `b`, estamos diciendo que para correr esta función el usuario debe dar dos números en ese orden, los cuales serán guardados en `a` y `b`.

Luego, siguen llaves que encapsulan el código de la función. Aquí es donde hacemos nuestros cálculos con nuestras variables y regresamos el valor de la salida.

Declaramos otro entero `c` y le asignamos la suma de `a` y `b`, luego lo regresamos, es decir estamos diciendo que queremos que `c` sea nuestra salida.

Todas las funciones tienen que estar colocados antes de `main` de tal manera:

Listing 13: Función de suma

```
#include <iostream>  
  
using namespace std;  
  
int suma(int a, int b) {  
    int c = a + b;  
    return c;  
}  
  
int main() {  
  
}
```

Ahora, podemos llamar nuestra función dentro de main dándole los parámetros que queremos:

Listing 14: Función de suma

```
#include <iostream>

using namespace std;

int suma(int a, int b) {
    int c = a + b;
    return c;
}

int main() {
    cout << suma(3, 5) << endl;
}
```

[Liga al código](#)

Al ejecutar este código, nos desplegara un 8.
Otro ejemplo es una función que regresa la división de dos enteros:

Listing 15: Función de división

```
#include <iostream>

using namespace std;

float divide(int a, int b) {
    return (float)a / b;
}

int main() {
    cout << divide(5, 2) << endl;
}
```

[Liga al código](#)

Este código imprime 2.5, lo que hace es que recibe dos enteros a y b, y luego regresa su división como si fuera un número flotante. El (float) antes del a significa que quieres convertir la división en un número flotante antes de que redondea.

7 Flujo de datos

La pieza fundamental detrás de cualquier programa es su lógica, y no puede haber lógica sin flujo de datos. El flujo de datos consiste en correr ciertas piezas

de código bajo ciertas condiciones. Por ejemplo, si queremos saber si un número es par el código debe de tener dos secciones, uno que corre cuando sí es par y otro que corre cuando no es par.

7.1 Condicional if

La manera mas sencilla de comparar datos es con el condicional **if**. Este es capaz de correr código solo cuando se cumple cierta condición. Para utilizar este, se debe iniciar escribiendo **if** y se debe incluir una condición entre paréntesis, luego el código que se desea correr entre llaves.

Listing 16: Condiciones

```
#include <iostream>

using namespace std;

int main() {
    int a, b;
    cin >> a >> b;
    if(a + b > 5) {
        cout << "La suma es mayor a cinco" << endl;
    }
}
```

Aqui, tenemos una condición que checa si la suma de A y B es mayor a 5. En el caso de que sí se cumple, se imprimirá "La suma es mayor a cinco" y en caso que no, no se imprimirá nada.

Ahora modificaremos el código para que se imprima "La suma es menor o igual que cinco" cuando no se cumple la primera condición.

Listing 17: Condiciones

```
#include <iostream>

using namespace std;

int main() {
    int a, b;
    cin >> a >> b;
    if(a + b > 5) {
        cout << "La suma es mayor a cinco" << endl;
    }
    if(a + b <= 5) {
        cout << "La suma es menor o igual a cinco" << endl;
    }
}
```

Podemos simplificar este código con el condicional **else**, que se ejecuta cuando una condición **if** NO se cumple. Este se puede agregar después de la ultima llave y deberá tener otro par de llaves después.

Listing 18: Else

```
#include <iostream>

using namespace std;

int main() {
    int a, b;
    cin >> a >> b;
    if(a + b > 5) {
        cout << "La suma es mayor a cinco" << endl;
    } else {
        cout << "La suma es menor o igual a cinco" << endl;
    }
}
```

Digamos que ahora queremos checar si la suma es menor, igual o mayor que cinco. Se puede agregar un tercer condicional **else if** que ocurre cuando la primera condición falla pero otra condición se cumple. Se puede tener varios **else if** encadenados.

Listing 19: Else if

```
#include <iostream>

using namespace std;

int main() {
    int a, b;
    cin >> a >> b;
    if(a + b > 5) {
        cout << "La suma es mayor a cinco" << endl;
    } else if (a + b == 5) {
        cout << "La suma es igual a cinco" << endl;
    } else {
        cout << "La suma es menor a cinco" << endl;
    }
}
```

En este caso, primero se checa si el número es mayor que cinco, luego se checa si es igual a cinco, y finalmente por lógica si no cumple ninguna de estas dos condiciones debe ser menor que cinco.

7.2 Condiciones

Como se ha visto, existen maneras de comparar datos para verificar condiciones. Para checar si una variable es menor que otro, se debe usar el simbolo `<`, mientras que para checar si es mayor, se usa el simbolo `>`.

Tambien se puede ver cuando un valor es menor o igual a otro valor con `<=` y cuando es mayor o igual con `>=`. Nota que el signo de igual siempre va despues de la comparación de mayor o menor.

Para preguntar si dos valores son iguales, se utilizan dos signos de igual `==`, ya que al utilizar solo uno se estaria asignando un valor a la variable de la izquierda. Para checar si no son iguales, se debe usar un signo de admiración seguido por un signo de igual `!=`.

Finalmente, existen los operadores lógicos `&&` y `||` que pueden ser utilizados para juntar condiciones. El primero se utiliza para correr código siempre y cuando las condiciones pegadas a él sean ciertos, mientras que el segundo sirve cuando cualquiera de los dos son ciertos.

Listing 20: Condiciones

```
#include <iostream>

using namespace std;

int main() {
    int a, b;
    cin >> a >> b;
    if(a == b) {
        cout << "A_es_igual_a_B" << endl;
    }
    if(a > 5) {
        cout << "A_es_mayor_a_5" << endl;
    }
    if(b <= 10) {
        cout << "B_es_menor_o_igual_que_10" << endl;
    }
    if(b != -6) {
        cout << "B_no_es_igual_a_-6" << endl;
    }
    if(a < 1 && b > 1) {
        cout << "A_es_menor_que_1_y_B_es_mayor_que_1" << endl;
    }
    if(a == 1 || b == 1) {
        cout << "Una_de_las_variables_A_o_B_es_igual_a_1" << endl;
    }
}
```

[Liga al código](#)

7.3 Ciclo for

Digamos que quieres repetir un proceso cierta cantidad de veces pero no quieres tener que volver a escribir el código cientos de veces con cambios pequeños. Se puede automatizar este proceso con el uso de ciclos for.

Un ciclo for repite código cierta cantidad de veces y tiene una variable que incrementa o decrementa con cada turno del ciclo, esta variable puede ser útil porque puede indicar el ciclo actual.

Digamos que ocupas saber la suma de 1 a 100. Esto se puede escribiendo cien líneas que suman cada valor o escribiendo una línea dentro de un ciclo for.

Aquí les mostramos el ejemplo con cien líneas:

Listing 21: Suma línea por línea

```
#include <iostream>

using namespace std;

int main() {
    int suma = 0;
    suma += 1;
    suma += 2;
    suma += 3;
    ...
    suma += 98;
    suma += 99;
    suma += 100;
    cout << suma << endl;
}
```

Y esto es la simplificación con el ciclo for:

Listing 22: Suma con ciclo for

```
#include <iostream>

using namespace std;

int main() {
    int suma = 0;
    for(int i = 1; i <= 100; i++) {
        suma += i;
    }
    cout << suma;
}
```

Como se puede ver, el ciclo for ahorra muchas líneas y puedes cambiar los parámetros de iteración.

Se debe notar que el for tiene un sintaxis similar al if, excepto que tiene tres divisiones en la parte condicional.

La primera parte corresponde a la asignación del valor inicial, aquí se debe declarar e igualar una variable al valor inicial del ciclo. A esta variable se le tiende a dar el nombre **i**, o **j** cuando existe otro ciclo que este usando **i** al mismo tiempo.

Despues, sigue una condición que al no cumplirse terminará el ciclo. En este ejemplo, se utilizó **i <= 100** para asegurarnos que una vez que **i** llegue a 100, se termine el ciclo.

Finalmente, la tercera parte debe modificar la variable del ciclo para que progrese hasta que deje de cumplirse la condición en la segunda parte.

Con esta información, podemos ver que primero se guarda 1 en la variable **i**, luego se checa en cada iteración del ciclo que no sea mayor que 100, luego se suma 1 a la variable **i**.

Si queremos hacer un ciclo que nos de los números 0 a 4 en orden ascendiente, podemos modificar el código de la siguiente manera:

Listing 23: Ciclo for

```
#include <iostream>

using namespace std;

int main() {
    for(int i = 0; i < 5; i++) {
        cout << i << endl;
    }
}
```

Y en caso de que queremos que nos den los valores de 4 a 0 en orden descendiente, podemos modificarlo para que reste en lugar de sumar.

Listing 24: Ciclo for

```
#include <iostream>

using namespace std;

int main() {
    for(int i = 4; i >= 0; i--) {
        cout << i << endl;
    }
}
```

7.4 Ciclo while

Si no te interesa la cantidad de veces que corra un ciclo y simplemente quieres repetir algo mientras que una condición sea cierto, se puede utilizar el ciclo while.

Se debe de tener cuidado con los ciclos while debido a que pueden correrse infinitamente.

Para crear un ciclo while, se utiliza el mismo sintaxis que el condicional if con el nombre **while**. El código dentro de las llaves seguirá corriendo hasta que la condición sea falsa.

Listing 25: Ciclo while

```
#include <iostream>

using namespace std;

int main() {
    int potenciaDos = 1;
    while(potenciaDos < 100) {
        potenciaDos *= 2;
    }
    cout << potenciaDos << endl;
}
```

Como se ve en este ejemplo, el código calculará todas las potencias de dos hasta llegar a uno que no sea menor a 100, en este caso 128, luego lo imprimirá en la consola.

7.5 Break

Si tienes algún ciclo y quisieras salirte de ese ciclo bajo una condición que no esta incluida en la parte donde se declara ese ciclo, se puede utilizar el termino **break**.

Digamos que queremos hacer un ciclo que termine después de 100 iteraciones o después de que se encuentren tres números consecutivos que suman 20. Esto se puede implementar con un ciclo for y un break de la siguiente manera:

Listing 26: Break

```
#include <iostream>

using namespace std;

int main() {
    bool existeSuma = false;
    for(int i = 0; i < 100; i++) {
        int suma = i + (i + 1) + (i + 2);
        if(suma == 20) {
            existeSuma = true;
            cout << "La suma es: " << endl;
            cout << i << ", " << i + 1 << ", " << i + 2 << endl;
            break;
        }
    }
}
```

```

    }
}
if(existeSuma == false) {
    cout << "La suma no existe!" << endl;
}
}

```

[Liga al código](#)

Como se puede ver, combinamos `break` con un booleano para ver si se llega a cumplir la condición de la suma, y en caso de que sí se cumple imprimimos la suma y dejamos de buscar más secuencias. En el caso contrario, no modificamos el booleano e imprimimos que la suma no existe.

7.6 Continue

A veces queremos evitar correr cierto código bajo una condición, pero no queremos tener que incluir nuestro código en muchas condicionales `if`. En un ciclo `for` o `while` se puede utilizar **continue** cada vez que se quiere saltar al siguiente ciclo.

Digamos que estamos iterando sobre todos los números menores a 100 y queremos saltarnos todos los números pares y divisibles entre 3. Esto se puede hacer de distintas maneras:

En la primera se puede tener un solo caso `if` con dos condiciones unidos con el operador `&&`

Listing 27: Condiciones con `and`

```

#include <iostream>

using namespace std;

int main() {
    for(int i = 0; i < 100; i++) {
        if(i % 2 != 0 && i % 3 != 0) {
            cout << i << endl;
        }
    }
}

```

Para la segunda manera se puede tener dos ifs anidados:

Listing 28: Dos condiciones

```

#include <iostream>

using namespace std;

int main() {

```

```

    for(int i = 0; i < 100; i++) {
        if(i % 2 != 0) {
            if(i % 3 != 0) {
                cout << i << endl;
            }
        }
    }
}

```

Y para la tercera podemos usar dos casos if con continue:

Listing 29: Continue

```

#include <iostream>

using namespace std;

int main() {
    for(int i = 0; i < 100; i++) {
        if(i % 2 == 0) {
            continue;
        }
        if(i % 3 == 0) {
            continue;
        }
        cout << i << endl;
    }
}

```

[Liga al código](#)

Al correr estos tres códigos, se verán los mismos resultados.

8 Errores comunes

8.1 Falta de inicialización

Este error es uno de los más graves y más difíciles de detectar debido a que la lógica puede estar completamente correcta, el programa se compila sin errores y en algunos casos tus variables te podrán dar los resultados correctos.

Esto pasa cuando defines una variable pero no le das un valor inicial. Por ejemplo:

Listing 30: Error de inicialización

```

#include <iostream>

using namespace std;

```

```

int main() {
    int i = 1;
    int suma;
    while(i <= 100) {
        suma += i;
        i++;
    }
    cout << suma << endl;
}

```

[Liga al código](#)

Este programa suma los números de 1 a 100 y los va guardando en la variable suma. Se esperaría que el resultado fuera 5050, pero si lo corres te despliega -1428474. Si vuelves a correr el código, ahora ves 82934. Este valor siempre cambia.

Esto ocurre debido a que nunca inicializamos o le damos un valor inicial a la variable suma. A la hora de declarar la variable, le estamos asegurando un espacio en la memoria del sistema pero esto no significa que ese espacio tenga guardado un cero. Puede ser que algún proceso anterior haya tenido que usar ese espacio y que haya dejado un valor que no nos interesa ahí.

La razón por la que siempre cambia es porque la memoria siempre esta siendo utilizado por otros procesos (Google Chrome, Word, etc.) y cada vez que corre el programa agarrara el primer espacio libre que encuentra para la variable suma, que no siempre será la misma.

Existe una posibilidad de que tu compilador automaticamente detecte que suma no fue inicializado y le asigna el valor de cero después, esto puede provocar que el programa funcione en tu computadora pero a la hora de probarlo en otra falla.

Entonces para asegurar que esto nunca pase, siempre debes de tener la costumbre de inicializar tus variables a menos que sabes que serán sobrescritos por otro valor o leídos desde la consola usando *cin*.

8.2 Ciclos con otras variables

Este error pasa muy seguido cuando se utilizan ciclos for anidados y cuando uno olvida que haya declarado una variable antes.

Digamos que vas tan mal en matemáticas que se te olvidaron las tablas multiplicativas por completo, y túquieres crear un programa que los genera. Puedes hacer esto con dos ciclos for anidados, uno que calcule las filas y otra que calcule las columnas.

Listing 31: Ciclos con otras variables

```

#include <iostream>

using namespace std;

```

```

int main() {
    for(int i = 0; i <= 12; i++) {
        for(int j = 0; i <= 12; i++) {
            cout << i * j << "\t";
        }
        cout << endl;
    }
}

```

[Liga al código](#)

Al parecer este código esta bien, pero si miras fijamente verás que en el segundo ciclo estamos modificando i en lugar de j. El compilador asume que sabes lo que estas haciendo, así que no te arroja ningún error al tener distintas variables en un solo ciclo.

Para asegurar que esto no pase, siempre debes asegurarte de que solo estes modificando una variable por ciclo y que no hayas usado una variable con ese mismo nombre antes.

8.3 Error por uno

Este error tiende a ser muy fácil de arreglar y ocurre por fallas de lógica.

Digamos que quieres encontrar la suma de todos los cuadrados $1^2 + 2^2 + 3^2 \dots + n^2$, y creas un programa para calcularlo.

Listing 32: Error por uno

```

#include <iostream>

using namespace std;

int main() {
    int n;
    cin >> n;
    int suma = 0;
    for(int i = 0; i < n; i++) {
        suma += i * i;
    }
    cout << suma << endl;
}

```

[Liga al código](#)

Corres tu programa con $n = 3$, y ya sabes que $1^2 + 2^2 + 3^2 = 1 + 4 + 9 = 14$, pero te dice que es 5. Ahora corres tu programa con $n = 4$ y ahora observas que la respuesta es 14 cuando debía ser 30. Esto ocurre porque estas contando de 0 a $n - 1$ (nota que el for tiene la expresión $i < n$ cuando debía ser $i \leq n$).

Esto pasó porque el for loop cuenta hasta n pero no lo incluye en la suma, y estos tipos de errores suelen ocurrir cuando se utilizan ciclos que inician desde 1 o cuando se debe incluir el ultimo valor en el proceso.

8.4 Segmentation fault

Este error es algo común y suele pasar cuando quieres sacar un valor que esta fuera de un arreglo.

Digamos que tenemos un código que le pregunta al usuario un valor y luego despliega el factorial de ese valor. El código asume que el usuario seleccionará un número entre 0 y 10, para luego desplegar su factorial el cual esta guardado en un arreglo.

Listing 33: Error de inicialización

```
#include <iostream>

using namespace std;

int main() {
    int factoriales[] = {1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800};
    cout << "Que factorial quisieras?" << endl;
    int deseado;
    cin >> deseado;
    if(deseado < 0) {
        cout << "No existe_" << deseado << "!";
    } else {
        cout << deseado << "_!" << factoriales[deseado] << endl;
    }
}
```

[Liga al código](#)

¿Ahora qué pasa si el usuario quiere saber el valor de 25565 factorial? Si ejecutas el código con tal valor verás que el programa se cierra inesperadamente y regresa el siguiente error: [1] 12937 segmentation fault (core dumped).

Esto ocurre porque estas intentando accesar el elemento 25565 de un arreglo que solo tiene 10 valores, lo que pasa es que estas intentando ver un lugar que no existe en la memoria de tu programa.

El 95% de las veces que verás un segmentation fault, será porque estas intentando modificar o ver un valor que esta fuera de un arreglo. Siempre debes asegurarte que un arreglo tiene suficiente espacio para guardar todos los posibles valores que ocupes.

8.5 Sobreflujos

Este error pasa cuando tienes un valor que es demasiado grande para una variable.

Ahora vamos a mejorar el código que habíamos hecho del factorial. En lugar de guardar los primeros 10 valores en un arreglo, vamos a generar cada valor con un ciclo.

Para checar que todo este bien, voy a desplegar los primeros 18 factoriales.

Listing 34: Sobreflujo

```
#include <iostream>

using namespace std;

int main() {
    int factorial = 1;
    for(int i = 1; i < 18; i++) {
        factorial *= i;
        cout << factorial << endl;
    }
}
```

[Liga al código](#)

Corres este código y te topas con una sorpresa, 17! es aparentemente -288522240. ¿Cómo fue que un factorial te dio un número negativo si siempre incrementa?

Lo que esta pasando es que el valor de 17! es demasiado grande para ser guardado en un valor de tipo int, si buscas en línea encontraras que un int solo puede guardar valores de -2,147,483,648 a 2,147,483,647, y 17! tiene un valor de 355,687,428,096,000.

El programa no te arroja ningun error, simplemente lo que hace es que cuando un número es más grande del valor máximo de 2,147,483,647, se traslapa a -2,147,483,648 y sigue sumando desde ese valor. Eso significa que si quiero guardo el valor de 2,147,483,649 en un int, me guardará en realidad el valor -2,147,483,647.

Si sigues imprimiendo más números, verás que se vuelven a hacer positivos y negativos debido a este traslape.

Para solucionar esto, debes reemplazar int por otro tipo de dato que pueda guardar números más grandes, como long long int o double.

8.6 Errores de redondeo

Este error pasa cuando se dividen dos enteros.

Sabes que $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots = 1$ y quisieras comprobarlo creando un programa. Lo implementas con el siguiente código:

Listing 35: Error de redondeo

```
#include <iostream>
```

```
using namespace std;

int main() {
    int suma = 0;
    int potencia = 2;
    for(int i = 1; i < 30; i++) {
        suma += 1 / potencia;
        potencia *= 2;
    }
    cout << suma << endl;
}
```

[Liga al código](#)

Pero cuando lo corres, te dice que tu suma es cero. Revisas la variable de potencia y te das cuenta que ese sí esta bien, pero la variable de suma se queda en cero.

Esto ocurre debido a que 1 y potencia estan siendo guardados como enteros, y cuando se dividen dos enteros el resultado siempre será otro entero redondeado para abajo. Es decir si hago $5/2$, esto me dará 2 porque el valor de 2.5 fue redondeado hacia abajo. Como potencia siempre es mayor que 1, esta división siempre se redondeara para abajo y dará cero.

Para corregir esto, se deben cambiar las variables de potencia y de suma de enteros a un tipo de dato flotante, es decir un número que permite puntos decimales.

8.7 Faltas de memoria

Esto sucede cuando tu programa ocupa demasiado espacio continuo y ocasiona un error.

A veces sucede que tienes que crear un arreglo con 10 millones de valores (es verdad, sí vas a ocupar algo así!) y te darás cuenta que tu programa deja de funcionar.

Listing 36: Faltas de memoria

```
#include <iostream>

using namespace std;

int main() {
    int arreglo[10000000];
    for(int i = 0; i < 10000000; i++) {
        arreglo[i] = i;
    }
    cout << "Arreglo_llenado" << endl;
}
```

[Liga al código](#)

Por suerte, esto se puede arreglar. Si cuentas los bytes, te darás cuenta que son 4 millones de bytes o 4 megabytes, y las computadoras modernas tienden a tener hasta 16 gigabytes de memoria RAM, así que no debe haber problema para conseguir 4 megabytes de memoria.

El problema viene del hecho que a la hora de crear un arreglo, el sistema busca 4 megabytes de espacio que estén libres y juntos dentro de la memoria del programa. Esto se puede arreglar preguntándole a tu sistema si puedes guardar estos 4 megabytes en una área especial extra de memoria llamado el heap, que viene siendo una área pública con muchísimo espacio, en lugar del espacio limitado que fue asignado a tu programa.

Para hacer que el programa busque dentro del heap, debes cambiar el arreglo a un puntero y utilizar **malloc**, **new** o alguna función que te permite obtener espacio del heap.

9 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, tendríamos que desplazar todos los demás datos que están enfrente hacia atrás, 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 explicarán algunas de las muchas estructuras de datos, y la siguiente semana se darán a conocer los demás.

9.1 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 específica de esta estructura llamada **vector**.

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

Listing 37: 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 38: Vectores

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

using namespace std;

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

Aquí 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 39: 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);
}
```

Aquí, 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 40: 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 41: Jugando con vectores

```

#include <iostream>
#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.

9.2 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 42: Declarando pares

```
#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 43: 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 44: 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.

9.3 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 búsqueda 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 45: 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 búsqueda de todos los amigos hasta encontrar a Juan. Esta es la simplificación con map:

Listing 46: 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 acceso 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 47: 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;
}
```

9.4 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 48: 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.

10 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 49: 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 50: 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 debiera 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 razones que esto debe ser dos veces más rápido porque eliminas la mitad de los pruebas. Así queda la implementación:

Listing 51: ¿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 prácticamente lo mismo que el programa anterior también. ¿Por qué será que al no probar la mitad de los números, el tiempo no se reduce a la mitad, si no que dura prácticamente lo mismo?

Esto puede ser explicado con complejidades de tiempo.

10.1 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 consola y vamos a denotar X como el número de operaciones. Podemos ver que en la siguiente línea se llama cout, así 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 líneas de código, una comparación y una igualación. Luego, se sale de ese ciclo y se hacen dos operaciones 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 fórmula 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:

$$A = 1 + \frac{N-3}{2} * B \quad (1)$$

$$B = 3 + (M-2) * C \quad (2)$$

$$C = 2 \quad (3)$$

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

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

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:

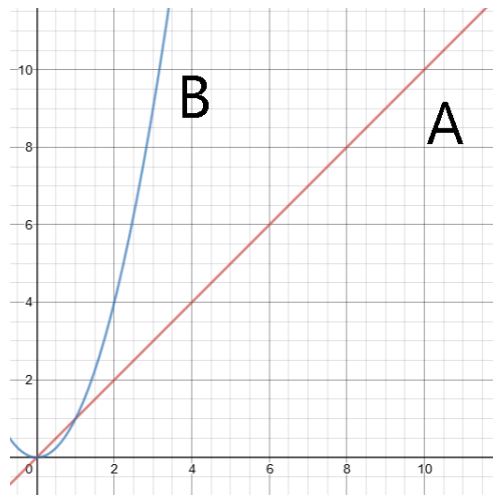
$$X \approx \frac{N}{2} * (M * 2) \quad (5)$$

Esta formula es muchísimo 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.

$$X \approx N * M \quad (6)$$

10.2 Límites

Nos conviene expresar el tiempo de ejecución de nuestros programas como una especie de función que depende de las entradas, porque así 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 días! Es muchísimo más conveniente tener la respuesta de un problema en un segundo que tener que esperar varios días.

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 terminarán instantáneamente. Por esa razón, 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 52: Encontrando primos menores que N

```
#include <iostream>

using namespace std;
```

```

int main() {
    int n;
    cin >> n;
    cout << "Los números de 0 a N-1:" << endl;
    for(int i = 0; i < n; i++) {
        cout << i << endl;
    }
    cout << "Los números pero ahora al revés:" << 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 $O(N)$ de nuevo. Esto significa que si se tienen 2, 3 o incluso más ciclos for que se están sumando aquí y que no están 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 significativo, 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 53: ¿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, $M = N$ así que podemos sustituir M por N para obtener que nuestro programa tiene $O(N^2)$. Si queremos mejorar nuestro programa, debemos cambiar las N s por términos más pequeños. No podemos quitar la primera N o el primer ciclo porque es la que imprimirá que números son primos, así 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. Así 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 54: ¿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;
        }
    }
}

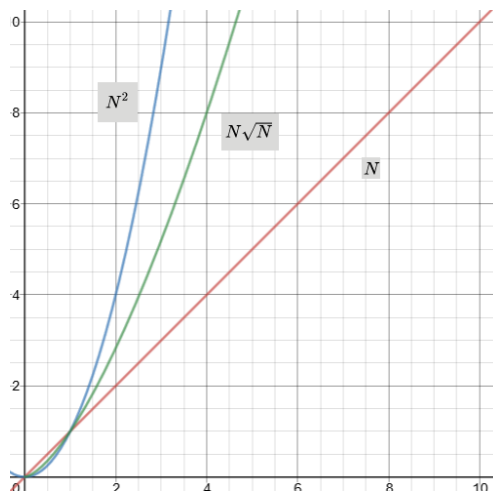
```

```

    }
  }
}

```

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 tan rápido como $O(N)$.



Si corremos el código, podemos ver una mejora enorme.

10.3 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, así 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 $O(1)$.

11 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.

11.1 Búsqueda lineal

Esta es la búsqueda 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, comparariamos 5, 7 y 9 hasta finalmente llegar al 1. Su implementación es muy sencilla:

Listing 55: ¿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, tendríamos que recorrer todo el arreglo hasta determinar que este valor ni si quiera esta en este arreglo. Por esta razón, este algoritmo tiene un peor caso de $O(N)$, que es cuando se tiene que recorrer todo el arreglo.

11.2 Búsqueda binaria

Este algoritmo es muchísimo más rápida que la búsqueda 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, así 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.

11.3 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.

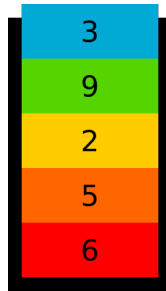
12 Estructuras de datos avanzadas

Hasta ahorita solo hemos visto estructuras de datos sencillas (vectores, pares, mapas y sets) que tienen el proposito de guardar valores de una manera conveniente para su acceso. Ahora se verán estructuras que guardan valores en cierta orden o configuración con el fin de simplificar programas más avanzadas.

12.1 Stack

El stack o la pila en español es una estructura de datos que, como indica su nombre, apila datos desde abajo hasta arriba. El concepto detrás de esta estructura es igual al de la vida real; si tienes una caja donde vas apilando hojas, debes sacar la hoja que esta hasta arriba antes de que puedas sacar la que esta debajo de esa y viceversa.

Digamos que tenemos 6 enteros, 6, 5, 2, 9, 3 y los insertamos en nuestra pila empezando con 6 y terminando con 3. Nuestra pila se verá de la siguiente manera:



Debido a la implementación interna de la pila, no podemos sacar un valor que este debajo de otro sin sacar todos los de arriba primero. Esta propiedad se llama **LIFO** en inglés o Last In First Out (el último en entrar es el primero en salir).

Si ahora sacamos todos los valores y los guardamos en un arreglo, veremos que se invirtió el orden de los datos debido a esta propiedad: 3, 9, 2, 5, 6.

Para incluir una pila en C++, se llama la librería **stack** y se declara con sintaxis **stack<tipo> nombre;**

Para agregar objetos a una pila, se puede utilizar la función **push(valor)** y para eliminar el dato que esta hasta arriba se debe hacer **pop()**. Para ver el dato que esta hasta arriba se puede utilizar **top()** y para ver cuantos elementos tiene se puede llamar a **size()**.

Vamos a crear un programa que invierte los valores que originalmente habíamos visto:

Listing 56: Pilas

```
#include <iostream>
#include <stack>

using namespace std;

int main() {
    stack<int> pila;
    int valores[] = {6, 5, 2, 9, 3};
    for(int i = 0; i < 5; i++) {
        pila.push(valores[i]);
    }
    while(pila.size() > 0) {
        cout << pila.top() << endl;
        pila.pop();
    }
}
```

[Liga al código](#)

Como se puede observar al correr el programa, los valores se imprimen al revés.

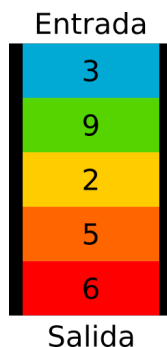
La pila no solo sirve para invertir cosas, sino que es útil para cuando quieres procesar los datos más recientes antes que las más antiguas.

Digamos que estás diseñando el buzón de una red social como Facebook o Twitter y tu quieres enseñarle a tus usuarios las publicaciones más recientes, pero también quieres eliminar las publicaciones vistas por el usuario para que no tengan que desplazarse mucho para ver más contenido.

Esto se puede implementar con una pila que va apilando las publicaciones más recientes y cada vez que el usuario se desplaza para abajo se quitan las publicaciones que están hasta arriba.

12.2 Queue

El queue o la fila es en cierto sentido el complemento de la pila. La fila, como también indica el nombre, funciona como una fila de datos donde el primero en entrar es el primero en salir. Esto se llama **FIFO** en inglés o First In First Out.



Como se puede observar, los datos están haciendo fila y saldrían en el mismo orden de la que entran, no se invierten. Al igual que una pila, un valor que está hasta atrás no puede salir primero.

Para implementarlo, se debe incluir la librería **queue** y se debe declarar como **queue<dato> nombre;**

Las funciones de esta estructura son algo distintas a las de la pila. Para insertar y quitar datos, se puede utilizar **push(valor)** y **pop()** respectivamente, pero para ver el valor que está hasta en frente se debe llamar **front()**. Como también tiene una parte de atrás, este se puede ver con **back()**.

Listing 57: Filas

```
#include <iostream>
#include <queue>

using namespace std;

int main() {
    queue<int> fila;
```

```

int valores[] = {6, 5, 2, 9, 3};
for(int i = 0; i < 5; i++) {
    fila.push(valores[i]);
}
while(fila.size() > 0) {
    cout << fila.front() << endl;
    fila.pop();
}
}

```

[Liga al código](#)

Esta estructura es bastante útil debido a que ahorra memoria al borrar datos que ya no ocupa y permite hacer una especie de buffer o vector que almacena datos que no se pueden procesar uno por uno.

Un ejemplo de la fila es cuando quieres promover un producto tuyo a muchas personas. Puedes empezar promoviéndole a un amigo tuyo y preguntándole si tiene amigos que le interesa tu producto, luego preguntas a cada uno de esos amigos por sus amigos y sigues repitiendo el proceso hasta haber promovido tu producto a una cantidad satisfactoria de gente.

La pila junto con la fila se aplicarán en las semanas que siguen para resolver problemas relacionados a grafos, como encontrar la respuesta a un laberinto o averiguar el camino más rápido entre dos ciudades.

12.3 Deque

El deque es una estructura especial debido a que puede funcionar como pila y fila a la vez. El nombre deque viene del termino double ended queue en inglés o fila con dos salidas.

Se llama de esta manera debido a que funciona como una pila porque puedes insertar datos y luego sacar el primer dato que fue ingresado, y además puedes sacar el último dato que fue ingresado.



Si usamos este ejemplo, no podemos obtener los valores de en medio, pero sí podemos sacar el 3 o el 6. Si quitamos el 3, ahora podríamos quitar el 9 o el 6.

Para implementar un deque, se debe incluir la librería **deque** y se puede crear con **deque<tipo> nombre;**

El deque fue implementado de una manera más flexible que las otras estructuras, y sí te permite ver los datos que están en su interior. Para guardar un dato en la parte de enfrente o quitar un dato, se pueden usar **push_front(valor)** y **pop_front(valor)** respectivamente, y para guardar datos en la parte de atrás de debe llamar **push_back(valor)** y **pop_back()**.

Se debe notar que el valor en frente es la que se encuentra en la posición 0 y el valor de atrás es la que está en la posición $N - 1$.

Para ver o modificar algún valor se puede usar el sintaxis de un arreglo, como **nombre[índice]**. Para ver el valor que esta hasta enfrente o hasta atrás, es recomendable utilizar **front()** y **back()**. Para insertar o eliminar valores existen las funciones **insert()** y **erase()**.

Un ejemplo práctico del deque es para determinar si una palabra es palindromo. Se puede insertar la palabra en el deque y luego se puede comparar el carácter de enfrente con el de atrás e irlos sacando de dos en dos hasta que quede cero o un valor en el deque.

Listing 58: Deques

```
#include <iostream>
#include <deque>

using namespace std;

int main() {
    string palindromo = "aibofobia";
    deque<char> dobleFila;
    //insertar a la fila
    for(int i = 0; i < palindromo.length(); i++) {
        dobleFila.push_back(palindromo[i]);
    }
    bool valido = true;
    while(dobleFila.size() > 1) {
        if(dobleFila.front() != dobleFila.back()) {
            valido = false;
            break;
        }
        dobleFila.pop_front();
        dobleFila.pop_back();
    }
    if(valido) {
        cout << palindromo << "es un palindromo" << endl;
    } else {
        cout << palindromo << "no es un palindromo" << endl;
    }
}
```


[Liga al código](#)

12.4 Priority queue

El priority queue es una especie de fila con prioridad como indica el nombre. Este guarda los valores como una fila normal, pero el primero en salir siempre será el valor mas grande en la fila.

Por ejemplo, un hospital usaría una fila de prioridad para atender sus pacientes, atendiendo al paciente más urgente primero.

Para usar este tipo de dato, se incluye la librería **queue** igual que cuando se estaba usando una fila y se declara con **priority_queue<tipo> nombre;**

Las funciones de este tipo son **push(valor)** y **pop()** para agregar o quitar valores y **top()** para ver el valor que esta enfrente de la fila.

Un ejemplo de estos sería un programa que ordena un arreglo de números flotantes.

Listing 59: Filas de prioridad

```
#include <iostream>
#include <queue>

using namespace std;

int main() {
    float valores[] = {2.22, 3.56, 1.9, 9.52, 3.42, 6.78, 0.11, 4.5};
    priority_queue<float> fila;
    for(int i = 0; i < 8; i++) {
        fila.push(valores[i]);
    }
    for(int i = 0; i < 8; i++) {
        cout << fila.top() << endl;
        fila.pop();
    }
}
```

[Liga al código](#)

Esta estructura es muy parecida a un heap, que guarda el dato más grande sobre los demás valores.

13 Recursividad

Imaginemos que encontraste un código extraño que parece correr una función dentro de esa misma función. Antes de llamarse a si mismo imprimirá hola y después imprimirá mundo.

Listing 60: Funciones

```
#include <iostream>

using namespace std;

void funcion() {
    cout << "hola" << endl;
    funcion();
    cout << "mundo" << endl;
}

int main() {
    funcion();
}
```

Si piensas en la lógica de este programa, tu función correrá, imprimirá "hola" y se correrá denuevo, volviendo a repetir este proceso. Lo que estarias haciendo es crear un ciclo infinito donde tu función corre una copia de esa función que corre otra copia de esa función y nunca se parará.

Si pruebas el código, verás que se imprime hola muchisimas veces hasta que el programa arroja un error. Este error pasa porque cada vez que se llama la función se esta ocupando memoria y llega a un punto donde ya no hay espacio para más funciones. También te darás cuenta que mundo nunca se imprime porque cada función que se llama nunca termina.

Este funcionamiento tiene muchas aplicaciones, por ejemplo, si decidimos correr la función con un parametro entero y decidimos restarle uno a ese número hasta que llegue a 0, podemos calcular la suma de todos los enteros naturales de 1 a N:

Listing 61: Suma de enteros

```
#include <iostream>

using namespace std;

int suma(int n) {
    if(n <= 0) {
        return 0;
    }
    return n + suma(n - 1);
}

int main() {
    cout << suma(100) << endl;
}
```

Si analizamos el código, suma con una entrada N primero checará si n es menor o igual a cero, y si no cumple con estas condiciones (es positivo), entonces

regresa $N + \text{suma}(N - 1)$. Si probamos esto con $N = 3$, se verá que $\text{suma}(3) = 3 + \text{suma}(2) = 3 + (2 + \text{suma}(1)) = 3 + (2 + (1 + \text{suma}(0))) = 3 + (2 + (1 + 0)) = 3 + 2 + 1 + 0$.

Podemos ver que gradualmente se van sumando los números empezando con N hasta llegar a 0, luego este se regresa en la función original. Cada llamada de $\text{suma}(N)$ depende del valor de $\text{suma}(N - 1)$.

A esto se le llama la recursividad y es útil para resolver problemas que dependen de un estado anterior. Otro ejemplo de la recursividad es para calcular el factorial de un número; se puede empezar con un número N y multiplicarle $N - 1$ hasta llegar a 1.

Listing 62: Factorial recursivo

```
#include <iostream>

using namespace std;

long long int factorial(long long int n) {
    if(n <= 1) {
        return 1;
    }
    return n * factorial(n - 1);
}

int main() {
    cout << factorial(10) << endl;
}
```

Pero donde brilla la recursividad es en problemas de combinatoria y permutaciones. Si quisieras escribir todas las permutaciones de los caracteres "ABCD", se puede escribir un programa sencillo que explora todas las posibilidades.

Listing 63: Permutaciones

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

using namespace std;

void permutacion(vector<char> letras , vector<char> usadas , int largo) {
    if(usadas.size() == largo) {
        for(int i = 0; i < usadas.size(); i++) {
            cout << usadas[i];
        }
        cout << endl;
        return;
    }
    for(int i = 0; i < letras.size(); i++) {
```

```

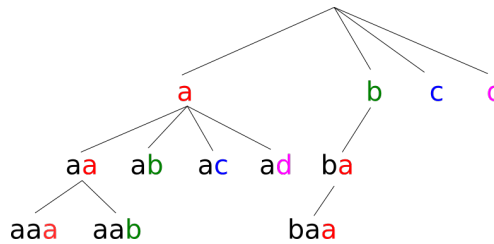
        usadas.push_back(letras[i]);
        permutacion(letras, usadas, largo);
        usadas.pop_back();
    }
}

int main() {
    vector<char> letras = {'a', 'b', 'c', 'd'};
    vector<char> usadas;
    permutacion(letras, usadas, 5);
}

```

[Liga al código](#)

Si probamos este código, se imprimirán todas las permutaciones de abcd con 5 letras, empezando con **aaaaa** y terminando con **ddddd**. El vector letras contiene las letras que queremos permutar y el vector usadas se va llenando con todas las permutaciones.



Tenemos un caso donde el vector usadas es igual al largo que queremos, en ese caso imprimimos las letras en ese vector, y si no lo que se hace es que se agregan todas las letras de 'a' a 'd'. Este código en cierto sentido navega un árbol de posibilidades, la primera letra tiene 4 posibilidades, y de esas 4 posibilidades hay 4 posibilidades que le siguen para la siguiente letra. Si calculamos el número total de caminos, obtenemos que es 4^5 o 1024 permutaciones.

14 Ordenamientos

En muchos problemas verás que es más conveniente o más rápido ordenar datos. Existen varios algoritmos de ordenamiento con varias ventajas y desventajas.

14.1 Ordenamiento por selección

Este tipo de ordenamiento tiende a ser conocido como "selection sort" en inglés y es fácil de programar con la desventaja que es muy lento (tiene complejidad de tiempo $O(N^2)$).

Este método busca el valor más bajo y lo intercambia con el valor en el primer lugar, luego busca el siguiente valor más bajo y lo intercambia con siguiente lugar hasta que se ordena todo el arreglo.

Por ejemplo, si tenemos el arreglo 5, 2, 3, 9, 6 y lo queremos ordenar con este método, intercambiaremos 2 con 5 para obtener 2, 5, 3, 9, 6. Como sabemos que 2 era el valor más bajo, el arreglo antes del 5 está ordenado y lo podemos ignorar. Esto significa que ahora solo tenemos que ordenar 5, 3, 9, 6. Podemos ver que el valor más bajo es 3 y lo podemos intercambiar con el primer lugar para obtener 3, 5, 9, 6. Nuestro arreglo original es ahora 2, 3, 5, 9, 6.

Después solo tendríamos que ordenar 5, 9, 6, pero como 5 ya está en el primer lugar podemos saltarnos ese valor y ordenar 9, 6. Al intercambiar estos valores, obtendríamos el arreglo ordenado 2, 3, 5, 6, 9.

Este método puede ser implementado con dos ciclos for, uno que va de 0 a N y otro que va del valor actual a N. El segundo ciclo buscará el valor más bajo de los valores que faltan de ordenarse y el primer ciclo lo irá poniendo en su lugar correcto.

Listing 64: Ordenamiento por selección

```
#include <iostream>

using namespace std;

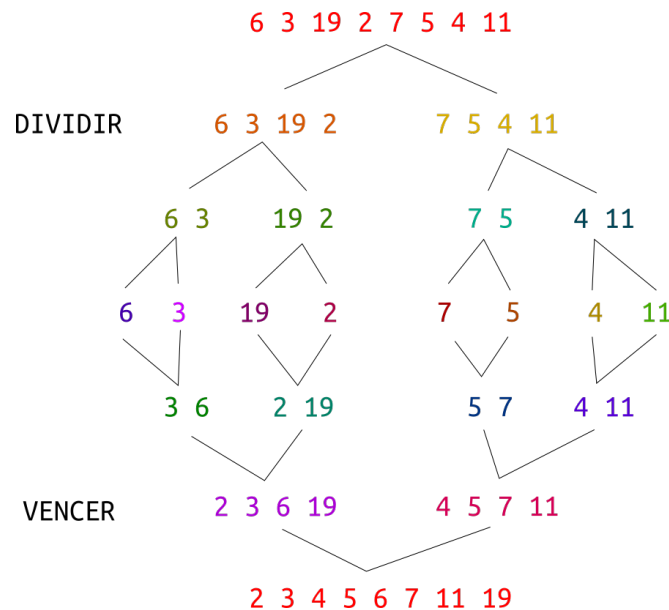
int main() {
    int valores[] = {5, 2, 3, 9, 6};
    for(int i = 0; i < 5; i++) {
        //Encuentra el valor mas pequeno de i a N
        int indiceMinimo = i;
        for(int j = i; j < 5; j++) {
            if(valores[j] < valores[indiceMinimo]) {
                indiceMinimo = j;
            }
        }
        //Si el valor esta en el primer lugar, ignoralo
        if(indiceMinimo == i) {
            continue;
        }
        //Intercambia los dos valores
        int intercambio = valores[i];
        valores[i] = valores[indiceMinimo];
        valores[indiceMinimo] = intercambio;
    }
    for(int i = 0; i < 5; i++) {
        cout << valores[i] << endl;
    }
}
```

[Liga al código](#)

Como se puede observar, su complejidad es de $O(N^2)$ porque tiene dos ciclos for anidados que van de 0 a N en el peor de los casos.

14.2 Ordenamiento por mezcla

También conocido como **merge sort** en inglés, este tipo de ordenamiento es muy rápido debido a que su complejidad es $O(N \log N)$. Este algoritmo se basa en lo que se llama "Divide y vencerás", esencialmente convierte un problema en más problemas pequeños pero más fáciles. Como sigue este modelo tiene una fase de división y parte el arreglo en mitades hasta llegar a un solo elemento. Como un solo elemento ya está ordenado, se puede juntar con otro arreglo de tamaño uno viendo cual de los dos es más pequeño y poniendo ese primero. Cuando se hace esto, se tendrán varios arreglos ordenados con dos elementos, y estos se vuelven a juntar para hacer arreglos ordenados de cuatro elementos y así sucesivamente.



Digamos que queremos ordenar los valores 6, 3, 19, 2, 7, 5, 4, 11. Partimos los arreglos a la mitad y ahora nos enfocamos en ordenar 6, 3, 19, 2 y 7, 5, 4, 11. Después decidimos partir estos a la mitad y enfocarnos en ordenar los arreglos 6, 3, 19, 2, 7, 5 y 4, 11. Los partimos una última vez para tener 6, 3, 19, 2, 7, 5, 4, 11.

Como habíamos partido 6, 3 para obtener 6 y 3, podemos juntarlos para hacer un nuevo arreglo ordenado escogiendo el más pequeño de los dos primero: 3, 6. Al repetir esto con los otros arreglos, se obtiene 3, 6, 2, 19, 5, 7, 4, 11.

Como antes queríamos ordenar 6, 3, 19, 2 y tenemos los dos subarreglos 3, 6 y 2, 19, podemos formar el arreglo ordenado de cuatro elementos volviendo a escoger el más pequeño de los dos arreglos en cada momento y poniendo ese primero. Al quedarse 2, 3, 6, 19 y 4, 5, 7, 11, se pueden volver a juntar estos arreglos para tener el arreglo original ordenado.

La manera más sencilla de implementar este algoritmo es usando la recursividad:

Listing 65: Ordenamiento por mezcla

```
#include <iostream>

using namespace std;

void ordenar(int* valores, int tamano) {
    if(tamano < 2) {
        return;
    }
    int mitadA = tamano / 2;
    int mitadB = tamano - mitadA;
    int valoresA[mitadA];
    int valoresB[mitadB];
    for(int i = 0; i < mitadA; i++) {
        valoresA[i] = valores[i];
    }
    for(int i = 0; i < mitadB; i++) {
        valoresB[i] = valores[i + mitadA];
    }
    ordenar(valoresA, mitadA);
    ordenar(valoresB, mitadB);
    int indiceA = 0;
    int indiceB = 0;
    while(indiceA < mitadA && indiceB < mitadB) {
        if(valoresA[indiceA] < valoresB[indiceB]) {
            valores[indiceA + indiceB] = valoresA[indiceA];
            indiceA++;
        } else {
            valores[indiceA + indiceB] = valoresB[indiceB];
            indiceB++;
        }
    }
    while(indiceA < mitadA) {
        valores[indiceA + indiceB] = valoresA[indiceA];
        indiceA++;
    }
    while(indiceB < mitadB) {
        valores[indiceA + indiceB] = valoresB[indiceB];
```

```

        indiceB++;
    }
}

int main() {
    int cantidad;
    cin >> cantidad;
    int valores[cantidad];
    for(int i = 0; i < cantidad; i++) {
        cin >> valores[i];
    }
    ordenar(valores, cantidad);
    for(int i = 0; i < cantidad; i++) {
        cout << valores[i] << endl;
    }
}

```

[Liga al código](#)

14.3 Ordenamiento estandar de C++

Para ahorrar tiempo y evitar tener que volver a escribir ordenamientos como el de mezcla, C++ tiene una librería que incluye su propia implementación eficiente de ordenamiento. Este ordenamiento se llama **csort** y tiene complejidad de tiempo $O(N \log N)$.

Para utilizar este ordenamiento, se tiene que incluir la librería **algorithm**. Luego, se debe llamar la función **sort()** con dos parámetros, el primer lugar en memoria del arreglo y el ultimo lugar del arreglo. Para un arreglo normal, se puede dar el nombre del arreglo como el primer parámetro y el nombre del arreglo sumado con su tamaño como el segundo parámetro.

Listing 66: Ordenamiento estandar para un arreglo

```

#include <iostream>
#include <algorithm>

using namespace std;

int main() {
    int valores[] = {6, 3, 19, 2, 7, 5, 4, 11};
    sort(valores, valores + 8);
    for(int i = 0; i < 8; i++) {
        cout << valores[i] << endl;
    }
}

```

Para vectores y otras estructuras definidas en librerías, se debe usar las funciones **begin()** y **end()**

Listing 67: Ordenamiento estandar para un vector

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

using namespace std;

int main() {
    vector<int> valores = {6, 3, 19, 2, 7, 5, 4, 11};
    sort(valores.begin(), valores.end());
    for(int i = 0; i < valores.size(); i++) {
        cout << valores[i] << endl;
    }
}
```

Si se desea ordenar un arreglo de una manera especial como ordenar pares por su primer valor, se puede crear una función que maneja el ordenamiento y se le puede pasar como tercer parámetro a la función **sort**.

Esta función debe ser de tipo booleano y debe tener dos parámetros que son del mismo tipo que el arreglo o vector que se desea ordenar. La función debe regresar verdadero si se quiere ordenar el primer parámetro antes que el segundo o falso en caso contrario.

Listing 68: Ordenamiento estandar sobrecargada

```
#include <iostream>
#include <algorithm>
#include <pair>

using namespace std;

//Ordena de manera ascendiente los primeros valores del par
bool ordenaPrimero(pair<int, int> a, pair<int, int> b) {
    if(a.first < b.first) {
        return true;
    }
    return false;
}

//Ordena de manera ascendiente los segundos valores del par
bool ordenaPrimero(pair<int, int> a, pair<int, int> b) {
    if(a.second < b.second) {
        return true;
    }
    return false;
}
```

```

int main() {
    vector<pair<int, int>> pares;
    pares.push_back(make_pair(6, 3));
    pares.push_back(make_pair(19, 2));
    pares.push_back(make_pair(7, 5));
    pares.push_back(make_pair(4, 11));
    sort(pares.begin(), pares.end(), ordenaPrimero);
    for(int i = 0; i < pares.size(); i++) {
        cout << pares[i].first << " " << pares[i].second << endl;
    }
    sort(pares.begin(), pares.end(), ordenaSegundo);
    for(int i = 0; i < pares.size(); i++) {
        cout << pares[i].first << " " << pares[i].second << endl;
    }
}

```

15 Programación dinámica

La programación dinámica consiste en guardar datos previos para ahorrar tiempo que normalmente se desperdiciaría recalculando la misma cosa multiples veces.

Vamos a analizar un programa que calcula la serie de fibonacci con la recursividad.

Listing 69: Fibonacci

```

#include <iostream>

using namespace std;

long long int fibonacci(int valor) {
    if(valor == 0) {
        return 0;
    }
    if(valor == 1) {
        return 1;
    }
    return fibonacci(valor - 1) + fibonacci(valor - 2);
}

int main() {
    cout << fibonacci(50);
}

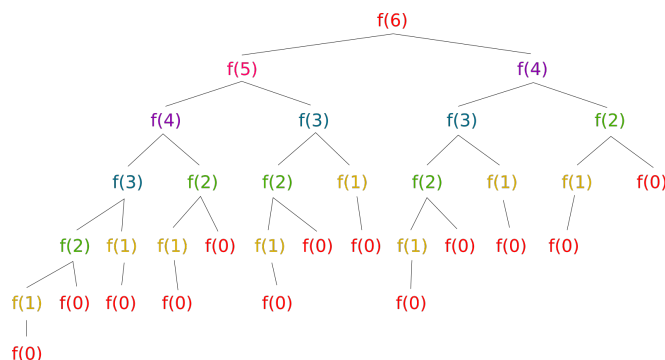
```

[Liga al código](#)

Si vemos el programa anterior, tenemos una función fibonacci que calcula un termino deseado. Si corremos el código con un valor bajo, por ejemplo 5

o 6 podemos ver que funciona correctamente, pero si intentamos calcular el valor número 50 el programa nunca parece querer terminar e incluso podríamos calcular ese valor más rápido a mano.

Si ponemos todas las llamadas a fibonacci(6) en una gráfica, podemos ver porque es ineficiente:



Se puede observar que el tamaño de fibonacci(N - 1) no es tan lejano del tamaño de fibonacci(N), así que al sumarle uno a N, estaríamos casi duplicando el número de cálculos necesarios. Esto significa que nuestra complejidad de tiempo es exponencial cuando debería ser lineal. En realidad, esta función tiene una complejidad de $O(1.618^N)$.

Para hacer esta función de complejidad lineal $O(N)$, debemos evitar la recursividad y guardar los valores de cada número de fibonacci en un vector mientras que los vayamos calculando. Si queremos calcular un valor de la serie que ya fue calculado, debemos regresar ese valor.

Esta sería la implementación con programación dinámica:

Listing 70: Programación dinámica

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

using namespace std;

vector<long long int> serie;

long long int fibonacci(int valor) {
    while(serie.size() <= valor) {
        serie.push_back(serie[serie.size() - 1] + serie[serie.size() - 2]);
    }
    return serie[valor];
}

int main() {
    serie.push_back(0);
```

```
    serie.push_back(1);  
    cout << fibonacci(50);  
}
```

Como se puede observar, nuestro programa ahora encuentra el termino número 50 en menos de un segundo.