Capitulo estudiantil ACM – ICPC UMSA

1. Introducción (en C++ a la)

2. {

3. Programación ;

4. Competitiva ;

5. }

ACM-ICPC UMSA
Student Chapter

CoderDojo

Hivos
people unlimited

Introducción en C++ a la Programación Competitiva

Oscar G. Carvajal Yucra

Rodrigo J. Castillo Gúzman

Diego A. Charca Flores

Branimir F. Espinoza Argollo

Brayan H. Gonzales Velazques

Gabriel A. Mazuelos Rabaza

Addis P. Rengel Sempértegui

Rolando Troche Venegas

Capitulo estudiantil ACM – ICPC UMSA

1. Introducción (en C++ a la)

2. {

3. Programación ;

4. Competitiva ;

5. }

ACM-ICPC UMSA
Student Chapter

CoderDojo

Hivos
people unlimited

Queremos agradecer al "Tech Hub" y a "Hivos" por ayudarnos en la elaboración de este libro:

Contenidos

Prefacio 1

Las Competencias de Programación fueron parte importante de mi vida universitaria y profesional los últimos años. Gracias a ellas obtuve mis primeros empleos y me hice de importantes armas teóricas en el área de las ciencias de la computación. Cuando empecé en este mundo, no existían muchas personas a quienes recurrir para preguntarles del tema ni documentación en español para poder leer. Hoy en día todo esto cambió gracias a varios grupos que existen por todo nuestro territorio, quienes socializaron el conocimiento de las competencias de programación y enseñaron a sus nuevas generaciones. Este libro que hicieron mis amigos del Capitulo Estudiantil de la Universidad Mayor de San Andrés me parece un gran avance con respecto a las épocas en las que a mi me tocaba investigar acerca del tema. Contiene todos los temas básicos que uno necesita para poder convertirse en un programador competitivo y poder participar en las competencias que exiten hoy en día. Sí eres un competidor de la ACM-ICPC o de la Olimpiada de Informática este es un material muy recomendado para empezar, también si solamente eres un apasionado por el tema y te interesan otro tipo de competencias como el Google Code Jam o el Facebook Hacker Cup. Hoy en día estas competencias han cobrado un rol importante en el área del desarrollo de software a nivel nacional e internacional, porque el programador competitivo programa sin bugs y las grandes empresas quieren eso. Grandes empresas como Google, Facebook, Amazon, Microsoft, Yahoo, contratan a sus empleados y pasantes haciendo entrevistas con preguntas idénticas a las de las competencias de programación, por esa razón los competidores mejor rankeados en cada país tienen un puesto asegurado en estas empresas. Desde estos parrafos extiendo una felicitación a mis compañeros que se esforzaron para entregar toda esta información al mundo de habla hispana y espero que estas iniciativas crezcan por todo nuestro país para que algún día no envidiemos el nivel de ningún otro en el mundo.

Happy Coding!

Jhonatan Castro Rocabado
marzo del 2015

Prefacio 2

Hace unos años (particularmente en Bolivia) ingresar en el campo de la programación competitiva era muy difícil, debido a que para empezar no se sabía ni de donde conseguir información que ayude en esta área. Hoy por hoy a alguien que quiera iniciarse en la programación competitiva podemos darle montones y montones de páginas que tienen problemas, libros, tutoriales; pero toda esta información está en inglés lo cual para una persona que no sabe nada de inglés es un gran problema (personalmente algunos de los autores de este libro saben algo de inglés gracias a esto). Con este libro lo que queremos es dar una pequeña ayuda a aquel que quiera iniciarse en el mundo de la programación competitiva explicando un poco del lenguaje que se usa y algunos algoritmos que les serán útiles en el futuro. Esperamos con el pasar de los años poder hacer un libro todavía más completo que contenga muchos más algoritmos y muchas más cosas útiles. Les deseamos mucha suerte y éxitos en su camino a todos aquellos que quieran ser programadores competitivos.

Addis Rengel Sempertegui
marzo del 2015

1. Introducción a la programación en c++

1.1 Introducción

1.1.1 ¿Qué es un Lenguaje de Programación?

Antes de hablar de C++, es necesario explicar que un lenguaje de programación es una herramienta que nos permite comunicarnos e instruir a la computadora para que realice una tarea específica. Cada lenguaje de programación posee una sintaxis y un léxico particular, es decir, forma de escribirse que es diferente en cada uno por la forma que fue creado y por la forma que trabaja su compilador para revisar, acomodar y reservar el mismo programa en memoria.
Existen muchos lenguajes de programación de entre los que se destacan los siguientes:

- C
- C++
- Basic
- Ada
- Java
- Pascal
- Python
- Fortran
- Smalltalk

1.1.2 ¿Qué es C++?

C++ es un lenguaje de programación orientado a objetos que toma la base del lenguaje C y le agrega la capacidad de abstraer tipos como en Smalltalk.
La intención de su creación fue el extender al exitoso lenguaje de programación C con mecanismos que permitieran la manipulación de objetos. En ese sentido, desde el punto de vista de los lenguajes orientados a objetos, el C++ es un lenguaje híbrido.
Posteriormente se añadieron facilidades de programación genérica, que se sumó a los otros dos paradigmas que ya estaban admitidos (programación estructurada y la programación orientada a objetos). Por esto se suele decir que el C++ es un lenguaje de programación multiparadigma.

1.1.3 Herramientas Necesarias

Las principales herramientas necesarias para escribir un programa en C++ son las siguientes:

1. Un equipo ejecutando un sistema operativo.

2. Un compilador de C++
   - Windows MingW (GCC para Windows) o MSVC (compilador de Microsoft con versión gratuita)
   - Linux (u otros UNIX): g++
   - Mac (con el compilador Xcode)
3. Un editor cualquiera de texto, o mejor un entorno de desarrollo (IDE)
   - Windows:
     - Microsoft Visual C++ (conocido por sus siglas MSVC). Incluye compilador y posee una versión gratuita (versión express)
     - Bloc de notas (no recomendado)
     - Editor Notepad++
     - DevCpp (incluye MingW - en desuso, no recomendado, incluye también un compilador)
     - Code::Blocks
   - Linux (o re-compilación en UNIX):
     - Gedit
     - Kate
     - KDevelop
     - Code::Blocks
     - SciTE
     - GVim
   - Mac:
     - Xcode (con el compilador trae una IDE para poder programar)
4. Tiempo para practicar
5. Paciencia

Adicional

- Inglés (Recomendado)
- Estar familiarizado con C u otro lenguaje derivado (PHP, Python, etc).

Es recomendable tener conocimientos de C, debido a que C++ es una mejora de C, tener los conocimientos sobre este te permitirá avanzar más rápido y comprender aún mas. También, hay que recordar que C++, admite C, por lo que se puede programar (reutilizar), funciones de C que se puedan usar en C++.

Aunque no es obligación aprender C, es recomendable tener nociones sobre la programación orientada a objetos en el caso de no tener conocimientos previos de programación estructurada. Asimismo, muchos programadores recomiendan no saber C para saber C++, por ser el primero de ellos un lenguaje imperativo o procedimental y el segundo un lenguaje de programación orientado a objetos.

1.1.4 Consejos iniciales antes de programar

Con la práctica, se puede observar que se puede confundir a otros programadores con el código que se haga. Antes de siquiera hacer una línea de código, si se trabaja con otros programadores, ha de tenerse en cuenta que todos deben escribir de una forma similar el código, para que de forma global puedan corregir el código en el caso de que hubieran errores o rastrearlos en el caso de haberlos.

También es muy recomendable hacer uso de comentarios (comenta todo lo que puedas, hay veces que lo que parece obvio para ti, no lo es para los demás) y tratar de hacer un código limpio y

comprensible, especificando detalles y haciendo tabulaciones, aunque te tome un poco más de tiempo, es posible que más adelante lo agradezcas tú mismo.

1.1.5 Ejemplo

Código 1.1: Ejemplo de C++

```cpp
#include <iostream>

using namespace std;

int main()
{
    int numero;
    cin>>numero;
    if(numero%2==0){
        cout<<"El numero es par\n";
    }else{
        cout<<"EL numero es impar\n";
    }
    return 0;
}
```

1.2 Lo mas básico

1.2.1 Proceso de desarrollo de un programa

Si se desea escribir un programa en C++ se debe ejecutar como mínimo los siguientes pasos:

1. Escribir con un editor de texto plano un programa sintácticamente válido o usar un entorno de desarrollo (IDE) apropiado para tal fin
2. Compilar el programa y asegurarse de que no han habido errores de compilación
3. Ejecutar el programa y comprobar que no hay errores de ejecución

Este último paso es el más costoso, porque en programas grandes, averiguar si hay o no un fallo prácticamente puede ser una tarea totémica.
Un archivo de C++ tiene la extensión *cpp* a continuación se escribe el siguiente código en c++ del archivo 'hola.cpp'

Código 1.2: Hola Mundo

```cpp
// Aquí generalmente se suele indicar qué se quiere con el programa a hacer
// Programa que muestra 'Hola mundo' por pantalla y finaliza

// Aquí se sitúan todas las librerias que se vayan a usar con include,
// que se verá posteriormente
#include <iostream> // Esta libreria permite mostrar y leer datos por consola

int main()
{
```

```
10      // Este tipo de líneas de código que comienzan por '//' son comentarios
11      // El compilador los omite, y sirven para ayudar a otros programadores o
12      // a uno mismo en caso de volver a revisar el código
13      // Es una práctica sana poner comentarios donde se necesiten,
14
15      std::cout << "Hola Mundo" << std::endl;
16
17      // Mostrar por std::cout el mensaje Hola Mundo y comienza una nueva línea
18
19      return 0;
20
21      // se devuelve un 0.
22      //que en este caso quiere decir que la salida se ha efectuado con éxito.
23  }
```

Mediante simple inspección, el código parece enorme, pero el compilador lo único que leerá para la creación del programa es lo siguiente:

Código 1.3: Hola Mundo Compilado

```
1  #include <iostream>
2  int main(void){ std::cout << "Hola Mundo" << std::endl; return 0; }
```

Como se puede observar, este código y el original no difieren en mucho salvo en los saltos de línea y que los comentarios, de los que se detallan posteriormente, están omitidos y tan sólo ha quedado "el esqueleto" del código legible para el compilador. Para el compilador, todo lo demás, sobra.
Aquí otro ejemplo

Código 1.4: Hello World

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "Hola Mundo" << std::endl;
6      std::cout << "Hello World" << std::endl;
7      std::cout << "Hallo Welt" << std::endl;
8      return 0;
9  }
```

Para hacer el código más corto debemos incluir *using namespace std* con lo cual ya no necesitaremos incluir cada vez que lo necesitemos, como en los objetos cout y cin, que representan el flujo de salida estándar (típicamente la pantalla o una ventana de texto) y el flujo de entrada estándar (típicamente el teclado).
Se vería de la siguiente manera:

Código 1.5: Using Namespace Std

```
1  #include <iostream>
2
3  using namespace std;
```

```
4
5   int main()
6   {
7       cout << "Hola Mundo" << endl;
8       cout << "Hello World" << endl;
9       cout << "Hallo Welt" << endl;
10      return 0;
11  }
```

Los pasos siguientes son para una compilación en GNU o sistema operativo Unix, para generar el ejecutable del programa se compila con g++ de la siguiente forma:

```
1  | g++ hola.cpp -o hola
```

Para poder ver los resultados del programa en acción, se ejecuta el programa de la siguiente forma:

```
1  |./hola
```

Y a continuación se debe mostrar algo como lo siguiente:

```
1  Hola Mundo
```

1.2.2 Sintaxis

Sintaxis es la forma correcta en que se deben escribir las instrucciones para el computador en un lenguaje de programación específico. C++ hereda la sintaxis de C estándar, es decir, la mayoría de programas escritos para el C estándar pueden ser compilados en C++.

1.2.2.1 El punto y coma

El punto y coma es uno de los símbolos más usados en C, C++; y se usa con el fin de indicar el final de una línea de instrucción. El punto y coma es de uso obligatorio como se pueden observar en los ejemplos anteriores.
El punto y coma se usa también para separar contadores, condicionales e incrementadores dentro de un sentencia **for**
Ejemplo

```
1  for (i=0; i < 10; i++) cout << i;
```

1.2.2.2 Espacios y tabuladores

Usar caracteres extras de espaciado o tabuladores (caracteres tab) es un mecanismo que nos permite ordenar de manera más clara el código del programa que estemos escribiendo, sin embargo, el uso de estos es opcional ya que el compilador ignora la presencia de los mismos. Por ejemplo se podría escribir así:

```
1  for (int i=0; i < 10; i++) { cout << i * x; x++; }
```

y el compilador no pondría ningún reparo.

1.2.2.3 Comentarios

Existen dos modos básicos para comentar en C++:

//	Comentan solo una línea de código
/*Comentario*/	Comentan estrofas de código

A continuación un ejemplo:

Código 1.6: Comentarios

```
#include <iostream>

using namespace std;

int main()
{
    /*
    cout    es para imprimir
    <<      se utiliza para separar elementos para cout
    endl    es un salto de linea
    */

    //En español
    cout << "Hola Mundo" << endl;
    //En ingles
    cout << "Hello World" << endl;
    //En aleman
    cout << "Hallo Welt" << endl;
    return 0;
}
```

1.2.3 Datos primitivos

En un lenguaje de programación es indispensable poder almacenar información, para esto en C++ están disponibles los siguientes tipos que permiten almacenar información numérica de tipo entero o real:

Nombre	Descripción	Tamaño	Rango de valores
bool	Valor booleano	1 byte	true o false
char	Carácter o entero pequeño	1 byte	De -128 a 127
short int	Entero corto	2 bytes	De -32768 a 32767
int	Entero	4 bytes	De -2147483648 a 2147483647
long long	Entero largo	8 bytes	De -9223372036854775808 a 9223372036854775807
float	Número de punto flotante	4 bytes	3.4e +/- 38 (7 dígitos)
double	Float con doble precisión	8 bytes	1.7e +/- 308 (15 dígitos)

Los valores dependen de la arquitectura utilizada. Los mostrados son los que generalmente se encuentran en una máquina típica de arquitectura 32 bits.

El Modificador unsigned

El modificador unsigned es utilizado únicamente con los enteros, su utilización permite utilizar en los enteros únicamente la parte positiva,

```
1 int a;
2 // Almacena valores entre -2,147,483,648 a 2,147,483,647
3 unsigned int a;
4 // Almacena valores entre 0 a 4,294,967,295
```

1.2.4 Variables y constantes

Una variable, como su nombre lo indica, es un determinado objeto cuyo valor puede cambiar durante el proceso de una tarea específica. Contrario a una variable, una constante es un determinado objeto cuyo valor no puede ser alterado durante el proceso de una tarea específica. En C, C++ para declarar variables no existe una palabra especial, es decir, las variables se declaran escribiendo el tipo seguido de uno o más identificadores o nombres de variables. Por otro lado, para declarar constantes existe la palabra reservada const, así como la directiva *#define*. A continuación se muestran ejemplos de declaración de variables y constantes.

Variables	Constantes	Constantes
int a;	const int a = 100;	#define a 100
float b;	const float b = 100;	#define b 100

A diferencia de las constantes declaradas con la palabra **const** los símbolos definidos con **#define** no ocupan espacio en la memoria del código ejecutable resultante.

El tipo de la variable o constante puede ser cualquiera de los listados en Tipos primitivos, o bien de un tipo definido por el usuario.

Las constantes son usadas a menudo con un doble propósito, el primero es con el fin de hacer más legible el código del programa, es decir, si se tiene (por ejemplo) la constante numérica 3.1416, esta representa al número pi, entonces podemos hacer declaraciones tales como:

```
1 #define pi 3.1416
```

En este caso podremos usar la palabra pi en cualquier parte del programa y el compilador se encargará de cambiar dicho símbolo por 3.1416 o bien,

```
1 const pi = 3.1416;
```

En este otro caso podremos usar la palabra pi en cualquier parte del programa y el compilador se encargará de cambiar dicho símbolo por una referencia a la constante pi guardada en la memoria.

1.2.5 Lectura e Impresión

Para la lectura e impresión de datos se pude realizar por dos métodos por **cin** y **cout** primitivos de C++ o por **printf** y **scanf** primitivos de C, en el programa se pueden usar ambos métodos si es necesario.

1.2.5.1 cin y cout

En sus programas, si usted desea hacer uso de los objetos cin y cout tendrá que incluir el uso de la biblioteca iostream (por medio de la directiva *#include*). La iostream es la biblioteca estándar en C++ para poder tener acceso a los dispositivos estándar de entrada y/o salida.
Si usted usa la directiva #include <iostream.h> o #include <iostream> en sus programas, automáticamente la iostream pone a su disposición los objetos cin y cout en el ámbito estándar (std), de tal manera que usted puede comenzar a enviar o recibir información a través de los mismos sin siquiera preocuparse de su creación. Así, un sencillo ejemplo del uso de los objetos mencionados se muestra en seguida.

Código 1.7: Iostream

```
1  // De nuevo con el hola mundo...
2  #include <iostream.h>
3
4  using namespace std;
5
6  int main()
7  {
8      cout << "Hola mundo";     // imprimir mensaje (en la pantalla)
9      cin.get();     // lectura ( entrada del teclado )
10     return 0;
11 }
```

Para el manejo de **cin** y **cout** se necesita también el uso de los **operadores de direccionamiento < < y > >**. Los operadores de direccionamiento son los encargados de manipular el flujo de datos desde o hacia el dispositivo referenciado por un stream específico. El operador de direccionamiento para salidas es una pareja de símbolos de "menor que" **< <**, y el operador de direccionamiento para entradas es una pareja de símbolos de "mayor que" **> >**. Los operadores de direccionamiento se colocan entre dos operandos, el primero es el Stream y el segundo es una variable o constante que proporciona o recibe los datos de la operación. Por ejemplo, en el siguiente programa y en la instrucción **cout < < "Entre su nombre: ";** la constante **"Entre su nombre: "** es la fuente o quien proporciona los datos para el objeto **cout**. Mientras que en la instrucción **cin > > nombre** la variable **nombre** es el destino o quien recibe los datos provenientes del objeto **cin**.

Código 1.8: Oper Direc

```
1  // De nuevo con el hola mundo...
2  #include <iostream.h>
3
4  using namespace std;
5
6  int main()
7  {
8      char nombre[80];
9      cout << "Entre su nombre: ";
10     cin  >> nombre;
11     cout << "Hola," << nombre;
12     cin.get();
13     return 0;
```

```
14 }
```

Observe que si en una misma línea de comando se desea leer o escribir sobre varios campos a la vez, no es necesario nombrar más de una vez al stream. Ejemplos:

```
1 cout << "Hola," << nombre;
2 cin >> A >> B >> C;
```

Otro ejemplo del manejo de **cout**:

Código 1.9: Impresión

```
1  // Programa que muestra diversos textos por consola
2
3  // Las librerías del sistema usadas son las siguientes
4  #include <iostream>
5
6  using namespace std;
7  // Es la función principal encargada de mostrar por consola diferentes textos
8  int main(void)
9  {
10     // Ejemplo con una única línea, se muestra el uso de cout y endl
11     cout << "Bienvenido. Soy un programa. Estoy en una linea de
           codigo." << endl;
12
13     // Ejemplo con una única línea de código que se puede fraccionar
14     // mediante el uso de '«'
15     cout << "Ahora "
16          << "estoy fraccionado en el codigo, pero en la consola me
                muestro como una unica frase."
17          << endl;
18
19     // Uso de un código largo, que cuesta leer para un programador,
20     // y que se ejecutará sin problemas.
21     // *** No se recomienda hacer líneas de esta manera,
22     // esta forma de programar no es apropiada ***
23     cout << "Un gran texto puede ocupar muchas lineas."
24          << endl
25          << "Pero eso no frena al programador a que todo se pueda
                poner en una unica linea de codigo y que"
26          << endl
27          << "el programa, al ejecutarse, lo situe como el
                programador quiso"
28          << endl;
29
30     return 0; // Y se termina con éxito.
31 }
```

Consola:

```
1 Bienvenido. Soy un programa. Estoy en una linea de codigo.
```

```
2  Ahora estoy fraccionado en el codigo, pero en la consola me muestro como una unica
       frase.
3  Un gran texto puede ocupar muchas lineas.
4  Pero eso no frena al programador a que todo se pueda poner en una unica linea de
       codigo y que
5  el programa, al ejecutarse, lo situe como el programador quiso
```

1.2.5.2 scanf y printf

En sus programas, si usted desea hacer uso de los objetos scanf y printf tendrá que incluir el uso de la biblioteca cstdio (por medio de la directiva **#cstdio**). Cstdio es la biblioteca estándar en C para poder tener acceso a los dispositivos estándar de entrada y/o salida.
Generalmente, **printf()** y **scanf()** funcionan utilizando cada una de ellas una "tira de caracteres de control" y una lista de "argumentos". Veremos estas características; en primer lugar en **printf()**, y a continuación en **scanf()**.
Las instrucciones que se han de dar a **printf()** cuando se desea imprimir una variable dependen del tipo de variable de que se trate. Así, tendremos que utilizar la notación **%d** para imprimir un entero, y **%c** para imprimir un carácter, como ya se ha dicho. A continuación damos la lista de todos los identificadores que emplea la función **printf()** y el tipo de salida que imprimen. La mayor parte de sus necesidades queda cubierta con los ocho primeros; de todas formas, ahí están los dos restantes por si desea emplearlos.

Identificador	Salida
%c	Carácter o entero pequeño
%s	Tira de caracteres
%d, %i	Entero decimal
%u	Entero decimal sin signo
%lld	Entero largo
%llu	Entero largo sin signo
%f	Número de punto flotante en notación decimal
%lf	Número de punto flotante en notación decimal con doble presición
%o	Entero octal sin signo
%x	Entero hexadecimal sin signo

Veamos ahora cómo se utilizan. El programa siguiente emplea algunos de los identificadores que acabamos de ver:

Código 1.10: Impresión

```
1  #include <cstdio>
2  #define PI 3.14159
3
4  using namespace std;
5
6  int main()
7  {
8      int numero = 5;
9      int coste = 50;
10     // No olvidar que '\n' es fin de linea
11     printf("Los %d jovenes tomaron %d helados.\n",numero,numero*2);
```

```
12      printf("El valor de PI es %f.\n");
13      printf("Esta es una linea sin variables.");
14      printf("%c%d", '$', coste);
15      return 0;
16  }
```

La salida es:

```
Los 5 jovenes tomaron 10 helados.
El valor de PI es 3.14159.
Esta es una linea sin variables.
\$50
```

El formato para uso de **printf()** es éste:

printf(Control, item1, item2, ...);

item1, item2, etc., son las distintas variables o constantes a imprimir. Pueden también ser expresiones, las cuales se evalúan antes de imprimir el resultado. **Control** es una tira de caracteres que describen la manera en que han de imprimirse los items. Por ejemplo, en la sentencia

```
printf("Los %d jovenes tomaron %d helados.\n", numero, numero*2);
```

Control sería la frase entre comillas (después de todo, es una tira de caracteres), y **numero** y **numero*2** serían los items; en este caso, los valores de dos variables.
También se debe hablar de **modificadores de especificaciones de conversión** en printf(), estos son apéndices que se agregan a los especificadores de conversión básicos para modificar la salida. Se colocan entre el símbolo **%** y el carácter que define el tipo de conversión. A continuación se da una lista de los símbolos que está permitido emplear. Si se utiliza más de un modificador en el mismo sitio, el orden en que se indican deberá ser el mismo que aparece en la tabla. Tenga presente que no todas las combinaciones son posibles.

-	El ítem correspondiente se comenzará a escribir empezando en el extremo izquierdo del campo que tenga asignado. Normalmente se escribe el ítem de forma que acabe a la derecha del campo. Ejemplo: **%-10d**
número	Anchura mínima del campo. En el caso de que la cantidad a imprimir (o la tira de caracteres) no quepa en el lugar asignado, se usará automáticamente un campo mayor Ejemplo: **%4d**
.número	Precisión. En tipos flotantes es la cantidad de cifras que se han de imprimir a la derecha del punto (es decir, el número de decimales). En el caso de tiras, es el máximo número de caracteres que se ha de imprimir. Ejemplo: **%4.2f** (dos decimales en un campo de cuatro caracteres de ancho)

Veamos ahora un ejemplo:

Código 1.11: Modificadores

```
1  #include <cstdio>
2
3  using namespace std;
4
```

```
5  int main()
6  {
7      printf("Modificador '-'\n");
8      printf("/%-10d/\n", 365);
9      printf("/%-6d/\n", 365);
10     printf("/%-2d/ ", 365);
11     printf(" en este caso no sirve de nada el modificador\n");
12     printf("Modificador 'numero'\n");
13     printf("/%2d/", 365);
14     printf(" en este caso tampoco hace efecto el modificador\n");
15     printf("/%6d/\n", 365);
16     printf("/%10d/\n", 365);
17     printf("Modificador '.numero'\n");
18     printf("/%.1f/\n", 365.897);
19     printf("/%.6f/\n", 365.897);
20     printf("/%.3s/\n", "AsDfGhJkL");
21     printf("/%.100s/", "AsDfGhJkL");
22     printf(" No se rellena con 0's como en los ejemplos anteriores\n
           ");
23     return 0;
24 }
```

La salida es:

```
1  Modificador '-'
2  /365       /
3  /365   /
4  /365/  en este caso no sirve de nada el modificador
5  Modificador 'numero'
6  /365/ en este caso tampoco hace efecto el modificador
7  /   365/
8  /       365/
9  Modificador '.numero'
10 /365.9/
11 /365.897000/
12 /AsD/
13 /AsDfGhJkL/ No se rellena con 0's como en los ejemplos anteriores
```

Para el uso de **scanf()** se utilizan los mismo identificadores que en **printf()**, y al igual que **printf()**, **scanf()** emplea una tira de caracteres de control y una lista de argumentos. La mayor diferencia entre ambas está en esta última; **printf()** utiliza en sus listas nombres de variables, constantes y expresiones; **scanf()** usa punteros a variable. Afortunadamente no se necesita saber mucho de punteros para emplear esta expresión; se trata simplemente de seguir las dos reglas que se dan a continuación:

- Si se desea leer un valor perteneciente a cualquier de los tipos básicos coloque el nombre de la variable precedido por un **&**.
- Si lo que desea es leer una variable de tipo tira de caracteres, no use **&**.

Otro detalle a tomar en cuenta es que **scanf()** considera que dos ítems de entrada son diferentes cuando están separados por blancos, tabulados o espacios. Va encajando cada especificador de conversión con su campo correspondiente, ignorando los blancos intermedios. La única excepción es la especificación **%c**, que lee el siguiente caracter, sea blando o no.
El siguiente programa es válido:

Código 1.12: Scanf

```cpp
#include <cstdio>

using namespace std;

int main()
{
    int edad;
    char nombre[30];
    printf("Ingrese su edad y nombre\n");
    scanf("%d %s", &edad, nombre); //Note que & no es necesario en nombre
    printf("Nombre: %s , Edad: %d ", nombre, edad);
    return 0;
}
```

Posible salida si la entrada fuera '19 Oscar':

```
Ingrese su edad y nombre
Nombre: Oscar , Edad: 19
```

1.2.6 Operadores

El C++ está lleno de operadores. Presentamos aquí una tabla de los mismos indicando el rango de prioridad de cada uno, y cómo se ejecutan. A continuación comentaremos brevemente los operadores.

Operadores	Sentido
( ) { } -> .	I-D
! ~++ – - *(tipo)* * & *sizeof(todos unarios)*	D-I
* / %	I-D
+ -	I-D
< < > >	I-D
< <= > >=	I-D
== !=	I-D
&	I-D
^	I-D
|	I-D
&& (and)	I-D
|| (or)	I-D
?: (Operador Condicional Ternario)	I-D
= += -= *= /= %=	D-I
,	I-D

La acción de estos operadores es la siguiente:

1.2.6.1 Operadores Aritméticos

+	Suma los valores situados a su derecha y a su izquierda.
-	Resta el valor de su derecha del valor de su izquierda.
-	Como operador unario, cambia el signo del valor de su izquierda.
*****	Multiplica el valor de su derecha por el valor de su izquierda.
/	Divide el valor situado a su izquierda por el valor situado a su derecha.
%	Proporciona el resto de la división del valor de la izquierda por el valor de la derecha (sólo enteros).
++	Suma 1 al valor de la variable situada a su izquierda (modo prefijo) o de la variable situada a su derecha (modo sufijo).
- -	Igual que ++, pero restando 1.

1.2.6.2 Operadores de Asignación

=	Asigna el valor de su derecha a la variable de su izquierda.

Cada uno de los siguientes operadores actualiza la variable de su izquierda con el valor de su derecha utilizando la operación indicada. Usaremos de *d* e *i* para izquierda.

+=	Suma la cantidad *d* a la variable *i*.
-=	Resta la cantidad *d* de la variable *i*.
***=**	Multiplica la variable *i* por la variable *d*.
/=	Divide la variable *i* entre la cantidad *d*.
%=	Proporciona el resto de la división de la variable *i* por la cantidad *d*

Ejemplo

conejos *= 1.6; //es lo mismo que conejos = conejos * 1.6;

1.2.6.3 Operadores de Relación

Cada uno de estos operadores compara el valor de su izquierda con el valor de su derecha. La expresión de relación formada por un operador y sus dos operandos toma el valor 1 si la expresión es cierta, y el valor 0 si es falsa.

<	menor que
<=	menor o igual que
==	igual a
>=	mayor o igual que
>	mayor que
!=	distinto de

1.2.6.4 Operadores Lógicos

Los operadores lógicos utilizan normalmente expresiones de relación como operadores. El operador **!** toma un operando situado a su derecha; el resto toma dos: uno a su derecha y otro a su izquierda.

&& and	La expresión combinada es cierta si ambos operandos lo son, y falsa en cualquier otro caso.
|| or	La expresión combinada es cierta si uno o ambos operandos lo son, y falsa en cualquier otro caso.
!	La expresión es cierta si el operador es falso, y viceversa.

1.2.6.5 Operadores Relacionados con punteros

& Operador de Dirección Cuando va seguido por el nombre de una variable, entrega la dirección de dicha variable *&abc* es la dirección de la variable *abc*.

*** Operador de Indirección** Cuando va seguido por un puntero, entrega el valor almacenado en la dirección apuntada por él

```
1 abc = 22;
2 def = &abc; // puntero a abc
3 val = *def
```

El efecto neto es asignar a *val* el valor 22

1.2.6.6 Operadores de Estructuras y Uniones

. El operador de pertenencia (punto) se utiliza junto con el nombre de la estructura o unión, para especificar un miembro de las mismas. Si tenemos una estructura cuyo *nombre* es nombre, y *miembro* es un miembro especificado por el patrón de la estructura, **nombre.miembro** identifica dicho miembro de la estructura. El operador de pertenencia puede utilizarse de la misma forma en uniones. Ejemplo:

```
1 struct{
2         int codigo;
3         float precio;
4 }articulo;
5 articulo.codigo = 1265;
```

Con esto se asigna un valor al miembro **código** de la estructura **artículo**.

-> El operador de pertenencia indirecto: se usa con un puntero estructura o unión para identificar un miembro de las mismas. Supongo que **ptrstr** es un puntero a una estructura que contiene un miembro especificado en el patrón de estructura con el nombre miembro. En este caso

```
1 ptrstr -> miembro;
```

identifica al miembro correspondiente de la estructura apuntada. El operador de pertenencia indirecto puede utilizarse de igual forma con uniones.

```
1 struct{
2         int codigo;
3         float precio;
4 }articulo, *ptrstr;
5 ptrstr = &articulo;
6 ptrstr->codigo = 3451;
```

De este modo se asigna un valor al miembro **código** de la estructura **articulo**. Las tres expresiones siguientes son equivalentes:

ptrstr->código articulo.código (*ptrstr).codigo

1.2.6.7 Operadores Lógicos y de Desplazamiento de Bits

Operadores Lógicos

Estos son cuatro operadores que funcionan con datos de clase entera, incluyendo **char**. Se dice que son operadores que trabajan "bit a bit" porque pueden operar cada bit independientemente del bit situado a su izquierda o derecha.

~ El **complemento a uno o negación en bits** es un operador unario, cambia todos los 1 a 0 y los 0 a 1. Así:

$$\sim(10011010) == 01100101$$

& El **and de bits** es un operador que hace la comparación bit por bit entre dos operandos. Para cada posición de bit, el bit resultante es 1 únicamente cuando ambos bits de los operandos sean 1. En terminología lógica diríamos que el resultado es cierto si, y sólo si, los dos bit que actúan como operandos lo son también. Por tanto

$$(10010011) \ \& \ (00111101) == (00010001)$$

ya que únicamente en los bits 4 y 0 existe un 1 en ambos operandos.

| El **or para bits** es un operador binario realiza una comparación bit por bit entre dos operandos. En cada posición de bit, el bit resultante es 1 si alguno de los operandos o ambos contienen un 1. En terminología lógica, el resultado es cierto si uno de los bits de los operandos es cierto, o ambos lo son. Así:

$$(10010011) \ | \ (00111101) == (10111111)$$

porque todas las posiciones de bits con excepción del bit número 6 tenían un valor 1 en, por lo menos, uno de los operandos.

^ El **or exclusivo de bits** es un operador binario que realiza una comparación bit por bit entre dos operandos. Para cada posición de bit, el resultante es 1 si alguno de los operandos contiene un 1; pero no cuando lo contienen ambos a la vez. En terminología, el resultado es cierto si lo es el bit u otro operando, pero no si lo son ambos. Por ejemplo:

$$(10010011) \ \hat{} \ (00111101) == (10101110)$$

Observe que el bit de posición 0 tenía valor 1 en ambos operandos; por tanto, el bit resultante ha sido 0.

Operadores de Desplazamiento de Bits

Estos operadores desplazan bits a la izquierda o a la derecha. Seguiremos utilizando terminología binaria explicita para mostrar el funcionamiento.

< < El **desplazamiento a la izquierda** es un operador que desplaza los bits del operando izquierdo a la izquierda el número de sitios indicando por el operados de su derecha. Las posiciones vacantes se rellenan con ceros, y los bits que superan el límite del byte se pierden. Así:

$$(10001010)\text{«}2 == (1000101000)$$

cada uno de los bits se ha movido dos lugares hacia la izquierda

> > El **desplazamiento a la derecha** es un operador que desplaza los bits del operando situado a su izquierda hacia la derecha el número de sitios marcado por el operando situado a su derecha. Los bits que superan el extremo derecho del byte se pierden. En tipos **unsigned**, los lugares vacantes a la izquierda se rellenan con ceros. En tipos con signo el resultado depende del ordenador utilizado; los lugares vacantes se pueden rellenar con ceros o bien con copias del signo (bit extremo izquierdo). En un valor sin signo tendremos

$$(10001010)\text{»}2 == (00100010)$$

en el que cada bit se ha movido dos lugares hacia la derecha.

1.2.6.8 Misceláneas

sizeof Devuelve el tamaño, en bytes, del operando situado a su derecha. El operando puede ser un especificador de tipo, en cuyo caso se emplean paréntesis; por ejemplo, *sizeof(float)*. Puede ser también el nombre de una variable con concreta o de un array, en cuyo caso no se emplean paréntesis: *sizeof foto*

(tipo) Operador de moldeado, convierte el valor que vaya a continuación en el tipo especificado por la palabra clave encerrada entre los paréntesis. Por ejemplo, *(float)9* convierte el entero *9* en el número de punto flotante *9.0*.

, El operador coma une dos expresiones en una, garantizando que se evalúa en primer lugar la expresión situada a la izquierda, una aplicación típica es la inclusión de más información de más información en la expresión de control de un bucle *for*:

```
1 for ( chatos=2, ronda=0; ronda<1000; chatos*=2){
2         ronda += chatos;
3 }
```

?: El operador condicional se explicara mas adelante

1.3 Estructuras de Control

Las estructuras de control permiten modificar el flujo de ejecución de las instrucciones de un programa.
Con las estructuras de control se puede:

- De acuerdo a una condición, ejecutar un grupo u otro de sentencias (**If**)
- De acuerdo al valor de una variable, ejecutar un grupo u otro de sentencias (**Switch**)
- Ejecutar un grupo de sentencias mientras se cumpla una condición (**While**)
- Ejecutar un grupo de sentencias hasta que se cumpla una condición (**Do-While**)

- Ejecutar un grupo de sentencias un número determinado de veces (**For**)

Todas las estructuras de control tienen un único punto de entrada y un único punto de salida.

1.3.1 Sentencias de decisión

Definición

Las sentencias de decisión o también llamadas de **control de flujo** son estructuras de control que realizan una pregunta la cual retorna verdadero o falso (evalúa una condición) y selecciona la siguiente instrucción a ejecutar dependiendo la respuesta o resultado.

1.3.1.1 Sentencia if

La instrucción if es, por excelencia, la más utilizada para construir estructuras de control de flujo.

Sintaxis

Primera Forma

Ahora bien, la sintaxis utilizada en la programación de C++ es la siguiente:

```
1 if (condicion)
2    {
3        Set de instrucciones
4    }
```

siendo **condición** el lugar donde se pondrá la condición que se tiene que cumplir para que sea verdadera la sentencia y así proceder a realizar el **set de instrucciones** o código contenido dentro de la sentencia.

Segunda Forma

Ahora veremos la misma sintaxis pero ahora le añadiremos la parte **Falsa** de la sentencia:

```
1 if (condicion)
2    {
3        Set de instrucciones     //Parte VERDADERA
4    }
5 else
6    {
7        Set de instrucciones 2   //Parte FALSA
8    }
```

La forma mostrada anteriormente muestra la unión de la parte **verdadera** con la nueva secuencia la cual es la parte **falsa** de la sentencia de decisión **if**.

La palabra **Else** indica al lenguaje que de lo contrario al no ser verdadera o no se cumpla la parte verdadera entonces realizara el **set de instrucciones 2**.

Ejemplos De Sentencias If

Ejemplo 1:

```
if(numero == 0) //La condicion indica que tiene que ser igual a Cero
    {
      cout<<"El Numero Ingresado es Igual a Cero";
    }
```

Ejemplo 2:

```
if(numero > 0) // la condicion indica que tiene que ser mayor a Cero
    {
      cout<<"El Numero Ingresado es Mayor a Cero";
    }
```

Ejemplo 3:

```
if(numero < 0) // la condicion indica que tiene que ser menor a Cero
    {
      cout<<"El Numero Ingresado es Menor a Cero";
    }
```

Ahora uniremos todos estos ejemplos para formar un solo programa mediante la utilización de la sentencia **else** e introduciremos el hecho de que se puede escribir en este espacio una sentencia if ya que podemos ingresar cualquier tipo de código dentro de la sentencia escrita después de un **else**.
Ejemplo 4:

```
if(numero == 0) //La condicion indica que tiene que ser igual a Cero
   {
     cout<<"El Numero Ingresado es Igual a Cero";
   }
else
   {
      if(numero > 0) // la condicion indica que tiene que ser mayor a Cero
      {
          cout<<"El Numero Ingresado es Mayor a Cero";
      }
      else
      {
         if(numero < 0) // la condicion indica que tiene que ser menor a Cero
         {
           cout<<"El Numero Ingresado es Menor a Cero";
         }
      }
    }
```

1.3.1.2 Sentencia switch

Switch es otra de las instrucciones que permiten la construcción de estructuras de control. A diferencia de if, para controlar el flujo por medio de una sentencia switch se debe de combinar con el uso de las sentencias **case** y **break**. Cualquier número de casos a evaluar por switch así como la

sentencia default son opcionales. La sentencia switch es muy útil en los casos de presentación de menús.

Sintaxis

Ahora bien, la sintaxis utilizada en la programación de C++ es la siguiente:

```
switch (condicion)
{
    case primer_caso:
         bloque de instrucciones 1
    break;

    case segundo_caso:
         bloque de instrucciones 2
    break;

    case caso_n:
         bloque de instrucciones n
    break;

    default: bloque de instrucciones por defecto
}
```

Ejemplos De Sentencias Switch

Ejemplo 1

```
switch (numero)
{
    case 0: cout << "numero es cero";
}
```

Ejemplo 2

```
switch (opcion)
{
    case 0: cout << "Su opcion es cero"; break;
    case 1: cout << "Su opcion es uno"; break;
    case 2: cout << "Su opcion es dos";
}
```

Ejemplo 3

```
switch (opcion)
{
    case 1: cout << "Su opcion es 1"; break;
    case 2: cout << "Su opcion es 2"; break;
    case 3: cout << "Su opcion es 3"; break;
```

```
6       default: cout << "Elija una opcion entre 1 y 3";
7   }
```

1.3.1.3 Operador condicional ternario ?:

En C/C++, existe el operador condicional (?:) el cual es conocido por su estructura como ternario. El operador condicional ?: es útil para evaluar situaciones tales como:
Si se cumple tal condición entonces haz esto, de lo contrario haz esto otro.

Sintaxis

```
1   ( (condicion) ? proceso1 : proceso2 )
```

En donde, condición es la expresión que se evalúa, proceso 1 es la tarea a realizar en el caso de que la evaluación resulte verdadera, y proceso 2 es la tarea a realizar en el caso de que la evaluación resulte falsa.

Ejemplos De Sentencias Operador condicional ternario ?:

Ejemplo 1

```
1   int edad;
2   cout << "Cual es tu edad: ";
3   cin >> edad;
4   //Usando el operador condicional ternario
5   cout << ((edad<18) ? "Eres joven" : "Ya tienes la mayoria de edad");
6   //Usando if else
7   if (edad < 18)
8           cout << "Eres joven aun";
9   else
10          cout << "Ya tienes la mayoria de edad";
```

Ejemplo 2
Vamos a suponer que deseamos escribir una función que opere sobre dos valores numéricos y que la misma ha de regresar 1 (true) en caso de que el primer valor pasado sea igual al segundo valor; en caso contrario la función debe retornar 0 (false).

```
1   int es_igual( int a, int b)
2   {
3       return ( (a == b) ? 1 : 0 )
4   }
```

1.3.2 Sentencias de iteración

Definición

Las Sentencias de Iteración o Ciclos son estructuras de control que repiten la ejecución de un grupo de instrucciones. Básicamente, una sentencia de iteración es una estructura de control condicional, ya que dentro de la misma se repite la ejecución de una o más instrucciones mientras que una a condición específica se cumpla. Muchas veces tenemos que repetir un número definido o indefinido de veces un grupo de instrucciones por lo que en estos casos utilizamos este tipo de sentencias. En C++ los ciclos o bucles se construyen por medio de las sentencias for, while y do - while. La sentencia for es útil para los casos en donde se conoce de antemano el número de veces que una o más sentencias han de repetirse. Por otro lado, la sentencia while es útil en aquellos casos en donde no se conoce de antemano el número de veces que una o más sentencias se tienen que repetir.

1.3.2.1 Sentencias For

Sintaxis

Ahora bien, la sintaxis utilizada en la programación de C++ es la siguiente:

```
1 for(contador; final; incremento)
2 {
3     Codigo a Repetir;
4 }
```

Donde **contador** es una variable numérica, **final** es la condición que se evalúa para finalizar el ciclo (puede ser independiente del contador) e **incremento** es el valor que se suma o resta al contador.
Hay que tener en cuenta que el **for** evalúa la condición de finalización igual que el while, es decir, mientras esta se cumpla continuaran las repeticiones.

Ejemplos De Sentencias For

Ejemplo 1

```
1 for(int i=1; i<=10; i++)
2 {
3     cout<<"Hola Mundo";
4 }
```

Esto indica que el contador **i** inicia desde **1** y continuará iterando mientras **i** sea menor o igual a **10** (en este caso llegará hasta 10) e **i++** realiza la suma por unidad lo que hace que el for y el contador se sumen repitiendo 10 veces **Hola Mundo** en pantalla.
Ejemplo 2

```
1 for(int i=10; i>=0; i--)
2 {
3     cout<<"Hola Mundo";
4 }
```

Este ejemplo hace lo mismo que el primero, salvo que el contador se inicializa a 10 en lugar de 1; y por ello cambia la condición que se evalúa así como que el contador se decrementa en lugar de ser incrementado.
La condición también puede ser independiente del contador:
Ejemplo 3

```
int j = 20;
for(int i=0; j>0; i++){
    cout<<"Hola"<<i<<" - "<<j<<endl;
    j--;
}
```

En este ejemplo las iteraciones continuaran mientras **i** sea mayor que 0, sin tener en cuenta el valor que pueda tener i.

1.3.2.2 Sentencia While

Sintaxis

Ahora bien, la sintaxis utilizada en la programación de C++ es la siguiente:

```
while(condicion)
{
    codigo a Repetir
}
```

Donde **condición** es la expresión a evaluar.

Ejemplos De Sentencias While

Ejemplo 1

```
int contador = 0;

while(contador<=10)
{
    contador=contador+1;
    cout<<"Hola Mundo";
}
```

El contador indica que hasta que este llegue a el total de 10 entonces se detendrá y ya no se realizará el código contenido dentro de la sentencia while, de lo contrario mientras el **contador** sea menor a 10 entonces el código contenido se ejecutará desplegando hasta 10 veces **Hola Mundo** en pantalla.

1.3.2.3 Sentencia Do - While

Sintaxis

La sentencia do es usada generalmente en cooperación con while para garantizar que una o más instrucciones se ejecuten al menos una vez. Por ejemplo, en la siguiente construcción no se ejecuta nada dentro del ciclo while, el hecho es que el contador inicialmente vale cero y la condición para que se ejecute lo que está dentro del while es *"mientras el contador sea mayor que diez"*. Es evidente que a la primera evaluación hecha por while la condición deja de cumplirse.

```
int contador = 0;

```

```
3  while(contador > 10)
4  {
5      contador ++;
6      cout<<"Hola Mundo";
7  }
```

Al modificar el segmento de código anterior usando do tenemos:

```
1  int contador = 0;
2  do
3  {
4      contador ++;
5      cout<<"Hola Mundo";
6  }
7  while(contador > 10);
```

Observe cómo en el caso de **do** la condición es evaluada al final en lugar de al principio del bloque de instrucciones y, por lo tanto, el código que le sigue al *do* se ejecuta al menos la primera vez.

1.3.3 Sentencias Break y Continue

En la sección (Sentencia switch) vimos que la sentencia break es utilizada con el propósito de forzar un salto dentro del bloque switch hacia el final del mismo. En esta sección volveremos a ver el uso de break, salvo que esta ocasión la usaremos junto con las sentencias for y la sentencia while. Además, veremos el uso de la sentencia continue.

1.3.3.1 Break

La sentencia break se usa para forzar un salto hacia el final de un ciclo controlado por for o por while.

Ejemplo

En el siguiente fragmento de código la sentencia break cierra el ciclo for cuando la variable **i** es igual a 5.

```
1  for (int i=0; i<10; i++) {
2      if (i == 5) break;
3      cout << i << " ";
4  }
```

La salida para el mismo será:

```
1  0 1 2 3 4
```

1.3.3.2 Continue

La sentencia continue se usa para ignorar una iteración dentro de un ciclo controlado por for o por while.

Ejemplo

En el siguiente fragmento de código la sentencia continue ignora la iteración cuando la variable **i** es igual a 5.

```
for (int i=0; i<10; i++) {
    if (i == 5) continue;
    cout << i << " ";
}
```

La salida para el mismo será:

```
0 1 2 3 4 6 7 8 9
```

1.3.3.3 Uso de break y continue junto con while

Los dos ejemplos anteriores se presentan en seguida, salvo que en lugar de for se hace uso de while. Observe que la construcción del ciclo while para el caso de la sentencia continue es diferente, esto para garantizar que el ciclo no vaya a caer en una iteración infinita.

Break

```
int i = 0;
while (i<10) {
    if (i == 5) break;
    cout << i << " ";
    i++;
}
```

Continue

```
int i = -1;
while (i<10) {
    i++;
    if (i == 5) continue;
    cout << i << " ";
}
```

2. Estructura de datos 1

2.1 Introducción

En este capítulo veremos más de **C++**, el uso de sus librerías. **C++** provee de librerías muy útiles que nos harán de mucha ayuda, en este capítulo veremos cómo utilizarlas y como trabajan en el interior.

2.1.1 ¿Qué es una estructura de datos?

Una estructura de datos es una forma de organizar un conjunto de datos elementales con el objetivo de facilitar su manipulación. Un dato elemental es la mínima información que se tiene en un sistema. Una estructura de datos define la organización e interrelación de estos y un conjunto de operaciones que se pueden realizar sobre ellos. Cada estructura ofrece ventajas y desventajas en relación a la simplicidad y eficiencia para la realización de cada operación. De está forma, la elección de la estructura de datos apropiada para cada problema depende de factores como la frecuencia y el orden en que se realiza cada operación sobre los datos.

2.1.2 Genericidad

Una clase genérica encapsula a una estructura cuyo comportamiento es independiente del tipo de las componentes, esto favorece la reusabilidad. Lo cual implica que estas clases pueden llegar a ser de cualquier tipo de dato incluso de otra clase genérica o no.

2.2 Estructuras Estáticas

Las estructuras estáticas son aquellas que el tamaño de las mismas no puede ser modificadas en la ejecución del algoritmo. Así su tamaño tiene que ser defino en la creación de la misma.
Estas se dividen en dos:

- Estructuras Estáticas Simples
- Estructuras Estáticas Compuestas

2.2.1 Estructuras Estáticas Simples

Estas estructuras de datos son también llamadas **Primitivas**, en la mayoría de los lenguajes existen las siguientes estructuras de datos primitivas:

Booleano	Está estructura es la más simple solo ocupa un byte puede llegar a almacenar 1 (verdad) o 0 (falso).
Carácter	Está estructura llega a almacenar un carácter en general como números, letras y símbolos, ocupa en memoria 8 bits, desde -128 a 127.
Entero	Está estructura puede almacenar a un entero desde -21474836482147483647, ocupa en memoria 32 bits.
Punto Flotante	Está estructura almacena números reales desde 1.2e-308 a 3.4e-38, ocupa en memoria 32 bits.
Entero Largo	Está estructura almacena también enteros pero con más capacidad desde -9223372036854775808 a 9223372036854775807, ocupa en memoria 64 bits.
Double	Está estructura almacena también reales como los float pero con más capacidad desde 2.2e-308 a 3.4e-38, ocupa en memoria 64 bits.

Implementación de estructuras estáticas simples:

Código 2.1: Implementación: Estructuras Estáticas Simples

```
1
2  #include <iostream> //Está es nuestra cabecera
3  using namespace std;
4  int main(){
5          //Dentro se escribirá todo el código
6          //Creacion de variables
7          bool dato_boolean;
8          char dato_char;
9          int dato_integer;
10         float dato_float;
11         long long dato_long_integer;
12         double dato_double;
13         //Asignando valores
14         dato_boolean = 0;
15         dato_char = 'A';
16         dato_integer = 12345678;
17         dato_float = 12.161;
18         dato_long_integer = 123456789123;
19         dato_double = 12.1235;
20         //También podemos asignar valores en la creación
21         int integer = 123;
22         //Impresion
23         cout<<"Boolean:"<<dato_boolean<<endl;
24         cout<<"Caracter:"<<dato_char<<endl;
25         cout<<"Integer:"<<dato_integer<<endl;
26         cout<<"Float:"<<dato_float<<endl;
27         cout<<"Long integer:"<<dato_long_integer<<endl;
28         cout<<"Double:"<<dato_double<<endl;
29         return 0;
30 }
```

2.2.2 Estructuras Estáticas Compuestas

2.2.2.1 Arreglos

Está estructura almacena varios elementos del mismo tipo en forma lineal.
Este es un arreglo de enteros de tamaño [10].

Índice	0	1	2	3	4	5	6	7	8	9
int A[10]	445	123	523	-6546	867	0	-7890	23	3453	190

Este es un arreglo de caracteres de tamaño [3].

Índice	0	1	2
char A[3]	'f'	'0'	'?'

2.2.2.2 Matrices

Está estructura almacena varios elementos del mismo tipo en forma planar.
Este es una matriz de doubles de tamaño [5][4].

	Índices	0	1	2	3	4
double M[8][4]	0	5.345	1.0	5.0	-546.114	7.124
	1	12.12	-45.7	-4.8	56.5	7.4
	2	13.3	4.12	-79.3	13.51	45.13
	3	45.1	6.41	1.1315	2.5	3.12343

Este es una matriz de booleanos de tamaño [5][3].

	Índices	0	1	2	3	4
bool M[5][3]	0	0	1	0	1	1
	1	0	1	1	0	1
	2	0	1	0	0	1

2.2.2.3 Pares

Está estructura almacena solo dos elementos del mismo tipo o diferente tipo.
Este es un par de <caracter, entero>.

Miembros	First	Second
Pair<char,int> P	'A'	65

Este es un pair de <double, booleano>.

Miembros	First	Second
Pair<double,bool> P	12.231441	1

2.2.2.4 Implementación de estructuras estáticas compuestas

Código 2.2: Implementación: Estructuras Estáticas Simples Compuestas

```
1 #include <iostream>
2
3 using namespace std;
4
```

```
5  int main() //Dentro se escribirá todo el código
6  {
7          //Creación de variables
8          bool est_array[5];
9          int matriz[2][2];
10         pair<int, char> dato_pair;
11         //Asignando valores
12         est_array[0] = true;
13         est_array[1] = false;
14         est_array[2] = false;
15         est_array[3] = true;
16         est_array[4] = false;
17         matriz[0][0] = 123;
18         matriz[0][1] = 1245;
19         matriz[1][0] = 7896;
20         matriz[1][1] = 1654;
21         dato_pair.first = 122314;
22         dato_pair.second = 'Z';
23         //Acceso a los valores
24         for(int i=0; i<5; i++){
25                 cout<<est_array[i]<<" ";
26         }
27         cout<<endl;
28         //Los índices empiezan desde 0
29         for(int i=0; i<2; i++){
30                 for(int j=0; j<2; j++)
31                         cout<<matriz[i][j]<<" ";
32                 cout<<endl;
33         }
34         cout<<endl;
35         //Acceso a los miembros del Pair
36         cout<<"Pair:"<<dato_pair.first<<","<<dato_pair.second<<endl;
37         //También podemos asignar valores en la creación
38         char est_array_char[] = {'1', '3', 'h'};
39         //Impresion
40         for(int i=0; i<3; i++){
41         cout<<est_array_char[i]<<" ";
42         }
43         cout<<endl;
44         return 0;
45 }
```

2.3 Estructuras Dinámicas

Las estructuras dinámicas son aquellas que el tamaño no está definido y puede cambiar en la ejecución del algoritmo. Estas estructuras suelen ser eficaces y efectivas en la solución de problemas complejos. Su tamaño puede reducir o incrementar en la ejecución del algoritmo.

2.3.1 Estructuras Dinámicas Lineales

Entre las estructuras dinámicas lineales están:

- Vector
- Pila
- Cola

2.3.1.1 Vector

Para la utilización de está estructura es necesario añadir en la cabecera del programa **#include <vector>**, gracias a esto ya podemos utilizar la estructura de otro modo no. Vector es una clase genérica y está pensada para operar con arreglos unidimensionales de datos del mismo tipo.

2.3.1.1.1 Creación

Para la creación de un vector se le debe mandar el tipo de dato. De la siguiente manera:

```
1 vector<int> Vec;
```

Otra manera de creación puede realizarse mandando al constructor el tamaño inicial del vector:

```
1 vector<int> Vec(5);
```

De está manera se reservara 8 espacios de memoria de tipo entero decimal para la utilización de estos se debe primero llenar de lo contrario retornara datos basura si se llega a acceder a uno de ellos.

.begin() Está función retorna el iterador del primer elemento, si el vector está vacío retorna **.end()**.

.end() Está función retorna el iterador del elemento siguiente al último elemento del vector

2.3.1.1.2 Iteradores

Estos son punteros que son utilizados para el acceso directo a sus elementos.

2.3.1.1.3 Acceso

El acceso al vector se puede hacer directamente mediante los corchetes **[índice]** como si fuera un array estático de la siguiente manera, siempre y cuando el vector tenga creado ese espacio de memoria.

```
1  vector<int> Vec(5);
2 Vec[0] = 445;
3 Vec[1] = 123;
4 Vec[2] = 523;
5 Vec[3] = -6546;
6 Vec[4] = 867;
7 //Vec[5] = 458; //Vec[9] no existe
```

Existe también dos funciones auxiliares para el acceso al vector estas son **.front()** que retorna el primer elemento del vector y **.back()** que retorna al último elemento del vector, el uso es el siguiente.

```
1 int primer_elemento = Vec.front();
2 int ultimo_elemento = Vec.back();
```

2.3.1.1.4 Ilustración

Índice	0	1	2	3	4
vector<int> vec(5)	445	123	523	-6546	867
Acceso	.front(), vec[0]	vec[1]	vec[2]	vec[3]	.back(), vec[4]

2.3.1.1.5 Funciones de capacidad

Las más importantes son dos:

.empty() Está función retorna verdad si está vacía y retorna falso si es que por lo menos tiene un elemento.

.size() Está función retorna cuantos elementos tiene el vector, es muy necesaria en estructuras dinámicas ya que el tamaño suele variar.

A continuación un ejemplo de manejo.

```
int tam = Vec.size();
if(Vec.empty()){
    cout<<"El vector esta vacio\n";
}else{
    cout<<"El vector lleva "<<tam<< " elementos\n";
}
```

2.3.1.1.6 Funciones modificadoras

Existen varias funciones útiles en el manejo de vectores entre ellas están:

.clear() Está función elimina todos los elemtos del vector logrando así que el tamaño (.size()) sea 0

.insert(pos, cant, valor) Está función inserta cierta cantidad (cant) de valores(valor) en una determinada posición(pos), **pos** debe ser un iterador, **cant** un numero entero y **valor** un tipo de dato aceptado por el vector

.erase(inicio, fin) Está función elimina elemtos desde una posición inicial (inicio) hasta otra posición (fin), **inicio** y **fin** deben ser iteradores

.push_back(valor) Está función adiciona un valor (valor) al final del vector, **valor** debe ser un tipo de dato aceptado por el vector.

.pop_back() Está función remueve el último elemento del vector.

.resize(tam) Está función modifica el tamaño del vector, **tam** debe ser un número entero, si **tam** es más pequeño que el tamaño del vector entonces elimina los elementos del final pero si **tam** es más grande que el tamaño del vector entonces asigna 0's a los elementos nuevos.

.swap(vec) Está función intercambia todo el contenido, **vec** debe ser un vector del mismo tipo de datos.

Código 2.3: Implementación: Vector

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

using namespace std;

void printVector(std::vector<int>& vec)
{
```

```
8       for (int i =0; i< vec.size(); i++)
9       {
10          std::cout << vec[i] << " ";
11      }
12  }
13
14  int main()
15  {
16      vector<int> v1(2);
17      v1[0] = 100;
18      v1[1] = 200;
19      cout<<"v1: ";
20      printVector(v1);
21      cout<<"\nv1 luego de .clear():";
22      v1.clear();
23      printVector(v1);
24      cout<<"\nv1 luego de .insert(v1.begin(), 2, 159):";
25      v1.insert(v1.begin(), 2, 159);
26      printVector(v1);
27      cout<<"\nv1 luego de .erase(v1.begin()):";
28      v1.erase(v1.begin());
29      printVector(v1);
30      cout<<"\nv1 luego de .push_back(753):";
31      v1.push_back(753);
32      printVector(v1);
33      cout<<"\nv1 luego de .pop_back():";
34      v1.pop_back();
35      printVector(v1);
36      cout<<"\nv1 luego de .resize(4):";
37      v1.resize(4);
38      printVector(v1);
39
40      vector<int> v2;
41      v2.push_back(123);
42      cout << "\nv2: ";
43      printVector(v2);
44      cout << "\nSWAP\n";
45      v2.swap(v1);
46      cout << "v1: ";
47      printVector(v1);
48      cout << "\nv2: ";
49      printVector(v2);
50  }
```

Salida:

```
1  v1: 100 200
2  v1 luego de .clear():
3  v1 luego de .insert(v1.begin(), 2, 159):159 159
4  v1 luego de .erase(v1.begin()):159
5  v1 luego de .push_back(753):159 753
6  v1 luego de .pop_back():159
7  v1 luego de .resize(4):159 0 0 0
```

```
v2: 123
SWAP
v1: 123
v2: 159 0 0 0
```

2.3.1.2 Pila

Para la utilización de está estructura es necesario añadir en la cabecera del programa **#include <stack>**, gracias a esto ya podemos utilizar la estructura de otro modo no.
Una pila es una estructura de datos en la que el modo de acceso a sus elementos es de tipo **LIFO** (del inglés Last In First Out, último en entrar, primero en salir) que permite almacenar y recuperar datos. Para el manejo de los datos se cuenta con dos operaciones básicas: **apilar**, que coloca un objeto en la pila, y su operación inversa, **des-apilar** (retirar), que retira el último elemento apilado. La clase **stack** es genérica.
Las pilas suelen emplearse en los siguientes contextos:

- Evaluación de expresiones en notación postfija (notación polaca inversa).
- Reconocedores sintácticos de lenguajes independientes del contexto
- Implementación de recursividad.

2.3.1.2.1 Creación

Para la creación de una pila se le debe mandar el tipo de dato. De la siguiente manera:

```
stack<int> stc;
```

Otra manera de creación puede realizarse mandando al constructor otra pila:

```
stack<int> stc2(stc1);
```

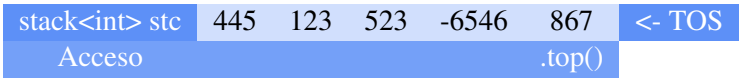
De está manera se copiará los elementos de la pila **stc1** a la nueva pila **stc2**.

2.3.1.2.2 Acceso

En cada momento sólo se tiene acceso a la parte superior de la pila, es decir, al último objeto apilado (denominado TOS, Top Of Stack en inglés). La función **.top()** la obtención de este elemento, que cuando es retirado de la pila permite el acceso al siguiente (apilado con anterioridad), que pasa a ser el nuevo TOS.

```
stack<int> stc;
stc.push(2);
stc.push(12);
int tos = stc.top() // tos llegaría a ser 12
```

2.3.1.2.3 Ilustración



2.3.1.2.4 Funciones de capacidad

Las dos únicas funciones existentes son:

.empty() Está función retorna verdad si la pila está vacía y retorna falso si es que por lo menos tiene un elemento como tos.

.size() Está función retorna cuantos elementos tiene la pila, pero sin embargo no se puede acceder a ellas por lo que no es muy usual el uso de está función.

A continuación un ejemplo de manejo.

```
stack<int> stc;
int tam = stc.size; //resultado 0
while(tam < 5){
    stc.push(100);
    tam = stc.size();
}
cout<<stc.size()<<"\n"; //resultado 5
```

2.3.1.2.5 Funciones modificadoras

Existen dos funciones en el manejo de pilas:

.push(valor) Está función apila un valor (valor) en el tope de la pila llegando a ser el nuevo **tos** de la pila, **valor** debe ser un tipo de dato aceptado por la pila.

.pop() Está función des-apila el elemento tope de la pila dando paso así a un nuevo **tos**.

A continuación un ejemplo de manejo.

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

using namespace std;

int main()
{
    stack<int> stc;
    stc.push(100);
    stc.push(200);
    stc.push(300);
    cout<<stc.top()<<"\n"; //resultado 300
    stc.pop();
    cout<<stc.top()<<"\n"; //resultado 200
}
```

2.3.1.3 Cola

Para la utilización de está estructura es necesario añadir en la cabecera del programa **#include <queue>**, gracias a esto ya podemos utilizar la estructura de otro modo no. La clase **queue** es genérica.

Una cola es una estructura de datos en la que el modo de acceso a sus elementos es de tipo **FIFO** (del inglés First In First Out, primero en entrar, primero en salir) que permite almacenar y recuperar datos.Así, los elementos sólo se pueden eliminar por el principio y sólo se pueden añadir por el final de la cola.

2.3.1.3.1 Creación

Para la creación de una cola se le debe mandar el tipo de dato. De la siguiente manera:

```
queue<int> que;
```

Otra manera de creación puede realizarse mandando al constructor otra cola:

```
queue<int> que2(que1);
```

De está manera se copiará los elementos de la cola **que1** a la nueva cola **que2**.

2.3.1.3.2 Acceso

En una cola solo podemos acceder al primero o bien al último elemento de la estructura, para esto tenemos dos funciones, **.front()** que nos permite acceder al primer elemento y **.back()** que nos permite acceder al último elemento.

```
queue<int> que;
que.push(100);
que.push(200);
que.push(300);
cout<<que.front()<<"\n"; //resultado 100
cout<<que.back()<<"\n"; //resultado 300
```

2.3.1.3.3 Ilustración

2.3.1.3.4 Funciones de capacidad

Las dos únicas funciones existentes son:

.empty() Está función retorna verdad si la cola está vacía y retorna falso si es que por lo menos tiene un elemento.

.size() Está función retorna cuantos elementos tiene la cola.

A continuación un ejemplo de manejo.

```
queue<int> que;
que.push(500);
int tam = que.size(); //tam llegaría a ser 1
while(tam < 5){
    que.push(100);
    tam = que.size();
}
cout<<que.size()<<"\n"; //resultado 5
cout<<que.front()<<" "<<que.back()<<"\n"; //resultado 500 100
```

2.3.1.3.5 Funciones modificadoras

Existen dos funciones en el manejo de colas:

.push(valor) Está función inserta un elemento (valor) al final de la cola, **valor** debe ser un tipo de dato aceptado por la cola.

.pop() Está función elimina el primer elemento de la cola.

A continuación un ejemplo de manejo.

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

using namespace std;

int main()
{
    //freopen("out.txt", "w", stdout);
    queue<int> que;
    que.push(500);
    cout<<que.front()<<" "<<que.back()<<"\n"; //resultado 500 500
    que.push(1000);
    cout<<que.front()<<" "<<que.back()<<"\n"; //resultado 500 1000
    que.pop();
    cout<<que.front()<<" "<<que.back()<<"\n"; //resultado 1000 1000
    return 0;
}
```

2.3.2 Estructuras Dinámicas No Lineales

Estas estructuras se caracterizan por que el acceso y la modificación ya no son constantes, es necesario especificar la complejidad de cada función de ahora en adelante.
Entre las estructuras dinámicas no lineales están:

- Arboles
- Grafos
- Cola de Prioridad
- Conjunto
- Aplicación

2.3.2.1 Cola de Prioridad

Para la utilización de está estructura es necesario añadir en la cabecera del programa **#include <queue>**, gracias a esto ya podemos utilizar la estructura de otro modo no.
Una cola de prioridad es una estructura de datos en la que los elementos se atienden en el orden indicado por una prioridad asociada a cada uno. Si varios elementos tienen la misma prioridad, se atenderán de modo convencional según la posición que ocupen, puede llegar a ofrecer la extracción constante de tiempo del elemento más grande (por defecto), a expensas de la inserción logarítmica, o utilizando **greater<int>** causaría que el menor elemento aparezca como la parte superior con **.top()**.
Trabajar con una cola de prioridad es similar a la gestión de un **heap**

2.3.2.1.1 Creación

Para la creación de una cola prioridad se le debe mandar el tipo de dato, si el dato es cualquier tipo de entero decimal, la prioridad sera el valor del mismo elemento.

```cpp
priority_queue<int> pq;
```

Otra manera de creación puede realizarse mandándole el tipo de dato, vector del mismo dato y el comparador:

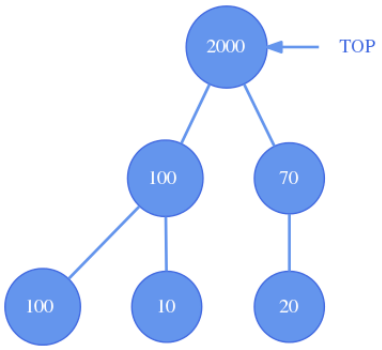
```
priority_queue< int, vector<int>, greater<int> > pqg;
```

De está manera la prioridad sera inversa al valor del elemento.

2.3.2.1.2 Acceso

En una cola de prioridad solo podemos acceder a la raíz del **heap** con la función **.top()** está función si es constante ya que siempre se sabe dónde está la raíz del **heap**.

2.3.2.1.3 Ilustración



2.3.2.1.4 Funciones de capacidad

Las dos únicas funciones existentes son:

.empty() Está función retorna verdad si la cola de prioridad está vacía y retorna falso en otro caso.

.size() Está función retorna cuantos elementos tiene la cola de prioridad.

A continuación un ejemplo de manejo.

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

using namespace std;

int main()
{
    //Creación de una cola de prioridad similar a la ilustración
    priority_queue< int> pq;
    //Añadimos a la cola de prioridad los elementos
    pq.push(100); pq.push(10);
    pq.push(20); pq.push(100);
    pq.push(70); pq.push(2000);
    if(pq.empty()){
        cout<<"La cola de prioridad esta vacia\n";
    }else{
        cout<<"La cola de prioridad lleva ";
        cout<<pq.size()<<" elementos\n";
    }
    //Salida :  La cola de prioridad tiene 6 elementos
    return 0;
}
```

2.3.2.1.5 Funciones modificadoras

Existen dos funciones en el manejo de colas:

.push(valor) Está función inserta un elemento (valor) a la cola de prioridad, **valor** debe ser un tipo de dato aceptado por la cola de prioridad. Está función tiene complejidad logarítmica **O(log_2 N)**

.pop() Está función elimina el **top** de la cola de prioridad. Está función tiene una complejidad logarítmica también **O(log_2 N)**.

A continuación un ejemplo de manejo.

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

using namespace std;

int main()
{
    //Creación de una cola de prioridad similar a la ilustración
    priority_queue< int> pq;
    //Añadimos a la cola de prioridad los elementos
    pq.push(100); pq.push(10);
    pq.push(20); pq.push(100);
    pq.push(70);
    pq.push(2000);
    while(!pq.empty()){
        // O(1) el acceso es constante
        int a = pq.top();
        // O(log n) la eliminación no es constante
        pq.pop();
        cout<<a<<"\n";
    }//Salida:  2000 100 100 70 20 10
    return 0;
}
```

2.3.2.2 Conjunto

Conjunto es un contenedor asociativo que contiene un conjunto ordenado de objetos únicos. La ordenación se realiza utilizando la función de comparación. Los conjuntos se implementan normalmente como árboles rojo-negro, por lo que sus funciones principales (búsqueda, eliminación e inserción) tienen complejidad logarítmica **O(log_2 N)**.

2.3.2.2.1 Creación

Para la creación de un conjunto se debe mandar el tipo de dato siempre y cuando tenga un comparador por defecto, los datos primitivos si llevan comparadores definidos por defecto.

```
set<int> conj;
```

Si se desearía crear un conjunto de otro tipo de datos que no sean primitivos de c++ se debe implementar una función booleana que acepta dos variables del tipo de dato deseado. Para la creación de este tipo de conjuntos se debe mandar aparte del Tipo_de_Dato lo siguiente, **bool**

(*)(Tipo_de_Dato , Tipo_de_Dato) y en el constructor mandarle el comparador que llegaría a ser la función que vimos hace un momento. Quedaría de la siguiente manera:

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

using namespace std;

bool comvec(vector<int> v1, vector<int> v2){ //Está es el comparador
    return v1.size()< v2.size();
}

int main()
{
    //Creación de un conjunto con 'vector<int>' como Tipo_de_Dato
    set<vector<int>, bool (*)(vector<int> , vector<int>)> conjvec(
        comvec);
    return 0;
}
```

2.3.2.2.2 Iteradores

Estos son punteros que se utilizan para acceder a los elementos, un itereador soporta el operador **++**, lo cual significa que podemos usarlo en sentencias de iteración(**for**).

.begin() Está función retorna el iterador del primer elemento, si el conjunto está vacio retorna **.end()**.

.end() Está función retorna el iterador del elemento siguiente al último elemento del conjunto

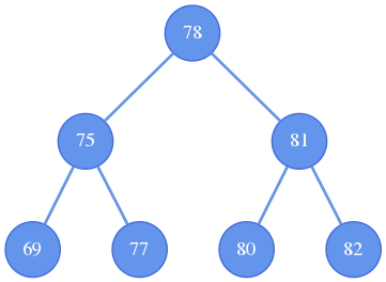
A continuación un ejemplo de manejo.

```
set<int> conj;
conj.insert(69); conj.insert(80);
conj.insert(77); conj.insert(82);
conj.insert(75); conj.insert(81);
for(set<int>::iterator it=conj.begin(); it!=conj.end(); it++){
    cout<<*it<<" ";
}//Resultado:  69 75 77 80 81 82
```

2.3.2.2.3 Acceso

El acceso a está estructura se debe hacer con iteradores, o a ciertos datos por medio de funciones.

2.3.2.2.4 Ilustración



2.3.2.2.5 Funciones de capacidad

Las dos únicas funciones existentes son:

.empty() Está función retorna verdad si el conjunto vacio y retorna falso en otro caso.
.size() Está función retorna cuantos elementos tiene el conjunto.

A continuación un ejemplo de manejo.

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

using namespace std;

int main()
{
    //Creación de una conjunto similar a la ilustración
    set< int> conj;
    //Añadimos al conjunto los elementos
    conj.insert(69); conj.insert(80);
        conj.insert(77); conj.insert(82);
        conj.insert(75); conj.insert(81);
        conj.insert(78);
    if(conj.empty()){
        cout<<"El conjunto esta vacia\n";
    }else{
        cout<<"El conjunto lleva ";
        cout<<conj.size()<<" elementos\n";
    }
    //Salida :  El conjunto lleva 7 elementos
    return 0;
}
```

2.3.2.2.6 Funciones modificadoras

Existen varias funciones útiles en el manejo de conjuntos entre ellas están:

.clear()	Está función elimina todos los elemtos del conjunto logrando así que el tamaño (.size()) sea 0
.insert(pos)	Está función inserta un elemento al conjunto si este ya existe lo ignorara, **valor** debe ser un tipo de dato aceptado por el conjunto. La complejidad de está función es **O**(log_2 **N)**.
.erase(inicio, fin)	Está función elimina elemtos desde una posición inicial (inicio) hasta otra posición (fin), **inicio** y **fin** deben ser iteradores. La complejidad es lineal
.erase(valor)	Está función busca el elemnto y lo elimina, **valor** debe ser un tipo de dato aceptado por el conjunto. La complejidad de está función es **O**(log_2 **N)**.

2.3.2.2.7 Funciones especificas

.count(valor)	Está función retorna la cantidad de elementos coincidentes con **valor** lo cual está entre 0 y 1 ya que no existen elementos dobles o repetidos, **valor** debe ser un tipo de dato aceptado por el conjunto. La complejidad de está función es **O**(log_2 **N)**.
.find(valor)	Está función retorna el iterador apuntando al elemnto **valor** si existe y si no retorna **.end()**, **valor** debe ser un tipo de dato aceptado por el conjunto. La complejidad de está función es **O**(log_2 **N)**.
.lower_bound(valor)	Está función retorna un iterador apuntando al primer elemento no menos que **valor**, **valor** debe ser un tipo de dato aceptado por el conjunto. La complejidad de está función es **O**(log_2 **N)**.
.upper_bound(valor)	Está función retorna un iterador apuntando al primer elemento mas grande que **valor**, **valor** debe ser un tipo de dato aceptado por el conjunto. La complejidad de está función es **O**(log_2 **N)**.

A continuación un ejemplo de manejo.

```
1  #include <iostream>
2  #include <set>
3
4  using namespace std;
5
6  int main()
7  {
8      set<int> conj;
9      conj.insert(69); conj.insert(80);
10         conj.insert(77); conj.insert(82);
11         conj.insert(75); conj.insert(81);
12         for(set<int>::iterator it=conj.begin(); it!=conj.end(); it
               ++){
13         cout<<*it<<" ";
14         }cout<<"\n";
15         //Resultado:  69 75 77 80 81 82
16         if(conj.count(77)==1)
17         cout<<"77 esta en el conjunto\n";
18     else
19         cout<<"77 no esta en el conjunto\n";
20     //Resultado:  77 está en el conjunto
21     if(conj.find(100)!=conj.end())
22         cout<<"100 esta en el conjunto\n";
```

```
23      else
24          cout<<"100 no esta en el conjunto\n";
25      //Resultado:  100 no está en el conjunto
26      if(conj.lower_bound(80)!=conj.upper_bound(80))
27          cout<<"80 esta en el conjunto\n";
28      else
29          cout<<"80 no esta en el conjunto\n";
30      //Resultado:  80 está en el conjunto
31      if(conj.lower_bound(70)==conj.upper_bound(70))
32          cout<<"70 no esta en el conjunto\n";
33      else
34          cout<<"70 esta en el conjunto\n";
35      //Resultado:  70 no está en el conjunto
36      return 0;
37  }
```

2.3.2.3 Mapa

Mapa es un contenedor asociativo que contiene pares de llaves y valores, las llave son únicas más no así los valores. La ordenación se realiza utilizando la función de comparación. Las aplicaciones se implementan normalmente como árboles rojo-negro, por lo que sus funciones principales (búsqueda, eliminación e inserción) tienen complejidad logarítmica **O(**log_2 **N)**.

2.3.2.3.1 Creación

Para la creación de un mapa se debe mandar el tipo de dato de las llaves y el tipo de dato de los valores, siempre y cuando el tipo de dato de las llaves tenga un comparador por defecto, los datos primitivos sí llevan comparadores definidos por defecto.

```
1  map<char, int> apl;
```

Si se desearía crear un conjunto de otro tipo de datos que no sean primitivos de c++ se debe implementar una función booleana que acepta dos variables del tipo de dato deseado. Para la creación de este tipo de conjuntos se debe mandar aparte del Tipo_de_Dato lo siguiente, **bool (*)(Tipo_de_Dato , Tipo_de_Dato)** y en el constructor mandarle el comparador que llegaría a ser la función que vimos hace un momento. Quedaría de la siguiente manera:

```
1   #include <iostream>
2   #include <map>
3   #include <vector>
4
5   using namespace std;
6
7   bool comvec(vector<int> v1, vector<int> v2){ //Está es el comparador
8       return v1.size()< v2.size();
9   }
10
11  int main()
12  {
13      //Creación de una aplicación con 'vector<int>' como Tipo_de_Dato de las
            llaves
```

```
14      map<vector<int>, int, bool (*)(vector<int> , vector<int>)>
            aplvec(comvec);
15      return 0;
16  }
```

2.3.2.3.2 Iteradores

Estos son punteros utilizados para acceder a los elementos un itereador soporta el operador **++**, lo cual significa que podemos usarlo en sentencias de iteración **for**.

.begin() Está función retorna el iterador apuntando al primer elemento, si la aplicación está vacía retorna **.end()**.

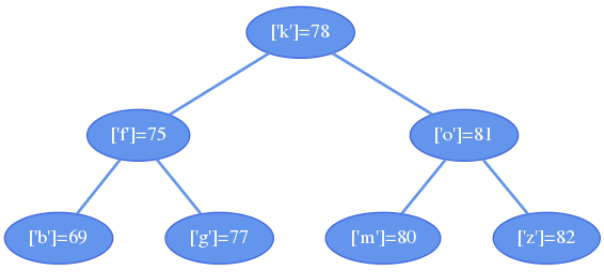
.end() Está función retorna el iterador apuntando al elemento siguiente al último elemento de la aplicación.

2.3.2.3.3 Acceso

Un mapa tiene la ventaja de acceder a los valores directamente mediante los corchetes **[llave]** y llaves, como si fuera un vector. Si se accede a un elemento mediante la llave y el corchete que no existe, se crea automáticamente asignando 0's como valores.
También se puede acceder a las llaves y valores mediante iteradores de la siguiente manera:

```
1   map<char, int> apl;
2   apl.insert(make_pair('a', 13));
3   apl.insert(make_pair('b', 98));
4   cout<<apl['a']<<"\n";
5   cout<<apl['b']<<"\n";
6   //Notese que no existe apl['c] por lo que se creara y pondra como valor 0
7   cout<<apl['c']<<"\n";
8
9   //Acceso a las llaves y valores mediante iteradores
10  for(map<char, int>::iterator it=apl.begin(); it!=apl.end(); it++){
11      cout<<it->first<<" "<<it->second<<"\n";
12  }
13  //Resultado:  a 13 b 98 c 0
```

2.3.2.3.4 Ilustración



2.3.2.3.5 Funciones de capacidad

Las dos únicas funciones existentes son:

.empty() Está función retorna verdad si la aplicación está vacía y retorna falso en otro caso.

.size() Está función retorna cuantos elementos tiene la aplicación.

A continuación un ejemplo de manejo.

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

using namespace std;

int main()
{
    map<char, int> apl;
    apl.insert(make_pair('a', 13));
    apl.insert(make_pair('b', 98));
    cout<<apl['a']<<"\n";
    cout<<apl['b']<<"\n";
    cout<<apl['c']<<"\n";

    if(apl.empty()){
        cout<<"La aplicacion esta vacia\n";
    }else{
        cout<<"La aplicacion lleva ";
        cout<<apl.size()<<" elementos\n";
    }
    //Salida :  La aplicacion lleva 3 elementos
    return 0;
}
```

2.3.2.3.6 Funciones modificadoras

Existen varias funciones útiles en el manejo de aplicaciones entre ellas están:

.clear()	Está función elimina todos los elemtos de la aplicación logrando así que el tamaño (.size()) sea 0
.insert(make_pair(llave, valor))	Está función inserta un par de elementos la llave y el valor designado a la llave, **llave** y **valor** debe ser un tipo de dato aceptado por la aplicación. La complejidad de está función es **O**(log_2 **N**).
.erase(inicio, fin)	Está función elimina elemtos desde una posición inicial (inicio) hasta otra posición (fin), **inicio** y **fin** deben ser iteradores. La complejidad es lineal
.erase(llave)	Está función busca el elemnto y lo elimina, **llave** debe ser un tipo de dato aceptado por la aplicación. La complejidad de está función es **O**(log_2 **N**).

2.3.2.3.7 Funciones especificas

.count(llave) Está función retorna la cantidad de llaves coincidentes con **llave** lo cual está entre 0 y 1 ya que no existen llaves dobles o repetidos, **llave** debe ser un tipo de dato aceptado por la aplicación. La complejidad de está función es **O**(log_2 **N)**.

.find(llave) Está función retorna el iterador apuntando a la **llave** si existe y si no retorna **.end()**, **llave** debe ser un tipo de dato aceptado por el conjunto. La complejidad de está función es **O**(log_2 **N)**.

.lower_bound(llave) Está función retorna un iterador apuntando a la primera llave no menos que **llave**, **llave** debe ser un tipo de dato aceptado por la aplicación. La complejidad de está función es **O**(log_2 **N)**.

.upper_bound(llave) Está función retorna un iterador apuntando a la primera llave mas grande que **llave**, **llave** debe ser un tipo de dato aceptado por el conjunto. La complejidad de está función es **O**(log_2 **N)**.

A continuación un ejemplo de manejo.

```
1  #include <iostream>
2  #include <map>
3
4  using namespace std;
5
6  int main()
7  {
8      map<char, int> apl;
9      apl.insert(make_pair('a', 13));
10     apl.insert(make_pair('b', 98));
11     cout<<apl['a']<<"\n";
12     cout<<apl['b']<<"\n";
13     //Notese que no existe apl['c] por lo que se creara y pondra como valor 0
14     cout<<apl['c']<<"\n";
15
16     //Acceso a las llaves y valores mediante iteradores
17     for(map<char, int>::iterator it=apl.begin(); it!=apl.end(); it
           ++){
18         cout<<it->first<<" "<<it->second<<"\n";
19     }
20     //Resultado:  a 13 b 98 c 0
21     if(apl.count('a')==1)
22         cout<<"'a' esta en la aplicacion\n";
23     else
24         cout<<"'a' no esta en la aplicacion\n";
25     //Resultado:  'a' está en la aplicacion
26     if(apl.find('d')!=apl.end())
27         cout<<"'d' esta en la aplicacion\n";
28     else
29         cout<<"'d' no esta en la aplicacion\n";
30     //Resultado:  'd' no está en la aplicacion
31     if(apl.lower_bound('c')!=apl.upper_bound('c'))
32         cout<<"'c' esta en la aplicacion\n";
33     else
```

```
34          cout<<"'c' no esta en la aplicacion\n";
35      //Resultado:  'c' está en la aplicacion
36      if(apl.lower_bound('Z')==apl.upper_bound('Z'))
37          cout<<"'Z' no esta en la aplicacion\n";
38      else
39          cout<<"'Z' esta en la aplicacion\n";
40      //Resultado:  70 no está en la aplicacion
41      return 0;
42  }
```

3. Programación modular

3.1 Introduccion

3.1.1 ¿Qué es la Programación Modular?

La programacion modular es un paradigma (modelo) de programacion, que consiste en dividir un problema en subproblemas con el fin de simplificarlo.
Aplicando la Programacion Modular podemos llevar problemas grandes y tediosos, a pequeños subproblemas, y estos a su ves en otros, hasta poder resolverlos facilmente con un lenguaje de programacion, a esta tecnica la llamaremos divide y venceras.
Un modulo es cada una de las partes del programa que resuelve un subproblema en las que se dividio el problema original.[4]

3.2 Modulos

3.2.1 Concepto de Modulo

Un Modulo es una segmento de codigo separado del bloque principal, que puede ser invocado en cualquier momento desde este o desde otro modulo.

3.2.2 Elementos de declaracion del Modulo

Un Moduo o subrutina se declara, generalmente, por:

- Un nombre unico con el que se lo identifica y distingue de otros.
- Un tipo de dato de retorno.
- Una lista de parametros, puede ser cero, uno o varios.
- Conjunto de ordenes que debe ejecutar la subrutina.

3.2.3 Ejemplos

El siguiente ejemplo muestra una subrutina que tiene un parametro de entrada (x) del cual calcula su factorial y lo devuelve como parametro de salida.
El factorial de un numero n se define como el producto de todos los numeros desde 1 hasta n, y se simboliza $n!$.

$$5! = 1x2x3x4x5 = 120$$

Código 3.1: Factorial

```
#include <iostream>
using namespace std;
//"int" Tipo de dato de retorno.
//"fact" Nombre de la subrutina.
//"(int x)" Paranetros que recibe.
int fact(int x){
        //Ordenes que ejecuta
        int f=1
        for(int i=1; i<=x;i++)
                f=f*i;
        return f;
}
int main(){
        int n;
        cin>>n;
        //"fact(n" Invocacion de la subrutina.
        cout<<fact(n)<<endl;
        return 0;
}
```

3.3 Recursividad

3.3.1 Que es recursvidad?

La recursividad es una tecnica de programacion en la que un modulo hace una llamada a si mismo con el fin de resolver el problema. La llamada a si mismo se conoce como llamada recursiva.
Dicho formalmente, un algoritmo se dice recursivo si calcula instancias de un problema en función de otras instancias del mismo problema hasta llegar a un caso base, que suele ser una instancia pequeña del problema, cuya respuesta generalmente está dada en el algoritmo y no es necesario calcularla.[5]
Aun asi la definicion puede ser confusa, por eso ahora presentaremos 2 ejemplos que ilustran la recursividad:

3.3.2 Ejemplos

- Al igual que en el primer ejemplo tendremos una subrutina o funcion que calcule el factorial de un numero, en este caso lo hara de manera recursiva.

Código 3.2: Factorial Recursivo

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

int fact(int n){
        /*
        Caso base: donde el algoritmo se detendra por que tendra
        un valor conocido y no necesitara calcularlo pues 1!=1
        */
```

```
9           if(n==1) return 1;
10          /*
11          Si el valor es diferente de 1, entonces es un valor que
12          necesitamos calcular, por la definicion de factorial
13          podemos apreciar que, por ejemplo: 5!=5x4! y que a su
14          vez 4!=4*3! y asi sucesivamente, esto es recursividad
15          ya que esta funcion se llama asi misma para calcular el
16          siguiente factorial.
17          */
18          return n*fact(n-1);
19  }
20  int main(){
21          int n;
22          cin>>n;
23          cout<<fact(n)<<endl;
24          return 0;
25  }
```

- Ahora calcularemos el n-simo termino de la sucesion Fibonacci por medio de un algoritmo recursivo.
  La sucecion Fibonacci, comienza con los valores 1 y 1 y a partir de estos cada termino es la suma de los 2 anteriores:

$$1,1,2,3,5,8,13,21,34,...$$

Código 3.3: Sucesion Fibonacci

```
1   #include <iostream>
2   using namespace std;
3
4   int Fibo(int n){
5           //Caso base:  El primer y segundo termino de la sucesion
6           //son 1.
7           if(n==1 or n==2) return 1;
8           //Por la definicion de la sucecion un termino es igual a
9           //la suma de los 2 terminos anteriores.
10          return Fibo(n-1)+Fibo(n-2);
11  }
12  int main(){
13          int n;
14          cin>>n;
15          cout<<Fibo(n)<<endl;
16          return 0;
17  }
```

3.4 Operaciones de bits

3.4.1 Introduccion

El sistema de numeracion decimal (Base 10) esta compuesto por 10 digitos: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 y todos los numeros estan compuestos pora la convinacion de estos, por otro lado el Sistema Binario (Base 2) esta compuesto unicamente por dos digitos: 0 y 1, Bit es el acronimo de Binary digit (digito binario), entonces un bit representa uno de estos valores 0 o 1, se puede interpretar como un foco que tiene 2 estados: Encendido(1) y Apagado(0). Existen varios metodos par convertir numeros de una base a otra.

Los datos que usamos en nuestros programas, internamente, estan representados en Binario con una cadena de bits. Por ejemplo un "int" tiene 32 bits. Entonces muchas veces se requiere hacer operaciones de bits, ya sea por que estas se ejecutaran mas rapido que otras mas complejas como la multiplicacion, o por que se quiere modificarlos.

En esta seccion trataremos las operaciones de bits que nos seran de mucha ayuda para la resolucion de problemas de algoritmia,y posteriormente usaremos estas en un algortimo bastente sencillo para calcular todos los subconjuntos de un conjunto.

3.4.2 Operaciones bit a bit

Son operaciones logicas que se ejecutan sobre bits individuales.

3.4.2.1 NOT

El NOT bit a bit, es una operación unaria que realiza la negación lógica en cada bit, invirtiendo los bits del número, de tal manera que los ceros se convierten en 1 y viceversa.

a	$\sim a$
0	1
1	0

Ejemplo:

$\sim$	1010
	0101

3.4.2.2 AND

El AND bit a bit, toma dos números enteros y realiza la operación AND lógica en cada par correspondiente de bits. El resultado en cada posición es 1 si el bit correspondiente de los dos operandos es 1, y 0 de lo contrario. La operacion AND se representa con el signo ”&”.

a	b	$a\&b$
0	0	0
0	1	0
1	0	0
1	1	1

Ejemplo:

&	10101
	01011
	00001

3.4.2.3 OR

Una operación OR de bit a bit, toma dos números enteros y realiza la operación OR inclusivo en cada par correspondiente de bits. El resultado en cada posición es 0 si el bit correspondiente de los dos operandos es 0, y 1 de lo contrario. La operacion OR se representa con el signo "|".

a	b	$a\|b$
0	0	1
0	1	1
1	0	1
1	1	0

Ejemplo:

|	00101
	01001
	01101

3.4.2.4 XOR

El XOR bit a bit, toma dos números enteros y realiza la operación OR exclusivo en cada par correspondiente de bits. El resultado en cada posición es 1 si el par de bits son diferentes y 0 si el par de bits son iguales. La operacion XOR se representa con el signo "^".

a	b	$a\hat{}b$
0	0	0
0	1	1
1	0	1
1	1	0

Ejemplo:

^	00101
	01001
	01100

3.4.3 Operaciones de Desplazamiento

Otra operacion importante es la de desplazar bits, se tiene 2 movientos: El desplazamiento a la derecha y el desplazamiento a la izquierda. Al desplazar los bits los espacios faltantes son rellenados con ceros.

3.4.3.1 Left Shift

El desplazamiento a la izquierda (left shift) es la operacion de mover todos los bits una cantidad determinada de espacion hacia la izquierda. Esta operacion esta representado por los signos " << ". Por cada desplazamiento a la izquierda se agregara un cero a la derecha de todo el numero.
Ejemplo:

$$101011 << 2 = 10101100$$

3.4.3.2 Rigth SHift

El desplazamiento a la derecha (rigth shift) es la operacion de mover todos los bits una cantidad determinada de espacios hacia la derecha. Esta operacion esta representada por los signos " >> ". Al dezplazar el numero hacia la derecha los bits menos significativos (los ultimos) se perderan. Ejemplo:

$$101011 >> 2 = 1010$$

3.4.4 Aplicaciones

Con todas las operaciones anteriormente vistas podemos hacer muchas cosas interesantes a la hora de programar, ahora les mostraremos algunas de las mas importante aplicaciones de estas operaciones:

3.4.4.1 Estado de un bit

Con el uso de las operaciones AND y left shift podemos determinar el estado de un bit determinado. Por ejemplo: Supongamos que tenemos el numero 17 y queremos saber si su quinto bit esta encendido, lo que haremos sera desplazar cuatro posiciones el numero 1, notese que se desplaza n-1 veces los bits, y realizamos la operacion AND si el resultado es diferente de 0 el bit esta encendido, por el contrario esta apagado.

$$17 = 10001$$
$$1 << 4 = 10000$$

&	10001
	10000
	10000

El resultado es diferente de 0 por lo tanto el quinto bit de 17 esta encendido.

3.4.4.2 Apagar un bit

Usando las operaciones AND, NOT y left shift podemos apagar un determinado bit. Por ejemplo: Supogamos que tenemos el numero 15 y queremos apagar su segundo bit, lo que haremos sera desplazar una posicion el numero 1, aplicamos NOT a este numero y luego AND entre ambos, con esto habremos apagado el segundo bit del numero 15.

$$15 = 1111$$
$$1 << 1 = 10$$
$$\sim 10 = ...11101$$

|	1111
	1101
	1101

3.4.4.3 Encender un bit

Usando las operaciones OR y left shift encenderemos un bit determinado de un numero. Por ejemplo: Supongamos que tenemos el numero 21 y queremos encender su cuarto bit, lo que haremos sera desplazar tres posiciones el numero 1, y realizamos la operacion OR entre ambos numeros.

$$21 = 10011$$
$$1 << 3 = 1000$$

&	10101
	01000
	11101

3.4.4.4 Multiplicacion y Division por 2

Una forma rapida de multilicar o dividir un numero por 2 es haciendo uso del desplazamiento de bits, pues si tenemos un numero entero n, y lo desplazaos una pocicion a la derecha el numero se dividira entre 2 con resultado entero (n/2) y si por el contrario desplazamos el numero una posicion a la izquierda el numero se multiplicara por 2 (n*2).

$$29 = 11101$$
$$29 >> 1 = 1110 = 14$$
$$29 << 1 = 111010 = 58$$

3.4.4.5 Dos elevado a la n

Si tenemos el numero 1 y lo desplazamos n veces a la izquierda obtendremos como resultado 2^n.

$$1 << 5 = 100000 = 2^5 = 32$$

3.5 Mascara de bits

Mascara de bits o BitMask es un algoritmo sencillo que se utiliza para calcular todos los subconjuntos de un conjunto. Este algoritmo tiene complejidad de $\Theta(2^n)$ por lo que su uso se limitara a conjuntos que tengan a lo mucho 20 elementos.
El algoritmo funciona de esta manera, si se tiene n elemenos en un conjunto entonces generara todos los numeros desde 0 hasta 2^n junto a sus representaciones binarias, se tomara en cuenta las n cifras menos significativas de cada numero en binario y cada columna representara a un elemento del conjunto, entonces si el bit esta encendido tomaremos ese elemento en el subconjunto.
Supongamos que tenemos un conjunto con 3 elementos, las representaciones binarias de los numeros de 0 a $2^3 - 1$ son:

0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Los subconjuntos seran cada numero en ese rango, y los elementos estan representados por cada columna, entonces si el bit esta encendido quiere decir que el elemento se toma para ese subconjunto, por le contrario si el bit esta apagado este se omite en este subconjunto.

Código 3.4: Mascara de Bits

```
for(int i=0; i<(1<<n); i++){
        for(int j=0; j<n; j++){
                if(i&(1<<j))
                        A[j];
        }
}
```

EL codigo funciona de la siguiente manera:

- El primer for generara todos los numeros de 0 a $2^n - 1$
- El segundo for recorrera todos los bits de los numeros que va generando el primero.
- El if verifica si ese bit esta encendido, de ser asi se toma ese elemento como parte del subconjunto.

3.5.1 Ejemplo

Para ilustrar este algoritmo usaremos el siguiente problema: Se nos dare un entero n, seguido de n enteros que representan la longitud de unas barras de metal, seguidamente se son dara un entero a la longitud de una barra que necesitamos, lo que se quiere saber es si juntando varias de las barras podemos formar una de longitud a. Aca tenemos el codigo en C++:

Código 3.5: UVA 12455: Bars

```
#include <bits/stdc++.h>
using namespace std;
int main(){
        int a,n,b;
        cin>>n>>a;
        vector <int> A(n);
        for(int i=0; i<n; i++){
                cin>>A[i];
        }
        bool sw=0;
        for(int i=0; i<(1<<n); i++){
                int s=0;
                for(int j=0; j<n; j++){
                        if(i&(1<<j)){
                                s+=A[j];
                        }
                }
                if(s==a){
                        sw=1;break;
                }
        }
        if(sw)
                cout<<"YES"<<endl;
        else
                cout<<"NO"<<endl;
}
```

4. Matemáticas

4.1 Introducción

La aparición de los problemas de matemáticas relacionados con las competencias de programación no es sorprendente ya que las Ciencias de la Computación esta profundamente enraizada en las Matemáticas.

4.2 Series o sucesiones

Una serie es un conjunto de cosas (normalmente números) una detrás de otra, en un cierto orden.

4.2.1 Tipos de Series

4.2.1.1 Series aritméticas

En una sucesión aritmética cada término se calcula sumando el anterior por un número fijo.
Ejemplos:

$$1,2,3,4,5,6,7,8,9,10,...$$
Esta sucesión tiene una diferencia de 1 entre cada dos términos.
La regla es $x_n = n$

$$2,4,6,8,10,12,14,16,18,...$$
Esta sucesión tiene una diferencia de 3 entre cada dos términos.
La regla es $x_n = 2n$

$$1,3,5,7,9,11,13,15,17,...$$
Esta sucesión también tiene una diferencia de 2 entre cada dos términos, pero los elementos son impares.
La regla es $x_n = 2n+1$

$$1,4,7,10,13,16,19,22,25,...$$
Esta sucesión tiene una diferencia de 3 entre cada dos términos.
La regla es $x_n = 3n-2$

$$3,8,13,18,23,28,33,38,...$$
Esta sucesión tiene una diferencia de 5 entre cada dos términos.
La regla es $x_n = 5n-2$

4.2.1.2 Series geométricas

En una sucesión geométrica cada término se calcula multiplicando el anterior por un número fijo.
Ejemplos:

$$2,4,8,16,32,64,128,256,...$$
Esta sucesión tiene un factor 2 entre cada dos términos.
La regla es $x_n = 2^n$

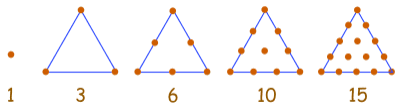
$$3,9,27,81,243,729,2187,...$$
Esta sucesión tiene un factor 3 entre cada dos términos.
La regla es $x_n = 3^n$

$$4,2,1,0.5,0.25,...$$
Esta sucesión tiene un factor 0.5 (un medio) entre cada dos términos.
La regla es $x_n = 4 \times 2^{-n}$

4.2.1.3 Series Especiales

Números triangulares

$$1,3,6,10,15,21,28,36,45,...$$
Esta sucesión se genera a partir de una pauta de puntos en un triángulo.
Añadiendo otra fila de puntos y contando el total encontramos el siguiente número de la sucesión.



Pero es más fácil usar la regla
$x_n = n(n+1)/2$

Números cuadrados

$$1,4,9,16,25,36,49,64,81,...$$
El siguiente número se calcula elevando al cuadrado su posición.
La regla es $x_n = n^2$

Números cúbicos

$$1,8,27,64,125,216,343,512,729,...$$
El siguiente número se calcula elevando al cubo su posición.
La regla es $x_n = n^3$

Números de Fibonacci

$$0,1,1,2,3,5,8,13,21,34,...$$
El siguiente número se calcula sumando los dos que están antes de él.
El 2 se calcula sumando los dos delante de él $(1+1)$
El 21 se calcula sumando los dos delante de él $(8+13)$
La regla es $x_n = x_{n-1} + x_{n-2}$
Esta regla es interesante porque depende de los valores de los términos anteriores.
Por ejemplo el 6º término se calcularía así:
$x_6 = x_{6-1} + x_{6-2} = x_5 + x_4 = 5+3 = 8$

4.3 Números Primos

Un entero $N > 1$ que tiene como divisores solo a N y 1 es un ***número primo***. El primer y el único numero primo par es el 2. Los siguientes números primos son: $3,5,7,11,13,17,19,23,29,...$, etc. Hay 25 números primos en el rango $[0..100]$, 168 primos en $[0..1000]$, 1000 primos en $[0..7919]$, 1229 primos en $[0..10000]$, etc. Algunos números primos grandes son 104729, 1299709, 15485863, 179424673, 2147483647, etc.

Los números primos son un importante tópico en las matemáticas y la razón de varios problemas de programación.

4.3.1 Prueba de Primalidad

El primer algoritmo presentado en esta sección es para probar si un numero natural N es primo, es decir, **bool esPrimo(N)**. La version mas simple es probar por definición, es decir, probar que N solo tenga 2 divisores 1 y N, para hacer esto tendremos que dividir N veces. Esta no es la mejor forma para probar si un numero N es primo y hay varias posibles mejoras.

La primera mejora es verificar si N es divisible por $divisores \in [2..\sqrt{N}]$, es decir, pararemos de dividir cuando el $divisor$ sea mas grande que $\sqrt{N}$.

La segunda mejora es verificar si N es divisible por $divisores \in [3,5,7,..\sqrt{N}]$, es decir, solo verificaremos números impares hasta $\sqrt{N}$. Esto es porque solo hay un numero primo par, el numero 2, que puede ser verificado por separado.

El código es el siguiente:

Código 4.1: Prueba de Primalidad Optimizada

```
#include <iostream>

using namespace std;

bool esPrimo(int N){
   if(N==1)return false;
   if(N==2)return true;
   for(int i=3;i*i<=N;i=i+2){ // i*i<=N es equivalente a i<=sqrt(N)
      if(N%i==0)return false;
   }
   return true;
}
int main(){
   int N;
   cin>>N;
   if(esPrimo(N))cout<<"El numero es primo";
   else cout<<"El numero no es primo";
   return 0;
}
```

4.4 Criba de Eratóstenes

Si queremos generar una lista de números primos en rango de $[0..N]$, existe un mejor algoritmo que probar si cada numero en el rango es primo o no. Este algoritmo es llamado "Criba de Eratóstenes" inventado por Eratóstenes de Alexandria. Esto funciona de la siguiente manera.

Primero, hacemos que todos los números en el rango sean 'probablemente primos', pero hacemos que los números 0 y 1 no sean primos. Luego, tomamos al 2 como primo marcamos todos los múltiplos de 2 empezando por $2+2=4,6,8,10,...$ hasta que el múltiplo sea mas grande que N. Luego tomamos el siguiente numero no marcado como primo que en este caso seria el 3 y marcamos todos los múltiplos de 3 empezando por $3+3=6,9,12,...$ Luego tomamos al el siguiente numero no marcado que en este caso seria el 5 y marcamos todos los múltiplos de 5 empezando por $5+5=10,15,20,...$ y así sucesivamente. Después de esto cualquier numero no marcado dentro del rango $[0..N]$ sera primo.

El código seria el siguiente:

Código 4.2: Criba de Eratóstenes

```
#include <iostream>

using namespace std;

const int MAX=1000000;
bool primos[MAX+1];

void iniciar_criba(){
   for(int i=0;i<=MAX;i++) // para asignar inicialmente todos los numeros
      primos[i]=true;    // como primos
   primos[0]=primos[1]=false;  // excepto el 0 y el 1
   for(int i=2;i<=MAX;i++){ // empezamos desde el numero 2
      if(primos[i]){        // verificamos que no haya sido marcado
         for(int j=i+i;j<=MAX;j=j+i){
            primos[j]=false;    // marcamos los multiplos
         }
      }
   }
}
int main(){
   iniciar_criba();

   cout<<primos[2]<<endl; // es primo, true
   cout<<primos[19]<<endl; // es primo, true
   cout<<primos[10]<<endl; // no es primo, false
   cout<<primos[51]<<endl; // no es primo, false
   return 0;
}
```

4.5 Factorización de enteros

Por el teorema de fundamental de la aritmética o también conocido como teorema de factorización única tenemos que todo numero entero positivo $X > 1$ puede ser representado como el producto de sus factores primos.
Por cada factor primo P de un entero X se dice que la multiplicidad de P es un entero positivo a tal que este es un máximo para que P^a sea divisor de X.
Ej:
Los factores primos de 60 son: $2,3,5$ ya que la multiplicación de $2^2 \times 3 \times 5$ es 60.
En este ejemplo vemos que el factor primo 2 tiene multiplicidad 2, 3 tiene multiplicidad 1 y 5 tiene multiplicidad 1.
Dado un numero en un rango $1 < X < N+1$ usualmente con N no mayor a 10^7 dado que la solución requiere generar, como primer paso, la Criba de Eratóstenes de tamaño N con una pequeña modificación. Damos entonces como solución un algoritmo que se ejecuta en tiempo $O(log_2 X)$.
Como un primer paso se necesita encontrar un factor primo máximo P_k que divida a X modificando la Criba de la siguiente manera:

Código 4.3: Modificación de la Criba de Eratóstenes

```
1  void iniciar_criba(){
2     for(int i=0;i<=N;i++)C[i]=i; // Si i es primo entonces
3     // el unico factor primo que tiene es si mismo.
4     C[0]=-1;C[1]=-1;
5     for(int i=2;i*i<=N;i++)
6        if(C[i]==i)
7           for(int j=i+i;j<=N;j+=i)
8              C[j]=i; // Como i es primo
9           // es un probable primo maximo que divide a j
10 }
```

Dado que i es un primo que divide a j, se tiene que si i es el máximo factor primo de j este será el último en pasar por j y si no entonces existirá otro posible i máximo que pase por j.
También tenemos que al final del procedimiento $C[X]$ es el primo máximo que divide a X el cual lo podemos obtener desde ahora en un tiempo de ejecución $O(1)$.
Ahora tenemos un $X > 1$ y deseamos encontrar el conjunto de factores primos que siguen la definición:

$$X = P_1^{a_1} \times P_2^{a_2} \times P_3^{a_3} \times ... \times P_k^{a_k}$$

Definimos la función $fp(x)$ para todo $x > 1$ la cual nos devuelve el conjunto de factores primos y multiplicidad de x ordenado ascendentemente.

$$fp(X) = \{P_1^{a_1}, P_2^{a_2}, P_3^{a_3}, \ldots, P_k^{a_k}\}$$

Pero tenemos que $P_k = C[x]$ definido como el primo más grande que divide a X, por lo tanto:

$$fp(X) = \{P_1^{a_1}, P_2^{a_2}, P_3^{a_3}, ..., P_{k-1}^{a_{k-1}}\} \cup \{C[x]^{a_k}\}$$

Podemos tomar un numero $Y = P_1^{a_1}/timesP_2^{a_2}/timesP_3^{a_2}/times.../timesP_{k-1}^{a_{k-1}}$ aunque aun no conocemos la multiplicidad de P_k sabemos que si el siguiente primo máximo que divide a Y es igual a P_k o sea que $P_k = P_{k-1}$ podemos decir que la multiplicidad de P_k es 2 y continuar haciendo el mismo proceso hasta que sean diferentes.
Entonces cuando P_k no sea igual a P_{k-1}:

$$X = Y \times P_k^{a_k}$$
$$Y = \frac{X}{P_k^{a_k}} = \frac{X}{C[x]^{a_k}}$$

Ahora dado que Y es el producto de todos los factores primos de X excepto por el mayor entonces podemos hacer lo siguiente:

$$fp(Y) = \{P_1^{a_1}, P_2^{a_2}, P_3^{a_3}, ..., P_{k-1}^{a_{k-1}}\}$$
$$fp(X) = fp(Y) \cup \{C[x]^{a_k}\}$$

reemplazando:

$$fp(X) = fp(\frac{X}{C[x]^{a_k}}) \cup \{C[x]^{a_k}\}$$

Desde luego $fp(Y)$ será un conjunto vacio si X es un numero primo, por lo que tendremos la igualdad siguiente:

$$fp(X) = \begin{cases} \{\} & \text{si } X \leq 1 \\ fp(Y) \cup \{C[x]^a\} & \text{si } X > 1 \end{cases}$$

Donde $Y = \frac{X}{C[x]^{a_k}}$ y a es la multiplicidad de $C[X]$.
El código lee un X por entrada e imprimirá los factores primos y su multiplicidad de X de forma ordenada.

Código 4.4: Factorización de enteros

```
#include <bits/stdc++.h>
using namespace std;
int C[10000001];
int N=10000000;
void iniciar_criba(){
   for(int i=0;i<=N;i++)C[i]=i;
   C[0]=-1;C[1]=-1;
   for(int i=2;i*i<=N;i++)
      if(C[i]==i)
         for(int j=i+i;j<=N;j+=i)
            C[j]=i;
}
void fp(int X){
   if(X<=1)return;
   int a=1;
   int Y=X/C[X];
   while(C[X]==C[Y]){
      a++;
```

```
19          Y=Y/C[Y];
20      }
21      fp(Y);
22      cout<<C[X]<<" ^ "<<a<<endl;
23  }
24  int main(){
25      int x;
26      iniciar_criba();
27      while(cin>>x){
28          fp(x);
29      }
30      return 0;
31  }
```

Aunque no podemos encontrar una función específica que describa el tiempo de ejecución, debido a la característica impredecible de los números primos, podemos estar seguros que $O(log_2 X)$ describe el peor tiempo de ejecución para un todos los X en el rango de 1 a N. Es decir que el máximo tiempo de ejecución dado un X será en un tiempo proporcional a $log_2 X$ o mucho menor en un buen caso.
Para verlo de una manera más simple volvemos a nuestra definición:

$$X = P_1^{a_1} \times P_2^{a_2} \times P_3^{a_3} \times ... \times P_k^{a_k}$$

A simple vista vemos que el número de factores primos está estrechamente relacionado al valor de los mismos. Mientras más grandes son los factores menor será la multiplicidad da cada factor. Dado este hecho el peor de los casos es uno en el que la suma de la multiplicidad de los factores es más grande por tanto los factores tienen que ser mínimos.
Si X tiene la forma de 2^M, o sea está conformado de M factores primos iguales a 2 entonces:

$$\begin{aligned} 2^k &= X \\ log_2 2^k &= log_2 X \\ M &= log_2 X \end{aligned}$$

Por lo que tenemos que en el peor de los casos tenemos $O(log_2 X)$ en un peor caso de prueba ya que nuestro algoritmo tiene un tiempo de ejecución en función al número de factores primos y su multiplicidad.

4.6 Máximo Común Divisor (MCD) y Mínimo Común Múltiplo (mcm)

El Máximo Común Divisor (MCD) de dos números (a,b) denotado por $MCD(a,b)$, es definido como el entero positivo mas grande d tal que $d \mid a$ y $d \mid b$ donde $x \mid y$ implica que x divide a y.
Ej: $MCD(4,8) = 4$, $MCD(10,5) = 5$, $MCD(20,12) = 4$.
Un uso practico del MCD es el de simplificar fracciones, es decir, $\frac{4}{8} = \frac{4/MCD(4,8)}{8/MCD(4,8)} = \frac{4/4}{8/4} = \frac{1}{2}$.
Para calcular el MCD, ingenuamente podemos empezar desde el mas pequeño de los dos números e ir decrementando hasta que encontremos un número que divida a los dos números.
El código seria el siguiente:

Código 4.5: MCD lento

```cpp
#include <iostream>

using namespace std;

int MCD_lento(int a,int b){
   for(int i=min(a,b);i>=1;i--){
      if(a%i==0 && b%i==0){
         return i;  // i es igual al MCD
      }
   }
}

int main(){

   cout<<MCD_lento(4,8)<<endl; //4
   cout<<MCD_lento(10,5)<<endl; // 5
   cout<<MCD_lento(5,10)<<endl; // 5
   cout<<MCD_lento(9,4)<<endl; //1
   cout<<endl;
   return 0;
}
```

Este método es lento para un problema en el cual la cantidad de consultas del MCD es muy grande, existe un método mas rápido llamado *Algoritmo de Euclides*.
El *Algoritmo de Euclides* itera sobre dos números hasta que encontremos un resto igual a 0. Por ejemplo, supongamos que queremos encontrar el MCD de 2336 y 1314. Empezamos expresando el numero mas grande (2336) en términos del numero pequeño (1314) mas un resto:

$$2336 = 1314 \times 1 + 1022$$

Ahora hacemos lo mismo con 1314 y 1022:

$$1314 = 1022 \times 1 + 292$$

Continuamos este proceso hasta que alcancemos un resto igual a 0:

$$1022 = 292 \times 3 + 146$$
$$292 = 146 \times 2 + 0$$

El ultimo resto distinto de cero es el MCD. Así el MCD de 2336 y 1314 es 146. Este código algoritmo puede ser fácilmente codificado como una función recursiva (mirar abajo).
El MCD esta muy relacionado con el Mínimo Común Múltiplo (mcm). El mcm de dos números enteros (a,b) denotado por $mcm(a,b)$, es definido como el numero entero mas pequeño l tal que $a \mid l$ y $b \mid l$.
Ej: $mcm(4,8) = 8$, $mcm(10,5) = 10$, $mcm(20,12)60$.
Usando el *Algoritmo de Euclides* también podemos encontrar el mcm (mirar abajo).

Código 4.6: MCD y mcm

```cpp
#include <iostream>

using namespace std;

int MCD(int a,int b){
    if(b==0)return a;
    return MCD(b,a%b); // recursion
}
int mcm(int a,int b){
    return a*(b/MCD(a,b)); // recursion
}

int main(){

    cout<<MCD(4,8)<<endl; //4
    cout<<MCD(10,5)<<endl; // 5
    cout<<MCD(5,10)<<endl; // 5
    cout<<MCD(9,4)<<endl; //1
    cout<<endl;
    cout<<mcm(4,8)<<endl; //8
    cout<<mcm(10,5)<<endl; //10
    cout<<mcm(20,12)<<endl; // 60
    cout<<mcm(9,4)<<endl; //36

    return 0;
}
```

4.7 Exponenciación binaria modulo m

El siguiente algoritmo tiene como objetivo responder de forma optima la siguiente operación:
Dados los enteros a, $b \geq 0$, pero por lo menos uno de ellos distintos de 0 y $m > 0$ se desea encontrar el valor de c tal que $c \equiv a^b (mod\ m)$ o lo que será para nosotros $c = a^b \% m$.
Nos concierne entonces entender la nueva operación modulo $(\%)$ para así usarla correctamente.
La operación binaria X modulo Y $(X\%Y)$ con X e Y números enteros es igual al resto de de la división entera de X entre Y (X/Y).
Ej:
$13\%4 = 1$ puesto que la parte entera de $13/4$ es igual a 3 y de resto tenemos 1. La operación modulo tiene las siguientes propiedades. Siendo a, b, m números enteros positivos y $m > 0$:

$$(a+b)\%m = (a\%m + b\%m)\%m$$
$$(a-b)\%m = (a\%m - b\%m)\%m$$
$$(a \times b)\%m = (a\%m \times b\%m)\%m$$

Si $b = a\%m$ entonces $0 \leq b < m$ la cual es una propiedad la cual no deja crecer cualquier operación que este en compuesta por multiplicaciones y sumas de un valor arbitrario m.
Volviendo a nuestro tema de interés, veremos algunas propiedades que tiene la función exponencial.
Dados los enteros $a, b, c, d \geq 0, a$ o b distintos de 0.
$POW(a,b) = a^b = a \times a \times a \times ... \times a$ con a repetida b veces.

$$\begin{gathered} a^0 = 1 \\ a^b = a^{(c+d)} = a^c \times a^d \quad , si\ b = c+d \\ (a^b)^c = a^{bc} \end{gathered}$$

Ahora resolver el problema a "fuerza bruta" es fácil ya que lo único que debemos hacer es realizar una multiplicación b veces lo que nos daría una solución en $O(b)$. Sin embargo es posible optimizar esto y obtener el resultado en $O(log_2 b)$.
Definimos una función que calculara $POW(a,b) = a^b$
Sabemos que $POW(a,b) = a^{2\frac{b}{2}} = (a^2)^{\frac{b}{2}} = POW(a \times a, \frac{b}{2})$
Tendríamos un problema si b es un número impar así que haremos lo siguiente:
$c = b-1$ como b es impar entonces c es par.
$POW(a,b) = a^{b-1+1} = a^c \times a = POW(a \times a, \frac{c}{2}) \times a = POW(a \times a, \lceil \frac{b}{2} \rceil) \times a$
Donde $\lceil x \rceil$ es la parte entera de x.
También podemos deducir fácilmente que $\lceil \frac{b}{2} \rceil = \frac{b-1}{2}$ si b es un número entero impar.
Entonces ya tenemos descrita nuestra función recursiva POW.

$$POW(a,b) = \begin{cases} 1 & \text{si } b = 0 \\ POW(a \times a, \lceil \frac{b}{2} \rceil) \times a & \text{si } b\ es\ impar \\ POW(a \times a, \frac{b}{2}) & \text{si } b\ es\ par \end{cases}$$

Sin embargo, como sabemos, la función exponencial crece mucho y no podemos guardar ese número en variables de tipo "int" o "long long" por lo que aplicaremos modulo a la función POW.

$$POW(a,b)\%m = \begin{cases} 1 & \text{si } b = 0 \\ POW(((a\%m \times a\%m)\%m, \lceil \frac{b}{2} \rceil) \times a\%m)\%m & \text{si } b\ es\ impar \\ POW((a\%m \times a\%m)\%m, \frac{b}{2}) & \text{si } b\ es\ par \end{cases}$$

Por lo que el código sería el siguiente:

Código 4.7: Exponenciación binaria modulo m

```
#include <iostream>

using namespace std;

int POW(int a, int b, int m){
   if(b==0)
      return 1;
   return (POW((a * a)%m, b/2, m) * (b&1?a:1) )%m;
}
int main(){
   int a,b,m;
   while(cin>>a>>b>>m){
      cout<<POW(a,b,m)<<endl;
   }
   return 0;
}
```

Como podemos ver el tiempo de ejecución está en función de b (exponente) y no de a (base) por tanto sabemos que b se dividirá en 2 hasta que esta llegue a 1 y de esta manera realizara k procesos.

$$1 = \frac{b}{2^k}$$
$$2^k = b$$
$$k = log_2 b$$

Por lo tanto este algoritmo tiene un tiempo $O(log_2 b)$ el cual es totalmente eficiente.

5. Algoritmos Basicos

5.1 Algoritmos de Ordenanmiento

5.1.1 Ordenamiento Rapido

El ordenamiento rapido (Quicksort en ingles) es un algortimo basado en la tecnica divide y venceras, que permite ordenar n elementos en un tiempo proporcional a $\Theta(n\log(n))$

5.1.1.1 Descripcion del Algoritomo

El algoritmo trabaja de la siguiente forma:

- Elegir un elemento del vector a ordenar, al que llamaremos pivote.
- Mover todos los elementos menores que el pivote a un lado y los mayores al otro lado.
- El vector queda separado en 2 subvectores una con los elementos a la izquierda del pivote y otra con los de la derecha,
- Repetir este proceso recursivamente en las sublistas hasta que estas tengan un solo elemento.

Aca tenemos el codigo en C++:

Código 5.1: QuickSort

```
#include <iostream>
using namespace std;
int pivot(int a[], int first, int last) {
    int  p = first;
    int pivotElement = a[first];

    for(int i = first+1 ; i <= last ; i++) {
        if(a[i] <= pivotElement){
            p++;
            swap(a[i], a[p]);
        }
    }

    swap(a[p], a[first]);

    return p;
}
void quickSort( int a[], int first, int last ) {
    int pivotElement;
```

```
20
21      if(first < last){
22          pivotElement = pivot(a, first, last);
23          quickSort(a, first, pivotElement-1);
24          quickSort(a, pivotElement+1, last);
25      }
26  }
27  void swap(int& a, int& b){
28      int temp = a;
29      a = b;
30      b = temp;
31  }
```

5.1.2 Ordenamiento por mezcla

El algoritmo de ordenamiento por mezcla (Merge Sort) es un algoritmo de ordenacion externo estable basado en la tecnica divide y venceras. Su complejidad es $\Theta(n \log n)$

5.1.2.1 Descripcion del Algoritomo

Conceptualmente el algoritmo funciona de la siguiente manera:

- Si la longitud del vector es 1 o 0, entonces ya esta ordenado, en otro caso:
- Dividir el vector desordenado en dos subvectores de aproximadamente la mitad de tamaño.
- Ordenar cada subvector recursivamente aplicando el ordenamiento por mezcla.
- Mezclar los dos subvectores en un solo subvector ordenado.

Aca tenenmos el codigo en C++.

Código 5.2: Merge Sort

```
1   #include <bits/stdc++.h>
2   using namespace std;
3
4   void mergeSort(int list[], int lowerBound, int upperBound){
5       int mid;
6
7       if (upperBound > lowerBound){
8           mid = ( lowerBound + upperBound) / 2;
9           mergeSort(list, lowerBound, mid);
10          mergeSort(list, mid + 1, upperBound);
11          merge(list, lowerBound, upperBound, mid);
12      }
13  }
14  void merge(int list[], int lowerBound, int upperBound, int mid){
15      int* leftArray = NULL;
16      int* rightArray = NULL;
17      int i, j, k;
18      int n1 = mid - lowerBound + 1;
19      int n2 = upperBound - mid;
20      leftArray = new int[n1];
21      rightArray = new int[n2];
22      for (i = 0; i < n1; i++)
```

```
23          leftArray[i] = list[lowerBound + i];
24      for (j = 0; j < n2; j++)
25          rightArray[j] = list[mid + 1 + j];
26
27      i = 0;
28      j = 0;
29      k = lowerBound;
30
31      while (i < n1 && j < n2){
32          if (leftArray[i] <= rightArray[j]){
33              list[k] = leftArray[i];
34              i++;
35          }
36          else{
37              list[k] = rightArray[j];
38              j++;
39          }
40
41          k++;
42      }
43      while (i < n1){
44          list[k] = leftArray[i];
45          i++;
46          k++;
47      }
48      while (j < n2){
49          list[k] = rightArray[j];
50          j++;
51          k++;
52      }
53      delete [] leftArray;
54      delete [] rightArray;
55  }
```

5.1.3 Ordenamiento por Monticulos

5.1.3.1 Descripcion del Algoritmo

Es un algoritmo de ordenacion no recursivo, con complejidad $O(n \log n)$
Este algoritmo consiste en almacenar todos los elementos del vector a ordenar en un montículo (heap), y luego extraer el nodo que queda como nodo raíz del montículo (cima) en sucesivas iteraciones obteniendo el conjunto ordenado. Basa su funcionamiento en una propiedad de los montículos, por la cual, la cima contiene siempre el menor elemento (o el mayor, según se haya definido el montículo) de todos los almacenados en él. El algoritmo, después de cada extracción, recoloca en el nodo raíz o cima, la última hoja por la derecha del último nivel. Lo cual destruye la propiedad heap del árbol. Pero, a continuación realiza un proceso de "descenso" del número insertado de forma que se elige a cada movimiento el mayor de sus dos hijos, con el que se intercambia. Este intercambio, realizado sucesivamente "hunde" el nodo en el árbol restaurando la propiedad montículo del arbol y dejándo paso a la siguiente extracción del nodo raíz.
El algoritmo, en su implementación habitual, tiene dos fases. Primero una fase de construcción

de un montículo a partir del conjunto de elmentos de entrada, y después, una fase de extracción sucesiva de la cima del montículo. La implementación del almacén de datos en el heap, pese a ser conceptualmente un árbol, puede realizarse en un vector de forma fácil. Cada nodo tiene dos hijos y por tanto, un nodo situado en la posición i del vector, tendrá a sus hijos en las posiciones 2 x i, y 2 x i +1 suponiendo que el primer elemento del vector tiene un índice = 1. Es decir, la cima ocupa la posición inicial del vector y sus dos hijos la posición segunda y tercera, y así, sucesivamente. Por tanto, en la fase de ordenación, el intercambio ocurre entre el primer elemento del vector (la raíz o cima del árbol, que es el mayor elemento del mismo) y el último elemento del vector que es la hoja más a la derecha en el último nivel. El árbol pierde una hoja y por tanto reduce su tamaño en un elemento. El vector definitivo y ordenado, empieza a construirse por el final y termina por el principio.

5.1.4 Ordenamiento por Cuentas

5.1.4.1 Descripcion del Algortimo

El ordenamiento por cuentas es un algortimo en el que se cuenta el número de elementos de cada clase para luego ordenarlos. Sólo puede ser utilizado por tanto para ordenar elementos que sean contables (como los números enteros en un determinado intervalo, pero no los números reales, por ejemplo).
El primer paso consiste en averiguar cuál es el intervalo dentro del que están los datos a ordenar (valores mínimo y máximo). Después se crea un vector de números enteros con tantos elementos como valores haya en el intervalo [mínimo, máximo], y a cada elemento se le da el valor 0 (0 apariciones). Tras esto se recorren todos los elementos a ordenar y se cuenta el número de apariciones de cada elemento (usando el vector que hemos creado). Por último, basta con recorrer este vector para tener todos los elementos ordenados.

5.2 Busqueda Binaria

5.2.1 Descripcion del Algoritmo

El algortimo de busqueda binaria esta basado en la tecnica divide y venceras. Se usa sobre un conjunto de elementos ordenados y tiene complejidad de $\Theta(n \log n)$
En una forma general el algoritmo de busqueda binaria se puede usar en cualquier funcion "binaria", es decir que sea falsa para un primer intervalo y verdadera para el resto, o viceversa, ya que lo que haremos sera buscar el punto de inflexion donde la funcion cambie su estado.

5.2.2 Ejemplo

El ejemplo mas comun de Busqueda Binaria es el de buscar un elemento x dentro de un arreglo de nuemeros ordenados. Usando la definicion anterior veremos como podemos aplicar busqueda binaria sobre este tipo de funciones.
Supongamos que tenemos el siguiente arreglo de numeros ordenados:

0	1	2	3	4	5	6	7	8	9	10	11
0	4	9	11	13	29	51	52	64	71	77	99

Y se quiere saber si en el se enuentra el numero 13, bien pues lo que haremos sera escribir la funcion binaria:

$$F(x)\begin{cases} True: & x \leq 13 \\ False: & x > 13 \end{cases}$$

Si tomamos la recta real como los posibles valores de x en nuestra funcion tenemos que todos los valores desde $-\infty$ hasta 13 son verdad y todos los valores desde 13 hasta $+\infty$ son falsos. Lo que queremos encontrar es el punto donde la funcion cambia de estado.
Para esto necesitamos don valores: a un valor siempre verdadero para la funcion y b un valor siempre falso para la funcion. En nuestro caso podemos tomar la "pocision" -1 del arreglo como a ya que sabes que sera menor a todos los elementos del arreglo y tomar la "pocision" 12 como b por que sera mayor a cualquier elemento del arreglo.

$$a = -1 \quad b = 12$$

A partir de este momento lo que haremos es tomar el elemento medio entre estos y ver que sucede con la funcion, si dicho elemento nos devuelve verdad este sera nuestro nuevo a y si por el contrario es falso sera nuestro nuevo b, asi sucesivamente hasta que la distancia entre a y b sea 1, pues en ese momento habremos encontrado el punto de imflexion de la funcion. Es decir a sera un valor verdarero y b uno falso, por lo tanto el elemento que buscabamos se encuentra en a.
EL codigo en C++ es el siguiente:

Código 5.3: Binary Search

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    int V[12]={0,4,9,11,13,29,51,52,64,71,77,99};
    int a=-1,b=12;
    while(abs(a-b)!=1){
        int c=(a+b)/2;
        if(V[c]<=13)
            a=c;
        else
            b=c;
    }
    cout<<V[a]<<endl;
    return 0;
}
```

6. Estructura de Datos ++

6.1 Introducción

En este capitulo estudiaremos estructura de datos más avanzadas las cuales no las prevee **C++**, por lo que estaremos oblidados a implementarlas. En este capitulo abarcaremos tres estrcuturas:

- Arboles (no lo implemetaremos)
- Unión Find
- Segment Tree

6.2 Arboles

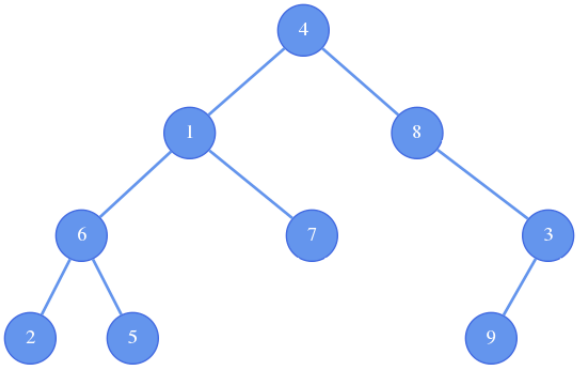
A los arboles lo veremos de manera más teorica, por lo que no nos enfocaremos como implementarlo como en las demas estructuras, ya que tenemos en **C++** una implementación de arboles Rojos y Negros implementada (**set**).

Primero empezemos definiendo algunos terminos:

Nodo En estructuras de datos definiremos un nodo como la unidad fundamental de un árbol, también podemos llamarlo vértice.

Arco Es un enlace, conección o unión entres dos nodos.

Un arbol es un conjunto de nodos y arcos, un nodo puede tener entre uno o dos arcos, una ilustración puede ser la siguiente.



Si volteamos la imagen se logra ver un arbol como en la vida real. Ahora vamos a definir algunos elementos fundamentales de un arbol:

Nodo Hijo	Es aquel nodo que desciende de otro, en la imagen el nodo 6 es el hijo del nodo 1.
Nodo Padre	Es aquel nodo del que un nodo hijo desciende, en la imagen el nodo 4 es el padre de los nodos 1 y 8.
Nodos Hermanos	Son aquellos nodos que comparten el mismo padre, en la imagen los nodos 2 y 5 comparten el nodo padre 6.
Nodo Raíz	Es el nodo superior del árbol, cada arbol solo tiene una raiz, en la imagen es el nodo 4.
Nodo Hoja	Es aquel nodo que no tienen hijos, en la imagen son los nodos 2, 5, 7 y 9.
Nodo Rama	Aunque esta definición apenas la usaremos, son los nodos que no son hoja ni raiz, en la imagen son los nodos 1, 8, 6 y 3.

Tambien podemos hablar de otros conceptos que definen las caracteristicas de un árbol, que más adelanten nos seran de mucha ayuda:

Orden	Es el número potencial de hijos que puede tener cada elemento de árbol. De este modo, diremos que un árbol en el que cada nodo puede apuntar a otros dos es de orden dos, si puede apuntar a tres será de orden tres, etc.
Grado	Es ell número de hijos que tiene el elemento con más hijos dentro del árbol. En el árbol del ejemplo, el grado es dos, ya que los nodos 4, 1 y 6 tienen dos hijos, y no existen nodos con más de dos hijos.
Nivel	Se define para cada nodo del árbol como la distancia a la raíz, medida en nodos. El nivel de la raíz es cero y el de sus hijos uno. Así sucesivamente. En la imagen, el nodo 8 tiene nivel 1, el nodo 7 tiene nivel 2, y el nodo 9 nivel 3.
Altura	La altura de un árbol se define como el nivel del nodo de mayor nivel. Como cada nodo de un árbol puede considerarse a su vez como la raíz de un árbol, también podemos hablar de altura de ramas. El árbol de la imagen tiene altura 3, la rama 8 tiene altura 2, la rama 6 tiene altura 1 y la rama 7 tiene altura cero.

Ahora podemos hablar de dos clases de arboles rapidamente.

- Arboles binarios
- Arboles multirrama

6.2.1 Arboles Binarios

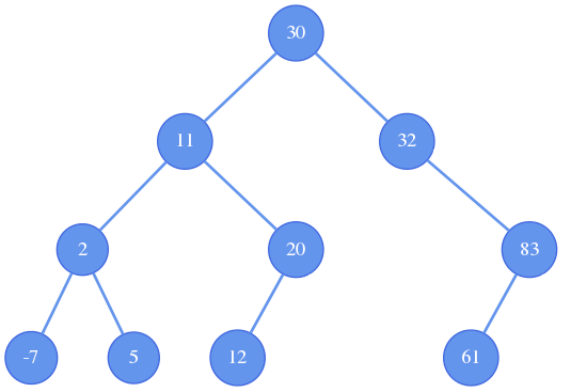
Estos arboles se distinguen por no tener más de dos hijos (de ahí el nombre "binario"), o tambien podemos definirlo como aquel arbol tal el grado de cada nodo no es mayor a tres.

6.2.1.1 Arboles Binarios de Busqueda

En programación es necesario conocer tambien los arboles binarios de busqueda, tambien llamados **BST** del acronimo del ingles Binary Search Tree, es un tipo particular de árbol binario. Un árbol binario no vacio, con raíz **R**, es un árbol binario de busqueda si:

- En caso de tener subárbol izquierdo, la raíz **R** debe ser mayor que el valor máximo almacenado en el subárbol izquierdo, y que el subárbol izquierdo sea un árbol binario de búsqueda.
- En caso de tener subárbol derecho, la raíz **R** debe ser menor que el valor mínimo almacenado en el subárbol derecho, y que el subárbol derecho sea un árbol binario de búsqueda.

Aqui una ilustración de como luciria un arbol binario de busqueda.



6.2.1.2 Arboles Binarios de Busqueda Auto-balanceable

Un árbol de busqueda auto-balanceable o equilibrado es un arbol de búsqueda que intenta mantener su altura, o el número de niveles de nodos bajo la raíz tan pequeños como sea posible en todo momento, automáticamente. Esto es importante, ya que muchas operaciones en un árbol de búsqueda binaria tardan un tiempo proporcional a la altura del árbol, y los árboles binarios de búsqueda ordinarios pueden tomar alturas muy grandes en situaciones normales, como cuando las claves son insertadas en orden. Mantener baja la altura se consigue habitualmente realizando transformaciones en el árbol, como la rotación de árboles, en momentos clave.

Existen estructuras de datos populares que implementan este tipo de árbol:

Árboles AVL El árbol AVL toma su nombre de las iniciales de los apellidos de sus inventores, Georgii Adelson-Velskii y Yevgeniy Landis, fue el primer árbol de búsqueda binario auto-balanceable que se ideó.

Árboles Rojo-Negro Un árbol rojo-negro es un árbol binario de búsqueda en el que cada nodo tiene un atributo de color cuyo valor es rojo o negro. En adelante, se dice que un nodo es rojo o negro haciendo referencia a dicho atributo, cada nodo hoja se convierte en padre de dos nodos "NULL". Además de los requisitos impuestos a los árboles binarios de búsqueda convencionales, se deben satisfacer las siguientes reglas para tener un árbol rojo-negro válido:

- Todo nodo es o bien rojo o bien negro.
- La raíz es negra.
- Todas las hojas (NULL) son negras.
- Todo nodo rojo debe tener dos nodos hijos negros.
- Cada camino desde un nodo dado a sus hojas descendientes contiene el mismo número de nodos negros.

Árbol AA Los árboles AA reciben el nombre de su inventor, Arne Andersson. Son una variación del árbol rojo-negro, que a su vez es una mejora del árbol binario de búsqueda. A diferencia de los árboles rojo-negro, los nodos rojos en un árbol AA sólo pueden añadirse como un hijo derecho. En otras palabras, ningún nodo rojo puede ser un hijo izquierdo.

6.2.2 Arboles Multirrama

Un árbol multirrama o multicamino posee un grado g mayor a dos, donde cada nodo de información del árbol tiene un máximo de g hijos. Sea un árbol de m-caminos A, es un árbol m-caminos si y solo si:

- Cada nodo de A muestra la siguiente estructura: [$nClaves$, $Enlace_0$, $Clave_1$, ..., $Clave_{nClaves}$, $Enlace_{nClaves}$] $nClaves$ es el número de valores de clave de un nodo, pudiendo ser $0 <= nClaves <= g-1$. $Clave_i$, son los valores de clave, pudiendo ser $1 <= i <= nClaves$ $Enlace_i$, son los enlaces a los subárboles de A, pudiendo ser $0 <= i <= nClaves$
- $Clave_i < Clave_{i+1}$
- Cada valor de clave en el subárbol Enlacei es menor que el valor de $Clave_{i+1}$.
- Los subárboles $Enlace_i$, donde $0 <= i <= nClaves$, son también árboles m-caminos.

Existen muchas aplicaciones en las que el volumen de la información es tal, que los datos no caben en la memoria principal y es necesario almacenarlos, organizados en archivos, en dispositivos de almacenaminento secundario. Esta organización de archivos debe ser suficientemente adecuada como para recuperar los datos del mismo en forma eficiente.

6.3 Union-Find en Conjuntos Disjuntos

Llamado UFDS por el acronimo en ingles Union Find Disjoint Set es una estructura de datos que nos ayudara a manejar conjuntos disjuntos. Dos conjuntos A y B son disjuntos si $A \cap B = \emptyset$
Esta estructura tiene dos operaciones principales las cuales lo implementaremos mediante funciones.

- Union(x, y)
- Find(x)

Empezemos a explicar el funcionamiento de esta estructura, supongamos que inicialmente tenemos 6 elementos (para nosotros nodos) donde inicialmente todos estan separados, en otras palabras tenemos 6 conjuntos unitarios.
Si todos están separados entonces podríamos decir que el padre de cada uno ellos es si mismo. Esto en código se traduciría como:

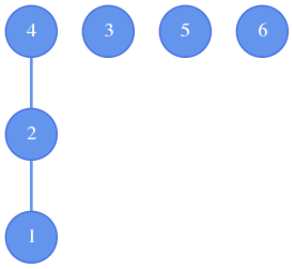
```
1 int padre[N+10];//donde N es el número de nodos
2 void init(int N){
3     for(int i=1;i<=N;i++){
4         padre[i]=i;
5     }
6 }
```

Hasta acá hemos logrado representar los elementos. Ahora podemos explicar las funciones de nuestra estructura de datos.

6.3.1 Find

La función **Find** nos permite hallar a que conjunto pertenece nuestro nodo **x**, lo que hace en si es ir buscando al padre de **x** recursivamente, hasta llegar a la raíz que es la que nos indica a que conjunto pertenece.

En la imagen anterior si desearíamos encontrar el padre de 3 **Find(3)** devolveria 3 por que el nodo 3 es padre del nodo 3 (ya que estan disjuntos).



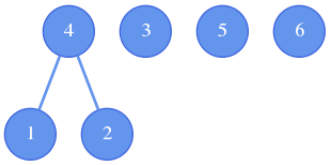
En la imagen anterior si desearíamos encontrar el padre de 1 **Find(1)** devolveria **Find(2)** por que el padre de 1 es 2, pero 2 no es la raiz, entonces seguimos buscando, **Find(2)** devolveria **Find(4)** por que el padre de 2 es 4 y **Find(4)** devolvera 4 por que el padre de 4 es 4, así llegando a la raíz de arbol. La manera de codificar seria la siguiente.

```
int Find(int x){
    if(x==padre[x]){ //cuando ya se llego a la raíz
        return x; // retornamos la raíz
    }else{
        return Find( padre[x] );
        //Si es que todavía no se llego a la raíz se la sigue buscando
}
```

Cabe destacar que esta función puede y debe ser optimizada para que no realice tantas llamadas recursivas.

```
int Find(int x){
    if(x==padre[x]){ //cuando ya se llego a la raíz
        return x; // retornamos la raíz
    }
    else {
                return padre[x]=Find(Padre[x]);
                //asignamos la raíz como el padre de x
   }
}
```

Ahora vamos a explicar esta breve optimización supongamos que hemos usado está función optimizada en el nodo 1 en la imagen anterior, el resultado sería lo siguiente.

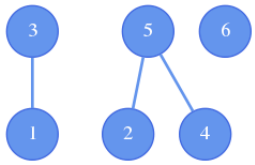


En este caso lo que hace nuestro Find optimizado es actualizar el padre de cada nodo por el que pase a la raíz, entonces todos los nodos que estaban en nuestro camino a la raíz quedan directamente unidos a esta, lo cual nos ayuda mucho en cuánto a tiempo se refiere. Imaginemos que en la imagen

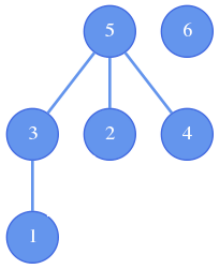
anterior en vez de haber un solo nodo entre 1 y la raíz hay 100000 nodos por los que queramos saber cuál es el padre de 1 tendríamos que pasar por esos 1000000 si tuviéramos que preguntar varias veces cuál es la raíz de 1 no resulta muy conveniente, en cambio con está optimización al preguntar varias veces por el padre de 1 sabremos instantáneamente que su padre es el nodo 4. Y lo propio con los 1000000 nodos que había en el camino.

6.3.2 Union

Como su nombre lo indica esta función une dos conjuntos disjuntos, logrando que ambos conjuntos tengan una ráíz en común, lo que unimos en este caso no son exactamente el nodo **x** y el nodo **y**, sino todos los elementos que son parte del mismo conjunto que **x** y todos los elementos que son parte del conjunto **y**. Supongamos que tenemos todos los nodos disjuntos como y unimos los nodos 1 , 3 y por otra lado los nodos 2, 4 y 5, tendríamos algo como esto.



De este modo se uniria dos conjuntos unitarios. Pero si unieramos el conjunto de 4 con el conjunto de 1 quedaria algo así.



Los lectores más hábiles se habrán percatado que lo que hacemos para unir dos conjuntos (partiendo de dos nodos) es buscar la raíz de esos dos conjuntos y unirlas como claramente se puede apreciar en la imagen anterior. A continuación se presenta el código de cómo implementar la función de **Union(x, y)**. no realice tantas llamadas recursivas.

```
1 void Union(int x,int y){
2     int u=Find(x),v=Find(y);//hallamos las raíces de ambos nodos
3     padre[u]=v;//el padre de u se convierte en el nodo v :)
4 }
```

Para no realizar uniones innecesarias (que los nodos **x** y **y** ya pertenezcan al mismo conjunto) podemos valernos de una función para verificar eso.

```
1 bool mismoConjunto(int x,int y){
2          //hallamos las raíces de ambos nodos
3     int u=Find(x),v=Find(y);
4 //si u es igual a v, ya son parte del mismo conjunto, caso contrario no lo son
```

```
5       return u==v;
6   }
```

Con esto hemos descrito ya el funcionamiento de esta importante estructura de datos, antes de concluir mencionaré unas últimas cosas sobre la estructura.

- Esta estructura es muy importante en el campo de los grafos ya que nos permite hallar el número de componentes conexas (por así decirlo el número de conjuntos que hay) para esto solo tendríamos que agregar a nuestra función init(N) una variable **conjuntos=N** y cada vez que realicemos una unión restar uno a la variable conjuntos.
- Otra aplicación importante que tiene es en dos algoritmos importantes de grafos Kruskal y Prim que se valen de está estructura para poder funcionar.
- Como el lector debió haber notado en las imágenes cada conjunto es un árbol a este conjunto de árboles se le denomina bosque, entonces Union-Find es un bosque.
- Si se quisiera separar un nodo de un conjunto es un tanto más difícil desafío al lector para que lo intente. Pista si el nodo es un nodo hoja es fácil retirarlo porque no tiene más hijos y sino se debe ver la forma de retirarlo sin separar a todos sus hijos del conjunto, mucha suerte.

6.4 Árbol de Segmentos

Tambien es llamado Segment Tree (en ingles), esta estructura es un tipo especial de árbol binario. Este árbol como su nombre lo indica trabaja con segmentos. En este capítulo veremos el funcionamiento y la implementación de esta estructura que nos ayudara bastanate.

6.4.1 Introducción

Antes de entrar al funcionamiento en si del árbol de segmentos, primero veremos un pequeño ejemplo para así poder entenderel porqué del uso de Segment Tree.
Supongamos que tenemos un arreglo de N elementos ($1 \leq N \leq 100000$), sobre el cual se te hará M ($1 \leq M \leq 100000$) consultas del tipo **¿Cuál es la suma en el rango L a R?** ($1 \leq L \leq R \leq N$)
Por ejemplo tenemos el siguiente arreglo.

Índice	1	2	3	4	5	6	7	8	9
int A[10]	12	7	6	1	0	12	2	3	4

Y nos preguntan cuál es la suma en el rango $[2,7]$ que en este caso seria $7+6+1+0+12+2=28$
Una primera idea (y la más obvia) para resolver el problema sería usar el siguiente código.

```
1   while(M--) { //mientras haya alguna otra consulta
2       //pedimos el rango de la consulta e iniciamos la suma en 0
3       int sum=0;
4       cin>>L>>R;
5       //sumar todo en el rango [L,R]
6       for(int i=L; i<=R; i++) {
7           sum+=v[i];
8       }
9       cout<<sum<<endl; //respondemos a la consulta
10  }
```

Claramente es algo muy sencillo de codificar; pero es muy lenta si se realizan muchas consultas. Podría tomarle hasta horas realizar la tarea. Entonces luego de pensar un poco viene a la mente la siguiente idea. Acumular las sumas hasta cada posición con la ayuda de un segundo array.

Índice	1	2	3	4	5	6	7	8	9
int Array[10]	12	7	6	1	0	12	2	3	4
int Acumulado[10]	12	19	25	26	26	38	40	43	47

Esto nos permitirá saber fácilmente cual es la suma en un rango mediante una simple resta entre **Acumulado[R]-Acumulado[L-1]**, por ejemplo si queremos saber la suma en el rango **[2, 7]** solo tendríamos que calcular $Acumulado[7] - Acumulado[1] = 40 - 12 = 28$, esto es mucho más rápido que el anterior método y tambien sencillo de codificar.

```
1  int acumulado[N+1];
2  acumulado[0]=0;//el valor de haber sumado 0 elementos del arreglo es 0
3  for(int i=1; i<=N; i++) {
4      acumulado[i]=acumulado[i-1]+v[i];
5      //la suma hasta la posición anterior lo sumamos con la posición actual
6  }
7  while(M--) {
8      cin>>L>>R;//pedimos el rango de la consulta
9      cout<<acumulado[R]-acumulado[L-1];//respondemos a la consulta
10 }
```

Con eso hemos resuelto el problema de manera correcta. Ahora pido al lector pensar en las siguientes situaciones.

- Aparte de existir consultas en las que te piden la suma del rango **[L, R]** hay otro tipo de consultas **Actualizar x y** donde se pide que le dé un valor **y** al elemento **Array[x]**.
- En lugar de preguntar por la suma en el rango **[L, R]** se pregunta por el máximo elemento en el rango **[L, R]**, además de existir actualizaciones.

Analicemos estás dos situaciones, en la primera ya no nos sirve de mucho el arreglo extra con los acumulados, porque al actualizar un valor la suma hasta esa posición se verá afectada, entonces tendríamos que volver a una situación similar al primer código donde se calculaba la suma para ese rango. En el segundo caso directamente no podemos usar el arreglo de acumulados para preguntar cuál es el mayor en el rango **[L, R]** y a eso hay que agregarle el hecho de que también existen actualizaciones, así que igual estaríamos en una situación similar al primer código, pero en lugar de sumar preguntando por el máximo.

Antes de seguir tómese un tiempo para pensar en alguna solución para estos problemas.

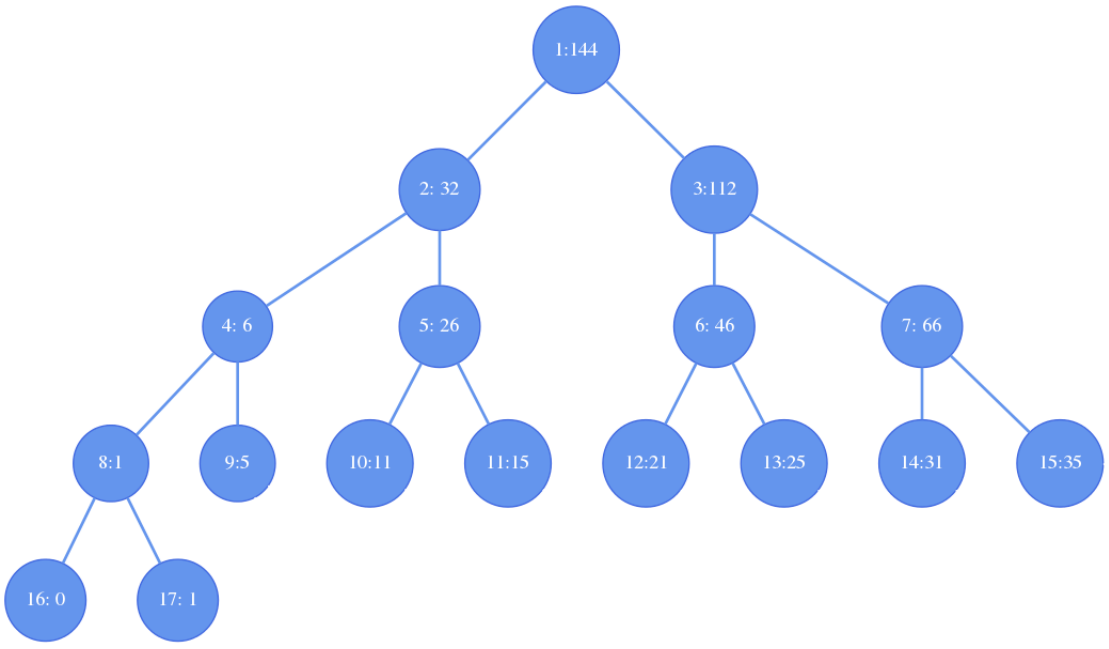
Si es que el lector no está familiarizado con Segment tree u otra estructura de datos que resuelva este tipo de problemas no habrá podido dar con una solución óptima. Este tipo de problemas es el que resuelve un Segment tree, problemas en los que se da un arreglo de elementos y varias consultas en las que algunas pueden ser actualizaciones de algún elemento, o responder por alguna pregunta en un rango [L,R], estás preguntas pueden ser cosas como el mínimo en el rango, el máximo en el rango, suma de los elementos del rango, multiplicación de los elementos en el rango, etc. Con la condición de que la operación que estemos realizando sea conmutativa, y asociativa.

6.4.2 Construcción

Tal y como su nombre indica está estructura es un árbol, en el cual nosotros guardamos la información de un determinado intervalo [L,R] en un nodo de este árbol, por ejemplo el nodo raíz contiene la información de todo el arreglo [1,n], ya sea esta información la suma de un intervalo, o hallar mínimos y máximos en un intervalo, o cualquier otra información de importancia. Ahora veamos cómo construir este tipo de árboles, partamos las dos ideas mencionadas en el anterior párrafo: Un nodo guarda la información de un intervalo [L,R] y el nodo raíz guarda la información de todo el arreglo, ahora diremos que todos los nodos en nuestro árbol tienen 0 o 2 hijos, 0 en el caso de ser nodos hoja. Cada nodo tiene la información de un intervalo, entonces su hijo izquierdo tendrá la información del intervalo del rango [L,mitad] y el derecho la información del intervalo (mitad,R] (donde mitad es (L+R)/2), estos dos nodos hijos a su vez tendrán la información de otros dos intervalos más pequeños. Por ejemplo supongamos que tenemos el siguiente arreglo.

Índice	1	2	3	4	5	6	7	8
int Array[9]	1	5	11	15	21	25	31	35

El árbol que generaríamos es el siguiente.



En la imagen podemos apreciar que los nodos hoja pertenecen al arreglo original (nodos 8 al 15), mientras que los demás nodos son la unión de algún intervalo, por ejemplo el nodo 4 representa al intervalo [1,2], el nodo 2 al intervalo [1,4], el nodo 1 nuestra raíz representa al intervalo [1,n]. Para implementar esto en código podríamos usar una función recursiva.

```
1 // todos estos valores son constantes en la primera llamada
2 void init(int node=1, int l=0, int r=N-1) {
3     //caso base cuando te encuentras en un nodo hoja
4     //y no puedes dividir en más intervalos
```

```
5       if(l == r) T[node] = a[l];
6       else {
7           int mi = (l + r) / 2; // mitad
8           init(2 * node, l, mi);//llamamos al hijo izquierdo
9           init(2 * node + 1, mi + 1, r);//llamamos al hijo derecho
10          T[node] = ( T[2 * node] + T[2 * node + 1] );
11          //como hemos ido llamando recursivamente
12          //ya sabremos el valor de T[2*node] y T[2*node+1] en este caso son sumas
13          //también podrían ser mínimos u otro tipo de datos.
14      }
15  }
```

En esta función iniciamos node=1 ya que 1 es nuestro nodo raíz que contendrá la información de todo el arreglo, l y r son los límites de nuestro intervalo inicialmente tomamos todo el rango de elementos. Como se puede apreciar tenemos un arreglo a[] que es en el que almacenamos los números originales, el otro arreglo T[] es nuestro árbol el tamaño aconsejable para este arreglo es 4*N ya que como se habrá podido ver con la anterior imagen el tamaño del árbol es mucho más grande que el arreglo original y en los peores casos necesitaremos aproximadamente 4 veces el número de elementos original. Si el lector se preguntaba porque existen los nodos 16 y 17 en la anterior imagen estos corresponden a a[0] (un elemento que no existe en nuestro arreglo y al valer 0 no afecta en nada a la suma) y a a[1]. Acá presentamos una tabla que indica que intervalo cubre cada nodo:

Nodo	L	R
1	0	8
2	0	4
3	5	8
4	0	2
5	3	4
6	5	6
7	7	8
8	0	1
9	2	2
10	3	3
11	4	4
12	5	5
13	6	6
14	7	7
15	8	8
16	0	0
17	1	1

Como se puede ver los nodos hoja contienen al arreglo original y sus rangos son [L,L].

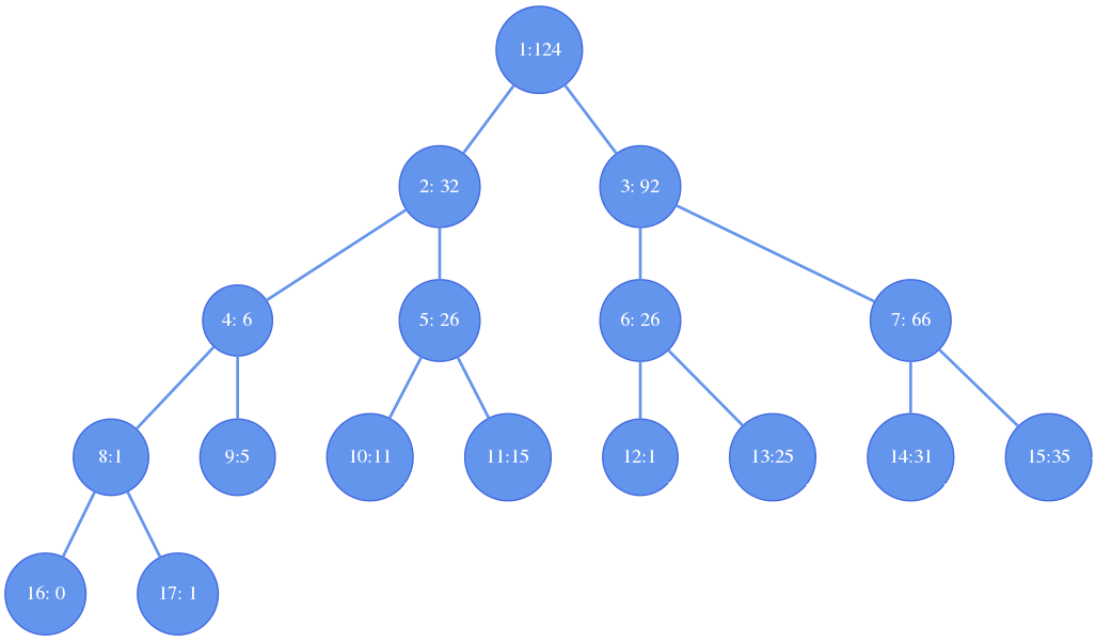
6.4.3 Update

Hasta ahora podemos generar el árbol que contiene la información de los intervalos ahora proseguiremos con las actualizaciones, cambiar el valor de un elemento del arreglo por otro, para esto vamos a usar una función muy parecida a la función previa para construir el árbol, con la diferencia de que

ahora necesitamos dos parámetros más un entero x que es la posición que queremos actualizar por un entero val, también cambia el caso base de nuestra recursión:

```
1  void update(int x, int val, int node=1, int l=0, int r=N-1) {
2      if(r < x || l > x) return; //si nos salimos del rango
3      //actulización cuando hemos llegado al nodo hoja buscado
4      if(l == r) T[node] = val;
5      else {
6          int mi = (l + r) /2;
7          update(x, val, 2 * node, l, mi);//hijo izquierdo
8          update(x, val, 2 * node + 1, mi + 1, r);//hijo derecho
9          //actualización de los otros nodos
10         T[node] = ( T[2 * node] + T[2 * node + 1] );
11     }
12 }
```

Como se puede ver en el código es prácticamente igual que la función build, solo que ahora cambia nuestro caso base. Por ejemplo si en el arreglo de la anterior imagen actualizamos el Array[5] con un 1 quedaría así.

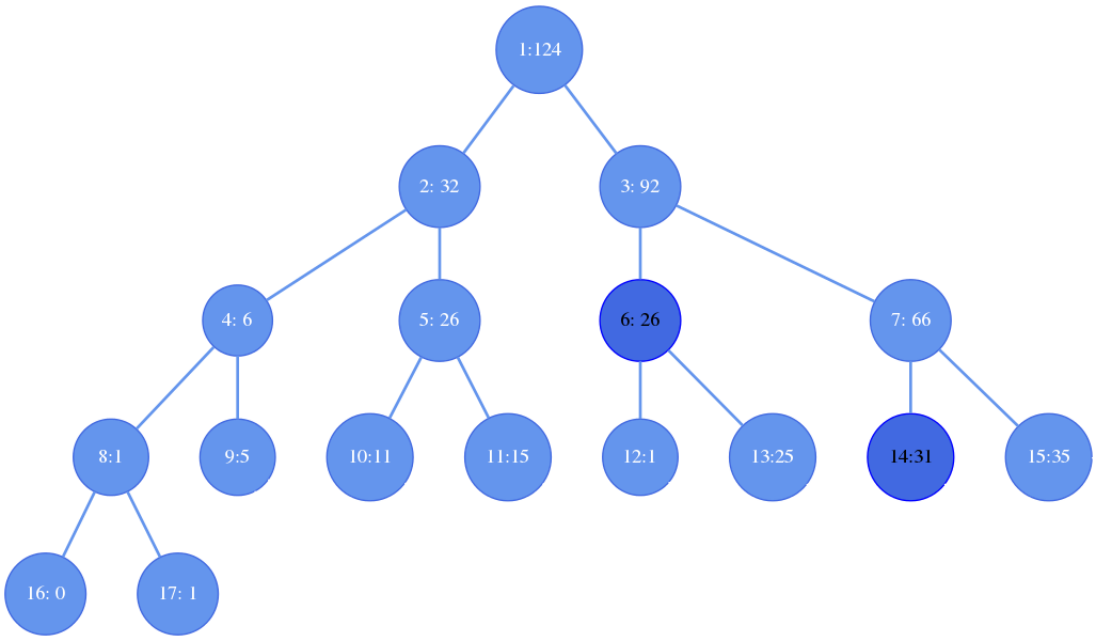


El árbol resultante luego de aplicar una actualización al elemento 5 del arreglo.
Como se puede ver en la anterior imagen solo los nodos que estaban asociados al nodo 12 que era el que contenía a la posición 5 cambiaron, mientras que los otros nodos mantienen sus valores originales. (Comparar con las anteriores dos imagenes).

6.4.4 Query

Finalmente llegamos a la parte en la que realizamos las de cuánto es la suma (o el mínimo o algún otro dato importante) en el rango [L,R], es probable que se haya percatado de un pequeño “error”

en nuestro árbol si por ejemplo se quisiera saber la suma en el intervalo [5,7] no tenemos un nodo que tenga la respuesta para ese intervalo tenemos uno que guarda el intervalo [5,6] (nodo 6), otro que guarda el intervalo [5,8] (nodo 3). Sin embargo la función query corrige esté aparente problema tomando el rango [5,6] (nodo 6) y el rango [7,7] (nodo 14) a continuación los suma.



Como vemos en la imagen anterior nuestro algoritmo tomará los dos nodos pintados para responder a nuestra consulta, el código para resolver esto es el siguiente.

```
1  //inicialización igual pero con el rango x y
2  int query( int x, int y, int node=1, int l=0, int r=N-1) {
3      if(r < x || l > y) return 0;
4      //caso base en el que un intervalo no debe ser tomado en cuenta en
5      //este caso retornamos 0 porque sumar algo+0=algo
6      //si se tratará de hallar el mínimo deberíamos poner un número muy grande
7      //ya que min(numgrande,otronumero) será el otro número
8      if(x <= l && r <= y) {
9          return T[node];
10         //si el rango forma parte de la solución tomamos en cuenta ese nodo
11     } else {
12         int mi = (l + r) /2;//la mitad
13         return query(x, y, 2 * node, l, mi) + query( x, y, 2 * node
               + 1, mi + 1, r);
14         //similar a las otras funciones hijo izquierdo e hijo derecho
15     }
16 }
```

Ahora solo nos falta tomar en cuenta algunas cosas:

- Como el lector ya hará podido notar en esta implementación los rangos son tomados de 1 a N y no de 0 a N-1, también es posible implementar de esta forma.

- Segment tree puede ser extendido a más de una dimensión, por ejemplo en matrices. Se debe tener muy en cuenta la cantidad de espacio que se utiliza para guardar la estructura en estos casos.
- Existen consulta de actualización en un rango en segment tree, para esto hay que modificar un poco el algoritmo y usar un algoritmo llamado “lazy propagation”.
- Segment tree suele acompañar a problemas de geometría, y servir en varios problemas de estructuras de datos.

7. Grafos

7.1 Introducción

Representar una ciudad, una red de carreteras, una red de computadoras, etc., nos resulta un poco dificil con las estructuras de datos que conocemos asi es que vamos a definir un grafo como un conjunto de "objetos" relacionados entre si, pero a diferencia de un conjunto normal un grafo G contiene dos conjuntos, el primer conjunto contiene a los objetos y el segundo conjunto contiene a las relaciones que existen entre ellos.

7.1.1 ¿Qué es un Grafo?

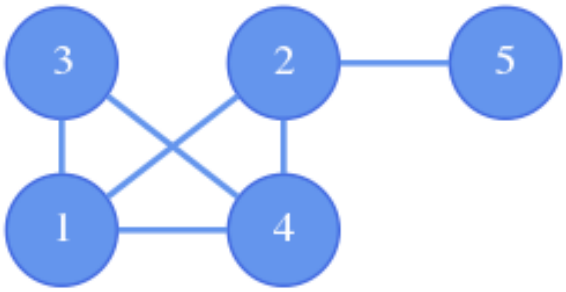
Un grafo G es un par ordenado G=(V,E), donde:

- V es un conjunto de vértices o nodos.
- E es un conjunto de aristas o arcos, que relacionan estos nodos.

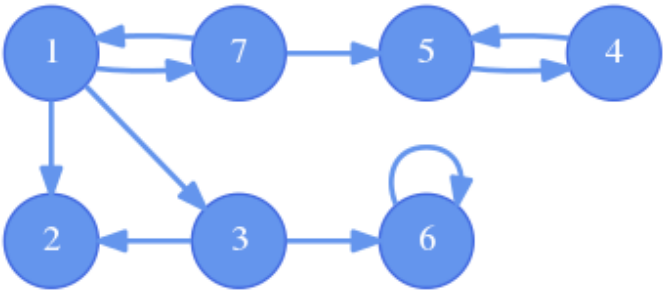
V suele ser un conjunto finito, aunque hay quienes se dedican a investigar grafos infinitos, en este libro nos dedicaremos exclusivamente a grafos finitos.

7.1.2 ¿Para que sirven los grafos?

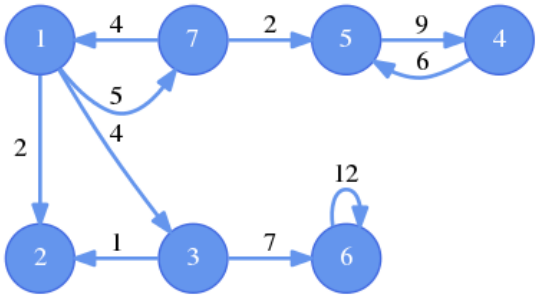
Podemos utilizar a los grafos para modelar una ciudad, laberintos, carreteras, redes de computadoras, etc. Volviendo a la definición anterior un grafo es un conjunto de vértices y arcos, entonces podemos modelar cualquier problema que tenga un conjunto de objetos (nodos) y otro conjunto que los relacione entre si (arcos).
Ejemplos:
Grafo no dirigído:



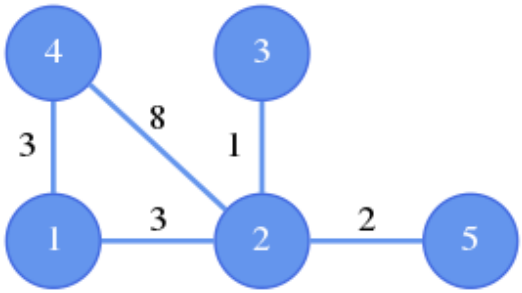
Grafo dirigído:



Grafo dirigído ponderado:



Grafo no dirigído ponderado:



Como vimos en los ejemplos anteriores, un grafo puede ser armado de distintas formas. Ahora que ya sabemos que son los grafos y para que sirven, necesitamos saber ¿Cómo los representamos en un lenguaje de programación?.

7.2 Representaciones de un grafo

Podemos representar un grafo de varias formas, especialmente veremos 3.

- Matriz de adyacencia.
- Lista de adyacencia.
- Lista de arcos.

7.2.1 Matriz de adyacencia

Esta representación es la mas fácil de implementar y la mas obvia al momento de pensar en grafos. Un grafo se lo representara en una matriz (de tipo bool, int, double, etc.) el tipo de la matriz

dependera del tipo de grafo, por ejemplo si queremos representar un grafo sin costos (dirigido o no dirigido) nos basta con una matriz de variables booleanas.
A continuacion su codigo.

Código 7.1: Matriz de adyacencia grafo no dirigido

```
#include <iostream>

using namespace std;

bool Grafo[100][100]; // para 100 nodos

int main(){
   int n,m,u,v;
   for(int i = 0; i < 100; i++)
      for(int j = 0; j < 100; j++)
         Grafo[i][j] = false; // inicializamos el grafo

   cin>>n>>m; // leemos número de nodos y número de arcos
   for( int i = 0; i < m; i++){
      cin>>u>>v; // leemos los arcos
      Grafo[u][v] = true; // grafo dirigido (camino de U a V)
      // Grafo[v][u] = true;
      // quitar comentario para grafo no dirigido (camino de V a U)
   }
   return 0;
}
```

Para un grafo ponderado:

Código 7.2: Matriz de adyacencia grafo ponderado

```
#include <iostream>

using namespace std;

int Grafo[100][100]; // para 100 nodos

int main(){
   int n,m,u,v,p;
   for(int i = 0; i < 100; i++)
      for(int j = 0; j < 100; j++)
         Grafo[i][j] = 0;  // inicializamos el grafo

   cin>>n>>m; // leemos número de nodos y número de arcos
   for( int i = 0; i < m; i++){
      cin>>u>>v>>p; // arcos ( nodo U, nodo V, costo P)
      Grafo[u][v] = p; // grafo dirigido
      // Grafo[v][u] = p; quitar comentario para grafo no dirigido
   }

   return 0;
```

```
21 }
```

El número 100 nos indica cuantos nodos tiene nuestro grafo, la idea de esta representacion es poner true cuando existe un arco desde el subindice i (fila) al subindice j (columna) o false cuando no hay un arco de i a j.
Una de las ventajas que nos ofrece esta representación es la facilidad al momento de codificar. Una de las desventajas es que el momento de recorrer el grafo debemos revisar toda la matriz, haciendo que el recorrido sea de orden $O(n^2)$.

7.2.2 Lista de adyacencia

Representar un grafo mediante una lista de adyacencia es muy util y óptimo ya que al recorrer el grafo solo revisamos los arcos existentes, la implementación de una lista de adyacencia se puede hacer de varias formas.

Código 7.3: Lista de adyacencia con Array estático

```
1  #include <iostream>
2  #include <vector>
3
4  using namespace std;
5
6  vector<int> Grafo[100]; // para 100 nodos
7
8  int main(){
9     int n,m,a,b;
10    cin>>n>>m; // leemos numero de nodos y arcos
11    for(int i = 0; i < m; i++){
12       cin>>a>>b;// leemos nuestros arcos (Nodo A, Nodo B)
13       G[a].push_back(b); // grafo dirigido
14    }
15
16    // ...
17    // nuestro codigo para solucionar problemas
18    // ...
19
20    //para limpiar el grafo (si lo necesitamos)
21    for(int i = 0; i < n; i++)
22       G[i].clear();
23
24    return 0;
25 }
```

Código 7.4: Lista de adyacencia con Array estático grafo ponderado

```
1  #include <iostream>
2  #include <vector>
```

```
3
4  using namespace std;
5
6  vector< pair<int,int> > Grafo[100];
7  // para 100 nodos la primera variable de pair es el nodo destino
8  // la segunda variable de pair es el peso
9
10 int main(){
11    int n,m,a,b,p;
12    cin>>n>>m; // leemos numero de nodos y arcos
13    for(int i = 0; i < m; i++){
14       cin>>a>>b>>p; // leemos arcos (nodo A, nodo B, costo P)
15       G[a].push_back(make_pair(b,p)); // grafo dirigido
16       G[b].push_back(make_pair(a,p)); // grafo no dirigido
17    }
18
19    // ...
20    // nuestro codigo para solucionar problemas
21    // ...
22
23    //para limpiar el grafo (solo si es necesario)
24    for(int i = 0; i < n; i++)
25       G[i].clear();
26
27    return 0;
28 }
```

Código 7.5: Lista de adyacencia con Array dinámico

```
1  #include <iostream>
2  #include <vector>
3
4  using namespace std;
5
6  vector< vector < int > > Grafo;
7
8  int main(){
9     int n,m;
10    cin>>n>>m; // leemos nodos y arcos
11    Grafo.assign( n , vector<int>() );
12    // inicializamos el grafo para n nodos
13    // lectura igual que en el ejemplo anterior
14
15    Grafo.clear(); // limpiamos
16    return 0;
17 }
```

Nota: Para grafos ponderados solo reemplazar la linea 6 por vector < vector < pair < int,int > > > Grafo;

7.2.3 Lista de arcos

Esta representación solo sirve cuando necesitemos ordenar los arcos.

Código 7.6: Lista de arcos

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

using namespace std;

vector< pair < int, pair <int,int> > > Grafo;
// la primera variable es el costo
// la segunda es el nodo origen y la tercera el nodo destino

int main(){
   int n,m,a,b,p;
   cin>>n>>m; // leemos nodos y arcos
   for(int i = 0; i< m; i++){
      cin>>a>>b>>p;
      Grafo.push_back(make_pair(p,make_pair(a,b)));
      // insertamos peso, nodo A, nodo B
   }

   Grafo.clear(); // limpiamos
   return 0;
}
```

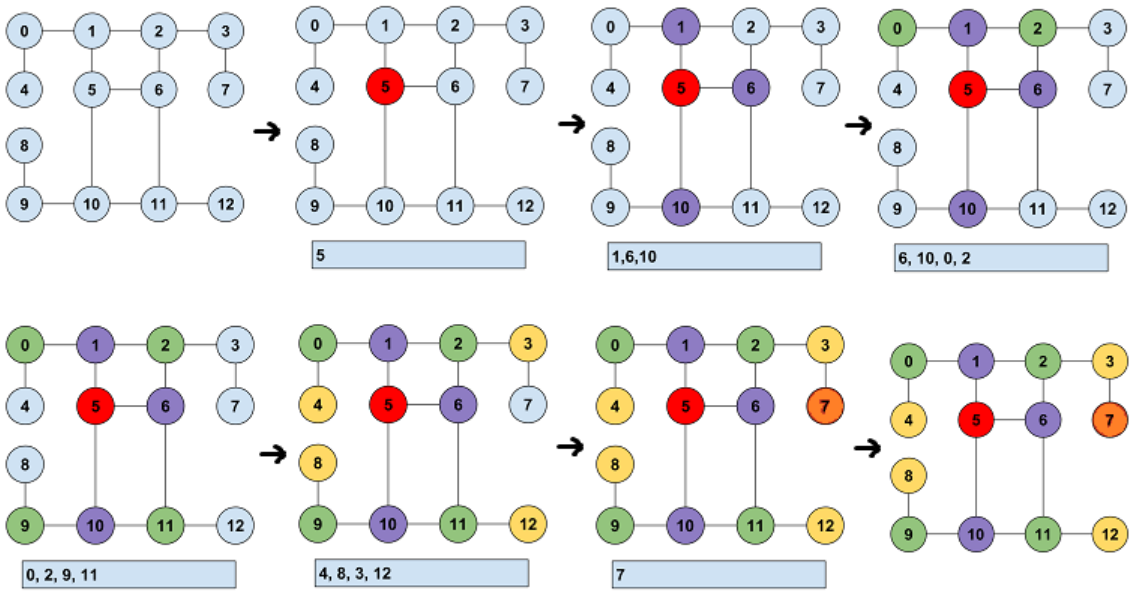
7.3 Recorrido de un grafo

Una ves armado el grafo, viene lo realmente importante que es que hacer con el en este caso lo que haremos seran 2 recorridos simples en todo el grafo, estos son:

7.3.1 BFS (Breadth First Search)

Este algoritmo recorre el grafo de manera progresiva y ordenada, lo recorre por niveles. BFS hace uso de una “cola” (queue) en la que coloca los nodos que visitara, de esta manera al terminar de visitar a los vecinos de la raíz continuara con los vecinos de estos y así sucesivamente.

Usaremos un ejemplo para explicar este recorrido.



Ahora vamos a explicar la ejecución, empezamos en el nodo 5, el primer nodo en ponerse en cola esta en color rojo, posteriormente revisamos a sus vecinos que estan de color lila y también los ponemos en cola, luego los nodos lila se ponen en cola y se revisa a sus vecinos (color verde) y también los ponemos en cola, asi sucesivamente hasta que todo el grafo este visitado.
Para evitar que el algoritmo sea infinito vamos a instanciar un Array que nos indique si un nodo fue visitado o no. A continuación el codigo de la búsqueda en profundidad.

Código 7.7: Busqueda en anchura

```
#include <iostream>
#include <vector>
#include <string.h>
#include <queue>

using namespace std;

vector< int > Grafo [100];
int visitado[100]; //para controlar los visitados
// visitado[i] quiere decir en que tiempo visitamos el nodo i
void bfs(int nodo){
   int u,v;
   queue<int> q;
   q.push(nodo);
   visitado[nodo] = 0;
   while(!q.empty()){
      u = q.front();q.pop();
          for(int i = 0; i < Grafo[u].size(); i++){
         v = Grafo[u][i];
                  if(visitado[v] == -1){
                          //si el nodo v no esta visitado
```

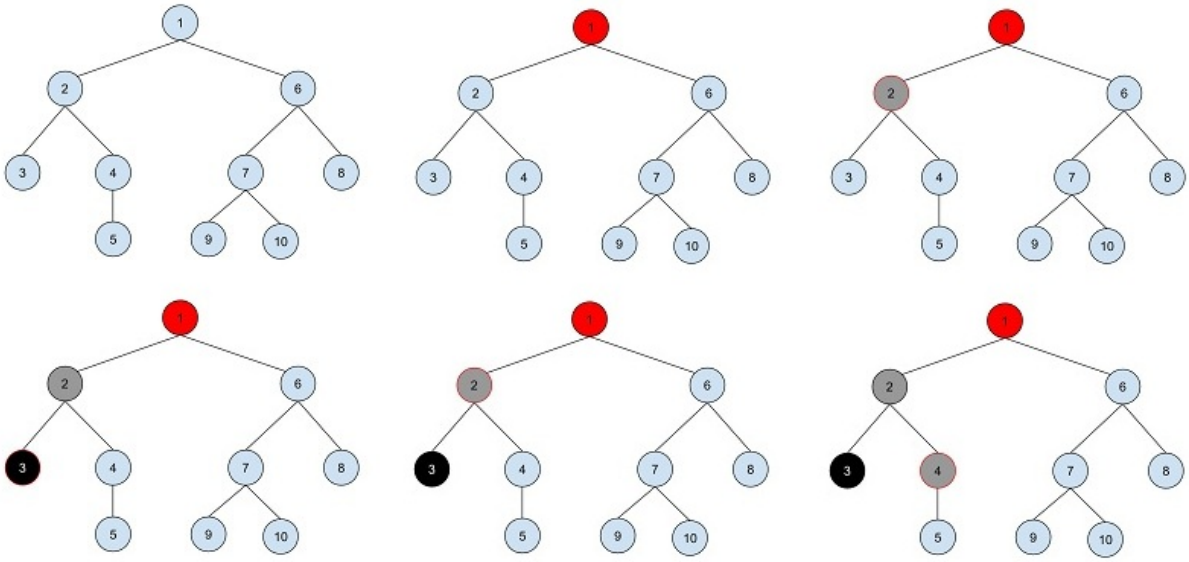
```
22                              q.push(v);
23                              visitado[v] = visitado[u] + 1;
24                              // al nodo V llegamos con 1 paso adicional
25                      }
26              }
27      }
28  }
29
30  int main(){
31      // ...
32      // lectura del grafo
33      // ...
34      memset(visitado,-1,sizeof(visitado)); // -1 no visitado
35      bfs(5); // bfs desde un nodo empírico
36
37      // ya recorrimos todo el grafo :)
38      return 0;
39  }
```

Si TODOS LOS ARCOS PESAN LO MISMO entonces con el anterior codigo podemos hallar el camino mas corto desde 5 hasta todos los demás nodos, también podemos saber si el nodo X esta conectado con el nodo Y, además podemos hallar en número de Componentes Fuertemente Conexas (Si el grafo es NO DIRIGIDO).

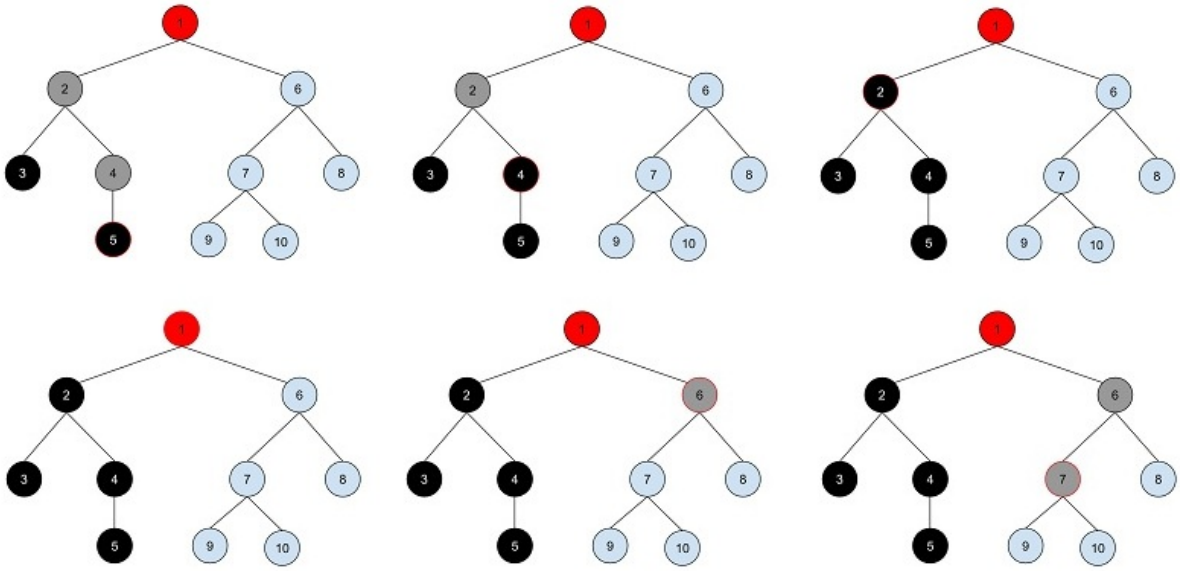
Nota: Un grafo es llamado fuertemente conexo si para cada par de vértices U y V existe un camino de U hacia V y un camino de V hacia U. Las componentes fuertemente conexas (SCC por sus siglas en inglés) de un grafo dirigido son sus subgrafos máximos fuertemente conexos. Estos subgrafos forman una partición del grafo.

7.3.2 DFS (Depth First Search)

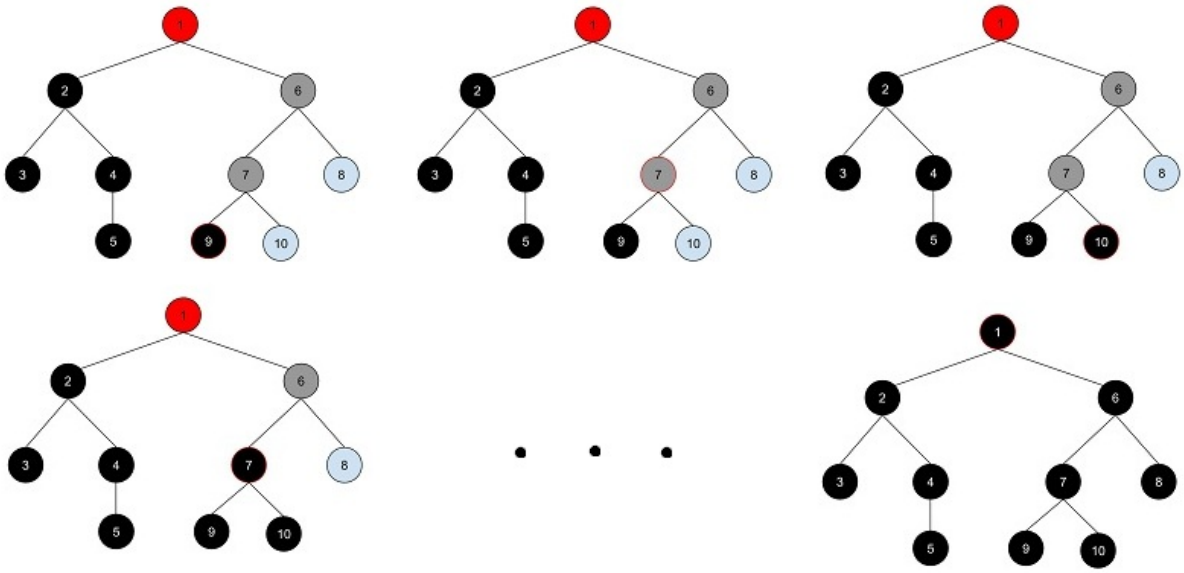
Este recorrido es recursivo, y como su nombre indica (Busqueda en Profundidad) lo que hace es entrar hasta lo mas profundo del grafo y luego empieza a retornar, para visualizar como funciona pintaremos los vertices de 2 colores, de gris si este nodo se esta procesando y de negro cuando ya fue procesado, tambien usaremos un borde rojo en el nodo en el que nos encontramos.



Como podemos observar en el gráfico anterior, la raiz será el nodo 1, despues con la recursión vamos "bajando" hasta lo más profundo de esa rama, cuando llegamos a un nodo hoja (sin mas hijos) entonces volvemos atras y "bajamos" por las otras ramas que aún no han sido visitadas.



El algoritmo se encarga de visitar todo el grafo en profundidad, llegando hasta lo más profundo y luego volviendo atrás para ir por otras ramas no visitadas.



Después de haber terminado con la recursión se ve que el último nodo en terminar de visitar es el primer nodo. A continuación el codigo:

Código 7.8: Busqueda en profundidad

```
#include <iostream>
#include <vector>
#include <string.h>
using namespace std;

vector< int > Grafo [100];
bool visitado[100]; //para controlar los visitados
// visitado[i] quiere decir si visitamos el nodo i
void dfs(int nodo){
    visitado[nodo] = true;
    for(int i = 0; i < Grafo[nodo].size(); i++){
        int v = Grafo[nodo][i];
            if(!visitado[v]){
                //si el nodo v no esta visitado
                    dfs(v);
            }
    }

}

int main(){
    // ...
    // lectura del grafo
    // ...
    memset(visitado,false,sizeof(visitado)); // false no visitado
    dfs(1); // bfs desde un nodo empírico

```

```
28      // ya recorrimos todo el grafo :)
29      return 0;
30  }
```

De este recorrido podemos sacar más información que de la búsqueda en anchura, debido a que este recorrido nos genera un árbol llamado árbol de DFS, esto lo veremos en el siguiente capitulo.

8. Algoritmos ++

8.1 Árbol de expansión mínima

Un árbol de recubrimiento mínimo es un subgrafo que conecta todos los nodos con el menor costo posible, como tratamos de minimizar el costo de conectar todos los nodos partimos de la observación: para conectar 3 nodos solo necesitamos 2 arcos y asi generalizamos que para conectar n nodos solo necesitamos n-1 arcos.
Al tener un grafo de n nodos no dirigido ponderado y conexo la pregunta es ¿Cómo elegimos los n-1 arcos para minimizar el costo?. Para solucionar este problema hay muchos algoritmos, los más conocidos son el algoritmo de Kruskal (que veremos en el libro) y el algoritmo de Prim.

8.1.1 Algoritmo de Kruskal

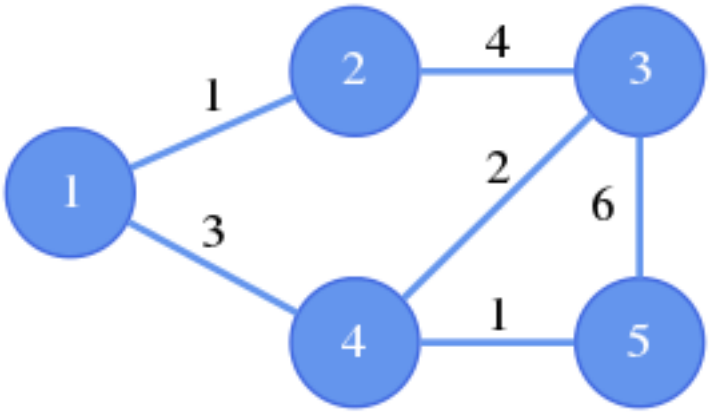
Para poder comprender el algoritmo de kruskal será necesario revisar el capitulo 6 (Union-Find). Este algoritmo trata de unir los arcos con menor costo siempre que no formen ciclos, de esta manera garantizamos de tomar solo los arcos necesarios y minimizar el costo.
A continuación el código:

Código 8.1: Algoritmo de Kruskal

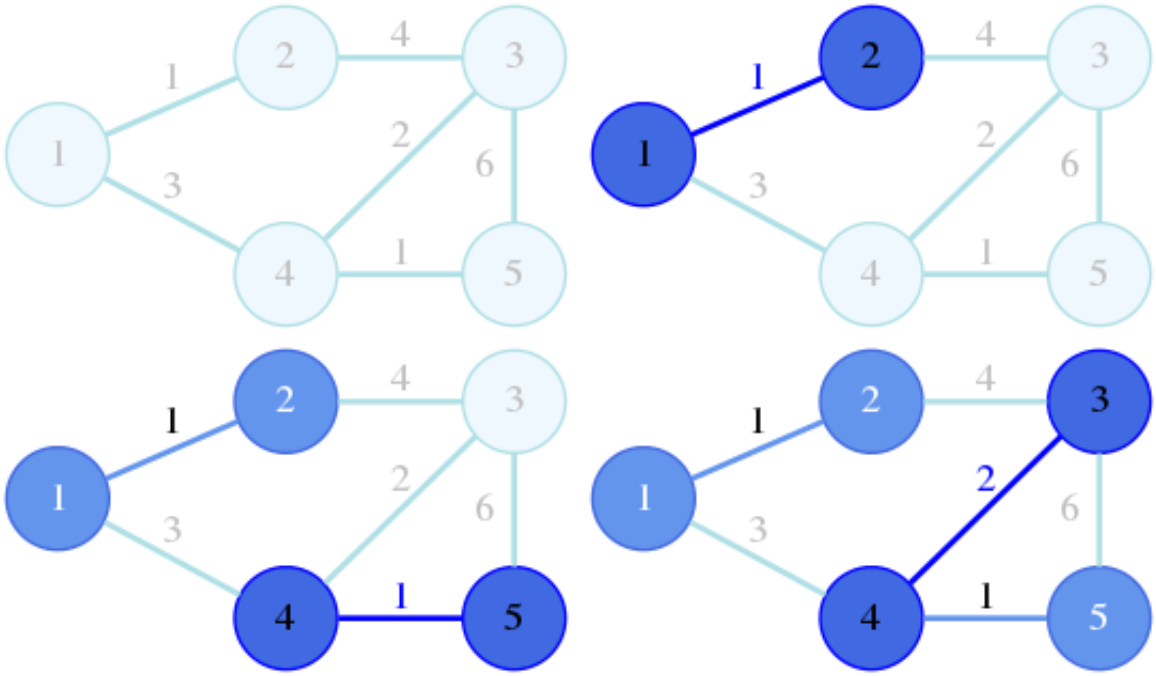
```
int N,lider[100];
// N es el numero de nodos
vector<pair<int,pair<int,int> > >  v;
// v es la lista de arcos (ver capítulo 7)
void init(){
   sort(v.begin(),v.end());
   for(int i=0;i<N;i++)
      lider[i]=i;
}
void Union(int a,int b){
   lider[b] = a;
}
int Find(int n){
   if(n==lider[n])return n;
   return (lider[n] = Find(lider[n]));
}

int Kruskall(){
```

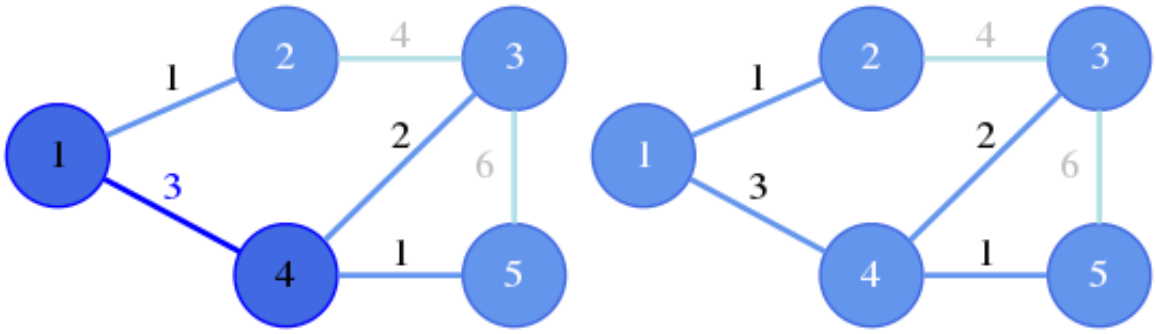
```
19      int a;int b;int sum=0;
20      init();
21      for(int i=0; i < v.size();i++){
22          a = Find(v[i].second.first);
23          b = Find(v[i].second.second);
24          if(a != b){
25              sum += (v[i].first);
26              Union(a,b);
27          }
28      }
29      return sum;
30  }
```

Sea el siguiente grafo:



Lo que hace el algoritmo de kruskal:





8.2 Camino más corto

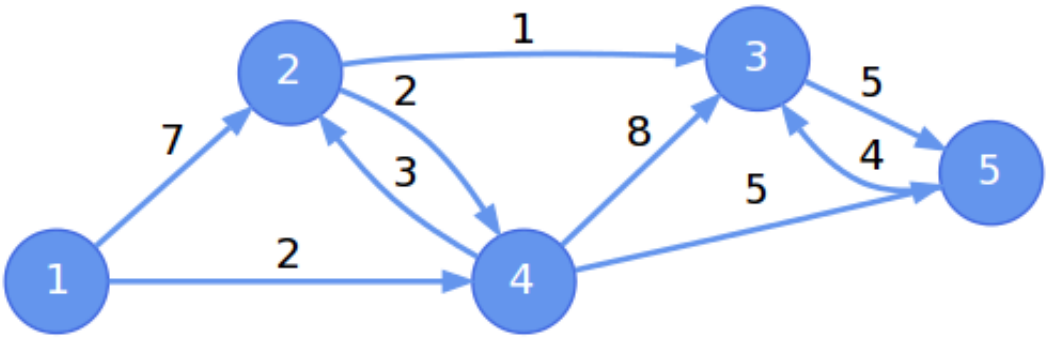
El problema del camino más corto es clásico en los grafos, como el nombre lo indica se trata de hallar el camino más corto desde un nodo hacia todos los demás.
Para resolver este problema podemos utilizar una técnica greedy, tomando la mejor opción entre ir directo o ir con algunos nodos intermedios.
De lo que esta técnica trata es de marcar todos los vértices como no utilizados. El algoritmo parte de un vértice origen que será ingresado, a partir de ese vértices evaluaremos sus adyacentes, como dijkstra usa una técnica greedy - La técnica greedy utiliza el principio de que para que un camino sea óptimo, todos los caminos que contiene también deben ser óptimos- entre todos los vértices adyacentes, buscamos el que esté más cerca de nuestro punto origen, lo tomamos como punto intermedio y vemos si podemos llegar más rápido a través de este vértice a los demás. Después escogemos al siguiente más cercano (con las distancias ya actualizadas) y repetimos el proceso. Esto lo hacemos hasta que el vértice no utilizado más cercano sea nuestro destino. Al proceso de actualizar las distancias tomando como punto intermedio al nuevo vértice se le conoce como relajación.
El algoritmo de Dijkstra es muy similar a BFS, si recordamos BFS usaba una Cola para el recorrido para el caso de Dijkstra usaremos una Cola de Prioridad o Heap, este Heap debe tener la propiedad de Min-Heap es decir cada vez que extraiga un elemento del Heap me debe devolver el de menor valor, en nuestro caso dicho valor será el peso acumulado en los nodos.

Código 8.2: Algoritmo de Dijkstra

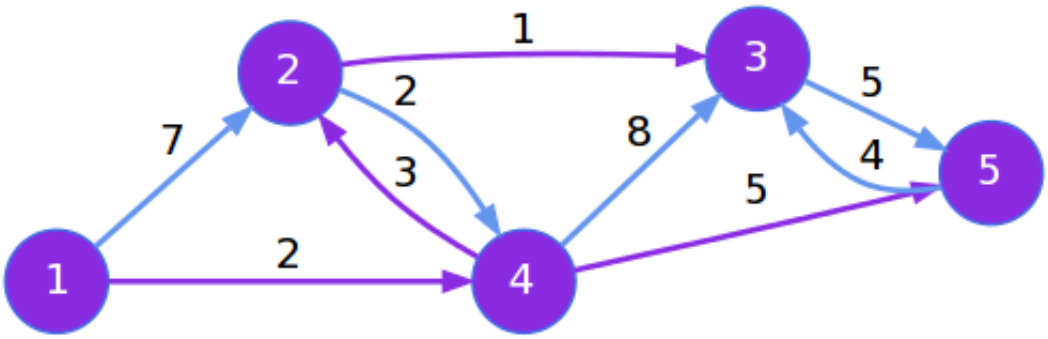
```
1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  typedef vector<int> vi;
6  typedef pair<int,int> ii;
7  typedef vector<ii> vii;
8  typedef vector<vii> vvii;
9
10 const int MAX = 1001;
11 const int MAXINT = 1000000000;
12
13 int n;
```

```
14 vvii G(MAX);
15 vi D(MAX, MAXINT);
16
17 void Dijkstra(int s){
18    //usaremos un set para obtener el menor
19    set<ii> Q;
20    D[s] = 0;
21    Q.insert(ii(0,s));
22    while(!Q.empty()){
23       ii top = *Q.begin();
24       Q.erase(Q.begin());
25       int v = top.second;
26       int d = top.first;
27       vii::const_iterator it;
28       for ( it = G[v].begin(); it != G[v].end(); it++){
29          int v2 = it->first;
30          int cost = it->second;
31          if (D[v2] > D[v] + cost){
32             if (D[v2] != 1000000000){
33                Q.erase(Q.find(ii(D[v2], v2)));
34             }
35             D[v2] = D[v] + cost;
36             Q.insert(ii(D[v2], v2));
37          }
38       }
39    }
40 }
41
42 int main(){
43     int m, ini, fin = 0;
44     scanf("%d %d %d %d", &n, &m, &ini, &fin);
45         // nodos, arcos, inicio, destino
46     for (int i = 0; i < m; i++){
47         int a, b, p = 0;
48         scanf("%d %d %d", &a, &b, &p);
49         G[a].push_back(ii(b, p));
50         G[b].push_back(ii(a, p));
51     }
52     Dijkstra(ini);
53     printf("%d\n", D[fin]);
54     return 0;
55 }
```

El anterior código nos halla los caminos más cortos desde un nodo hasta todos los demás, por ejemplo en el siguiente grafo:



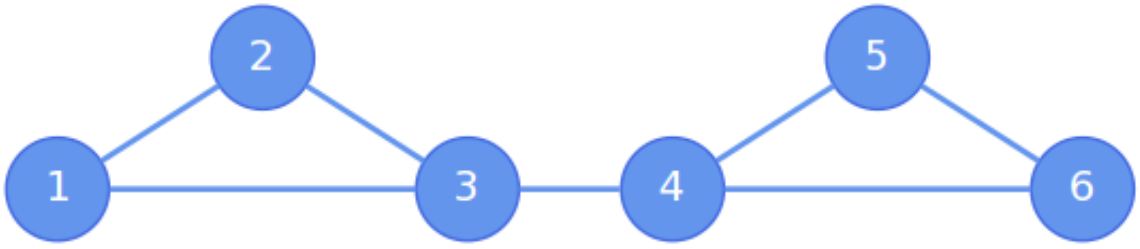
Ejecutamos el algoritmo de dijkstra desde el nodo 1, obteniendo el siguiente resultado:



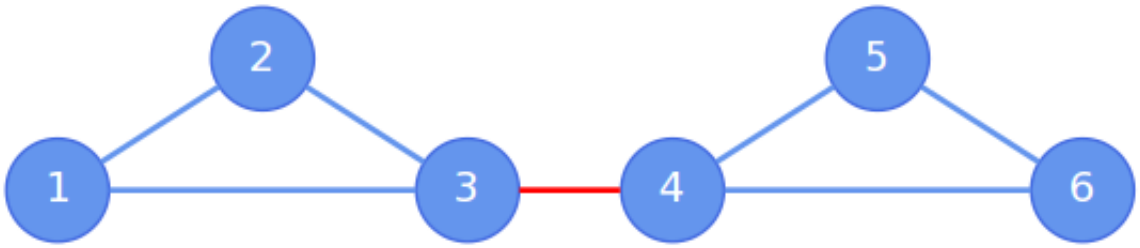
Los caminos violetas forman parte del arbol de caminos cortos, se puede ver que solo hay un camino desde el nodo 1 hasta cualquier otro nodo, siendo este camino el camino mas corto.

8.3 Puntos de Articulación y Puentes

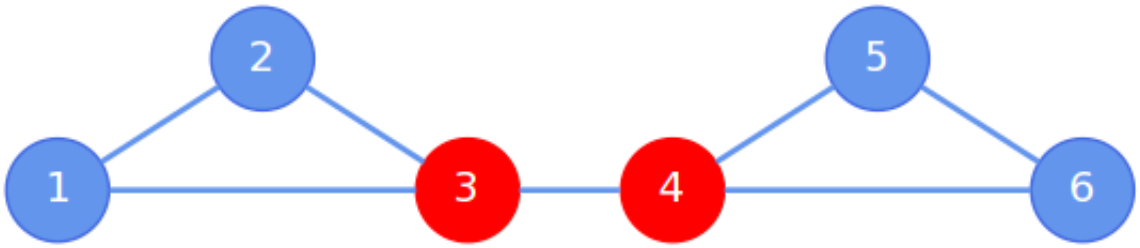
Un ***Punto de Articulación*** en un grafo es un vértice que al ser eliminado divide al grafo original en dos o más partes el grafo original convirtiéndolos en dos grafos.
Cuando en un grafo no existe un punto de articulación se dice que es un grafo biconexo ya que existen dos o más caminos que conectan al vértice.
Un ***Puente*** en un grafo es un arco que al ser eliminado divide al grafo original en dos o más partes el grafo original convirtiéndolos en dos grafos.
Vamos a identificar en el siguiente grafo los puentes y puntos de articulación:



El único arco que al eliminarlo nos desconecta el grafo es el arco 3,4.



Los únicos nodos que al eliminarlos nos desconecta el grafo son: 4,3.



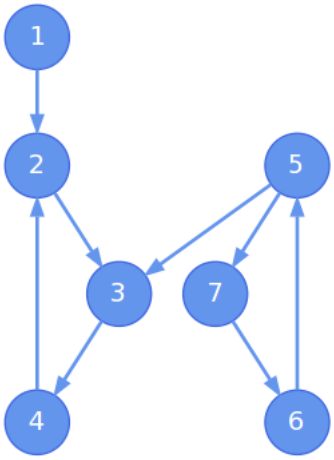
Para poder encontrar un punto de articulación en un grafo se tiene el siguiente procedimiento:

1. Crear un árbol del grafo (Arbol del DFS).
2. Elegir un nodo como raíz, este será el que tenga mayor conexión (hijos).
3. Se recorre al siguiente nodo más cercano por la izquierda.
4. Se repite el paso 2 hasta completar todos los nodos.

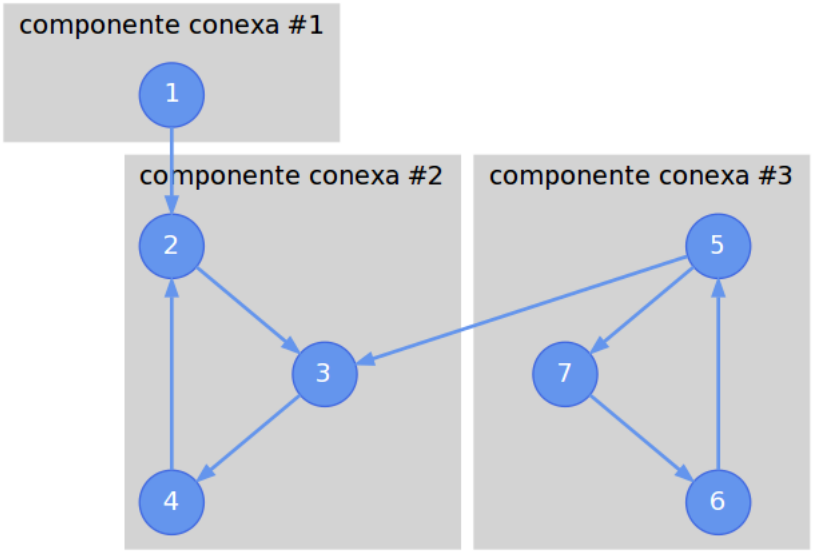
Los anteriores pasos se los puede realizar mediante un recorrido en profundidad (DFS).
En el momento de hacer el recorrido DFS estamos generando el arbol del DFS, podemos aprovechar este recorrido para hacer el siguiente paso de nuestro algoritmo.
Instanciamos dos arrays que se llamen time y low, estos nos indicaran el momento en el que lo visitamos y el nodo mas bajo al que llegamos con un backedge. (Un backedge se lo detecta cuando el hijo de i tiene un padre, que tambien es el padre de i).
Para que un nodo se pueda considerar como punto de articulación debe cumplir con low[V] >= time[V] (en el gráfico bajo[v] >= time[v]). Para que un arco sea considerado puente solo hay que verificar que (low[v] > time[u] para todo padre v e hijo u).

8.4 Componentes Conexas

Como habíamos visto en el capítulo anterior en un grafo no dirigido hallar una componente conexa es fácil, pero cuando son grafos dirigidos el algoritmo deja de ser tan obvio.
Definición: Un grafo dirigido es fuertemente conexo si hay un camino desde cada vértice en el grafo a cualquier otro vértice.
Definición: Una componente fuertemente conexa de un grafo dirigido es un conjunto maximal de vértices en el cual existe un camino desde cualquier vértice en el conjunto a cualquier otro vértice en el conjunto.



En el anterior grafo solo hay 3 componentes fuertemente conexas, ¿ Te animas a hallar las componentes conexas ? recuerda que en una componente conexa hay camino de todos los nodos a todos los nodos.
La respuesta es 3:



La técnica Búsqueda en profundidad (DFS) puede ser usada para determinar las componentes fuertemente conexas (SCC) de un grafo dirigido eficientemente mediante el ***Algoritmo de Kosaraju***.
Este algoritmo se basa en el hecho de que si invertimos todas las aristas de un grafo (hallamos el grafo transpuesto), las SCC son las mismas que en el grafo original.
El algoritmo de Kosaraju lo que hace para hallar las SCC de un grafo dirigido G:

1. Realiza un DFS sobre G y enumera los vértices en orden de terminación de las llamadas recursivas (como en el anterior algoritmo).
2. Construye un nuevo grafo GT invirtiendo la dirección de todo arco en G.
3. Realiza un DFS sobre GT empezando por el vértice que fue enumerado con el mayor valor de acuerdo a la numeración asignada en el paso (1). Si el DFS no alcanza todos los vértices, empieza el siguiente DFS desde el vértice restante enumerado con el mayor valor.

4. Cada árbol obtenido luego del DFS es una componente fuertemente conexa.

Nota: En lugar de enumerar los vértices, también podemos irlos insertando en una pila (stack).
Los anteriores algoritmos se los puede implementar en el mismo código de DFS ya que solo se necesitan poner algunas variaciónes.
Con los grafos se pueden hacer muchas cosas más que estos algoritmos, asi que se sugiere al lector seguir investigando.

9. Programación Dinámica

9.1 Un poco de introducción

Richard Bellman es el creador de la programación dinámica. La programación dinámica es un método que nos ayuda a reducir el tiempo de ejecución de algunos algoritmos. Parte del denominado principio de optimalidad enunciado por *Bellman*:

"En una sucesión óptima de decisiones u opciones, toda sub-secuencia de decisiones debe ser también óptima".

La programación dinámica es un tópico muy recurrente en los concursos de programación. Aprender programación dinámica nos dará una gran ventaja sobre otros competidores, es mas te aseguro que en algunos casos nos podría asegurar una victoria, al poder resolver un problema que requiere este enfoque y que es muy probable que los demás no podrán resolverlo.

En un principio te puede parecer un poco difícil de entender, pero te recomiendo que le des unas 2 lecturas a esta sección para entenderla y que pruebes los códigos que se exponen y te convenzas de que funcionan y dan respuestas correctas.

Para abordar y solucionar problemas con esta técnica es necesario contar con una habilidad:

1. Poder detectar una relación de recurrencia más algunas condiciones base. Para calcular valores en función de valores más pequeños, apelando al principio de optimalidad. Hasta utilizar las condiciones base establecidas explícitamente.

La única forma de obtener esta habilidad es con la práctica, haciendo un montón de problemas que requieren la utilización de esta técnica. Una forma de ir adquiriendo esta habilidad, es iniciar resolviendo problemas clásicos de programación dinámica, los cuales veremos luego.

El resto de esta introducción es una traducción al español(con algunas adaptaciones mías) de [2], el cual es un sitio famoso en internet donde un usuario hace una pregunta y otros usuarios pueden responder, es un lugar parecido a "Yahoo Answers", pero *Quora.com* es un sitio mas formal, en especial la traducción que presentamos a continuación es la respuesta de *Michal Danilák* mas conocido como el usuario *"Mimino"* en sitios como *TopCoder.com* y *Codeforces.com*, *Mimino* es un competidor muy reconocido en el mundo de las competencias de programación.

He decidido poner esto aquí ya que explica de una manera muy clara y didáctica el proceso que debe seguir uno, al resolver problemas usando programación dinámica. La primera vez que lo leí me sorprendió la claridad y facilidad con la que lo explica.

Y gracias a que las respuestas que los usuarios publican en *Quora.com* vienen con la licencia *Creative Commons* podemos darnos el lujo de replicarla aquí. Así que manos a la obra.

La programación dinámica es un tema muy específico en los concursos de programación. No importa cuántos problemas haya usted solucionado usando programación dinámica, aun puede sorprenderse

con nuevos problemas. Pero como todo en la vida, la práctica hace al maestro.
El los siguientes párrafos tratare de explicar como llegar a la solución de un problema utilizando programación dinámica.

9.1.1 Iteración vs Recursión

Después de leer algunos textos de introducción a la programación dinámica (que recomiendo encarecidamente), casi todos los ejemplos de código fuente en ellos usan la técnica de bottom-up con iteración (es decir, utilizando ciclos). Por ejemplo el cálculo de la longitud de la sub-secuencia común más larga (LCS) de dos cadenas A y B de longitud N, se vería así:

Código 9.1: Código LCS

```
int dp[N+1][N+1];
for (int i = 0; i <= N; ++i)
    dp[0][i] = dp[i][0] = 0;
for (int i = 1; i <= N; ++i)
    for (int j = 1; j <= N; ++j) {
        dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
        if (A[i-1] == B[j-1])
            dp[i][j] = max(dp[i][j], dp[i-1][j-1]+1);
    }

int answer = dp[N][N];
```

Hay un par de razones por las que se codifica de esta manera:

1. La iteración es mucho más rápida que la recursión.
2. Uno puede ver fácilmente la complejidad del algoritmo.
3. El código es más corto y limpio.

En cuanto a dicho código, uno puede entender cómo y por qué funciona, pero es mucho más difícil llegar a él.
El mayor avance en el aprendizaje de programación dinámica es, cuando uno se pone a pensar en la solución de los problemas en la forma top-down usando recursión, en lugar de bottom-up usando iteración.
En primer aspecto no se ve como una idea tan revolucionaria, pero estos dos enfoques se traduce directamente en dos códigos fuentes diferentes. Uno utiliza iteración (bottom-up) y el otro utiliza recursividad (top-down). Este último también se llama la técnica de memoización. Las dos soluciones son más o menos equivalentes y siempre se puede transformar una en la otra.
En los siguientes párrafos se va a mostrar cómo llegar a una solución de un problema utilizando la técnica de memoización.

9.1.2 Problema de Motivación

Imagine que tiene una colección de N vinos colocados uno junto al otro en un estante. Para simplificar, vamos a numerar los vinos de izquierda a derecha, ya que están de pie en la plataforma con números enteros de 1 a N, respectivamente. El precio del i-ésimo vino es p_i (los precios de diferentes vinos pueden ser diferentes).
Debido a que los vinos mejoran cada año, suponiendo que hoy es el año 1, en el año "y" el precio del vino $i-esimo$ vino será $y * p_i$, es decir y-veces más cada año que el año en curso.

Quieres vender todos los vinos que tienes, pero quieres vender exactamente un vino al año, a partir de este año. Una limitación será que cada año se te permite vender sólo el vino más a la izquierda o el vino más a la derecha en el estante y no se te permite reordenar los vinos(es decir, deben permanecer en el mismo orden en el que están al inicio).
Usted quiere saber, ¿Cuaĺ es el máximo beneficio que puede obtener, si vende los vinos en orden óptimo?
Así por ejemplo, si los precios de los vinos son (en el orden en que se colocan en el estante, de izquierda a derecha):

$$p_1 = 1, p_2 = 4, p_3 = 2, p_4 = 3$$

La solución óptima sería vender los vinos en el orden

$$p_1, p_4, p_3, p_2$$

para un beneficio total de

$$1*1+3*2+2*3+4*4 = 29$$

9.1.3 Solución Equivocada

Después de jugar con el problema por un tiempo, es probable que se tenga la sensación, de que la solución óptima es vender los vinos caros lo más tarde posible. Entonces usted probablemente llegara a la siguiente estrategia greddy(estrategia golosa):
Todos los años, vendes el más barato de los dos vinos disponibles (el de más a la izquierda o el de más a la derecha).
Aunque la estrategia no menciona qué hacer cuando los dos vinos cuestan lo mismo, uno siente que esta estrategia es correcta. Pero, por desgracia, no lo es, como demuestra el siguiente ejemplo.
Si los precios de los vinos son:

$$p_1 = 2, p_2 = 3, p_3 = 5, p_4 = 1, p_5 = 4$$

La estrategia greedy sería vender en el orden

$$p_1, p_2, p_5, p_4, p_3$$

para un beneficio total de

$$2*1+3*2+4*3+1*4+5*5 = 49$$

Pero podemos obtener mas si vendemos los vinos en el orden

$$p_1, p_5, p_4, p_2, p_3$$

para un beneficio total de

$$2*1+4*2+1*3+3*4+5*5 = 50$$

Este contraejemplo debe convencerte, de que el problema no es tan fácil como puede parecer a primera vista, vamos a resolverlo utilizando programación dinámica.

9.1.4 Escribir un Backtrack

Para una solución con memoización para un problema, siempre se empieza escribiendo un backtrack que siempre encuentra la respuesta correcta. La solución backtrack enumera todas las respuestas válidas para el problema y elige la mejor. Para la mayoría de los problemas, es fácil llegar a dicha solución.

Pero para poder utilizar posteriormente nuestro backtrack en nuestra solución final de programación dinámica, debemos tener en cuenta las siguientes restricciones:

- Debería ser una función, que calcule la respuesta utilizando recursividad.
- Debe devolver la respuesta con una instrucción de retorno, es decir no almacenar en algún otro lugar.
- Todas las variables no locales que se usan en la función deben ser solo de lectura, es decir la función solo puede modificar las variables locales y sus parámetros.

Así que para el problema de los vinos, la solución con backtrack se verá así:

Código 9.2: Código Problema de los vinos(Backtrack)

```
1  int N;//Numero de vinos
2  int p[MAXN];//Array de solo lectura, con los precios de los vinos
3
4  int ganancia(int anio, int izq, int der) {
5      if (izq > der)//No hay mas vinos en el estante
6          return 0;
7      //Vender el vino de la izquierda o de la derecha
8      int opcion1 = p[izq] * anio + ganancia(anio + 1, izq + 1, der);
9      int opcion2 = p[der] * anio + ganancia(anio + 1, izq, der - 1);
10     return max(opcion1, opcion2);
11 }
```

Podemos obtener la respuesta llamando a:

```
1  int solucion = ganancia(1, 1, N); // N es el número total de vinos
```

Esta solución simplemente trata de todos los posibles órdenes válidas de la venta de vinos. Si hay N vinos al principio, intentará 2^N posibilidades(cada año tenemos 2 opciones). Así que, aunque ahora tenemos la respuesta correcta, la complejidad del tiempo de ejecución de nuestra solución crece exponencialmente.

Una función backtrack escrita correctamente siempre representa una respuesta a una pregunta bien planteada.

En nuestro caso la función de ganancia representa una respuesta a la pregunta:

¿Cuál es el mejor beneficio que podemos obtener de la venta de los vinos con precios almacenados en el vector p, cuando la variable *anio* es el año en curso y el intervalo de los vinos sin vender se extiende a través de [izq, der] (incluidos los vinos en las posiciones *izq* y *der*)?".

Usted siempre debe tratar de crear una pregunta de este tipo para su función de backtrack para ver si lo ha hecho bien y entidende exactamente lo que hace la función.

9.1.5 Reducir la cantidad de parámetros de la función

En este paso quiero que pienses, que algunos parámetros que se pasa a la función son redundantes. O bien podemos construirlos a partir de otros parámetros o que no los necesitamos en absoluto.
En la función ganancia, el parámetro *anio* es redundante. Es equivalente a la cantidad de vinos que ya hemos vendido más uno. Si creamos una variable global de sólo lectura N, que representa el número total de vinos al principio, podemos reescribir nuestra función como sigue:

Código 9.3: Código Problema de los vinos(Reducción de estados)

```
int N;
int p[MAXN];

int ganancia(int izq, int der) {
    if (izq > der)
         return 0;
    //(der - izq + 1) es el numero de vinos no vendidos
    int anio = N - (der - izq + 1) + 1;//Total - No vendidos + 1
    int opcion1 = p[izq] * anio + ganancia(izq + 1, der);
    int opcion2 = p[der] * anio + ganancia(izq, der - 1);
    return max(opcion1, opcion2);
}

//...  y la llamamos así:
int solucion = ganancia(1, N);
```

También quiero que pienses en el rango de posibles valores de los parámetros de la función que serian parámetros validos. En nuestro caso, cada uno de los parámetros "izq" y "der" pueden tomar valores entre 1 y N. En las entradas válidas también esperamos que $izq <= der$. Utilizando la notación O-grande podemos decir que hay $O(n^2)$ diferentes parámetros para nuestra función, con las cuales puede ser llamada.

9.1.6 Ahora cachéalo

De momento tenemos hecho el 99% de la solución. Ahora para transformar la función de backtrack con una complejidad de tiempo $O(2^N)$, a una solución de memoización con complejidad de tiempo $O(n^2)$ vamos a utilizar un pequeño truco que no requiere pensar mucho.
Como se señaló anteriormente, existen $O(n^2)$ diferentes parámetros posibles para nuestra función con las cuales podría ser llamada. En otras palabras, sólo hay $O(n^2)$ estados diferentes que en realidad podemos calcular. Entonces, ¿De dónde viene la complejidad $O(2^N)$ y como se procesa?.
La respuesta es:
La complejidad de tiempo exponencial viene de la recursividad repetida y debido a eso, calcula los mismos valores una y otra vez. Si ejecuta el código anterior para un vector arbitrario de N vinos ($N = 20$) y calcula cuántas veces la función fue llamada con los parámetros [izq, der](izq = 10 y der = 10) obtendrá el número 92378. Esa es una enorme pérdida de tiempo para calcular la misma respuesta y responder muchas veces. Lo que podemos hacer para mejorar esto es almacenar los valores una vez que los calculamos y cada vez que la función pregunta por un valor ya calculado retornamos el valor almacenado, así no necesitamos volver a correr la recursión de nuevo. Ver el código de abajo:

Código 9.4: Código Problema de los vinos(Cacheando el backtrack)

```
int N;
int p[MAXN];
int cache[MAXN][MAXN];//Todos los valores inicializados en "-1"
                      //o en valor que indique
                      //que aun no esta calculado
int ganancia(int izq, int der) {
    if (izq > der)
        return 0;
    //Estas 2 lineas salvan el dia
    if (cache[izq][der] != -1)//¿Ya fue calculado?
        return cache[izq][der];

    int anio = N - (der - izq + 1) + 1;
    int opcion1 = p[izq] * anio + ganancia(izq + 1, der);
    int opcion2 = p[der] * anio + ganancia(izq, der - 1);
    //Lo guardamos mientras retornamos
    return cache[izq][der] = max(opcion1, opcion2);
}
```

Y eso es todo. Con ese pequeño truco el código funciona en tiempo $O(n^2)$, porque hay $O(n^2)$ posibles parámetros en nuestra función, que puede ser llamada varias veces y para cada uno de ellos, la función se ejecuta sólo una vez con O(1) de complejidad de tiempo.

9.2 Problemas clásicos de Programación Dinámica

Si bien la programación dinámica no trata de un algoritmo especifico como lo son los algoritmos de ordenación o hallar caminos cortos en grafos sino mas bien de una técnica de programación, existen problemas específicos con restricciones establecidas que pueden ser resueltas con programación dinámica, las cuales fueron estudiadas por ciertos autores, quienes desarrollaron las soluciones y su respectiva demostración de correctitud. Como ya mencionamos anteriormente el mayor avance a la hora de aprender programación dinámica es cuando uno se pone a pensar en las soluciones en la forma Top-Down la cual implica recursión, estudiar las soluciones de algunos problemas clásicos nos ayudara a obtener una visión clara de la forma que deberían tener nuestras recursiones para poder aplicar programación dinámica, haciendo énfasis en la forma, tipo y cantidad de parámetros que utilizan estas.

9.2.1 La subsecuencia creciente mas larga

9.2.1.1 Problema

En ingles "*Longest increasing subsequence(LIS)*", el problema se describe a continuación:
"Dada una sucesión de N números, determinar la longitud de la subsecuencia mas larga, de modo que todos los elementos de la subsecuencia se encuentran en orden creciente"
Pero, ¿Por que solo nos piden el tamaño y no la subsecuencia creciente mas larga?, esto se debe a que podríamos obtener varias subsecuencias crecientes que cumplen la condición de tener el tamaño de la subsecuencia creciente mas larga, pero el tamaño de la subsecuencia creciente mas larga es único. Por ejemplo, dada la siguiente sucesión de números:

$$0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15$$

Una subsecuencia creciente mas larga posible que podemos obtener es:

$$0,4,6,9,11,15$$

y otra:

$$0,4,6,9,13,15$$

Entonces la solución sera 6, por ser el tamaño máximo que podemos obtener.

9.2.1.2 Análisis

Para resolver este problema pensemos en el ultimo elemento de una subsecuencia creciente, imaginemos que este ultimo elemento esta en la posición i, si i no es el ultimo elemento de la secuencia original quiere decir que existen elementos a la derecha de i.

Que pasaría si alguno de esos elementos es mayor que el elemento en la posición i?

Supongo que ya te diste cuenta, podemos adicionar este elemento al final de nuestra subsecuencia e incrementar su tamaño en 1.
Utilicemos esta observación para construir nuestra solución.
Si nos ponemos a pensar en la solución final(o por lo menos en una de ellas) notamos que nuestra subsecuencia podría terminar en cualquier posición de la secuencia original, llamemos a esta posición i y al tamaño de la subsecuencia creciente mas larga que termina en esa posición M,entonces de acuerdo con la observación que hicimos antes, debe existir una subsecuencia de tamaño $M-1$ a la izquierda de i donde el ultimo elemento de esta subsecuencia mas pequeña es menor que el elemento en la posición i.
Aquí es donde podemos notar con mas claridad el hecho de que el resultado no es único, si existieran varias subsecuencias crecientes con tamaño $M-1$ a la izquierda de i donde su ultimo elemento es menor que el elemento en la posición i, cada una formara una subsecuencia creciente que cumple con la condición de ser la mas larga.
Como habrá notado este mismo análisis se aplica a la subsecuencia de tamaño $M-1$, donde podríamos encontrar subsecuencias de tamaño $M-2$ a la izquierda de su ultimo elemento, y así hasta que solo tengamos un solo elemento. Entonces podemos plantear una pregunta a la cual deberá responder nuestra función recursiva:

¿Cual es el tamaño de la subsecuencia creciente mas larga que termina en la posición "i"?

Podemos notar que nuestra función solo recibirá un número como parámetro de entrada que representa a una posición y deberá retornar un numero que representa la respuesta a la pregunta. Pero esta no es la solución final ya que dijimos que la solución podría terminar en cualquier posición de la secuencia original, debemos hacer un ciclo preguntando sobre la subsecuencia creciente mas larga que termina en la posición actual y tomar la mas grande de todas ellas. Con todo esto hecho ya estamos en condiciones de entender y analizar el código que soluciona el problema, y tal vez posteriormente utilizar la misma idea para poder solucionar problemas parecidos o del mismo tipo.

9.2.1.3 Código

Código 9.5: Código de la subsecuencia creciente mas larga

```
int N; //El numero de elementos de la secuencia original
int v[1000]; //Elementos de la secuencia original
               //indexadas dede la posición 0
int dp[1000]; //Array donde cachearemos las llamadas a la función
int f(int i) {
    if (dp[i] != -1) return dp[i];
    //El elemento i cuenta como una subsecuencia de un solo elemento
    int sol = 1;
    for (int j = 0; j < i; j++) //Recorremos menores que "i"
        if (v[j] < v[i])
            sol = max (sol , 1 + f(j)) ;
    return dp[i] = sol ;
}
int LIS() {
    memset (dp , -1, sizeof(dp)) ; //Marcamos como no calculados
    int sol = 0; //Si N = 0 no hay elementos
    for (int i = 0; i < N; i++)
        sol = max(sol, f(i)) ;
    return sol ;
}
```

Pasemos a entender un poco el código, tenemos 2 funciones, la primera es "$f(i)$" que es la función que mencionamos en el análisis(la función que responde a la pregunta) y la otra función "$LIS()$" es la que se encarga de utilizar la primera función para solucionar el problema. Respecto a la primera función lo que hacemos es recorrer con un ciclo en "j" todos las posiciones que son menores que "i" y verificamos que el elemento en $v[j]$ es menor que $v[i]$, si es así entonces a la subsecuencia mas grande que termina en la posición "j" le podemos adicionar el elemento en la posición "i", es por eso el "$f(j)+1$", simplemente tomamos el mas grande de todos y lo retornamos.
No debemos olvidar cachear la respuesta de la llamada a la función, para no tener que calcular respuestas otra vez cuando ingresen los mismos parámetros, eso lo hacemos en el vector "dp". En cuanto a la segunda función, lo primero que hace es setear todos los elementos del vector "dp" en -1 para indicar que aun no fueron calculadas cuando se llame a la función f , y un ciclo el cual prueba con todos las posiciones que podrían ser el final de la subsecuencia creciente mas grande, llama a f con todas esas posiciones y toma el máximo de todos.

9.2.2 El problema de la mochila

9.2.2.1 Problema

En ingles "*Knapsack problem(KP)*", a continuación se describe el enunciado del problema:

Dados N items, cada uno con un valor(o ganancia) y un peso. Y una mochila con una capacidad de carga de W. Determinar el máximo valor que se puede obtener del proceso de llenar la mochila con algunos de los items de modo que la suma de pesos de esos items no exceda W.

Del mismo modo que en el caso de La subsecuencia creciente mas larga, ¿Por que nos piden solamente el máximo valor y no los items?, pues la respuesta es la misma, por que podrían existir

varias respuestas posibles correctas, por ejemplo un caso en el que podría ocurrir esto es cuando la suma de los valores y pesos respectivamente de algunos items que aun no están en la mochila suman lo mismo en su peso y valor que un solo objeto que también aun no esta en la mochila, podríamos meter cualquiera de los 2 en la mochila y obtener repuestas correctas con ambas.

9.2.2.2 Análisis

Primeramente para facilitar el análisis del problema definiremos un entero N que representa la cantidad de items disponibles, otro entero *CAPACIDAD* que representa la capacidad de la mochila y 2 vectores de enteros, $valor[i]$ y $peso[i]$ que representan el valor y peso respectivamente del i-esimo ítem. También por comodidad guardemos nuestros datos en estos vectores en las posiciones 1 a N, dejando vacía las posiciones 0 de ambos vectores.
En un principio uno podría ponerse a pensar en una solución greedy(golosa) en la que seleccionamos primero aquellos items que tienen mayor valor primero y nos detenemos cuando ya no quede espacio suficiente para meter otro objeto. Pero después de probar algunos casos podríamos darnos cuenta que no nos da una respuesta correcta.
La solución a este problema es un poco parecida al anterior problema, imaginemos que tenemos una solución con los primeros $N-1$ items, algunos items ya están en la mochila y algunos no.

¿Que sucede si aun queda espacio suficiente para meter el N-esimo item?

Pues podríamos meter el N-esimo item en la mochila y tener una mejor solución sumándole su valor al que ya teníamos con los $N-1$ primeros items.
Pero si no quedase espacio en la mochila para meter el N-esimo item nos quedamos con la duda de:

¿Que sucedería si no hubiese tomado algunos items, tal que quede espacio suficiente para meter el N-esimo item?, ¿No obtendríamos mejor respuesta?

Es posible, como también es posible que no, así que debemos probar con los 2 casos, si metiésemos el N-esimo item siempre y cuando haya suficiente espacio para él, y si no lo metiésemos dejando espacio tal vez para posibles respuestas mejores con otros items.
Esto nos hace notar que todo el tiempo debemos saber cuanto espacio disponible queda en la mochila y con que elementos estamos trabajando.
Así que nuestra solución contara con 2 parámetros, uno que nos dirá con que elementos estamos trabajando y otro que nos avise cuanto espacio libre queda en la mochila.
Entonces la pregunta que nos podemos plantear sera:

¿Cual es el máximo beneficio que se puede obtener con los primeros "i" items, con "libre" espacio libre en la mochila?

Vamos a ello.

9.2.2.3 Código

Código 9.6: Código de El problema de la mochila

```
1 #define MAXN 1010
2 int N, CAPACIDAD;
3 int peso[MAXN], valor[MAXN];
```

```
4  int dp[MAXN][MAXN];//Aqui es donde cachearemos las respuestas
5  int mochila(int i, int libre) {
6      if (libre < 0) return -100000000;//Metimos un objeto demasiado pesado
7      if (i == 0) return 0;//Si ya no hay objetos, ya no ganamos nada
8      if (dp[i][libre] != -1) return dp[i][libre]; //El DP
9      //Si no tomariamos el item
10     int opcion1 = mochila(i - 1, libre);
11     //Si tomariamos el item
12     int opcion2 = valor[i] + mochila(i - 1, libre - peso[i]);
13     //Devolvemos el máximo (y lo cacheamos)
14     return (dp[i][libre] = max(opcion1, opcion2));
15 }
16
17 //...  y lo llamamos así
18
19 memset(dp, -1, sizeof(dp));//Marcamos como no calculados
20 int solucion = mochila(N, CAPACIDAD);
```

Analizando un poco el código, lo primero que hacemos es revisar si no metimos un objeto demasiado pesado en la mochila tal que hizo que el espacio libre que tenemos sea negativo, de ser así retornamos un numero negativo muy grande para indicar que perdemos en vez de ganar, luego verificamos si i es igual a 0 de ser así significa que ya no hay mas objetos disponibles(recuerda que indexamos nuestros vectores desde 1), retornamos 0 por que ya no podemos ganar nada mas. Enseguida verificamos si la respuesta para esos parámetros no fueron calculados ya y retornamos de ser así. En la opcion1 guardamos el resultado que obtendríamos si no metiésemos el item actual, el parámetro *libre* no varia, en la opcion2 probamos la opción de si tomar el item actual, tomamos la ganancia que nos da el item sumándonos $valor[i]$ a la opción y llamamos recursivamente al resto de items que quedan por procesar pero en este caso debemos restarle al parámetro libre el peso del item que acabamos de tomar.
Finalmente devolvemos el máximo de las 2 opciones disponibles, sin olvidar cachear la respuesta para posibles futuras llamadas.

Bibliography

[1] Mario Ynocente. Componentes Conexas. *https://sites.google.com/site/ycmario/home/*.

[2] Quora.com *http://www.quora.com/Are-there-any-good-resources-or-tutorials-for-dynamic-programming-besides-the-TopCoder-tutorial*.

[3] Oswualdo Alquisiris Quecha. Puentes y puntos de articulación. *https://lamdiscreta.wordpress.com/2011/11/22/equipo-1/*.

[4] http://es.wikipedia.org/

[5] Agustin Gutierrez, *Programacion Dinamica*, Buenos Aires, 2014

[6] http://es.wikibooks.org/wiki/Programacion_en_C++

[7] TopCoder.com. *http://help.topcoder.com/data-science/competing-in-algorithm-challenges/algorithm-tutorials/*

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein *Introduction to Algorithms, third edition*

Lista de Códigos