Notas de curso
Algoritmos y Estructuras de Datos

Rodrigo Alexander Castro Campos
UAM Azcapotzalco, División de CBI
`https://racc.mx`
`https://omegaup.com/profile/rcc`

Última revisión: 19 de octubre de 2023

1. Algoritmia y eficiencia

Como la computadora es la evolución natural de la calculadora, es natural pensar que todo ingenierio debe aprender a programar en algún momento durante su formación profesional. Sin embargo, lo que diferencia a un ingenierio genérico de un ingenierio en computación o electrónica es que estos últimos deben aprender a usar la computadora de forma *eficiente*. En este curso, nuestro recurso más valioso será el *tiempo* y es desafortunado (pero muy común) encontrarnos con cálculos para los que el algoritmo más sencillo de programar tardaría horas en obtener la respuesta, mientras que un algoritmo más ingenioso ¡tardaría menos de un segundo! En este curso aprenderemos formas eficientes de realizar millones de consultas sobre colecciones de millones de datos almacenados en la memoria de la computadora.

Para aprender a idear formas eficientes de realizar algún cálculo, necesitamos desarrollar la intuición de si un algoritmo será eficiente o ineficiente sin tener que programarlo. Y para lograr lo anterior, necesitamos desarrollar la intuición de cuánto tardará un programa al ejecutarlo en una computadora. A final de cuentas, una computadora es una máquina con limitaciones físicas y no opera a una velocidad infinita.

En una computadora, el procesador es el que se encarga de ejecutar las operaciones de un programa y suele ser el principal determinante del tiempo de ejecución de un programa. Un procesador es un componente electrónico muy complicado, pero un aspecto que es fácil de entender y de comparar es su frecuencia de reloj (por ejemplo, se altamente probable que un procesador que vaya a una frecuencia de 3 GHz sea más rápido que uno que vaya a 2 GHz). La frecuencia de reloj del procesador se puede explicar como sigue: en un reloj discreto de manecillas, cada segundo el segundero se mueve bruscamente para cambiar de posición y el reloj permanecerá en este estado hasta que transcurra otro segundo. Como el reloj cambia de estado cada segundo, se dice que el reloj opera a 1.0 hz (un hertz o un ciclo por segundo). El procesador de una computadora también es una máquina de estados: si el procesador va a 3.0 GHz entonces realiza 3×10^9 ciclos o cambios de estado por segundo.

Nomenclatura	**Nombre**	**Ciclos por segundo**
`1 Hz`	hertz	Un ciclo por segundo (1)
`1 kHz`	kilohertz	Mil ciclos por segundo (10^3)
`1 MHz`	megahertz	Un millón de ciclos por segundo (10^6)
`1 GHz`	gigahertz	Mil millones de ciclos por segundo (10^9)

Los procesadores modernos pueden ejecutar operaciones aritméticas sencillas como sumas o comparaciones en un solo ciclo, aunque otras operaciones más complejas como multiplicaciones o divisiones requieren decenas de ciclos para completarse. Esto dificulta el cálculo exacto del tiempo que tarda un programa, pero casi siempre basta un estimado (por ejemplo, ¿el programa tardará menos de un segundo, varios segundos, minutos u horas?). Para simplificar la discusión, imaginaremos una computadora *ideal* que siempre opera a 1 GHz pero en donde cada operación básica (operación aritmética, comparación y acceso a memoria) tarda un ciclo. Corriendo a 1 GHz, una computadora podrá realizar 10^9 operaciones

básicas por segundo, lo que facilita las estimaciones. Por ejemplo, si un algoritmo ejecuta 10^8 operaciones entonces su ejecución tardará aproximadamente 0.1 de segundo.

Lo anterior es relevante porque casi siempre hay *más de un algoritmo* que resuelve un problema dado. Un ejemplo intuitivo de este hecho es el problema de calcular la sumatoria de n dada por $\sum_{i=1}^{n} i$. Implementando ciegamente la defición de sumatoria, obtenemos el siguiente programa:

> Los códigos de ejemplo que no definan funciones se deben escribir dentro de la función `main` y también se deben incluir las bibliotecas necesarias para poder ejecutar el programa.

Código	Entrada	Salida

`int n;`
`scanf("%d", &n);`
 	
`long long res = 0;`
`for (int i = 1; i <= n; ++i) {`
`  res += i;`
`}`
`printf("%lld\n", res);`
	2000000000	2000000001000000000

La ejecución del programa anterior no será instantánea si se ejecuta con un valor grande de n. Usando el valor de n dado en el ejemplo es prácticamente seguro que la ejecución del programa tarde más de un segundo, ya que el ciclo `for` se ejecutará 2×10^9 veces, que es el doble de la cantidad de instrucciones que una computadora ideal a 1 GHz puede ejecutar en un segundo. Por otra parte, se sabe que $\sum_{i=1}^{n} i = \frac{(n)(n+1)}{2}$ por lo que se puede obtener la misma respuesta usando sólo tres operaciones aritméticas. Esto producirá un programa que tarde menos de una millonésima de segundo, independientemente del valor de n.

Código	Entrada	Salida

`int n;`
`scanf("%d", &n);`
`printf("%lld\n", (n * (n + 1LL)) / 2);`
	2000000000	2000000001000000000

Otro problema similar es el de determinar si un natural n es primo o no. Por definición, un número primo sólo tiene dos divisores distintos: 1 y sí mismo. El 1 no es primo porque sólo tiene un divisor. Si $n > 1$ es primo entonces ningún entero entre 2 y $n-1$ lo divide exactamente. Dado un entero n, podemos determinar si existe o no un divisor en ese rango usando un ciclo `for` con un `if` adentro.

Código	Entrada	Salida

`#include <stdio.h>`
 	
`int es_primo(int n) {`
`  for (int i = 2; i < n; ++i) {`
`  if (n % i == 0) {`
`  return 0;`
`  }`
`  }`
`  return n > 1;`
`}`
 	
`int main( ) {`
`  int n;`
`  scanf("%d", &n);`
`  printf("%d", es_primo(n));`
`  return 0;`
`}`
	2000000011	1

Cuando n es primo, el ciclo `for` del código anterior ejecutará $n-2$ iteraciones. Si n es primo y también es mayor que 10^9, el código tardará más de un segundo en ejecutarse. Una forma ingeniosa de acelerar el cálculo anterior es darnos cuenta que los divisores de un entero n presentan un comportamiento "de espejo" con respecto a $\sqrt{n}$. Para ejemplificar esto, a continuación se muestran los divisores de 36:

División exacta	**Cociente**
`36 / 1`	36
`36 / 2`	18
`36 / 3`	12
`36 / 6`	6
`36 / 12`	3
`36 / 18`	2
`36 / 36`	1

Por este comportamiento "de espejo" podemos afirmar lo siguiente:

- Si d es un divisor de n, entonces $r = n \,/\, d$ también es un divisor de n, ya que $d = n \,/\, r$.
- Si $d \leq \sqrt{n}$ entonces $r \geq \sqrt{n}$.
- Si no existe ningún divisor de n menor a $\sqrt{n}$, tampoco existe ningún divisor de n mayor a $\sqrt{n}$.

Podemos modificar el algoritmo anterior para sólo buscar divisores que sean menores o iguales a $\sqrt{n}$. Debemos incluir $\sqrt{n}$ en el rango a probar para detectar que los cuadrados perfectos no son primos.

Código	**Entrada**	**Salida**

`#include <math.h>`
`#include <stdio.h>`
 	
`int es_primo(int n) {`
`  int tope = round(sqrt(n));`
`  for (int i = 2; i <= tope; ++i) {`
`  if (n % i == 0) {`
`  return 0;`
`  }`
`  }`
`  return n > 1;`
`}`
 	
`int main( ) {`
`  int n;`
`  scanf("%d", &n);`
`  printf("%d", es_primo(n));`
`  return 0;`
`}`
	`2000000011`	`1`

¿Qué tan rápido es este algoritmo en comparación con el anterior? Para el primo $n = 2000000011$, el primer algoritmo ejecutará $2000000011 \approx 2 \times 10^9$ iteraciones del `for`, por lo que tardará aproximadamente dos segundos. El segundo algoritmo ejecutará $\sqrt{2}000000011 \approx 4.5 \times 10^4$ iteraciones del `for`, por lo que tardará aproximadamente 5 cienmilésimas de segundo. ¡Una gran diferencia!

1.1. Ejercicios

El juez en línea omegaUp es una plataforma donde podemos enviar nuestro código para intentar resolver alguno de los problemas de programación disponibles. A partir de ahora, los ejercicios consistirán en problemas disponibles en esta plataforma. Puedes consultar un video sobre cómo usar omegaUp aquí.

1. Resuelve el problema `https://omegaup.com/arena/problem/Calculo-de-sumatoria`.
2. Resuelve el problema `https://omegaup.com/arena/problem/Suma-de-intervalos`.
3. Resuelve el problema `https://omegaup.com/arena/problem/Sumando-todos-los-subarreglos`.

2. Introducción al lenguaje C++ para programadores de C

El lenguaje de programación C++ es un lenguaje eficiente de alto nivel diseñado por Bjarne Stroustrup en 1979. El lenguaje C++ fue estandarizado por primera vez ante la Organización Internacional de Normalización (ISO) en 1989 y su versión más reciente fue publicada en 2020. Se usa ampliamente en la implementación de sistemas operativos, videojuegos y sistemas embedidos.

Los compiladores más famosos de C++ son GCC, Visual Studio y Clang. En Windows lo más común es usar Visual Studio o MinGW que es una variante de GCC, mientras que en MacOS se suele usar Clang y en Linux se suele usar GCC. Como Windows no trae preinstalado ningún compilador, se sugiere descargar la versión más reciente de Visual Studio o de MinGW. En Linux lo normal es que GCC ya esté preinstalado. En MacOS, la recomendación de Apple es instalar Xcode que trae Clang.

El lenguaje C++ es casi un superconjunto del lenguaje C. Sin entrar en demasiados detalles, esto significa que prácticamente todo código válido en C también compila en C++. Esta amplia compatibilidad permite usar en C++ las funciones de `<stdio.h>` y del resto de la biblioteca de C.

Código	Salida

`#include <stdio.h>  // Válido en C++`
 	
`int main( ) {`
`  printf("Hola mundo");`
`  return 0;`
`}`
	`Hola mundo`

Sin embargo y a pesar de su flexibilidad, las funciones `scanf` y `printf` tienen inconvenientes serios como el tener que elegir el especificador de formato `%` correcto para cada tipo de dato, o que olvidar el `&` en `scanf` al leer una variable no provoca un error de compilación, pero sí uno de ejecución.

Código	Diagnóstico

`int n;`
`scanf("%d", n);  // error en ejecución`
`printf("%f", n); // formato incorrecto`
	`Error en ejecución`

La entrada y salida al estilo C++ se realiza con las utilidades declaradas en el archivo de biblioteca `<iostream>` (que a diferencia de la biblioteca de C, no lleva extensión `.h`). Las variables `std::cin` y `std::cout` de `<iostream>` representan la entrada y salida de datos, respectivamente. El origen y el destino de los datos son los mismos que con `scanf` y `printf`, por lo que es fácil entender su semántica. Estas variables emplean una fea notación con los operadores `>>` y `<<` para realizar la lectura y la escritura respectivamente. Una forma fácil de aprender el sentido de los operadores es: si los datos van de la entrada (`std::cin`) a la variable, entonces el operador es `>>`, mientras que si los datos van de la variable a la salida (`std::cout`), entonces el operador es `<<`. Por ejemplo:

Código	Entrada	Salida

`#include <iostream>`
 	
`int main( ) {`
`  int a, b;`
`  std::cin >> a;  // leer a`
`  std::cin >> b;  // leer b`
`  std::cout << a + b;  // imprimir`
`  return 0;`
`}`
	`1 2`	`3`

Podemos leer e imprimir más de un valor en la misma línea.

Código	Entrada	Salida

`#include <iostream>`
 	
`int main( ) {`
`  int a, b;`
`  std::cin >> a >> b;`
`  std::cout << a + b << " " << a - b;`
`  return 0;`
`}`
	8 3	11 5

En el programa anterior, hay dos formas de reducir la cantidad de código a escribir. La primera es aplicar una regla que dice que si no regresamos explícitamente un valor en `main`, entonces se regresa `0` automáticamente (es importante recordar que esta regla sólo aplica para `main` y no para otras funciones que también regresen enteros). La segunda es usar la sentencia `using namespace std;` que le permite al compilador encontrar las utilidades de la biblioteca de C++ sin teclear `std::`. En este caso, `std` es lo que se denomina el espacio de nombres estándar de C++.

Código	Entrada	Salida

`#include <iostream>`
`using namespace std;`
 	
`int main( ) {`
`  int a, b;`
`  cin >> a >> b;`
`  cout << a + b << " " << a - b;`
`}`
	8 3	11 5

Los espacios de nombres buscan evitar *colisiones de nombres*: la ambigüedad que ocurriría si un usuario declara una función que actualmente no existe, pero una versión futura de la biblioteca la introduce con el mismo nombre. Todas las utilidades que son exclusivas de la biblioteca de C++ se declaran en `std`. Estas notas usarán `std::` cuando convenga hacer explícito que una utilidad es de la biblioteca de C++.

Una gran ventaja de usar `std::cin` y `std::cout` es que funcionan de manera uniforme con los tipos primitivos y con muchos otros tipos de la biblioteca. Por omisión, el operador `>>` siempre se salta espacios.

Código	Entrada	Salida

`int n; char c; float f;`
`std::cin >> n >> c >> f;`
`std::cout << f << " " << c << " " << n;`
	1 @ 3.14	3.14 @ 1

La variable `std::endl`, que también está disponible en `<iostream>`, provee una forma alternativa de imprimir un salto de línea en C++:

Código	Salida

`std::cout << "hola\n";`
`std::cout << "cómo estás" << endl;`
`std::cout << "adiós";`
	hola
	cómo estás
	adiós

Sin embargo, usar `std::endl` afecta gravemente el rendimiento del programa. Aunque no lo parezca, enviar datos a la salida es un proceso muy costoso que involucra la intervención del sistema operativo. Usar `std::endl` ordena que la salida le sea transferida al sistema operativo en ese momento (lo cual es muy tardado), mientras que la biblioteca hubiera preferido esperar a tener más datos qué imprimir o a que en verdad fuera urgente su despliegue. En este curso, el uso de `std::endl` está totalmente desaconsejado.

Las variables `std::cin` y `std::cout` son un poco más lentas que `scanf` y `printf` en la mayoría de los compiladores. Sin embargo, es válido usar ambos en el mismo programa. Si deseamos acelerar `std::cin` y `std::cout` a expensas de ya no poder usar `scanf` y `printf` correctamente, podemos hacer lo siguiente:

Código	Salida

`std::ios_base::sync_with_stdio(false);`
`std::cout << "más rápido\n";`
`printf("mmm...\n");  // desincronizado`
`std::cout << "más rápido\n";`
	`más rápido`
	`más rápido`
	`mmm...`

Otra característica útil del lenguaje C++ es el tipo de dato `bool`, el cual permite al programador volver explícita su intención de manejar los valores de verdad *verdadero* (`true`) o *falso* (`false`). En C esto suele simularse usando el tipo `int` con los valores `1` y `0` respectivamente. Cuando los valores booleanos se usan en expresiones numéricas o cuando éstos se imprimen, `true` se interpreta como 1 y `false` como 0.

Código	Salida

`#include <iostream>`
 	
`bool es_par(int n) {`
`  return n % 2 == 0;`
`}`
 	
`int main( ) {`
`  bool a = true, b = false;`
`  bool c = es_par(5);`
`  std::cout << a << b << c;`
`}`
	`100`

Adicionalmente, se puede declarar una variable de un tipo estructura sin usar la palabra `struct` en la declaración de la variable. Esto hace que el código sea más fácil de entender para un principiante. A su vez, la palabra `auto` permite inferir el tipo de una variable declarada con un inicializador.

Código

```
struct fecha {
   int dia, mes, anyo;
};

fecha f = { 1, 5, 2000 };   // ok en C++, error en C
auto t = f;                 // fecha t = f;
```

3. La pila del proceso, paso por valor y por referencia

Por omisión, cada llamada a función tiene sus propias variables. Cuando una función llama a otra, la función llamante transfiere una copia de los valores usados en la llamada y la función llamada captura estos valores en sus propias variables. A este mecanismo se le conoce como paso por valor. En el siguiente ejemplo, la variable `x` de `f` es independiente de la variable `x` de `main`:

Código	Salida

`#include <iostream>`
 	
`void f(int x) {`
`  x += 2;`
`  std::cout << "x f:" << x << "\n";`
`}`
 	
`int main( ) {`
`  int x = 5;`
`  std::cout << "x main:" << x << "\n";`
`  f(x);`
`  std::cout << "x main:" << x << "\n";`
`}`
	`x main: 5`
	`x f: 7`
	`x main: 5`

Durante la ejecución del programa, cada llamada a función tiene asociado un *marco de ejecución*, el cual contiene las variables propias de la llamada. Una instrucción que modifique la variable de un marco no modificará las variables de otros marcos. A continuación se muestran gráficamente los marcos de `main` y de `f` así como sus respectivas variables.

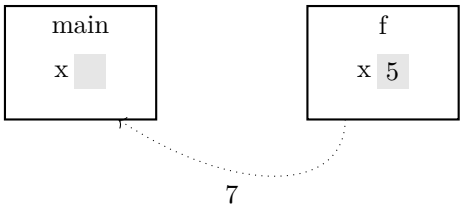
Los marcos de `main` y de `f` inmediatamente después de la llamada a función.

Los marcos de `main` y de `f` después de ejecutar la instrucción `x += 2;` en la función `f`.

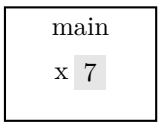
De forma similar, cuando una función regresa un valor, ésta transfiere una copia de éste a la función que la llamó. En cuanto una función termina, su marco desaparece y se destruyen todas sus variables.

Código	**Salida**

`#include <iostream>`
 	
`int f( ) {`
`  int x = 5;`
`  return x + 2;`
`}`
 	
`int main( ) {`
`  int x = f( );`
`  std::cout << x;`
`}`
	7



Los marcos de `main` y de `f` durante el retorno de valor.



El marco de `main` después del retorno de valor. El marco de `f` se destruye.

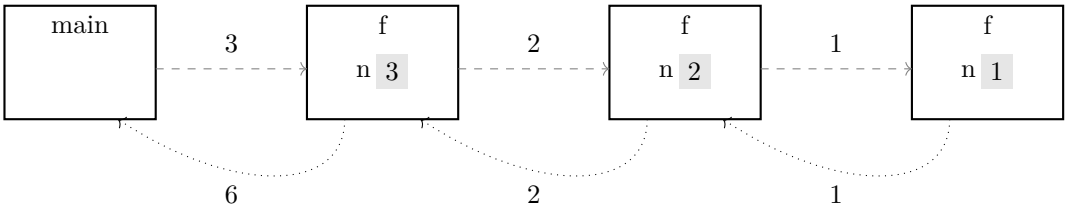
Durante la ejecución de un programa, existirán tantos marcos como llamadas activas a función haya. Como ya se mencionó, cada marco se liberará automáticamente cuando la llamada a función termine.

También se dice que las variables locales se almacenan en memoria automática, justamente porque los marcos donde están almacenadas se crean y se destruyen automáticamente. A la región de memoria que almacena los marcos de las llamadas activas de un programa se le denomina la pila del proceso.

Es importante resaltar que *existe un marco por llamada, no un marco por función*. Llamadas anidadas a la misma función (también denominadas llamadas recursivas) producirán marcos independientes:

Código	Salida

`#include <iostream>`
 	
`int f(int n) {`
`  if (n <= 1) {`
`  return 1;`
`  } else {`
`  return n * f(n - 1);`
`  }`
`}`
 	
`int main( ) {`
`  std::cout << f(3);`
`}`
	`6`



Los marcos de `main` y de las llamadas recursivas a la función `f`.

El lenguaje C++ proporciona un mecanismo que permite asignarle más de un nombre a una misma variable. A esto se le conoce formalmente como referencia. Una referencia se declara como una variable normal, excepto que se debe escribir un `&` después del tipo de la variable. Por ejemplo:

Código	Salida

`int a = 0;`
`int& b = a;  // b es un alias de a`
`b = 7;`
`std::cout << a;`
	`7`

Las referencias también se pueden usar para hacer que un nombre local a un marco en realidad se refiera a una variable de otro marco. Esto permite modificar las variables de un marco desde otro.

Código	Salida

`#include <iostream>`
 	
`void f(int& a) {`
`  a += 1;`
`}`
 	
`int main( ) {`
`  int n = 0;`
`  f(n);`
`  std::cout << n;`
`}`
	`1`

Los marcos de `main` y de `f` al momento de llamar a `f`.
El nombre `a` en la función `f` en realidad se refiere a la `n` de `main`.

Los marcos de `main` y de `f` después de ejecutar la instrucción `a += 1;` en la función `f`.

La característica anterior permite implementar el algoritmo de intercambio de dos variables fácilmente.

Código	Salida

`#include <iostream>`
 	
`void intercambia(int& a, int& b) {`
`  int c = a;`
`  a = b;`
`  b = c;`
`}`
 	
`int main( ) {`
`  int a = 5, b = 7;`
`  intercambia(a, b);`
`  std::cout << a << " " << b;`
`}`
	`7 5`

Afortunadamente, la biblioteca estándar de C++ ya tiene implementada esta función bajo el nombre de `std::swap` y está declarada en el archivo de encabezado `<algorithm>`. En este curso, usaremos la biblioteca estándar de C++ ampliamente.

Código	Salida

`int a = 5, b = 7;`
`std::swap(a, b);`
`std::cout << a << " " << b;`
	`7 5`

En C++ también existe el retorno por valor y el retorno por referencia. Este último debe usarse con cuidado: lo usual es regresar por referencia una variable que nosotros mismos recibimos por referencia. Jamás debemos regresar una variable local por referencia, porque ésta se destruirá al terminar la función.

3.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Forzando-la-caja-fuerte`.
2. Resuelve el problema `https://omegaup.com/arena/problem/Ordenando-numeros`.

4. Recursión y sus aplicaciones

Normalmente existe más de un algoritmo que resuelve un problema computacional dado. Por ejemplo, la exponenciación de números naturales a^b puede implementarse mediante iteración o mediante recursión, esto último siendo el nombre que recibe el empleo de llamadas anidadas o recursivas a función.

Ejemplo de uso

```
int exponenciacion_iterativa(int a, int b) {
   int res = 1;
   for (int i = 0; i < b; ++i) {
      res *= a;
   }
   return res;
}

int exponenciacion_recursiva(int a, int b) {
   if (b == 0) {
      return 1;
   } else {
      return a * exponenciacion_recursiva(a, b - 1);
   }
}
```

Para entender la naturaleza de ambos algoritmos, podemos decir que el algoritmo iterativo interpreta el cálculo de 2^4 como $2 \times 2 \times 2 \times 2$, mientras que el algoritmo recursivo lo interpreta como 2×2^3. Nótese que al decir que $2^4 = 2 \times 2^3$ estamos describiendo el cálculo de una exponenciación en términos del cálculo de otra exponenciación más sencilla (ya que, intuitivamente, las exponenciaciones con exponentes más chicos son más sencillas de calcular). Formalmente, un algoritmo recursivo es aquél que, para calcular la respuesta de una instancia de un problema, primero resuelve una o más instancias más fáciles del mismo problema y luego usa sus resultados para calcular la respuesta de la instancia original.

Así como las implementaciones iterativas evitan ciclos infinitos mediante condiciones de paro, las implementaciones recursivas evitan recursión infinita mediante *casos base*. Un caso base se da cuando el trabajo a realizar es tan simple que podemos realizarlo directamente sin necesidad de hacer más llamadas recursivas. En el ejemplo recursivo para exponenciación de naturales se definió un caso base cuando el exponente es 0, por lo que el planteamiento recursivo fue $a^0 = 1$ y $a^b = a \times a^{b-1}$. Sería válido definir casos base adicionales, ya que por ejemplo, calcular a^1 también es fácil.

En general se prefieren las implementaciones iterativas a las recursivas, ya que la recursión puede requerir crear muchos marcos de función, lo que consume tiempo y memoria de la pila del proceso. En la mayoría de los sistemas operativos, la capacidad por omisión de la pila del proceso es pequeña (1 MB en Windows y 8 MB en Linux) y el programa puede abortar si se hace una recursión muy profunda o si se declaran arreglos muy grandes. Sin embargo, algunos algoritmos son recursivos por naturaleza y no vale la pena intentar volverlos iterativos, o bien, pueden ser más eficientes que el algoritmo iterativo obvio.

Para el mismo problema de exponenciación de naturales, un planteamiento recursivo alternativo es $a^0 = 1$, $a^b = (a^{\frac{b}{2}})^2$ cuando b es par y $a^b = (a^{\lfloor \frac{b}{2} \rfloor})^2 \times a$ cuando b es impar. Por ejemplo, $2^{30} = (2^{15})^2$ y $2^{15} = (2^7)^2 \times 2$. En este nuevo planteamiento, el exponente b se divide entre dos al hacer recursión, por lo que sólo se necesitan $\lfloor log_2(b) \rfloor + 1$ llamadas recursivas para $b > 0$, sin contar la llamada original. Elevar un valor al cuadrado se puede manejar como una multiplicación por sí mismo, por lo que la cantidad total de multiplicaciones ejecutadas en este planteamiento también es logarítmica. Aún suponiendo que se ejecuten dos multiplicaciones por llamada recursiva, lo cual sólo ocurre cuando la llamada respectiva recibe un exponente impar, calcular 2^{60} requeriría menos de 14 multiplicaciones en total. Esto contrasta con las 60 multiplicaciones que ejecutaría el algoritmo iterativo obvio.

Ejemplo de uso

```
int exponenciacion_recursiva(int a, int b) {
   if (b == 0) {
      return 1;
   } else {
      int t = exponenciacion_recursiva(a, b / 2);
      return (b % 2 == 0 ? t * t : t * t * a);
   }
}
```

4.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/El-ciclo-de-vida-de-las-bacteri2`.
2. Resuelve el problema `https://omegaup.com/arena/problem/Lavos-el-devorador-de-planetas`.

5. Apuntadores y secuencias

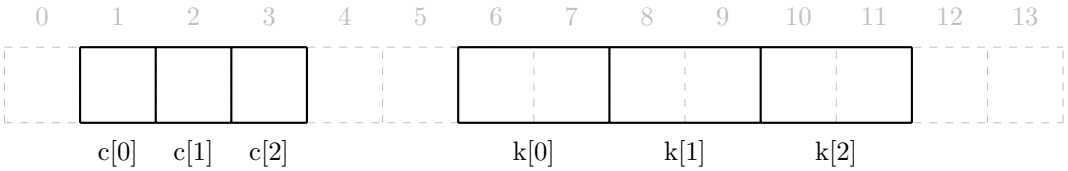
La memoria principal de la computadora está explícitamente dividida en bytes (y actualmente se acepta que un byte está formado por 8 bits). No todas las variables ocupan la misma cantidad de bytes en memoria. Por ejemplo, una variable de tipo `char` ocupa 1 byte mientras que una variable de tipo `int` suele ocupar 4 bytes. El operador `sizeof` regresa la cantidad de bytes que ocupa un tipo o variable.

Código	Salida
`std::cout << sizeof(char) << "\n";`	1
`int n;`	4
`std::cout << sizeof(n) << "\n";`	2
`std::cout << sizeof(short) << "\n";`	

Cuando compilamos un programa, es responsabilidad del compilador y del sistema operativo apartar fragmentos de esta memoria para almacenar nuestras variables. El lenguaje garantiza que todos los elementos de un mismo arreglo aparecen contiguos en memoria.

Código

```
char c[3];
short k[3];
```

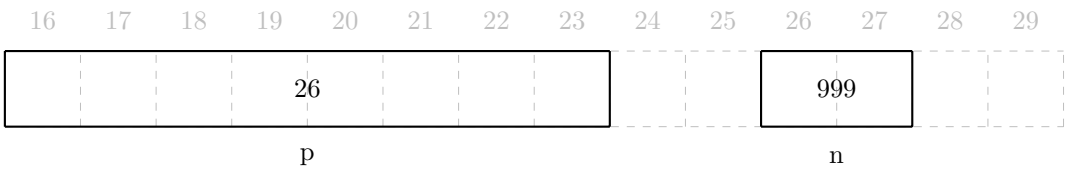


En gris se muestra la memoria y la dirección de sus bytes. En negro se muestran los arreglos `c` y `k`. Nótese como cada elemento de `k` ocupa más de un byte, ya que `sizeof(short)` $= 2$.

La dirección de una variable es la dirección del primer byte que ocupa y puede obtenerse con el operador `&` prefijo. La dirección de una variable de tipo `T` puede almacenarse en una apuntador de tipo `T*`. En una computadora de 32 bits que sólo puede direccionar 2^{32} bytes de memoria, se tiene `sizeof(T*)`$= 4$ bytes (ya que $4 \times 8 = 32$ bits), mientras que en una computadora de 64 bits se tiene que `sizeof(T*)`$= 8$ bytes. Esto es independiente del tipo `T`, ya que un apuntador siempre almacena una dirección.

Código

```
short n = 999;
short* p = &n;
```



Un apuntador es una variable que almacena la dirección de otra.
La dirección de una variable es la de su primer byte.

La característica más importante de los apuntadores es su capacidad de desreferencia. El operador prefijo `*` sobre un apuntador permite visitar la variable de la que conocemos su dirección. Por ejemplo:

Código	Salida

`int n = 999;`
`int* p = &n;`
`std::cout << *p << "\n";  // n mediante p`
`*p = 5;  // n mediante p`
`std::cout << n << "\n";`
	`999`
	`5`

Si reasignamos un apuntador a una nueva dirección, ahora se referirá a otra variable:

Código	Salida

`int a = 0, b = 1;`
`int* p = &a;`
`std::cout << *p << "\n";  // a mediante p`
`p = &b;`
`std::cout << *p << "\n";  // b mediante p`
	`0`
	`1`

Si una variable expone su dirección al llamar una función, la función llamada podrá modificar el valor de la variable usando el apuntador. A esto se le denomina paso por apuntador.

Código	Salida

`#include <iostream>`
 	
`void func(int* p) {`
`  *p += 1;  // n mediante p`
`}`
 	
`int main( ) {`
`  int n = 2;`
`  func(&n);`
`  std::cout << n;`
`}`
	`3`

Cuando un apuntador apunta a algún elemento de un arreglo, podemos usar los operadores `++` y `--` para movernos de un elemento al siguiente o al anterior en el arreglo. El lenguaje conoce el `sizeof(T)` de los elementos para adecuar correctamente la dirección almacenada en el apuntador.

Código	Salida

`int a[3] = { 5, 7, 8 };`
`int* p = &a[0];`
`std::cout << *p << " ";  // a[0]`
`++p;  // avanzamos`
`std::cout << *p << " ";  // a[1]`
`++p;  // avanzamos`
`std::cout << *p << " ";  // a[2]`
`--p;  // retrocedemos`
`std::cout << *p << " ";  // a[1]`
	`5 7 8 7`

También están definidos los operadores `+=` y `-=` para avanzar o retroceder varios elementos en un paso.

Código	Salida

`int a[3] = { 5, 7, 8 };`
`int* p = &a[0];`
`p += 2;  // avanzamos`
`std::cout << *p;  // a[2]`
	`8`

Los operadores `+` y `-` siguen la semántica de los operadores `+=` y `-=` en el sentido de que producen un apuntador que es el resultado de la suma o de la resta, pero sin modificar el apuntador original.

Código	Salida

`int a[3] = { 5, 7, 8 };`
`int* p = &a[0];`
`std::cout << *(p + 2) << "\n";  // a[2]`
`std::cout << *p << "\n";  // a[0]`
	`8`

Dos apuntadores se pueden comparar para saber si apuntan a la misma dirección (`p1 == p2`) o para saber si un apuntador apunta a una dirección que esté a la izquierda de otra (`p1 < p2`). El resto de los operadores relacionales `!=`, `>`, `<=` y `>=` también están definidos.

Código	Salida

`int a[3] = { 9, 9, 9 };`
`int* p1 = &a[0];`
`int* p2 = &a[0];`
`int* p3 = &a[2];`
`std::cout << (p1 == p2) << "\n";`
`std::cout << (p1 == p3) << "\n";`
`std::cout << (p1 < p3) << "\n";`
	`1`
	`0`
	`1`

En ese sentido, debe quedar clara la diferencia entre comparar apuntadores y comparar los valores de las variables que están siendo apuntadas. Por ejemplo:

Código	Salida

`int a = 2, b = 2;`
`int* pa = &a;`
`int* pb = &b;`
`// ¿misma dirección apuntada?`
`std::cout << (pa == pb) << "\n";`
`// ¿mismos valores apuntados?`
`std::cout << (*pa == *pb) << "\n";`
	`0`
	`1`

La resta de apuntadores `p2 - p1` es igual a la cantidad de veces que debemos avanzar el apuntador `p1` para que apunte a la misma dirección de `p2`.

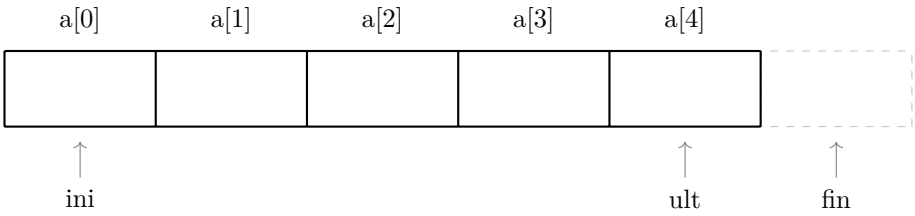
Código	Salida

`int a[10];`
`int* p1 = &a[2];`
`int* p2 = &a[7];`
`std::cout << p2 - p1 << "\n";`
	`5`

Diremos que un apuntador que apunte al primer elemento de un arreglo es el apuntador *al inicio del arreglo*. Diremos que un apuntador que apunte al último elemento de un arreglo es el apuntador *al último del arreglo*. El lenguaje C++ además garantiza que es posible construir la dirección del elemento que estaría inmediatamente después de un arreglo (aunque dicho elemento en realidad no exista). Diremos que un apuntador que apunte inmediatamente después del último elemento de un arreglo es el apuntador *al fin del arreglo*. El apuntador al fin sólo puede compararse con otro apuntador: es un error desreferenciarlo ya que el elemento en cuestión en realidad no existe.

Código

`int a[5];`
`int* ini = &a[0];`
`int* ult = &a[4];`
`int* fin = &a[5];`



Los apuntadores al inicio, al último y al fin.

Haciendo uso de los apuntadores anteriores, una forma rara pero válida de visitar los elementos de un arreglo consiste en declarar un apuntador que comience en el inicio y que siga avanzando hasta alcanzar el apuntador al fin, el cual no debe desreferenciarse. Por ejemplo:

Código	Salida

`int a[5] = { 3, 1, 4, 1, 6 };`
`for (int* p = &a[0]; p != &a[5]; ++p) {`
`  std::cout << *p << " ";`
`}`
	`3 1 4 1 6`

Si un arreglo `a` tiene `n` elementos, entonces el apuntador al fin siempre es `&a[n]`. Aunque siempre es seguro obtener el apuntador al fin, de todos modos es algo raro que `&a[n]` sea válido pero `a[n]` no lo sea. La expresión `&a[n]` es equivalente a `(&a[0] + n)`, la cual produce un apuntador al inicio que avanzó `n` elementos a la derecha. Esta última notación suele preferirse.

La pareja de apuntadores [`ini`, `fin`) denota un intervalo semiabierto porque los elementos desreferenciables del arreglo comienzan a partir de `ini` sin incluir `fin`. Ésta no es la primera vez que usamos un intervalo semiabierto. Por ejemplo, ésta es la manera típica de iterar sobre un arreglo mediante índices:

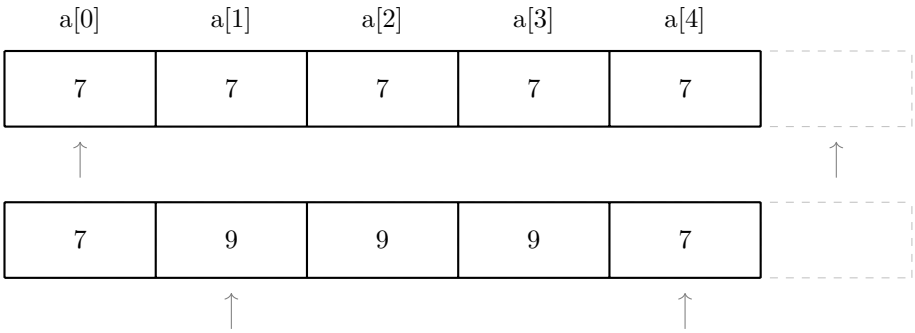
Código

```
int arr[5];                          // un arreglo de tamaño 5
for (int i = 0; i < 5; ++i) {        // desde 0 hasta antes de 5
   a[i] = 999;     // se visita desde a[0] hasta a[4], luego el ciclo termina
}
```

Dada una pareja de posiciones (generalmente apuntadores), diremos que los elementos desreferenciables en ese rango contituyen una *secuencia*. En una secuencia, el orden de los elementos importa: existe la noción del primer y del último elemento de la secuencia. El uso de apuntadores de inicio y fin constituyen una forma creativa de pasar una secuencia (la cual puede ser arreglo ¡o un subarreglo!) a una función:

Código	Salida

`#include <iostream>`
 	
`void llena(int* ini, int* fin, int v) {`
`  for (int* p = ini; p < fin; ++p) {`
`  *p = v;`
`  }`
`}`
 	
`int main( ) {`
`  int a[5];`
`  llena(&a[0], &a[0] + 5, 7);`
`  llena(&a[1], &a[0] + 4, 9);`
`  for (int i = 0; i < 5; ++i) {`
`  std::cout << a[i] << " ";`
`  }`
`}`
	`7 9 9 9 7`



Los apuntadores de inicio y fin para cada llamada a `llena`.

6. Algoritmos iterativos y recursivos sobre secuencias

A continuación se describen e implementan una serie de algoritmos elementales sobre secuencias, los cuales sirven como pilares fundamentales en la construcción de algoritmos más sofisticados. De ahí radica el hecho de que todo programador debe saber cómo implementarlos, aún cuando la biblioteca estándar de C++ ya proporciona implementaciones de éstos y otros algoritmos en `<algorithm>`.

Se presentarán dos implementaciones de cada algoritmo a estudiar en esta sección: una iterativa y una recursiva. Las implementaciones recursivas usan alguna de dos estrategias principales:

1. La llamada original hace parte del trabajo y luego delega lo que falta a una o más llamadas recursivas.
2. La llamada original primero hace una o más llamadas recursivas y luego completa el trabajo faltante.

Con frecuencia, los caso base de las implementaciones recursivas se dan cuando ya no hay más trabajo por hacer. Todas las implementaciones que se presentan a continuación toman una pareja de apuntadores `ini`, `fin` para denotar la secuencia. Éste es el mismo estilo que sigue la biblioteca estándar de C++. Al ser [`ini`, `fin`) un intervalo semiabierto, la secuencia vacía se da cuando `ini == fin`. Del mismo modo, el número de elementos de la secuencia se puede calcular con la resta de apuntadores `fin - ini`. Por tal motivo, supondremos que `ini` nunca apunta después de `fin`.

> Por cuestiones pedagógicas, las definiciones de las siguientes funciones pueden no ser óptimas y pueden no coincidir completamente con las que se encuentran la biblioteca estándar de C++. Sin embargo y para los fines de este curso, su uso básico coincide con las de la biblioteca de C++.

- Llenar una secuencia de valores (`std::fill`)
  Este algoritmo toma una secuencia y un valor, y copia dicho valor en todos los elementos de la secuencia. Una implementación iterativa simplemente visita y asigna cada elemento de la secuencia. Una implementación recursiva primero pregunta si aún faltan elementos por asignar. En caso afirmativo, le asigna valor al primer elemento de la secuencia y luego delega el resto del trabajo de forma recursiva.

Implementación iterativa

```
void fill(int* ini, int* fin, int v) {
   while (ini != fin) {
      *ini = v;
      ++ini;
   }
}
```

Implementación recursiva

```
void fill(int* ini, int* fin, int v) {
   if (ini != fin) {
      *ini = v;
      fill(ini + 1, fin, v);
   }
}
```

Ejemplo de uso

```
int a[5];
std::fill(&a[0], &a[0] + 5, 7);   // a vale { 7, 7, 7, 7, 7 }
```

- Contar las apariciones de un valor en una secuencia (`std::count`)
  Este algoritmo toma una secuencia y un valor, y cuenta cuántas veces aparece dicho valor en los elementos de la secuencia. Una implementación iterativa simplemente visita y compara cada elemento de la secuencia con el valor dado, llevando la cuenta de coincidencias en un acumulador. Una implementación recursiva primero pregunta si aún faltan elementos por comparar. En caso afirmativo, compara el primer elemento de la secuencia con el valor dado y a este resultado le suma la cantidad de coincidencias que encontró la llamada recursiva a la cual le delegó el resto de la secuencia.

Implementación iterativa

```
int count(int* ini, int* fin, int v) {
   int res = 0;
   while (ini != fin) {
      res += (*ini == v);
      ++ini;
   }
   return res;
}
```

Implementación recursiva

```
void count(int* ini, int* fin, int v) {
   int res = 0;
   if (ini != fin) {
      res += (*ini == v);
      res += count(ini + 1, fin, v);
   }
   return res;
}
```

Ejemplo de uso

```
int a[5] = { 3, 1, 4, 1, 6 };
int r = std::count(&a[0], &a[0] + 5, 1);   // r vale 2
```

- Búsqueda lineal de un valor (`std::find`)
  Este algoritmo toma una secuencia y un valor, y busca la primera aparición del valor en la secuencia. Una implementación iterativa avanza sobre la secuencia mientras no haya encontrado una coincidencia y mientras aún haya elementos a comparar. Una implementación recursiva primero pregunta si aún faltan elementos por comparar. En caso afirmativo, compara el primer elemento de la secuencia y si no es una coincidencia, entonces delega el resto del trabajo de forma recursiva. En ambos casos se regresa el apuntador al fin cuando no se encontró el valor buscado.

Implementación iterativa

```
int* find(int* ini, int* fin, int v) {
   while (ini != fin && *ini != v) {
      ++ini;
   }
   return ini;
}
```

Implementación recursiva

```
int* find(int* ini, int* fin, int v) {
   if (ini != fin && *ini != v) {
      return find(ini + 1, fin, v);
   }
   return ini;
}
```

Ejemplo de uso

```
int a[5] = { 1, 2, 3, 4, 5 };
int* p = std::find(&a[0], &a[0] + 5, 3);   // p vale &a[2]
if (p != &a[0] + 5) {
   // encontrado
} else {
   // no encontrado
}
```

- Encontrar el mínimo valor de una secuencia (`std::min_element`)
  Este algoritmo toma una secuencia y regresa un apuntador a la primera ocurrencia del mínimo valor de la secuencia. Una implementación iterativa avanza sobre la secuencia y mantiene en todo momento un apuntador al mínimo actual, reemplazando dicho apuntador en caso de encontrar un elemento aún menor. Una implementación recursiva es ligeramente más complicada, pues si la secuencia está vacía entonces la función regresará el fin (el cual no debe desreferenciarse). Para evitar este inconveniente, sólo haremos la llamada recursiva si ésta al menos recibirá un elemento. Si estamos en esta situación, recursivamente calculamos el mínimo elemento de la secuencia adelante del inicio y luego comparamos dicho mínimo contra el elemento inicial de la secuencia.

Implementación iterativa

```
int* min_element(int* ini, int* fin) {
   int* p = ini;
   while (ini != fin) {
      if (*ini < *p) {
         p = ini;
      }
      ++ini;
   }
   return p;
}
```

Implementación recursiva

```
int* min_element(int* ini, int* fin) {
   int* p = ini;
   if (fin - ini >= 2) {
      int* q = min_element(ini + 1, fin);
      if (*q < *p) {
         p = q;
      }
   }
   return p;
}
```

Ejemplo de uso

```
int a[5] = { 3, 4, 1, 1, 6 };
int* p = std::min_element(&a[0], &a[0] + 5);   // p vale &a[2], *p vale 1
```

- Invertir una secuencia (`std::reverse`)
  Este algoritmo toma una secuencia y coloca sus elementos en el orden inverso al que aparecían originalmente. Una implementación iterativa se coloca en los extremos de las secuencias e intercambia el primer elemento con el último, avanzando hacia el centro a cada paso. Una implementación recursiva primero pregunta si aún hay elementos por invertir. En caso afirmativo, intercambia el primer elemento con el último y luego delega el resto del trabajo de forma recursiva avanzando hacia el centro. En ambos casos, la inversión termina cuando ya no faltan elementos por invertir o sólo falta uno (el cual quedaría en su misma posición). Cabe recordar que el último elemento de la secuencia está atrás del fin.

Implementación iterativa

```
void reverse(int* ini, int* fin) {
   while (fin - ini >= 2) {
      std::swap(*ini, *(fin - 1));
      ++ini, --fin;
   }
}
```

Implementación recursiva

```
void reverse(int* ini, int* fin) {
   if (fin - ini >= 2) {
      std::swap(*ini, *(fin - 1));
      reverse(ini + 1, fin - 1);
   }
}
```

Ejemplo de uso

```
int a[5] = { 1, 2, 3, 4, 5 };
std::reverse(&a[0], &a[0] + 5);   // a vale { 5, 4, 3, 2, 1 }
```

- Copiar una secuencia en otra (`std::copy`)
  Este algoritmo toma una secuencia y el inicio de otra, y copia todos los elementos de la primera secuencia en la segunda secuencia. Una implementación iterativa simplemente visita cada elemento de la primera secuencia y avanza sobre la segunda secuencia conforme va copiando. Una implementación recursiva primero pregunta si aún faltan elementos por copiar. En caso afirmativo, copia el primer elemento de la primera secuencia en el primer elemento la segunda secuencia y luego delega el resto del trabajo de forma recursiva, avanzando sobre ambas secuencias.

Implementación iterativa

```
void copy(int* ini1, int* fin1,
          int* ini2) {
   while (ini1 != fin1) {
      *ini2 = *ini1;
      ++ini1, ++ini2;
   }
}
```

Implementación recursiva

```
void copy(int* ini1, int* fin1,
          int* ini2) {
   if (ini1 != fin1) {
      *ini2 = *ini1;
      copy(ini1 + 1, fin1, ini2 + 1);
   }
}
```

Ejemplo de uso

```
int a[5] = { 1, 2, 3, 4, 5 }, b[5];
std::copy(&a[0], &a[0] + 5, &b[0]);   // b vale { 1, 2, 3, 4, 5 }
```

- Determinar si dos secuencias son iguales (`std::equal`)
  Este algoritmo toma una secuencia y el inicio de otra, y determina si los elementos correspondientes de ambas secuencias son iguales. Una implementación iterativa avanza sobre ambas secuencias mientras siga encontrando coincidencias entre ellas (lo cual eventualmente debe terminar al llegar al fin de la primera secuencia) y luego pregunta si pudo hacer coincidir todos los elementos de las secuencias. Una implementación recursiva primero pregunta si aún faltan elementos por comparar. En caso afirmativo, compara el primer elemento de la primera secuencia contra el primer elemento la segunda secuencia y si son iguales, entonces se examina el resultado de delegar el resto del trabajo de forma recursiva.

Implementación iterativa

```
bool equal(int* ini1, int* fin1,
           int* ini2) {
   while (ini1 != fin1 &&
          *ini1 == *ini2) {
      ++ini1, ++ini2;
   }
   return ini1 == fin1;
}
```

Implementación recursiva

```
bool equal(int* ini1, int* fin1,
           int* ini2) {
   if (ini1 != fin1 &&
      *ini1 == *ini2) {
      return equal(ini1 + 1, fin1,
                   ini2 + 1);
   }
   return ini1 == fin1;
}
```

Ejemplo de uso

```
int a[5] = { 3, 1, 4, 1, 6 };
int b[5] = { 2, 7, 1, 8, 1 };
bool b = std::equal(&a[0], &a[0] + 5, &b[0]);   // falso
```

6.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Invirtiendo-palabras`.
2. Resuelve el problema `https://omegaup.com/arena/problem/subterraneo-camino-shamash`.

7. Programacion genérica y funciones de orden superior

El lenguaje C++ provee de características avanzadas para la especificación de algoritmos abstractos y eficientes. La primera de ellas es la sobrecarga de funciones: la capacidad de declarar varias funciones con el mismo nombre pero con distintos parámetros, de modo que el compilador decide cuál de ellas llamar con base en los argumentos usados en la llamada a función:

Código

```
#include <iostream>

void f(int n) {
   std::cout << "f de int\n";
}
void f(double d) {
   std::cout << "f de double"\n";
}

int main( ) {
   f(5);
   f(3.14);
}
```

Salida

```
f de int
f de double
```

La segunda característica son las plantillas. Por ejemplo, el algoritmo de intercambio de dos valores es general, pero una función ordinaria sólo compilaría si se usa con el tipo correcto:

Código

```
#include <iostream>

void intercambia(int& a, int& b) {
   int c = a;
   a = b;
   b = c;
}

int main( ) {
   int a = 5, b = 7;
   intercambia(a, b);   // ok
   double c = 3.14, e = 2.71;
   intercambia(c, e);   // error
}
```

Diagnóstico

```
error: cannot bind non-const lvalue
reference of type 'int&'
    intercambia(c, e);
```

Una plantilla se declara con la palabra `template` y permite generalizar una función para cualquier tipo:

Código

```
#include <iostream>

template<typename T>
void intercambia(T& a, T& b) {
   T c = a;
   a = b;
   b = c;
}

int main( ) {
   int a = 5, b = 7;
   intercambia(a, b);   // ok
   double c = 3.14, e = 2.71;
   intercambia(c, e);   // ok
   std::cout << a << " " << b << "\n";
   std::cout << c << " " << e << "\n";
}
```

Salida

```
7 5
2.71 3.14
```

Las plantillas dan soporte al paradigma de la programación genérica, el cual busca poder definir componentes de software términos en generales y eficientes. Los algoritmos de la biblioteca, tales como `std::swap`, `std::fill` o `std::find`, son plantillas. La biblioteca tambien proporciona las plantillas de función `std::min` y `std::max` que regresan el mínimo y el máximo de dos valores, respectivamente.

Código

```
int a = 5, b = 7;
std::cout << std::min(a, b) << "\n";
double c = 3.14, e = 2.71;
std::cout << std::max(c, d) << "\n";
```

Salida

```
5
3.14
```

La tercera característica es la capacidad de definir funciones que tomen otras funciones como parámetros. A éstas se les conoce como funciones de orden superior. El siguiente ejemplo define una variante de búsqueda lineal, la cual busca el primer valor que cumpla cierta propiedad. Dicha propiedad está dictada por una función que regresa verdadero o falso (también llamada predicado) y que es pasada como argumento junto con la secuencia sobre la que se realiza la búsqueda:

Código	Salida

`#include <iostream>`
 	
`template<typename T, typename F>`
`T* find_if(T* ini, T* fin, F pred) {`
`  while (ini != fin && !pred(*ini)) {`
`  ++ini;`
`  }`
`  return ini;`
`}`
 	
`bool es_par(int n) {`
`  return n % 2 == 0;`
`}`
 	
`int main( ) {`
`  int a[] = { 3, 1, 4, 1, 6 };`
`  int* p = find_if(&a[0],`
`  &a[0] + 5, es_par);`
`  std::cout << *p;`
`}`
	4

Esta variante de búsqueda lineal también está implementada en la biblioteca de C++. En el ejemplo anterior, el tipo `F` de la plantilla se deduce como `bool(*)(int)`, que es el tipo real de una función que toma un entero y regresa un booleano. Desafortunadamente, es poco conveniente usar dicho tipo.

El lenguaje C++ también permite definir una función dentro de una expresión, lo que nos ahorra la necesidad de definir la función aparte. A esto se le conoce como funciones lambda:

Código

```
int a[] = { 3, 1, 4, 1, 6 };
int* p = std::find_if(&a[0], &a[0] + 5, [](int n) {
  return n % 2 == 0;
});
std::cout << *p;   // 4
```

Por omisión, una función lambda no tiene acceso a las variables del ámbito local en el que fue declarada. Sin embargo, se puede especificar un `&` dentro de los corchetes de la declaración para poder tener acceso a dichas variables. El siguiente ejemplo busca el primer múltiplo de `m` dentro de una secuencia:

Código	Entrada	Salida

`int m;`
`std::cin >> m;`
`int a[] = { 3, 1, 4, 1, 6 };`
`int* p = std::find_if(`
`  &a[0], &a[0] + 5, [&](int n) {`
`  return n % m == 0;`
`});`
`if (p != &a[0] + 5) {`
`  std::cout << *p;`
`}`
	2	4

8. Algoritmos de ordenamiento

Usando la biblioteca estándar de C++, ordenar una secuencia es fácil con la función `std::sort` de `algorithm`. Esta función toma un apuntador al inicio y otro al fin de la secuencia y produce un

ordenamiento ascendente (es decir, de menor a mayor). Sin embargo, todo programador debe conocer cómo se implementan varios algoritmos clásicos de ordenamiento. A continuación se describen e implementan seis de ellos y posteriormente en el curso se estudiarán dos algoritmos más.

- Ordenamiento de burbuja (*bubble sort*)
  Este algoritmo barre la secuencia de izquierda a derecha e intercambia parejas de elementos consecutivos cuando el elemento de la izquierda es mayor que el de la derecha. Después de la primera pasada, se tiene la certeza de que el elemento más grande ahora está en la última posición. Si repetimos el proceso, ahora el segundo elemento más grande estará en la penúltima posición. Este proceso se repite hasta que todos los elementos estén en su posición. Si el arreglo tiene n elementos, basta realizar $n-1$ pasadas.

  En cuanto a la eficiencia de este algoritmo, la primera pasada hace $n-1$ comparaciones, la segunda pasada hace $n-2$ comparaciones, la tercera pasada hace $n-3$ comparaciones, etc. La última pasada hace una comparación, ya que un intercambio basta para ordenar los dos elementos restantes. La cantidad total de trabajo está dada por $(n-1)+(n-2)+(n-3)+\cdots+1 = \frac{n^2-n}{2}$ que es un polinomio cuadrático. Por ejemplo, para $n = 10^5$ tenemos que $\frac{n^2-n}{2} \approx 5 \times 10^9$, por lo que este algoritmo tardará aproximadamente 5 segundos en ordenar cien mil elementos.

  Código

```
void bubble_sort(int* ini, int* fin) {
   for (int i = 0; i < fin - ini - 1; ++i) {
      for (int* p = ini; p != fin - i - 1; ++p) {
         if (*p > *(p + 1)) {
            std::swap(*p, *(p + 1));
         }
      }
   }
}
```

- Ordenamiento por selección (*selection sort*)
  Este algoritmo primero busca el elemento más pequeño de la secuencia y lo coloca en la primera posición mediante un intercambio. Si repetimos este proceso ignorando el mínimo recién colocado, podremos encontrar el segundo elemento más pequeño de la secuencia para así colocarlo en la segunda posición. Este proceso se repite hasta que hayamos colocado en su lugar todos los elementos de la secuencia.

  En cuanto a la eficiencia de este algoritmo, la primera pasada hace $n-1$ comparaciones, la segunda pasada hace $n-2$ comparaciones, la tercera pasada hace $n-3$ comparaciones, etc. La última pasada hace una comparación, ya que un intercambio basta para ordenar los dos elementos restantes. La cantidad total de trabajo está dada por $(n-1)+(n-2)+(n-3)+\cdots+1 = \frac{n^2-n}{2}$ que es un polinomio cuadrático. Este algoritmo tardará aproximadamente 5 segundos en ordenar cien mil elementos.

  Código

```
void selection_sort(int* ini, int* fin) {
   while (fin - ini >= 2) {
      std::swap(*ini, *std::min_element(ini, fin));
      ++ini;
   }
}
```

- Ordenamiento por inserción (*insertion sort*)
  Este algoritmo va considerando uno a uno los elementos de la secuencia, de izquierda a derecha, pero a cada paso mantiene ordenados los elementos que ya vio. Cada nuevo elemento deberá colocarse en una posición que no rompa el orden parcialmente calculado. Esto lo podemos hacer recorriendo el nuevo elemento hacia atrás mediante intercambios, mientras aún haya elementos mayores atrás.

  La eficiencia de este algoritmo es difícil de determinar, porque no sabemos qué tan tardado vaya a ser colocar los nuevos elementos en la secuencia parcialmente ordenada. Por una parte, si los elementos ya

aparecen en orden ascendente, entonces basta una comparación por elemento para que este algoritmo se de cuenta de que los elementos no necesitan recorrerse. Por otra parte, si los elementos aparecen en orden descendente, cada nuevo elemento debe recorrer toda la secuencia de atrás para él colocarse en la primera posición. En este peor caso, primero se hace una comparación, luego se hacen dos comparaciones, luego se hacen tres comparaciones, etc. La última pasada hace $n-1$ comparaciones. La cantidad total de trabajo está dada por $1+2+\cdots+(n-1)=\frac{n^2-n}{2}$ que es un polinomio cuadrático. Este algoritmo tardará aproximadamente 5 segundos en ordenar cien mil elementos en el peor caso.

Código

```
void insertion_sort(int* ini, int* fin) {
   for (int* p = ini; p != fin; ++p) {
      for (int* q = p; q != ini && *(q - 1) > *q; --q) {
         std::swap(*(q - 1), *q);
      }
   }
}
```

- Ordenamiento por mezcla (*merge sort*)
  Este algoritmo primero ordena recursivamente cada mitad de la secuencia y luego las mezcla de forma ordenada. La mezcla ordenada de dos secuencias ordenadas se puede realizar como sigue. Comenzaremos viendo el primer elemento de cada secuencia, que también son sus mínimos respectivos, y los compararemos para determinar el mínimo elemento de la mezcla. Luego avanzaremos sobre la secuencia de donde provino el mínimo y repetiremos el proceso para encontrar ahora al segundo elemento más pequeño de la mezcla. Eventualmente terminamos de avanzar en una de las dos secuencias; la mezcla se completa con los elementos restantes de la otra secuencia. La mezcla ordenada de dos secuencias es más fácil de programar si escribimos el resultado de la mezcla en una tercera secuencia auxiliar.

  La eficiencia de este algoritmo se determina como sigue. Por una parte, la mezcla ordenada no siempre realiza la misma cantidad de comparaciones, ya que es más fácil mezclar si una secuencia debe ir completa antes que otra, y es más difícil mezclar cuando los elementos de las secuencias deben intercalarse. La mezcla hace $n-1$ comparaciones en el peor caso, aunque siempre hace n asignaciones. Sin embargo, la cantidad de elementos a mezclar no siempre es la misma, ya que cada llamada recursiva de *merge sort* trabaja sobre menos elementos que la llamada padre. A pesar de eso, aparece un patrón: la llamada original trabaja sobre la secuencia completa, el nivel recursivo inferior trabaja con las dos mitades, el siguiente nivel recursivo trabaja con los cuatro cuartos, etc. La profundidad de la recursión es $\log_2(n)$, por lo que la cantidad total de trabajo está dada por $n\log_2(n)$. Este algoritmo tardará aproximadamente un milisegundo en ordenar cien mil elementos.

  Código

```
void merge(int* ini1, int* fin1, int* ini2, int* fin2, int* aux) {
   while (ini1 != fin1 && ini2 != fin2) {          // std::merge en <algorithm>
      *aux++ = (*ini1 < *ini2 ? *ini1++ : *ini2++);
   }
   std::copy(ini1, fin1, aux);
   std::copy(ini2, fin2, aux);
}

void merge_sort(int* ini, int* fin, int* aux) {
// aux es el inicio de una secuencia auxiliar con el mismo tamaño que la otra
   if (fin - ini >= 2) {
      int* mitad = ini + (fin - ini) / 2;
      merge_sort(ini, mitad, aux);
      merge_sort(mitad, fin, aux);
      merge(ini, mitad, mitad, fin, aux);
      std::copy(aux, aux + (fin - ini), ini);
   }
}
```

- Ordenamiento rápido de Hoare (*quicksort*)
  A grandes rasgos, este algoritmo primero calcula un valor pivote (incluso de forma arbitraria) y luego reacomoda o particiona los elementos de la secuencia dependiendo de si son menores o no que el pivote. Después, el algoritmo ordena recursivamente cada parte. Una implementación concreta de este algoritmo toma el último elemento de la secuencia como pivote y particiona el resto de los elementos, colocando primero los que son menores y luego los que no lo son. Una vez hecho esto, el algoritmo coloca el pivote enmedio de ambas partes (ya que el pivote es igual a sí mismo y constituye una buena frontera entre ellas) y finalmente hace dos llamadas recursivas, excluyendo de ellas al pivote que ya está en su lugar y garantizando así que el tamaño del problema recursivo disminuye.

  La eficiencia de este algoritmo es difícil de determinar. La partición hará $n-1$ comparaciones y luego colocamos el pivote, pero no sabemos de antemano si la partición estará balanceada. El mejor caso ocurre cuando la partición siempre queda balanceada y cada llamada recursiva recibe aproximadamente la mitad de la secuencia. La llamada original trabajaría sobre la secuencia completa, el nivel recursivo inferior trabajaría con las dos mitades, el siguiente nivel recursivo trabajaría con los cuatro cuartos, etc. La profundidad de la recursión en el mejor caso es $\log_2(n)$, por lo que la cantidad total de trabajo está dada por $n\log_2(n)$. El peor caso se da cuando la partición siempre queda desbalanceada y todos los elementos (menos el pivote) aparecen en la misma parte. Esto ocurre cuando ordenamos una secuencia que ya está ordenada y el pivote es el máximo valor. En dicho peor caso, la llamada original haría $n-1$ comparaciones, el nivel recursivo inferior haría $n-2$ comparaciones, el siguiente nivel recursivo haría $n-3$ comparaciones, etc. La cantidad total de trabajo en el peor caso está dada por $(n-1)+(n-2)+(n-3)+\cdots+1=\frac{n^2-n}{2}$, que es un polinomio cuadrático. Este algoritmo tardará aproximadamente un milisegundo en ordenar cien mil elementos en el mejor caso y 5 segundos en el peor caso.

  Código

```
template<typename F>
int* partition(int* ini, int* fin, F pred) {   // std::partition en <algorithm>
   int* p = ini;
   for (int* q = ini; q != fin; ++q) {
      if (pred(*q)) {
         std::swap(*q, *p++);
      }
   }
   return p;
}

void quicksort(int* ini, int* fin) {
   if (fin - ini >= 2) {
      int* p = partition(ini, fin - 1, [&](int n) {
         return n < *(fin - 1);
      });
      std::swap(*p, *(fin - 1));
      quicksort(ini, p);
      quicksort(p + 1, fin);
   }
}
```

- Ordenamiento por cuenta (*counting sort*)
  En contraste con los algoritmos anteriores, este algoritmo ordena una secuencia de enteros *sin comparar* sus elementos, siempre y cuando conozcamos de antemano el rango de valores que pueden tomar éstos. Sin pérdida de generalidad, supondremos que todos los elementos de la secuencia están en el intervalo $[0,k)$, por lo que el algoritmo comienza llenando de ceros un arreglo de tamaño k. Una vez hecho esto, cada elemento de la secuencia será usado como un índice sobre el arreglo para incrementar la celda respectiva. Tras procesar la secuencia, el arreglo contiene la cantidad de veces que apareció cada valor y entonces puede verse como una tabla de frecuencias. El último paso del algoritmo consiste en visitar dicho arreglo en orden y reconstruir la secuencia usando las frecuencias calculadas.

Este algoritmo visita dos veces el arreglo de frecuencias (primero para inicializarlo y luego para reconstruir la secuencia) y dos veces la secuencia (una vez para leerla y otra vez para escribir en ella), por lo que el trabajo total realizado es $2k + 2n$, que es lineal en n pero que también depende del valor de k. Más aún, el valor de k también constituye un requerimiento de memoria. Este algoritmo es el más rápido de todos cuando k es pequeña, pero es lento e incluso inviable cuando k es muy grande.

Código

```
void counting_sort(int* ini, int* fin) {
   int frec[k] = { };     // suponer que k es una constante
   for (int* p = ini; p != fin; ++p) {
      frec[*p] += 1;
   }
   for (int i = 0; i < k; ++i) {
      for (int j = 0; j < frec[i]; ++j) {
         *ini++ = i;
      }
   }
}
```

La biblioteca de C++ garantiza que el algoritmo usado por `std::sort` trabaja mediante comparaciones de elementos y que realiza un trabajo proporcional a $n \log_2(n)$ en el peor caso. De los algoritmos anteriores, el único que cumple con estas garantías es *merge sort*. Sin embargo, la mayoría de las implementaciones de `std::sort` usan un algoritmo híbrido llamado *introsort*. Éste comienza con *quicksort* pero detecta si está ocurriendo el peor caso, en cuyo caso cambia a otro algoritmo que se estudiará posteriormente y que sí tiene garantía $n \log_2(n)$. Cuando quedan muy pocos elementos a ordenar en los niveles más profundos de la recursión, se cambia a *insertion sort* que es muy rápido para valores pequeños de n.

8.1. Ejercicios

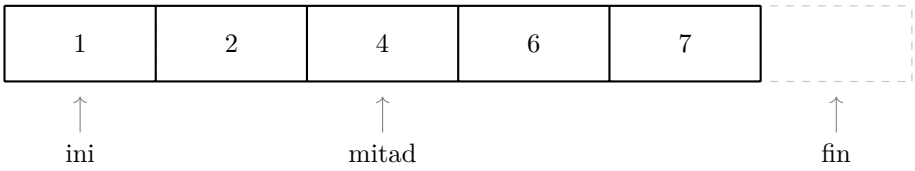
1. Resuelve el problema `https://omegaup.com/arena/problem/cactus_horizonte`.
2. Resuelve el problema `https://omegaup.com/arena/problem/Calculo-de-la-mediana`.
3. Resuelve el problema `https://omegaup.com/arena/problem/Muletillas-de-robots`.

9. Búsqueda eficiente en secuencias ordenadas

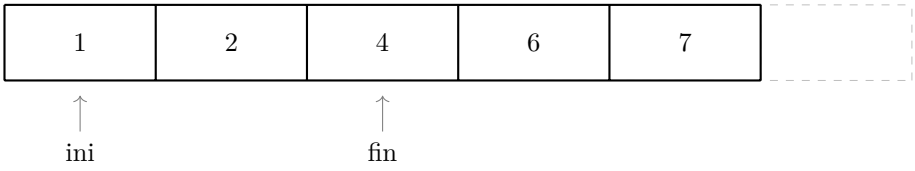
Desde que el ser humano aprendió a registrar información para poder consultarla después, la búsqueda de la misma se volvió una de las actividades más importantes de la sociedad moderna. Los libros incluyen tablas de contenidos e índices, los sistemas operativos ofrecen una opción de buscar entre nuestros archivos y los buscadores de internet entregan enlaces relevantes a páginas web, todo en un intento de encontrar rápidamente lo que estamos buscando en una cantidad de información cada vez más grande.

En este contexto, el algoritmo de búsqueda lineal (`std::find`) es posiblemente el más importante de los algoritmos fundamentales sobre secuencias. Sin embargo, éste también es un algoritmo de *fuerza bruta*, ya que su estrategia es simplemente examinar exhaustivamente todos los elementos de la secuencia hasta encontrar lo que buscamos o hasta determinar que no está. El algoritmo de búsqueda lineal no tiene (y no necesita tener) ningún conocimiento previo sobre la secuencia en la que buscará. Por otra parte, si sabemos de antemano que necesitaremos realizar muchas búsquedas sobre la misma secuencia, entonces debemos considerar formas más eficientes de buscar. En particular, una búsqueda lineal sobre una secuencia de n elementos puede requerir examinarlos todos: puede ocurrir que el valor buscado sea el último que revisemos o que lo que busquemos no esté. Si deseamos realizar m búsquedas sobre la misma secuencia y hacemos una búsqueda lineal para cada una, podríamos tener que realizar $n \times m$ comparaciones en total. Si $n \approx m \approx 10^5$ entonces $n \times m = 10^{10}$, por lo que un programa que realice lo anterior podría tardar hasta 10 segundos en terminar de contestar todas las búsquedas. En la producción de diccionarios, los árabes descubrieron hace más de 1000 años dos técnicas que resuelven el problema: el ordenamiento de la secuencia y el algoritmo de búsqueda binaria.

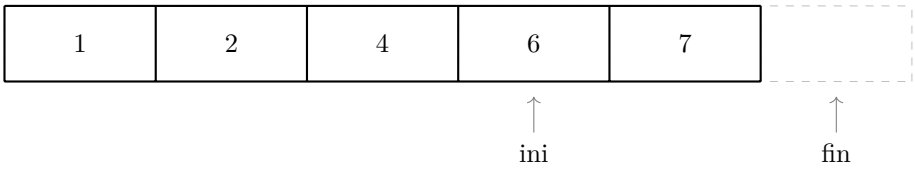
El algoritmo de búsqueda binaria sobre secuencias ya ordenadas trabaja como sigue: primero se calcula un apuntador al elemento de la mitad de la secuencia y luego comparamos el elemento apuntado contra el elemento que estamos buscando. Si ambos valores coinciden, entonces la búsqueda fue exitosa. En caso contrario, preguntamos si el valor buscado está a la izquierda o a la derecha y buscamos recursivamente en una de esas dos mitades. Nótese que pudimos descartar la mitad de la secuencia con una única comparación, pero que sólo fue posible lo anterior gracias a que la secuencia está ordenada.



El apuntador a la mitad se puede calcular con la expresión `ini + (fin - ini) / 2` donde `fin - ini` es la cantidad de elementos en juego, en este caso 5.



Al buscar un 2 en la secuencia anterior, el algoritmo de búsqueda binaria descarta la mitad derecha de la secuencia con una sola comparación.



Al buscar un 7 en la secuencia anterior, el algoritmo de búsqueda binaria descarta la mitad izquierda de la secuencia con una sola comparación.

Como búsqueda binaria descarta la mitad de la secuencia en tan sólo una comparación, la cantidad de comparaciones que necesita para determinar si un valor aparece o no en la secuencia es $\log_2(n)+1$ donde n es el tamaño de la secuencia. La subexpresión $\log_2(n)$ se puede deducir contando la cantidad de veces que podemos dividir n entre dos antes de llegar a 1, lo cual coincide con la cantidad de veces que podemos multiplicar 2 por sí mismo para alcanzar el valor de n. Cuando $n = 1$ entonces basta una comparación para terminar la búsqueda; de ahí aparece la comparación adicional en la expresión de arriba.

Tamaño de la secuencia (n elementos)	**Búsqueda lineal (n comparaciones)**	**Búsqueda binaria ($\log_2(n)+1$ comparaciones)**
1	1	1
10^3	10^3	11
10^6	10^6	21
10^9	10^9	31

La biblioteca estándar cuenta con tres funciones que implementan el algoritmo de búsqueda binaria:

- Determinar si un elemento aparece o no en la secuencia (`std::binary_search`)
  Este algoritmo toma una secuencia y un valor, y regresa verdadero si el valor aparece en la secuencia y falso en caso contrario. Este algoritmo es muy útil si nos basta determinar la mera existencia del valor en la secuencia, pero no nos da información sobre su posición en la misma.

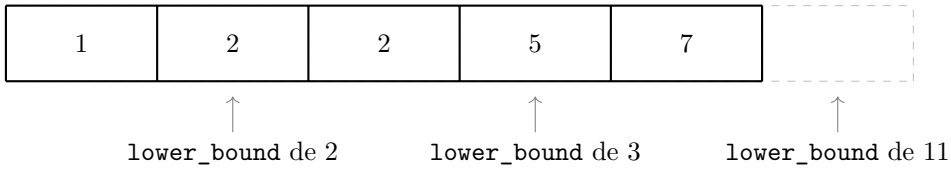
Código

```
bool binary_search(int* ini, int* fin, int v) {
   if (ini == fin) {
      return false;
   }

   int* mitad = ini + (fin - ini) / 2;
   if (v == *mitad) {
      return true;
   } else if (v < *mitad) {
      return binary_search(ini, mitad, v);
   } else {
      return binary_search(mitad + 1, fin, v);
   }
}
```

- Encontrar el primer elemento mayor o igual a un valor en la secuencia (`std::lower_bound`)
  Este algoritmo toma una secuencia y un valor, y regresa un apuntador al primer elemento mayor o igual que el valor buscado, o al fin si ningún elemento cumple lo anterior. Si el valor buscado sí aparece en la secuencia, entonces apuntaremos a su primera aparición (ya que pueden existir duplicados). Si el valor buscado no aparece en la secuencia, entonces el apuntador podría ser válido (y entonces apunta a un elemento mayor al buscado) o podría ser inválido y apuntar al fin. Este apuntador también coincide con la ubicación en la que podría colocarse el valor buscado sin romper el orden de la secuencia.

Código

```
int* lower_bound(int* ini, int* fin, int v) {
   if (fin - ini <= 2) {
      return std::find_if(ini, fin, [&](int actual) {
         return !(actual < v);
      });
   }

   int* mitad = ini + (fin - ini) / 2;
   if (*mitad < v) {
      return lower_bound(mitad + 1, fin, v);
   } else {
      return lower_bound(ini, mitad + 1, v);
   }
}
```

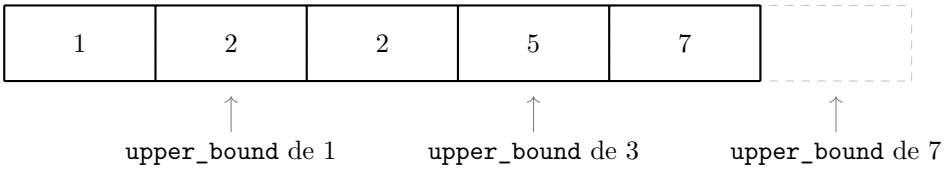


Apuntadores que regresa `std::lower_bound` al buscar algunos valores sobre la secuencia.

- Encontrar el primer elemento mayor a un valor en la secuencia (`std::upper_bound`)
  Este algoritmo toma una secuencia y un valor, y regresa un apuntador al primer elemento estrictamente mayor que el valor buscado o al fin si ningún elemento cumple lo anterior.

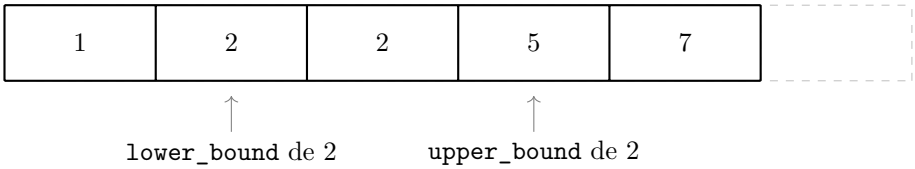
Código

```
int* upper_bound(int* ini, int* fin, int v) {
   if (fin - ini <= 2) {
      return std::find_if(ini, fin, [&](int actual) {
         return actual > v;
      });
   }

   int* mitad = ini + (fin - ini) / 2;
   if (*mitad > v) {
      return upper_bound(ini, mitad + 1, v);
   } else {
      return upper_bound(mitad + 1, fin, v);
   }
}
```



Apuntadores que regresa `std::upper_bound` al buscar algunos valores sobre la secuencia.

La pareja de apuntadores que regresa `std::lower_bound` y `std::upper_bound` buscando el mismo valor `v` constituye una subsecuencia [`ini`, `fin`) con todas las apariciones de `v` en la secuencia.



El inicio y fin de la subsecuencia más pequeña que contiene todas las apariciones de 2.

9.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Busqueda-binaria-p`.
2. Resuelve el problema `https://omegaup.com/arena/problem/El-zorro-y-las-uvas`.
3. Resuelve el problema `https://omegaup.com/arena/problem/El-juego-de-la-silla`.

10. Predicados de ordenamiento definidos por el usuario

A veces, el ordenamiento ascendente que produce `std::sort` no es suficiente. Por una parte, si queremos un orden descendente, lo único que sabríamos hacer en este momento es ordenarlo ascendentemente con `std::sort` y luego invertir la secuencia, lo cual es ineficiente (sería mucho mejor producir el orden descendente directamente). Por otra parte, `std::sort` no sabría cómo ordenar una secuencia de elementos de un tipo `struct` definido por el usuario, pero hacer esto a veces es necesario. Afortunadamente, las rutinas de ordenamiento y búsqueda binaria de la biblioteca de C++ tienen sobrecargas adicionales que nos permitirá configurar el comportamiento de dichas funciones para lograr lo anterior.

Dado un orden deseado, un predicado de ordenamiento es una función `p(a, b)` que regresa verdadero o falso y que nos permite determinar si los elementos de una secuencia aparecen o no en el orden deseado. En este sentido, la biblioteca estándar de C++ usa predicados que implementan la siguiente pregunta: ¿a debe ir antes que b en el orden deseado? Por omisión, la función `std::sort` usa un predicado de ordenamiento que simplemente regresa el valor de la expresión `(a < b)`, el cual se interpreta como "`a` debe ir antes que `b` si `a` es menor que `b`". Podemos usar otros predicados que se interpretarán de forma similar: si el predicado regresa verdadero entonces `a` debe aparecer antes que `b` en la secuencia.

Rutinas de ordenamiento y búsqueda binaria de `algorithm`	
Función de biblioteca	**Valor calculado**
`void sort(T* ini, T* fin);`	Ordena la secuencia con el predicado `a < b`.
`void sort(T* ini, T* fin, F p);`	Ordena la secuencia con el predicado `p(a, b)`.
`T* lower_bound(T* ini, T* fin, T v);`	Busca el primer `a` tal que `a < v` es falso.
`T* lower_bound(T* ini, T* fin, T v, F p);`	Busca el primer `a` tal que `p(a, v)` es falso.
`T* upper_bound(T* ini, T* fin, T v);`	Busca el primer `a` tal que `v < a` es verdadero.
`T* upper_bound(T* ini, T* fin, T v, F p);`	Busca el primer `a` tal que `p(v, a)` es verdadero.

A continuación se muestra cómo ordenar una secuencia de enteros de forma ascendente o descendente.

Código

```
int arr[] = { 3, 1, 4, 1, 6 };
std::sort(&arr[0], &arr[0] + 5);                        // { 1, 1, 3, 4, 6 }
std::sort(&arr[0], &arr[0] + 5, [](int a, int b) {      // { 1, 1, 3, 4, 6 }
   return a < b;
});

std::sort(&arr[0], &arr[0] + 5, [](int a, int b) {      // { 6, 4, 3, 1, 1 }
   return a > b;
});
```

Los predicados también pueden ser funciones nombradas. A continuación se muestra cómo ordenar una secuencia de enteros ascendentemente, excepto que los pares deben aparecer antes que los impares:

Código

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

bool predicado(int a, int b) {
   if (a % 2 != b % 2) {
      return a % 2 < b % 2;    // primero el de menor paridad
   } else {
      return a < b;            // empate en paridad: primero el menor
   }
}

int main( ) {
   int arr[] = { 3, 1, 4, 1, 6 };
   std::sort(&arr[0], &arr[0] + 5, predicado);     // { 4, 6, 1, 1, 3 }
   for (int i = 0; i < 5; ++i) {
      std::cout << arr[i] << " ";
   }
}
```

Como los tipos `struct` recién definidos por el usuario no cuentan de antemano un operador `<`, intentar ordenarlos directamente no compilará. Sin embargo, podemos proporcionarle un predicado de ordenamiento a `std::sort` para que el algoritmo sepa cómo compararlos:

Código

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

struct fecha {
   int dia, mes, anyo;
};

bool predicado(fecha a, fecha b) {
   if (a.anyo != b.anyo) {
      return a.anyo < b.anyo;  // primero la de menor año
   } else if (a.mes != b.mes) {
      return a.mes < b.mes;    // empate en año: primero la de menor mes
   } else {
      return a.dia < b.dia;    // empate en año y mes: primero la de menor día
   }
}

int main( ) {
   fecha arr[] = { { 3, 3, 2020 }, { 20, 5, 1995 }, { 5, 12, 1995 } };
   std::sort(&arr[0], &arr[0] + 3, predicado);
   //  orden final: { { 20, 5, 1995 }, { 5, 12, 1995 }, { 3, 3, 2020 } }
   for (int i = 0; i < 3; ++i) {
      std::cout << arr[i].dia << " "
                << arr[i].mes << " "
                << arr[i].anyo << "\n";
   }
}
```

Cuando usamos un predicado para ordenar, debemos usar el mismo predicado al momento de buscar con búsqueda binaria. Si no hacemos esto, la búsqueda binaria no comprenderá la forma en la que están acomodados los elementos de la secuencia e incluso el programa podría morirse o trabarse.

Código

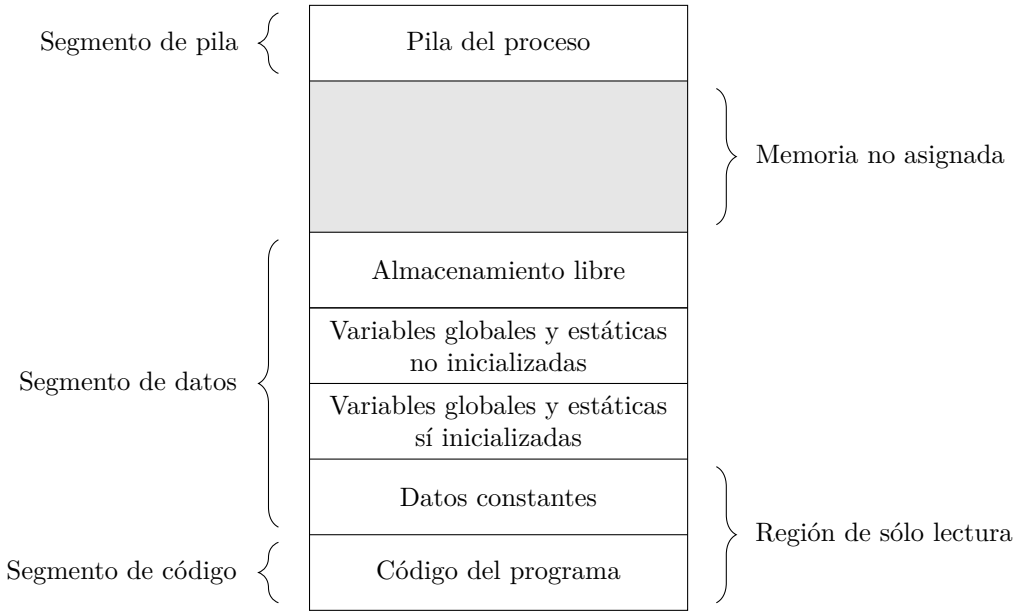
```
#include <algorithm>
#include <iostream>

bool predicado(int a, int b) {
   return a > b;    // primero el mayor (orden descendente)
}

int main( ) {
   int arr[] = { 3, 1, 4, 1, 6 };
   std::sort(&arr[0], &arr[0] + 5, predicado);     // ordenar con predicado
   bool b = std::binary_search(&arr[0], &arr[0] + 5, 3, predicado);
                                        // ¡buscar usando el mismo predicado!
}
```

10.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Ordenando-por-magnitud`.
2. Resuelve el problema `https://omegaup.com/arena/problem/Orden-raro-por-divisores`.
3. Resuelve el problema `https://omegaup.com/arena/problem/Ordenando-por-distancia-al-orige`.
4. Resuelve el problema `https://omegaup.com/arena/problem/Contando-fechas-en-intervalos`.

11. La memoria de un programa y el almacenamiento libre

Un programa cargado en memoria tiene una estructura relativamente compleja. Ya se ha hablado en secciones anteriores de la pila del proceso, pero ésta no es la única región de memoria de un programa. A continuación se muestra un diagrama que describe qué otras regiones existen.



Disposición de memoria de un programa en ejecución.

Que el código de un programa deba estar cargado en memoria, explica por qué un programa que usa `<iostream>` (que es una biblioteca bastante pesada) ocupa más memoria durante su ejecución que un programa que usa `<stdio.h>`. Con respecto a la pila del proceso, ya se ha dicho que esta región de memoria almacena los marcos de ejecución de las llamadas a función, junto con las variables locales de cada llamada. En esta sección se estudiará el segmento de datos, y en particular, el almacenamiento libre.

El lenguaje C++ permite declarar variables de sólo lectura con la palabra `const`, así como constantes en tiempo de compilación con la palabra `constexpr`. En general, el compilador decide si dichos valores estarán o no almacenados en la memoria del programa o si sólo se usarán durante la compilación. En todo caso, tanto las constantes como las variables originalmente declaradas de sólo lectura podrán ser almacenadas en la región del segmento de datos dedicada para ellas. Los sistemas operativos modernos detectan los intentos por modificar una valor almacenado en dicha región y previenen que esto ocurra.

Código	Diagnóstico

`constexpr int n = 5;`
`int arr[n];  // ¡ok!`
`n = 0;  // error de compilación`
 	
`const int m = rand( );`
`(int*)&m = 0;  // indefinido`
	`error: assignment of read-only variable`

Una variable global es una que está declarada fuera de toda función. Dicha variable es visible (y potencialmente modificable si no fue declarada `const`) por cualquier función que ya haya visto su declaración. A diferencia de las variables locales, una global que no sea explícitamente inicializada recibe el valor 0 de su tipo (es decir, `0` para `int`, `0.0` para `double`, el 0 de todos sus miembros para un `struct` y el 0 de todos sus elementos para un arreglo). Las variables globales se almacenan en el segmento de datos y una ventaja de esto es que el compilador sí apartará suficiente memoria para ellas. En los sistemas operativos donde la pila del proceso tiene una capacidad por omisión muy limitada, es frecuente que un arreglo local desborde la pila. Declarar un arreglo global incluso de 1 GB suele no provocar problemas.

Código	Salida

`#include <iostream>`
 	
`int arr[500000000];`
 	
`void f( ) {`
`  ++arr[0];`
`}`
 	
`int main( ) {`
`  f( ), f( ), f( );`
`  std::cout << arr[0];`
`}`
	3

Una variable declarada con la palabra `static` retiene su valor aún si la llamada a función en la que está declarada termina. En este sentido, una variable `static` realmente se almacena en el segmento de datos y no en el marco de la llamada que está en la pila del proceso. La memoria de las variables `static` se reserva de antemano, aunque la inicialización de la variable sólo ocurre la primera vez que el hilo de ejecución pasa por la declaración. Todas las llamadas a dicha función comparten la variable `static`.

Código	Salida

`#include <iostream>`
 	
`int f( ) {`
`  static int n = 1;`
`  return n++;`
`}`
 	
`int main( ) {`
`  std::cout << f( ) << "\n";`
`  std::cout << f( ) << "\n";`
`}`
	1
	2

La región más importante del segmento de datos es el almacenamiento libre (también llamado almacenamiento dinámico o memoria dinámica). Ésta es una región de memoria que puede crecer o decrecer durante la ejecución del programa y bajo solicitud del programador. El almacenamiento libre podrá disponer de toda la memoria principal con la que cuente la computadora.

En el lenguaje C++, el operador `new[]` nos permite solicitar memoria del almacenamiento libre para crear un arreglo, pero la memoria no migrará de un segmento a otro. Lo que el operador `new[]` regresa es la dirección del primer elemento del arreglo asignado. Esa dirección la podremos almacenar en un apuntador (sin importar si es local, global, etc) para así poder manipular la memoria de forma remota.

Código

```
int n;
std::cin >> n;
int* p = new int[n];
```

Un apuntador que vive en la pila apunta a un arreglo que vive en el almacenamiento libre.

Dependiendo de la cantidad de memoria que solicitamos, podremos desreferenciar los elementos `*p`, `*(p + 1)`, `*(p + 2)`, etc. Cuando `p` es un apuntador, el lenguaje C++ define la sintaxis `p[i]` como equivalente a `*(p + i)`, por lo que podemos usar la memoria solicitada como si fuera un arreglo.

Código	Entrada	Salida

`int n;`
`std::cin >> n;`
 	
`int* p = new int[n];`
`for (int i = 0; i < n; ++i) {`
`  std::cin >> p[i];`
`}`
 	
`std::reverse(&p[0], &p[0] + n);`
`for (int i = 0; i < n; ++i) {`
`  std::cout << p[i] << " ";`
`}`
	`5`
	`1 2 3 4 5`
		`5 4 3 2 1`

La memoria solicitada con `new[]` se retiene hasta que sea liberada con el operador `delete[]`. Al operador `delete[]` se le debe proporcionar un apuntador que apunte a la dirección devuelta por `new[]`. En el siguiente código, el primer ciclo asignará y liberará memoria de la pila en cada iteración (una variable local se destruye al llegar a la llave de cierre); el segundo ciclo pedirá memoria del almacenamiento libre y también la liberará en cada iteración con `delete[]`, pero el tercer ciclo pedirá memoria en cada iteración y no la liberará, por lo que el programa terminará abruptamente por falta de memoria.

Código	Diagnóstico

`for (int i = 0; i < 1000000; ++i) {`
`  int arr[100000];`
`}`
 	
`for (int i = 0; i < 1000000; ++i) {`
`  int* p = new int[100000];`
`  delete[] p;`
`}`
 	
`for (int i = 0; i < 1000000; ++i) {`
`  int* p = new int[100000];`
`}`
	`Error en ejecución`

Al fenómeno que ocurre en el tercer ciclo se le conoce como fuga de memoria. Se debe recalcar que el apuntador `p` que vive en la pila sí se crea y se destruye en cada iteración; lo que no se libera es la memoria dinámica a la que éste apunta. El apuntador `p` apuntará a un bloque distinto de memoria en cada iteración, ya que el anterior sigue en uso desde el punto de vista del asignador de memoria. Cuando el programa completo termina (ya sea de buena o de mala manera), el sistema operativo libera toda la memoria que el programa había solicitado y que aún no liberaba.

Para apartar y liberar elementos individuales en lugar de arreglos, se pueden emplear los operadores `new` y `delete` sin corchetes. No se debe usar `delete[]` para liberar memoria apartada con `new`, ni usar `delete` para liberar memoria apartada con `new[]`. Tampoco se debe liberar algo que ya fue liberado.

Código

```
int* p = new int;   // memoria para un elemento individual
*p = 123;
delete p;           // bien, no delete[]
delete p;           // ¡error! la memoria ya había sido liberada
```

En el archivo de biblioteca `<stdlib.h>` del lenguaje C también están declaradas las funciones `malloc` y `free` para solicitar y liberar memoria del almacenamiento libre, respectivamente. La gran diferencia

entre `new` y `malloc` es que `new` conoce el tipo de los elementos a construir, mientras que `malloc` no lo conoce y sólo aparta bytes. Por esta razón, debemos usar el operador `sizeof` para calcular la cantidad de bytes que necesitaremos, además de que `malloc` regresa un apuntador de tipo `void*`, el cual no puede usarse sin un moldeado explícito. El moldeado puede hacerse con el operador `static_cast` de C++.

Código

```
void* t = malloc(3 * sizeof(int));   // bytes suficientes para tres enteros
int* p = static_cast<int*>(t);       // un int* a partir del void*
p[0] = 5, p[1] = 8, p[2] = -3;
free(p);                             // liberar la memoria
```

En el ejemplo anterior, el moldeado también se puede realizar con la notación `(int*)t` que es la notación de moldeado de C. En C++, todo apuntador de tipo `T*` tiene una conversión implícita a un `void*` pero no al revés; de ahí la necesidad del moldeado explícito. La conversión implícita en el sentido opuesto sí existe en C y ésta constituye una de las pocas incompatibilidades que hay entre C y C++.

Sigue siendo un error liberar memoria ya liberada. Cuando `malloc` no puede satisfacer la solicitud de memoria, éste regresa un valor especial llamado apuntador nulo (`NULL` en C o `nullptr` en C++). Se puede usar un `if` después de una llamada a `malloc` para saber si la solicitud de memoria tuvo éxito.

Código

```
void* t = malloc(1000000000);
if (t == nullptr) {
   // no hubo suficiente memoria
} else {
   // sí hubo suficiente memoria
}
```

A continuación se presentan dos tablas que resumen las utilidades disponibles en C y C++. Se debe recordar que la biblioteca estándar de C también está disponible desde C++. Sin embargo, no se puede liberar con `free` memoria solicitada con `new` ni liberar con `delete` memoria solicitada con `malloc`.

Manipulación de memoria dinámica en C++	
Operador	**Acción realizada**
`new T[n]`	Aparta y construye un arreglo de `n` elementos de tipo `T` en el almacenamiento libre. Regresa un `T*` que apunta al primer elemento.
`new T`	Aparta y construye un elemento de tipo `T` en el almacenamiento libre. Regresa un `T*` que apunta al elemento.
`delete[] p;`	Destruye y libera un arreglo que fue reservado con `new[]` y cuyo primer elemento está siendo apuntado por `p`.
`delete p;`	Destruye y libera un elemento que fue reservado con `new` y que está siendo apuntado por `p`.

Manipulación de memoria dinámica en C (funciones de `stdlib.h`)	
Función de biblioteca	**Acción realizada**
`void* malloc(size_t n);`	Intenta asignar un bloque de `n` bytes y regresar un apuntador al mismo. Regresa `nullptr` en caso de error. El tipo `size_t` suele ser sinónimo de `unsigned long`.
`void* free(void* p);`	Libera un bloque de memoria reservado con `malloc`.

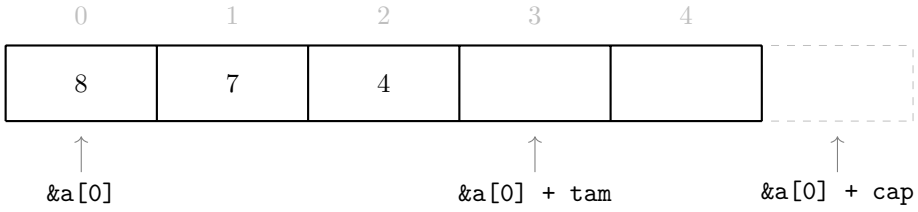
Existen funciones adicionales en C (`aligned_alloc`, `calloc` y `realloc`) que también manipulan la memoria del almacenamiento libre. Se puede consultar sobre ellas en `https://en.cppreference.com/w/c/memory`. Sin embargo, no se recomienda usar las funciones de gestión de memoria dinámica de C en C++, salvo en casos muy especiales en los que el programador sepa exactamente lo que está haciendo. En el lenguaje C++, se sugiere usar exclusivamente los operadores disponibles.

12. Arreglos como contenedores

Hasta el momento, los arreglos han jugado el rol de secuencias de tamaño fijo. Es decir, se decide la capacidad del arreglo, se le asignan valores a sus elementos y se procesa el arreglo de una u otra forma. Hasta el momento, no hemos permitido que el usuario agregue o quite elementos de un arreglo *a posteriori*. En esta sección se estudiará distintas formas de implementar el cómo permitir esto.

A partir de este momento, haremos una distinción entre la *capacidad* y el *tamaño* de un arreglo. La capacidad de un arreglo se definirá como la cantidad máxima de elementos que éste puede almacenar. El tamaño de un arreglo será entonces la cantidad de elementos activos o que en realidad estamos usando. Cuando hablemos de agregar, quitar o consultar elementos de un arreglo, siempre nos referiremos a los elementos activos. Buscaremos implementar las siguientes operaciones: agregar un elemento al final de un arreglo, quitar el último elemento de un arreglo e imprimir el i-ésimo elemento de un arreglo.

En nuestras implementaciones, supondremos que tenemos dos enteros `tam` y `cap` que denotan el tamaño y la capacidad del arreglo, respectivamente. El arreglo está vacío si `tam == 0` y está lleno si `tam == cap`.



Un arreglo `a` de tamaño 3 y capacidad 5.

Existen tres situaciones generales a considerar:

- La capacidad máxima del arreglo se conoce en tiempo de compilación.

  En este caso, podemos declarar `cap` como `constexpr` y el arreglo puede ser un arreglo local en la pila. Si el arreglo no está lleno, entonces `tam` es, además del tamaño actual, la siguiente posición válida sobre la cual agregar un elemento. Una vez agregado el elemento, debemos incrementar `tam`. Podemos simular que quitamos el último elemento del arreglo simplemente decrementando `tam`. El acceso al i-ésimo elemento de un arreglo es trivial.

Código

```
constexpr int cap = 5;
int a[cap], tam = 0;

// agregar elementos al final del arreglo
a[tam++] = 8;
a[tam++] = 7;
a[tam++] = 4;
a[tam++] = 9;

// quitar el último elemento del arreglo
--tam;

// imprimir el arreglo
for (int i = 0; i < tam; ++i) {
   std::cout << a[i] << " ";      // 8 7 4
}
```

- La capacidad máxima del arreglo la proporciona el usuario.

  En este caso, debemos declarar `cap` como una variable ordinaria y debemos usar el almacenamiento libre para apartar la memoria del arreglo. Sin embargo, el resto de la implementación es muy similar.

Código

```
int cap;
std::cin >> cap;                    // suponer que cap = 5
int* a = new int[cap], tam = 0;     // asignar memoria para el arreglo

// agregar elementos al final del arreglo
a[tam++] = 8;
a[tam++] = 7;
a[tam++] = 4;
a[tam++] = 9;

// quitar el último elemento del arreglo
--tam;

// imprimir el arreglo
for (int i = 0; i < tam; ++i) {
   std::cout << a[i] << " ";        // 8 7 4
}

delete[] a;                         // liberar la memoria del arreglo
```

- La capacidad máxima del arreglo no se conoce de antemano.

  Usaremos el almacenamiento libre como en el caso anterior, pero no hay ninguna garantía de que el bloque de memoria apartado inicialmente sea suficiente para almacenar todos los elementos que el usuario desee agregar. Desafortunadamente y salvo en casos excepcionales, no es posible simplemente agrandar un arreglo. La estrategia será la siguiente: cada vez que el usuario desee agregar un elemento y el arreglo se encuentre lleno, apartaremos un nuevo arreglo del doble de capacidad, copiaremos el contenido del arreglo viejo al nuevo y nos quedaremos con el nuevo arreglo, liberando el viejo.

Código

```
int cap = 1                         // capacidad inicial
int* a = new int[cap], tam = 0;     // asignar memoria para el arreglo
auto asegura_capacidad = [&]( ) {   // rutina para asegurar que hay capacidad
   if (tam == cap) {
      cap *= 2;
      int* t = new int[cap];
      std::copy(&a[0], &a[0] + tam, &t[0]);
      delete[] a, a = t;
   }
};

// agregar elementos al final del arreglo
asegura_capacidad( ), a[tam++] = 8;
asegura_capacidad( ), a[tam++] = 7;
asegura_capacidad( ), a[tam++] = 4;
asegura_capacidad( ), a[tam++] = 9;

// quitar el último elemento del arreglo
--tam;

// imprimir el arreglo
for (int i = 0; i < tam; ++i) {
   std::cout << a[i] << " ";        // 8 7 4
}

delete[] a;                         // liberar la memoria del arreglo
```

A la tercera implementación se le suele llamar arreglo redimensionable o dinámico. Además, la política de duplicar la capacidad del arreglo no es casualidad. Cada vez que un arreglo de n elementos se llena, necesitamos relocalizar o copiar todos sus elementos a un arreglo nuevo, lo cual es costoso. Duplicar la capacidad del arreglo con cada relocalización (o mejor dicho, hacer crecer la capacidad del arreglo de forma exponencial) garantiza que debemos relocalizar con cada vez menor frecuencia. Por ejemplo, la cantidad total de escrituras de elementos tras agregar 64 elementos está dada por $64 + r$ donde $r = 1 + 2 + 4 + 8 + 16 + 32 = 63$ es la suma de las escrituras hechas en las seis relocalizaciones. Se puede inferir que la cantidad total de escrituras realizadas tras agregar n elementos será aproximadamente $2n$, lo que en promedio son dos escrituras por elemento. Una política que aumente la capacidad del arreglo sólo en una constante provocaría que el costo de relocalización se incurra con mucha frecuencia y la cantidad de escrituras necesarias para insertar n elementos sea una función cuadrática.

La biblioteca de C++ ya provee de una implementación de arreglos redimensionables en `<vector>`. La plantilla `std::vector` nos permite declarar arreglos redimensionables de cualquier tipo y además se encarga por completo del manejo de la memoria dinámica, por lo que es innecesario usar `new` o `delete`. La sintaxis del uso de `std::vector` se asemeja a la de un `struct` con funciones miembro.

Código

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

int main( ) {
   std::vector<int> a;                 // arreglo inicialmente vacío

   // agregar elementos al final del arreglo
   a.push_back(8);
   a.push_back(7);
   a.push_back(4);
   a.push_back(9);

   // quitar el último elemento del arreglo
   a.pop_back( );

   // imprimir el arreglo
   for (int i = 0; i < a.size( ); ++i) {
      std::cout << a[i] << " ";        // 8 7 4
   }
}
```

La función miembro `push_back` permite agregar un elemento al final del arreglo, mientras que la función miembro `pop_back` permite quitar el último elemento del arreglo. La función miembro `size` permite obtener el tamaño actual del arreglo mientras que la función miembro `capacity` permite obtener la capacidad actual del bloque de memoria subyacente. El acceso a los elementos de un `std::vector` se puede hacer con la notación usual de corchetes. Cuando el ámbito de un `std::vector` termina, éste se destruye y libera automáticamente la memoria que esté usando del almacenamiento libre. También es posible inicializar un `std::vector` con una secuencia e incluso ordenarlo de la forma usual:

Código

```
std::vector<int> a = { 3, 1, 4, 1, 6 };     // inicializar el arreglo
std::sort(&a[0], &a[0] + a.size( ));        // ordenar el arreglo
```

Las contenedores de la biblioteca de C++ se pueden copiar y comparar con los operadores `=` y `==`, respectivamente. La copia del contenedor es completa: se aparta una nueva región de memoria y se copian todos los elementos del contenedor preexistente al nuevo. Una vez terminada la copia, ambos contenedores son iguales en valor pero independientes en memoria. Con esto, la biblioteca mantiene la semántica de copia por valor que caracteriza a los tipos escalares nativos.

Código

```
std::vector<int> v1(10, -1);   // arreglo con diez -1 iniciales
std::vector<int> v2 = v1;     // hacer una copia del arreglo
if (v1 == v2) {
   std::cout << "Los arreglos son iguales\n";
}

v2.pop_back( );               // quitar de v2, v1 permanece inalterado
std::cout << v1.size( ) << " " << v2.size( );     // 10 y 9 respectivamente
```

Es posible pasar un contenedor por valor y por referencia. Se puede usar el paso por referencia a `const` para permitir que una función inspeccione la estructura de datos original pero sin poder modificarla. Esto evita la copia del paso por valor, que puede resultar muy costosa en tiempo y memoria.

Código

```
void fv(std::vector<int> arr) {   // paso por valor: copiamos el arreglo
   arr.pop_back( );               // modificamos la copia, no el original
}
void fr(std::vector<int>& arr) {  // paso por referencia: tomamos el original
   arr.pop_back( );               // modificamos el original
}
void fc(const std::vector<int>& arr) {      // paso por referencia a const
   arr.pop_back( );               // error, no podemos modificar el original
   std::cout << arr[0];           // ok
}

int main( ) {
   std::vector<int> arr = { 1, 2, 3 };
   fv(arr);
   std::cout << arr.size( );     // sigue siendo 3
   fr(arr);
   std::cout << arr.size( );     // ahora vale 2
}
```

Siguiendo el mismo estilo, el archivo `<string>` de la biblioteca de C++ ofrece el tipo `std::string`, el cual implementa una cadena usando memoria dinámica. Con este tipo, se vuelve innecesario conocer de antemano la memoria requerida para almacenar la cadena. Las cadenas se pueden copiar y comparar.

Código

```
#include <iostream>
#include <string>

int main( ) {
   std::string s;
   std::cin >> s;

   if (s == "hola") {
      std::cout << "La cadena vale hola\n";
   } else if (s < "hola") {    // ¿es lexicográficamente menor que hola?
      std::string t = s;       // copiar la cadena en otra
   }
}
```

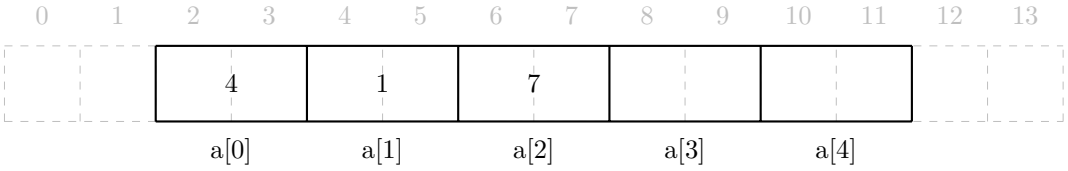
12.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/caracteres_subcadenas`.

13. Arreglos como estructuras de datos

Una *estructura de datos* es una forma de organizar datos en memoria, tal que dicha organización satisface un conjunto de propiedades y permite implementar eficientemente un conjunto de operaciones. En este sentido, un *arreglo* es una estructura de datos que almacena n elementos del mismo tipo y que:

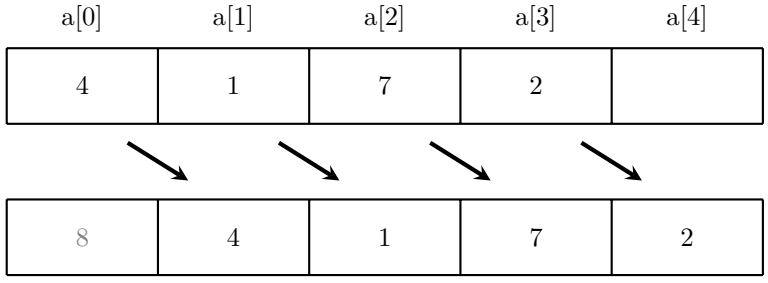
- Consume una cantidad óptima de memoria: si queremos almacenar n elementos de b bytes cada uno, entonces el arreglo consume $n \times b$ bytes. No hay huecos entre elementos.
- Permite calcular la dirección del i-ésimo elemento de un arreglo en tiempo constante a partir de la dirección inicial del arreglo: si la dirección del arreglo es d (en bytes), el i-ésimo elemento tiene dirección $d + (i \times b)$, donde b es el número de bytes por elemento. Los elementos son contiguos en memoria y consecutivos según su índice. El tiempo teórico para accesar a cualquiera de los elementos es el mismo.
- Permite agregar o quitar elementos del extremo derecho del arreglo en tiempo constante: si hay capacidad suficiente, los demás elementos pueden permanecer en su lugar durante la operación.



Un arreglo `short a[5] = { 4, 1, 7 };` asignado a partir de la dirección 2. Hay suficiente capacidad para asignarles valores explícitos a los últimos elementos sin relocalizar los demás.

Un arreglo es la única estructura de datos que tiene las propiedades anteriores y es la estructura de datos por excelencia cuando no deseamos agregar o eliminar elementos de forma arbitraria. Además, existen muchas estructuras de datos que son variantes de arreglos, siendo la más importante de ellas el *arreglo ordenado*. Esta estructura de datos cumple con las propiedades de un arreglo y además cumple con que el elemento i es menor o igual al elemento $i+1$. Esto nos permite encontrar el mínimo y el máximo del arreglo en tiempo constante (ya que corresponden con el primer y el último elemento, respectivamente), además de que ahora podemos usar el algoritmo de búsqueda binaria para buscar eficientemente valores en el arreglo. Sin embargo, no podemos modificar arbitrariamente el valor de los elementos de un arreglo ordenado o agregar elementos arbitrarios a la derecha del arreglo sin el temor de invalidar la propiedad de orden que caracteriza a un arreglo ordenado.

Dos operaciones que no son eficientes en un arreglo son agregar o quitar elementos por la izquierda, ya que ambas necesitan recorrer todos los demás elementos del arreglo, por lo que tardan tiempo lineal:



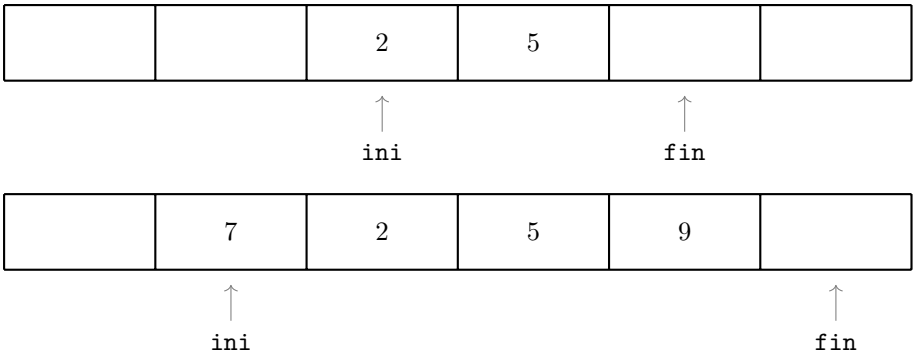
Agregar un elemento al inicio de un arreglo obliga a los elementos a recorrerse.
Quitar el primer elemento realiza el proceso contrario y los elementos también se recorren.

Los arreglos no son las únicas estructuras de datos que existen. Las siguientes secciones describen estructuras de datos que proveen operaciones eficientes distintas a las de un arreglo. Sin embargo, en computación no existe la estructura de datos perfecta: siempre que ideemos la forma de realizar alguna operación eficientemente, debemos sacrificar eficiencia en algún otro aspecto de la estructura de datos.

14. Dobles colas

Una doble cola (*double-ended queue* en inglés) es una estructura de datos que permite realizar eficientemente las siguientes operaciones: agregar o quitar elementos por la derecha, agregar o quitar elementos por la izquierda, consultar el tamaño de la doble cola y accesar al i-ésimo elemento de la doble cola (el primer elemento es el de más a la izquierda y el último elemento es el de más a la derecha).

Un lector astuto habrá notado que, en un arreglo, las inserciones por la derecha sólo son posibles en tiempo constante cuando hay capacidad de sobra por la derecha. En realidad, no hay ninguna razón por la cual no podamos usar el mismo truco por la izquierda.

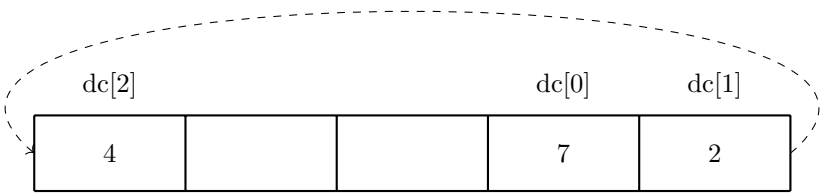


Si almacenamos una secuencia a la mitad de un bloque de memoria,
podemos simular las inserciones por ambos extremos usando la capacidad sobrante.

Siguiendo la idea anterior, podemos posicionar los apuntadores `ini` y `fin` a la mitad del bloque de memoria y podemos implementar las operaciones de la siguiente forma:

Doble cola con capacidad en ambos extremos	
Operación	**Implementación**
Agregar el valor `v` por la derecha.	`*fin++ = v;`
Agregar el valor `v` por la izquierda.	`*--ini = v;`
Quitar el último elemento.	`--fin;`
Quitar el primer elemento.	`++ini;`
Calcular el tamaño de la doble cola.	`int t = fin - ini;`
Accesar al i-ésimo elemento de la doble cola.	`int v = ini[i];`

La implementación anterior tiene dos problemas importantes. En primer lugar, necesita apartar una cantidad considerable de memoria para tener capacidad en ambos lados. En segundo lugar, no hay ninguna garantía de que los apuntadores permanezcan en el centro del bloque aún si la doble cola está vacía: si se agregan elementos a la derecha (avanzando `fin`) y se quitan elementos por la izquierda (avanzando `ini`) entonces la doble cola queda vacía pero los apuntadores ahora están cargados hacia la derecha. Una forma de resolver ambos problemas es emplear una estrategia circular: si la capacidad del lado derecho se acaba pero aún hay capacidad del lado izquierdo, entonces se usa ésa.



Una doble cola `dc` de tamaño 3 y capacidad 5. Los elementos de la doble cola son
`{ 7, 2, 4 }` y se almacenan circularmente a partir de cierta posición del bloque de memoria.

A la estrategia anterior se le conoce como arreglo circular. Un pequeño inconveniente es que no resulta natural implementarlo usando apuntadores, pero se pueden usar enteros y aritmética módular con respecto a la capacidad del bloque de memoria subyacente. La aritmética modular agrega sobrecarga a las operaciones básicas que coinciden con las que están disponibles en un arreglo, aunque dicha sobrecarga se tolera cuando la necesidad de agregar o quitar elementos al inicio de la secuencia es real. Un inconveniente no tan menor es la pérdida de la propiedad de que los elementos de la secuencia sean contiguos en memoria, lo cual puede ser una desventaja con respecto a los arreglos en algunas situaciones.

A continuación se describe una implementación que hace uso de un arreglo `a` de capacidad `cap` y de dos variables enteras `ini` y `tam`. Una doble cola vacía comienza con `ini = 0` y `tam = 0`.

Doble cola como un arreglo circular	
Operación	**Implementación**
Agregar el valor `v` por la derecha.	`a[(ini + tam) % cap] = v;` `++tam;`

Agregar el valor `v` por la izquierda.	`ini = (ini - 1 + cap) % cap;`
	`a[ini] = v`
	`++tam;`

Quitar el último elemento.	`--tam;`
Quitar el primer elemento.	`ini = (ini + 1) % cap;` `--tam;`
Calcular el tamaño de la doble cola.	`int t = tam;`
Accesar al i-ésimo elemento de la doble cola.	`int v = a[(ini + i) % cap];`

Afortunadamente, la biblioteca estándar de C++ ya implementa una doble cola que además usa memoria dinámica, por lo que no tiene limitaciones de capacidad. Dicha implementación se encuentra definida en el archivo `<deque>` bajo el nombre `std::deque` y su uso es muy similar al de `std::vector`.

Código

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

int main( ) {
   std::deque<int> dc;                // doble cola inicialmente vacía

   // agregar elementos al final de la doble cola
   dc.push_back(2);
   dc.push_back(4);
   dc.push_back(9);

   // agregar elementos al inicio de la doble cola
   dc.push_front(7);
   dc.push_front(5);

   // quitar el último elemento de la doble cola
   dc.pop_back( );

   // quitar el primer elemento de la doble cola
   dc.pop_front( );

   // imprimir el arreglo
   for (int i = 0; i < dc.size( ); ++i) {
      std::cout << dc[i] << " ";        // 7 2 4
   }
}
```

La función miembro `push_back` permite agregar un elemento al final de la doble cola, mientras que la función miembro `pop_back` permite quitar el último elemento. La función miembro `push_front` permite agregar un elemento al inicio de la doble cola, mientras que la función `pop_front` permite quitar el primer elemento. La función miembro `size` permite obtener el tamaño actual de la doble cola. El acceso a los elementos de un `std::deque` se puede hacer con la notación usual de corchetes. Un `std::deque` también se encarga de solicitar y liberar la memoria del almacenamiento libre automáticamente.

Dos funciones miembro de `std::deque` que también están disponibles en `std::vector` son `front` y `back`, las cuales regresan el primer y el último elemento, respectivamente. En ese sentido, `dc.front( )` es equivalente a `dc[0]` y `dc.back( )` es equivalente a `dc[dc.size( ) - 1]`. Dado que las funciones `pop_front` y `pop_back` no regresan el valor del elemento que quitan, suele ser común primero copiarlos en variables auxiliares antes de quitarlos de la estructura de datos.

Una consecuencia de la falta de contigüidad de memoria en una doble cola, es que el siguiente fragmento de código para ordenar una doble cola puede provocar un error de ejecución:

Código

```
std::deque<int> dc = { 3, 1, 4, 1, 6 };     // inicializar la doble cola
std::sort(&dc[0], &dc[0] + dc.size( ));     // ¡error en ejecución!
```

La biblioteca estándar de C++ provee de una solución que se estudiará en la siguiente sección.

14.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Dobles-colas`.

2. Resuelve el problema `https://omegaup.com/arena/problem/banco-clientes-no-preferentes`.

15. Pilas, colas, iteradores y abstracción

En la sección anterior se presentaron dos implementaciones distintas de la estructura de datos llamada doble cola. Lo mismo ocurre con otras estructuras de datos: generalmente siempre hay más de una posible implementación. Cuando hablemos de una estructura de datos en términos *abstractos*, nos limitaremos a listar las propiedades y operaciones que la estructura de datos debe proveer. Cuando hablemos de una estructura de datos en términos *concretos*, indicaremos una estrategia de implementación.

Una pila es una estructura de datos que permite realizar eficientemente las siguientes operaciones: agregar o quitar elementos del mismo extremo, consultar el elemento actual en ese extremo y consultar el tamaño de la pila. En una pila, si agregamos una secuencia de valores y luego los extraemos, el orden que se produce se denomina *último en entrar, primero en salir* (*LIFO* por sus siglas en inglés). Una cola es una estructura de datos que permite realizar eficientemente las siguientes operaciones: agregar elementos en un extremo, consultar y quitar elementos en el extremo opuesto y consultar el tamaño de la cola. En una cola, si agregamos una secuencia de valores y luego los extraemos, el orden que se produce se denomina *primero en entrar, primero en salir* (*FIFO* por sus siglas en inglés). Estas dos definiciones son abstractas, y con lo visto en las secciones anteriores ya contamos con implementaciones para ambas.

La política de la biblioteca estándar de C++ es sólo proporcionar una sintaxis cómoda para operaciones eficientes. El tipo `std::vector` es una posible implementación de pilas, ya que éste ofrece las funciones `push_back` y `pop_back` para agregar o quitar elementos del mismo extremo, el operador `[]` o la función `back` para consultar el elemento en dicho extremo y la función `size` para consultar el tamaño de la secuencia. El tipo `std::deque` también es una posible implementación de pilas. Por otra parte, `std::vector` no puede implementar una cola porque `pop_front` no se provee al ser ineficiente en arreglos. En contraste, `std::deque` sí es una posible implementación para colas ya que provee `push_back`, `pop_front`, `size` y el operador `[]` junto con la función `front` para accesar al primer elemento.

La biblioteca estándar de C++ ofrece los adaptadores `std::stack` y `std::queue` para declarar pilas y colas, respectivamente. Estas utilidades están declaradas en los archivos de biblioteca `<stack>` y `<queue>`. Ambos adaptadores usan `std::deque` internamente ya que esta estructura puede implementar las otras dos, pero estos adaptadores sólo proveen las funciones propias de cada estructura de datos bajo nombres de función simplificados y más usados en la literatura. A continuación se muestra un ejemplo de su uso.

Código

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

int main( ) {
   std::stack<int> pila;              // internamente es std::deque
   pila.push(1);                      // push_back(1) en std::deque
   std::cout << pila.top( );          // back( ) en std::deque
   pila.pop( );                       // pop_back( ) en std::deque

   std::queue<int> cola;              // internamente es std::deque
   cola.push(1);                      // push_back(1) en std::deque
   std::cout << cola.front( );        // front( ) en std::deque
   cola.pop( );                       // pop_front( ) en std::deque
}
```

No se recomienda usar `std::stack` y `std::queue` mas que en situaciones muy peculiares, ya que estos adaptadores no ofrecen el operador `[]` y entonces no se pueden imprimir las secuencias almacenadas sin extraer los elementos de las estructuras, destruyéndolas en el proceso.

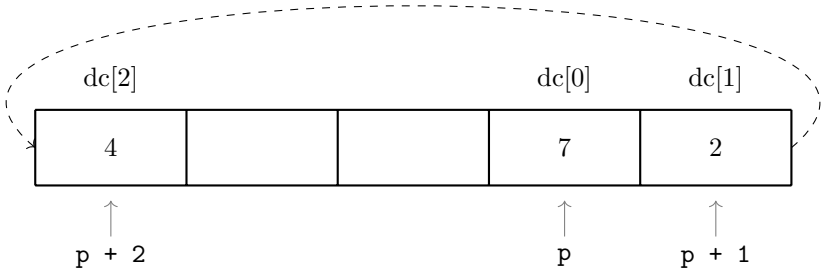
Las estructuras de datos no son las únicas entidades que pueden describirse en términos abstractos. Un *iterador* es un tipo de dato abstracto que puede señalar y visitar un elemento. Además, los iteradores se clasifican dependiendo de qué operaciones adicionales pueden proveer eficientemente. Por ejemplo:

- Iteradores hacia adelante: Pueden avanzar al siguiente elemento de una secuencia.
- Iteradores bidireccionales: Pueden avanzar o retroceder sobre los elementos de una secuencia.
- Iteradores de acceso arbitrario: Pueden saltar arbitrariamente sobre los elementos de una secuencia.
- Iteradores contiguos: Pueden saltar arbitrariamente sobre los elementos de una secuencia contigua.

Los apuntadores son implementaciones concretas de iteradores contiguos. Sin embargo, una doble cola no necesariamente es contigua en memoria, por lo que es un error recorrerla mediante apuntadores. La biblioteca estándar de C++ proporciona iteradores especializados para recorrer un `std::deque`, los cuales conocen la estructura interna de la doble cola y avanzan de la manera correcta.

Código

```
std::deque<int> dc = { 7, 2, 4 };          // inicializar la doble cola
std::deque<int>::iterator p = dc.begin( ); // iterador al inicio
std::cout << *p << " ";                    // 7
++p;
std::cout << *p << " ";                    // 2
++p;
std::cout << *p << " ";                    // 4
```



Un `std::deque<int>::iterator` sabría "dar la vuelta" al iterar sobre una doble cola implementada con un arreglo circular.

La función miembro `begin` regresa un iterador especializado de tipo `std::deque<T>::iterator`, el cual apunta al inicio de la secuencia almacenada en la doble cola. Este iterador desempeña el mismo rol que un apuntador desempeña sobre un arreglo, pero es capaz de manejar de forma transparente y correcta la no contigüidad de `std::deque`. Una doble cola se puede ordenar como sigue:

Código

```
std::deque<int> dc = { 3, 1, 4, 1, 6 };
std::sort(dc.begin( ), dc.begin( ) + dc.size( ));     // ¡ok!
std::sort(dc.begin( ), dc.end( ));                    // ¡ok! equivalente
```

La función miembro `end` regresa directamente un `std::deque<T>::iterator` que apunta al fin de la secuencia almacenada en la doble cola. Como siempre, el iterador al fin no debe desreferenciarse. Ya que el tipo `std::deque<T>::iterator` es largo de escribir, se suele usar `auto` para evitar tener que escribirlo:

Código

```
std::deque<int> dc = { 3, 1, 4, 1, 6 };
auto ini = dc.begin( ), fin = dc.end( );
std::sort(ini, fin);                        // ¡ok! ordenar la doble cola
for (int i = 0; i < dc.size( ); ++i) {      // imprimir mediante índices
   std::cout << dc[i] << " ";
}
for (auto p = ini; p != fin; ++p) {         // imprimir mediante iteradores
   std::cout << *p << " ";
}
```

El tipo `std::vector` también provee las funciones miembro `begin` y `end`, las cuales regresan iteradores de tipo `std::vector<T>::iterator`. Este tipo es formalmente incompatible con apuntadores. A su vez, C++ también provee una variante del ciclo `for` que recorre el contenedor del inicio al fin y que desreferencia automáticamente el elemento actual en cada iteración.

Código

```
std::vector<int> arr = { 3, 1, 4, 1, 6 };
int* p1 = &arr[0];          // ok
int* p2 = arr.begin( );     // error: std::vector<int>::iterator no es int*
auto p4 = arr.begin( );     // ok y preferible

for (int actual : v) {      // iterar de inicio a fin
   std::cout << actual << " ";
}
```

Una de las razones de lo anterior es que el sistema de tipos de C++ es nominal y no estructural: aunque dos tipos distintos sean estructuralmente equivalentes (y `std::vector<T>::iterator` es estructuralmente equivalente a `T*`), el lenguaje C++ no permite conversiones entre ellos.

Código

```
struct s1 {
   int a;
};
struct s2 {
   int a;
};

s1 v1 = { 5 };
s2 v2 = v1;      // error
```

El tipo `std::string` también ofrece las funciones `begin` y `end` para obtener iteradores al inicio y al fin de la cadena. Esto permite usarlo de forma muy similar al del resto de los contenedores de la biblioteca.

Código	Entrada	Salida

`std::string s;`
`std::cin >> s;`
`auto p = std::find(s.begin( ),`
`  s.end( ), '@');`
`if (p != s.end( )) {`
`  *p = '#';`
`}`
`std::cout << s;`
	`hola@gatito`	`hola#gatito`

15.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Sumando-con-pilas`.
2. Resuelve el problema `https://omegaup.com/arena/problem/grupos`.

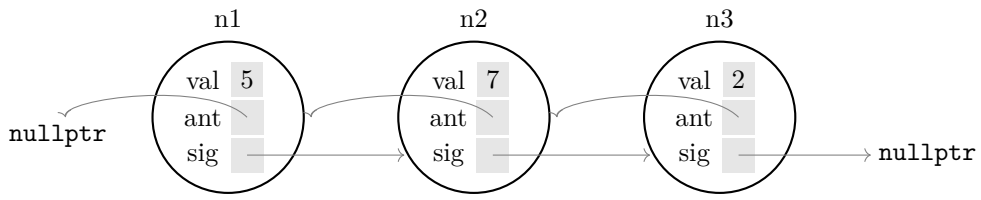
16. Listas enlazadas

Ninguna de las estructuras de datos vistas hasta el momento puede insertar o eliminar elementos en cualquier lugar de la secuencia de forma eficiente; siempre lo hemos hecho en los extremos. A su vez, cortar una secuencia (y obtener dos a cambio) o concatenar secuencias son operaciones que tampoco son eficientes en arreglos o dobles colas. Las listas enlazadas son estructuras de datos que permiten realizar las operaciones anteriores eficientemente, pero siempre sacrificando algo a cambio.

Una lista enlazada está formada por nodos conectados mediante apuntadores. Los nodos representan los elementos de la secuencia. En una lista doblemente enlazada, cada nodo está conectando al nodo anterior y al nodo siguiente de la lista, mientras que en una lista simplemente enlazada sólo hay conexiones en un sentido. Cuando hablemos de listas enlazadas sin especificar la variante, nos referiremos a listas doblemente enlazadas. Los nodos generalmente se implementan con un tipo `struct`.

Código

```
struct nodo {
   int val;
   nodo* ant;    // el apuntador al anterior de la secuencia
   nodo* sig;    // el apuntador al siguiente de la secuencia
};

int main( ) {
   nodo n1, n2, n3;
   n1.val = 5;
   n1.ant = nullptr, n1.sig = &n2;
   n2.val = 7;
   n2.ant = &n1, n2.sig = &n3;
   n3.val = 2;
   n3.ant = &n2, n3.sig = nullptr;
}
```



La lista enlazada denota la secuencia 5 7 2. No hay elementos antes del primero ni después del último.

Gracias a la estructura que poseen las listas enlazadas, se pueden realizar cambios locales a una lista simplemente ajustando los apuntadores entre elementos, sin tener que recorrer o relocalizar en memoria ninguno de los elementos. Más aún, los nodos en la lista enlazada pueden vivir en regiones completamente distintas en memoria y aún así la lista puede construirse.

Código

```
static nodo n1;         // nodo en el segmento de datos (variables estáticas)
nodo n2;                // nodo en la pila del proceso
nodo* p3 = new nodo;    // nodo en el almacenamiento libre, p3 apunta a él
n1.val = 5, n1.ant = nullptr, n1.sig = &n2;
n2.val = 7, n2.ant = &n1, n2.sig = p3;
(*p3).val = 2, (*p3).ant = &n2, (*p3).sig = nullptr;
```

En el ejemplo anterior, los paréntesis de la expresión `(*p3).val` son necesarios desafortunadamente. Esto es debido a que la expresión `*p3.val` se interpretaría como intentar accesar al miembro `val` del apuntador y para después desreferenciarlo, lo cual es inválido. Para simplificar la notación, los lenguajes C y C++ definen la notación `p->miembro` como equivalente a `(*p).miembro`.

Aunque para eliminar un nodo de una lista enlazada basta desconectarlo de la misma, generalmente también deseamos liberar la memoria que éste ocupa. Es por esta razón que la mayoría de las implementaciones de listas enlazadas apartan todos sus nodo en el almacenamiento libre uno a uno, de modo que se vuelve posible liberarlos individualmente. Cabe mencionar que el `sizeof` de un `struct nodo` con un valor de tipo `int` puede valer 12 o 24 bytes dependiendo del procesador y del sistema operativo. Si comparamos dicho valor con `sizeof(int)` que suele ser 4, la sobrecarga en memoria de una secuencia almacenada en una lista enlazada es mucho mayor que si simplemente la almacenamos en un arreglo.

Si contamos con un apuntador al nodo inicial de una lista enlazada que termina en `nullptr`, entonces podemos iterar sobre ella como se muestra a continuación:

Código

```
nodo* ini;
// crear la lista y asignar ini para que apunte al primer nodo
for (nodo* p = ini; p != nullptr; p = p->sig) {
   std::cout << p->valor << " ";
}
```

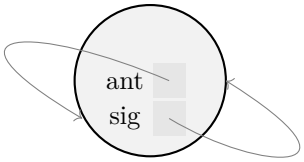
En el código anterior, la sentencia `p = p->sig` se interpreta como sigue: si actualmente estamos parados en el nodo `*p`, entonces `p->sig` corresponde al apuntador al siguiente nodo de la lista enlazada. Queremos hacer que `p` ahora apunte a dicho siguiente nodo. Si todos los nodos de una lista enlazada fueron apartados individualmente en el almacenamiento libre, entonces podemos eliminar y liberar un nodo que guarde cierto valor `v` de la siguiente forma.

Código

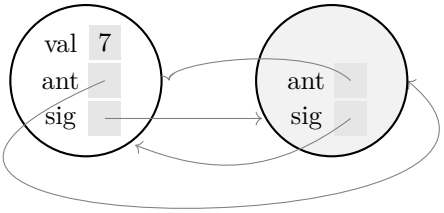
```
for (nodo* p = ini; p != nullptr; p = p->sig) {
   if (p->val == v) {
      if (p->ant != nullptr) {   // si hay un nodo antes, ajustamos su sig
         p->ant->sig = p->sig;
      }
      if (p->sig != nullptr) {   // si hay un nodo adelante, ajustamos su ant
         p->sig->ant = p->ant;
      }
      if (ini == p) {            // si borraremos el primer nodo, ajustamos ini
         ini = p->sig;
      }
      delete p;
      break;
   }
}
```

La implementación anterior no es la más fácil, porque nos obliga a tener cuidado con los apuntadores nulos o con los nodos que se invalidan al ser liberados. Una implementación que no tiene ninguno de estos inconvenientes es el de una lista doblemente enlazada *circular*, la cual tiene un nodo centinela que denota el fin de la lista. El nodo centinela tiene la propiedad de apuntar al último nodo de la lista con su apuntador `ant` y al primer nodo de la lista con su apuntador `sig`. Una lista enlazada vacía se ve como un apuntador centinela que se apunta a sí mismo con sus dos apuntadores.

Lista enlazada circular vacía



Lista enlazada circular con un elemento



Una lista enlazada circular con centinela no tiene apuntadores nulos. Además, el nodo centinela nos permite accesar al primer y al último elemento de la lista enlazada.

Implementaremos dos funciones que permiten insertar o eliminar un elemento en cualquier lugar de una lista enlazada circular. En particular, la función `insert` crea un nuevo nodo antes de uno preexistente, mientras que la función `erase` elimina un nodo preexistente que no es el centinela.

Código

```
struct lista {
   nodo centinela = { -1, &centinela, &centinela };
};

void insert(lista& li, nodo* p, int v) {    // insertar antes de p
   nodo* nuevo = new nodo;
   nuevo->val = v, nuevo->sig = p, nuevo->ant = p->ant;
   p->ant->sig = nuevo, p->ant = nuevo;
}

void erase(lista& li, nodo* p) {           // eliminar p
   p->ant->sig = p->sig, p->sig->ant = p->ant;
   delete p;
}
```

También definiremos funciones que regresan respectivamente un apuntador al inicio y al fin de la lista enlazada, donde inicio es un apuntador al primer nodo y el fin apunta al nodo centinela. Una vez más, tenemos que la lista está vacía si el inicio coincide con el fin.

Código

```
nodo* inicio(lista& li) {     // el siguiente del centinela es el primero
   return li.centinela.sig;
}

nodo* fin(lista& li) {        // el centinela es el fin de los elementos reales
   return &li.centinela;
}
```

Además, las funciones `push_back`, `push_front`, `pop_back` y `pop_front` pueden implementarse en términos de `insert` e `erase` de la siguiente forma:

Código

```
void push_back(lista& li, int v) {     // insertar antes del fin
   insert(li, fin(li), v);
}

void pop_back(lista& li) {             // quitar el último elemento
   erase(li, fin(li)->ant);
}

void push_front(lista& li, int v) {    // insertar antes del primero
   insert(li, inicio(li), v);
}

void pop_front(lista& li) {            // quitar el primero
   erase(li, inicio(li));
}
```

Si llegáramos a necesitar calcular el tamaño de la lista enlazada, basta agregar un entero al `struct lista` y actualizar su valor como corresponda en las funciones `insert` e `erase`. Aunque no implementaremos funciones que corten y peguen listas enlazadas, se puede emplear una estrategia muy similar.

La biblioteca estándar de C++ ya proporciona una implementación de listas enlazadas en el archivo de biblioteca `<list>`. El tipo de dato se llama `std::list` y proporciona las funciones de modificación de listas antes mencionadas, además de `begin` y `end` que regresan iteradores de tipo `std::list<T>::iterator`. Las funciones `insert` e `erase` toman iteradores en lugar de apuntadores. Un iterador de lista enlazada puede avanzar y retroceder usando `++` y `--`. No necesitamos (ni podemos) examinar los miembros del `struct nodo` que usa la biblioteca. La desreferencia de un iterador regresa el valor del elemento visitado.

Código

```
#include <iostream>
#include <iterator>          // para std::next, ver abajo
#include <list>

int main( ) {
   std::list<int> li;
   li.push_back(1);
   li.push_back(2);
   li.push_front(3);
   li.push_back(4);

   // imprimir la lista
   std::cout << "tam: " << li.size( ) << "\n";
   for (auto p = li.begin( ); p != li.end( ); ++p) {
      std::cout << *p << " ";
   }

   // eliminar el primer elemento
   li.pop_front( );
   // otra forma de eliminar el primer elemento
   li.erase(li.begin( ));
}
```

Los iteradores de lista enlazada son bidireccionales y no de acceso arbitrario ya que no podemos dar saltos sobre una lista enlazada. Por esta razón, `p + k` no funciona cuando `p` es un iterador de lista enlazada. Si queremos calcular el iterador que está dos elementos adelante, una posibilidad es hacer `auto t = p; ++t; ++t;` pero esto es feo. Las funciones `std::next` y `std::prev` disponibles en `<iterator>`

toman un iterador y un entero y avanzan o retroceden el iterador la cantidad especificada de veces. De ser necesario, estas funciones implementan un ciclo que hace `++` o `--` repetidamente sobre el iterador.

Código

```
std::list<int> li = { 3, 1, 7, 2, 5 };
auto p = li.begin( );              // p apunta al inicio
auto t = std::next(p, 2);          // t apunta dos elementos adelante del inicio
std::cout << *p << " " << *t;      // 3 y 7
```

Lo anterior permite ver el gran sacrificio que implica usar una lista enlazada: deja de ser posible accesar al i-ésimo elemento de la lista enlazada eficientemente. El tipo `std::list` no proporciona el operador `[]` y no nos quedará de otra más que visitar elemento a elemento hasta llegar al elemento deseado. Incluso deja de ser posible ejecutar el algoritmo de búsqueda binaria sobre una lista enlazada ordenada. Por otra parte, los iteradores de `std::list` cuenta con una propiedad que se denomina *estabilidad de iteradores*: tras una inserción o eliminación, los nodos no afectados no se relocalizan y sus iteradores siguen siendo válidos. En una lista enlazada, los nodos son independientes en memoria y la secuencia se simula mediante enlaces. En cambio, una estructura de datos que relocalice sus elementos no proveerá iteradores estables.

La función miembro `splice` de `std::list` transfiere una subsecuencia de elementos de la lista origen a una destino. Los elementos transferidos se insertan antes del iterador dado en el destino. Lo anterior permite simular cortes o concatenaciones de listas.

Código

```
std::list<int> fuente = { 0, 1, 2, 3, 4 };
auto fa = std::find(fuente.begin( ), fuente.end( ), 1);
auto fb = std::find(fuente.begin( ), fuente.end( ), 3);

std::list<int> destino = { 5, 6, 7, 8, 9 };
auto d = std::find(destino.begin( ), destino.end( ), 7);

destino.splice(d, fuente, fa, fb);    // mover [fa, fb) antes del 7
// fuente ahora tiene { 0, 3, 4 }
// destino ahora tiene { 5, 6, 1, 2, 7, 8, 9 }
```

Recordemos que las listas simplemente enlazadas son aquéllas donde sólo existen enlaces en un sentido (por ejemplo, al siguiente nodo de la lista y no al anterior). La biblioteca de C++ provee este tipo de lista en el archivo `forward_list`. Estas listas enlazadas usan menos memoria pero son menos poderosas. Por ejemplo, no podemos eliminar eficientemente el elemento actualmente visitado porque, al no poder accesar rápidamente al elemento anterior, no podremos reconectar el elemento anterior con el siguiente.

16.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/De-una-lista-a-otra`.
2. Resuelve el problema `https://omegaup.com/arena/problem/Reubicando-elementos-de-una-list`.

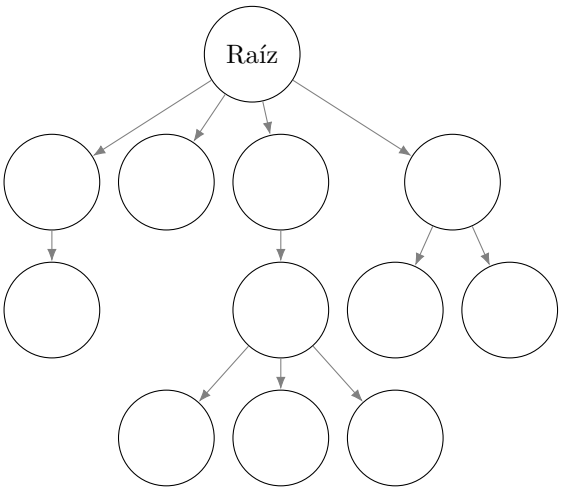
17. Colas de prioridad

Una cola de prioridad es una estructura de datos que permite realizar eficientemente las siguientes operaciones: agregar elementos, consultar el elemento de mayor prioridad y eliminar el elemento de mayor prioridad. Desde el punto de vista de la definición abstracta, realmente no nos importa el orden relativo de los elementos almacenados, siempre y cuando las tres operaciones anteriores sean eficientes. Por lo mismo, esta estructura de datos no denota una secuencia.

Ninguna de las estructuras de datos vistas con anterioridad permite implementar una cola de prioridad. La estructura que más se aproxima es un arreglo ordenado, ya que ésta permite ubicar el máximo elemento rápidamente (es el último) y también permite eliminar éste sin recorrer a los demás. Sin embargo, en un arreglo ordenado no podemos agregar elementos arbitrarios eficientemente. Para implementar una cola

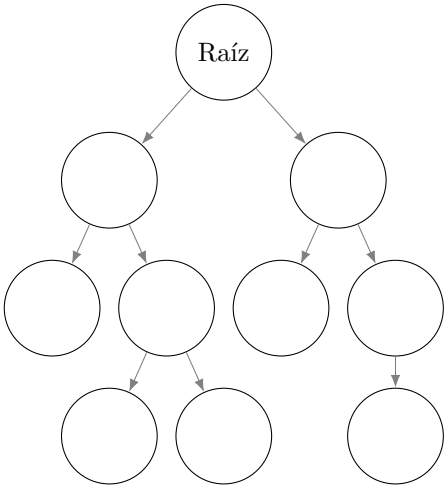
de prioridad, necesitaremos una idea radicalmente distinta. A una estructura de datos concreta que nos permitirá implementar una cola de prioridad se le conoce como montículo binario máximo.

Para implementar esta estructura de datos, necesitaremos de varios conceptos previos. El primero de ellos es el de *árbol enraizado*, aunque tampoco daremos una definición sino sólo la intuición de lo que es. En un árbol enraizado, cada nodo tiene cero o más hijos y cada nodo tiene un único padre (excepto el nodo *raíz*, que no tiene padre). A los nodos sin hijos se les denominan *hojas*. Por ejemplo:



Ejemplo de árbol enraizado

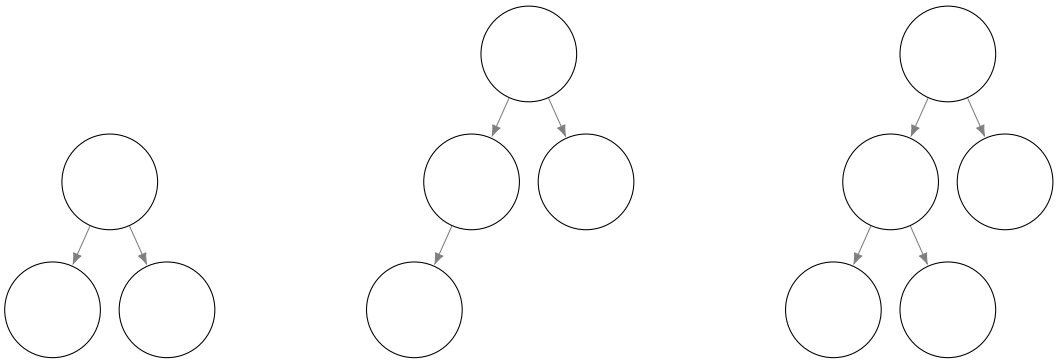
Los árboles enraizados generalmente se dibujan por niveles. En el primer nivel está la raíz, en el segundo nivel están los hijos directos de la raíz, en el tercer nivel están los nietos directos (es decir, los hijos directos de los hijos directos) de la raíz, etc. Nos concentraremos en los árboles enraizados que también son binarios. Esto quiere decir que cada nodo puede tener un máximo de dos hijos.



Ejemplo de árbol binario enraizado

Si definimos la altura de un nodo como el número de su nivel, entonces la raíz tiene altura 1, sus hijos directos tienen altura 2, etc. La altura de un árbol enraizado es el máximo de las alturas de sus nodos (es decir, la altura de un nodo del nivel más profundo).

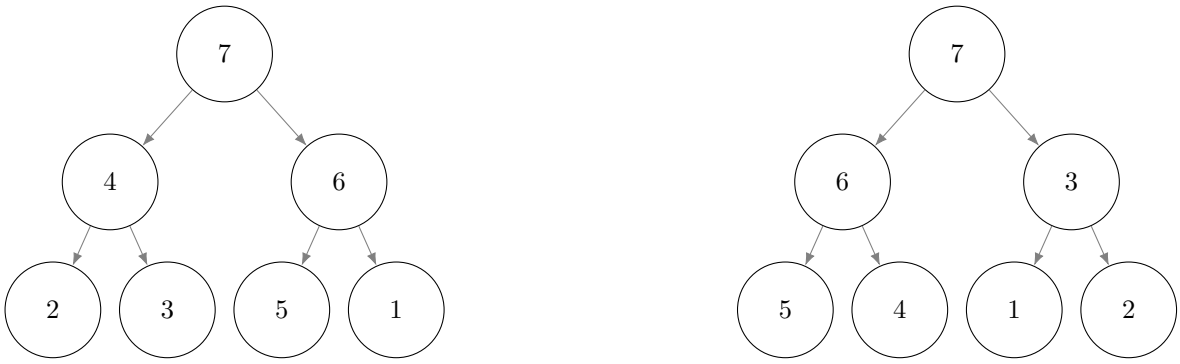
Un árbol binario completo es un árbol binario enraizado que además cumple que todos sus niveles están llenos, excepto posiblemente el último nivel, donde las hojas aparecen lo más a la izquierda posible. Lo anterior quiere decir que la forma de un árbol binario completo de n elementos es única.



Árboles binarios completos de 3, 4 y 5 nodos, respectivamente.

La capacidad en nodos de cada nivel está dada por 2^{k-1} donde k es el número del nivel. En el primer nivel cabe un nodo, en el segundo nivel caben dos nodos, en el tercer nivel caben cuatro nodos, en el cuarto nivel caben ocho nodos, etc. Es relativamente sencillo observar que la altura de un árbol binario completo de n nodos es $\lfloor \log_2(n) + 1 \rfloor$, por lo que un árbol completo de 10^6 nodos tiene apenas altura 21.

Un montículo binario máximo es un árbol binario completo donde el valor anotado en cada nodo es mayor o igual que los de sus hijos. Aunque la "silueta" de un montículo binario de n elementos es única (ya que ésta viene de ser un árbol binario completo), puede haber más de una forma de anotar un conjunto de valores en los nodos. Sin embargo, el valor máximo necesariamente debe aparecer en la raíz. A continuación se muestran dos montículos binarios máximos que almacenan los enteros del 1 al 7.



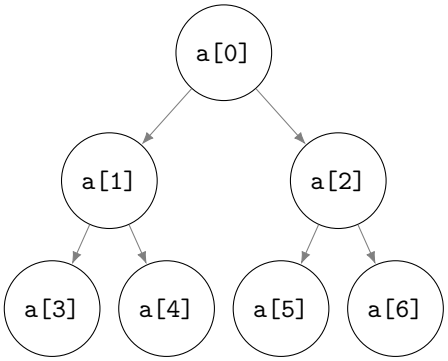
Dos montículos binarios máximos con los mismos elementos.

En esta estructura de datos, consultar el elemento máximo es trivial: siempre está en la raíz. Ahora se discutirá cómo agregar elementos al montículo y cómo eliminar el valor máximo del mismo.

- Agregar un elemento con valor v a un montículo binario máximo.
  Crearemos un nuevo nodo en la posición que dicte el árbol binario completo y anotaremos el valor v en ese nodo. Sin embargo, no hay ninguna garantía de que v sea menor o igual a su padre. Compararemos v con su padre e intercambiaremos sus valores si v es mayor. Repetiremos el proceso hacia arriba hasta encontrar un padre que es mayor o hasta llegar a la raíz. A lo mucho ocurre una comparación por nivel, por lo que el tiempo que toma agregar el elemento es proporcional a $\log_2(n)$.

- Eliminar el máximo valor de un montículo binario máximo.
  Es imprescindible conservar la forma de un árbol binario completo, pero estamos en una disyuntiva porque el valor que queremos eliminar (el máximo) no necesariamente está en el nodo que debe desaparecer. Lo que haremos será sobreescribir el valor del máximo con el valor del nodo que desaparecerá, para luego eliminar dicho nodo. Como el valor recién migrado a la raíz puede no ser mayor o igual que sus hijos, los compararemos. Si el valor de algún hijo es mayor, haremos un intercambio con el mayor de sus hijos y repetiremos el proceso hacia abajo mientras sea posible. A lo mucho ocurren dos comparaciones por nivel, por lo que el tiempo que toma eliminar el máximo es proporcional a $\log_2(n)$.

A pesar de que las figuras anteriores parecen sugerir el uso de apuntadores para implementar montículos binarios, hay una forma simple de emplear un arreglo `a` que almacene los elementos, nivel a nivel y de izquierda a derecha. El nodo almacenado en `a[i]` tendrá como padre al nodo `a[(i - 1) / 2]` y como hijos izquierdo y derecho a los nodos `a[2 * i + 1]` y `a[2 * i + 2]`, respectivamente.



Los nodos de un montículo binario pueden almacenarse en un arreglo.

Implementaremos un montículo binario máximo con cuatro funciones: `push` para agregar un elemento, `top` para accesar al máximo valor actual, `pop` para eliminar el máximo actual y `size` para consultar el tamaño del montículo. Primero definiremos la representación interna de un montículo y también un conjunto de funciones auxiliares nos permitirán calcular los índices del padre y los hijos de un nodo.

Código

```
#include <vector>

struct monticulo {
   std::vector<int> a;
};

int indice_padre(int pos) {
   return (pos - 1) / 2;
}

int indice_hijo1(int pos) {
   return 2 * pos + 1;
}

int indice_hijo2(int pos) {
   return 2 * pos + 2;
}
```

Las funciones para calcular el tamaño del montículo y para calcular el valor en la raíz del montículo simplemente se limitan a accesar al arreglo subyacente:

Código

```
int size(const monticulo& mont) {
   return mont.a.size( );
}

int top(const monticulo& mont) {
   return mont.a[0];
}
```

Las funciones para agregar y quitar elementos del montículo se pueden implementar como sigue:

Código

```
#include <algorithm>

void push(monticulo& mont, int v) {
   mont.a.push_back(v);
   int pos = mont.a.size( ) - 1;
   while (pos != 0 && mont.a[pos] > mont.a[indice_padre(pos)]) {
      std::swap(mont.a[pos], mont.a[indice_padre(pos)]),
      pos = indice_padre(pos);
   }
}

void pop(monticulo& mont) {
   int pos = mont.a.size( ) - 1, nueva_pos = 0;
   do {
      std::swap(mont.a[pos], mont.a[nueva_pos]), pos = nueva_pos;
      if (indice_hijo1(pos) < mont.a.size( ) - 1 &&
          mont.a[indice_hijo1(pos)] > mont.a[nueva_pos]) {
         nueva_pos = indice_hijo1(pos);
      }
      if (indice_hijo2(pos) < mont.a.size( ) - 1 &&
          mont.a[indice_hijo2(pos)] > mont.a[nueva_pos]) {
         nueva_pos = indice_hijo2(pos);
      }
   } while (pos != nueva_pos);
   mont.a.pop_back( );
}
```

Construir un montículo de n elementos mediante la llamada repetida de la función `push` tardaría tiempo proporcional a $n \log_2(n)$. Una forma más eficiente de construir un montículo es colocar directamente los elementos en el árbol completo y luego empezar a establecer la propiedad de montículo de abajo hacia arriba, con montículos cada vez más grandes que se van juntando en parejas. Este proceso será más rápido porque sólo un elemento (el de la raíz) tendrá que compararse con elementos de todos los demás niveles, a diferencia del proceso que hace n llamadas a `push` donde posiblemente todos los elementos del último nivel deban compararse con elementos de todos los demás niveles.

La biblioteca estándar de C++ provee de una implementación de colas de prioridad que también está basada en montículos binarios. Ésta se encuentra en `<queue>` bajo el nombre de `std::priority_queue`. Las funciones del tipo anterior también se llaman `size`, `top`, `push` y `pop`, aunque son funciones miembro.

Código	Salida

`#include <iostream>`
`#include <queue>`
 	
`int main( ) {`
`  std::priority_queue<int> cp;`
`  cp.push(3);`
`  cp.push(7);`
`  cp.push(5);`
 	
`  std::cout << cp.top( ) << "\n";`
`  cp.pop( );`
`  std::cout << cp.top( ) << "\n";`
`}`
	7
	5

La implementación de `std::priority_queue` considera que el valor de un elemento es también su prioridad. Así, `std::priority_queue` colocará un elemento de máxima prioridad (y máximo valor) en la

raíz del montículo. Por omisión, `std::priority_queue` no sabría cómo acomodar elementos de un tipo `struct` en el montículo. Sin embargo, C++ permite definir el significado del operador $<$ para un `struct`. Usando tal definición, `std::priority_queue` podrá determinar si un `struct` es menor que otro (o en términos de prioridad, menos importante que otro) para así colocar al más importante en la raíz.

Código

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

struct alumno {
   std::string nombre;
   int promedio;
};

// un alumno tendrá menor prioridad que otro si tiene menor promedio
// en caso de empate, el menos importante es el de mayor nombre lexicográfico
bool operator<(const alumno& a, const alumno& b) {
   if (a.promedio != b.promedio) {
      return a.promedio < b.promedio;
   } else {
      return a.nombre > b.nombre;
   }
}

int main( ) {
   std::priority_queue<alumno> cp;
   cp.push(alumno{"memo", 10});
   cp.push(alumno{"pablo", 7});
   cp.push(alumno{"martha", 10});
   cp.push(alumno{"carlos", 5});

   std::cout << cp.top( ).nombre << " " << cp.top( ).promedio << "\n";
   // "martha" tiene mayor prioridad (mayor promedio y menor nombre)
}
```

El algoritmo de ordenamiento por montículo (*heap sort*) consiste simplemente en agregar todos los elementos a un montículo, para luego sacarlos en orden. Desafortunadamente, la palabra *heap* también se usa en la literatura para denotar al almacenamiento libre, el cual no tiene relación con los monticulos.

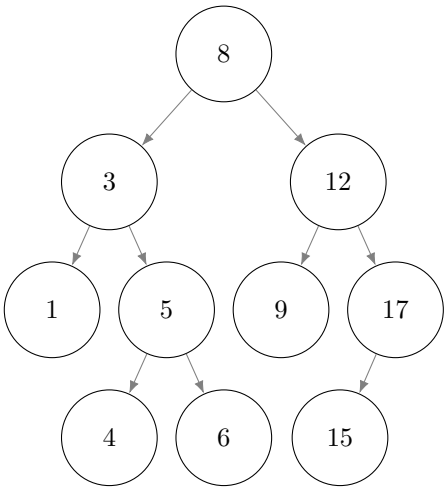
17.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/La-huida-de-los-jardineros-respo`.

18. Conjuntos y diccionarios ordenados

Un conjunto ordenado es una estructura de datos que permite realizar eficientemente las siguientes operaciones: agregar elementos, buscar elementos, eliminar elementos, e iterar sobre los elementos en orden. Como empieza a ser habitual, ninguna de las estructuras de datos vistas con anterioridad permite implementar un conjunto ordenado. La estructura que más se aproxima es un arreglo ordenado, pero ésta no permite agregar o eliminar elementos arbitrarios eficientemente.

Para implementar un conjunto ordenado, una vez más usaremos árboles binarios, aunque esta vez no necesitarán ser árboles completos. Un *árbol binario de búsqueda* es un árbol binario en donde, para cada nodo con valor v, todos los nodos del subárbol izquierdo tienen valores menores que v, mientras que todos los nodos del subárbol derecho tienen valores mayores que v.



Ejemplo de árbol binario de búsqueda. En el subárbol izquierdo de cada nodo v, todos los valores son menores. Del lado derecho, todos los valores son mayores.

Podemos construir un árbol binario de búsqueda insertando un elemento a la vez. El primer elemento será la raíz. Los siguientes elementos se compararán con él y bajarán por el lado izquierdo o derecho, según corresponda. Los nuevos elementos se insertarán en el primer lugar libre que encuentren. Si al bajar por el árbol encontramos otro nodo, repetiremos el proceso de compararnos con su valor y bajar por uno de sus lados tanto como sea necesario. Podemos implementar lo anterior como sigue:

Código

```
struct nodo {
   int valor;                    // el valor del nodo
   nodo* izq = nullptr;          // los apuntadores a los dos subárboles
   nodo* der = nullptr;
};

void inserta(nodo*& p, int v) {  // el apuntador original, no una copia
   if (p == nullptr) {
      p = new nodo{v};
   } else if (v < p->valor) {
      inserta(p->izq, v);
   } else if (v > p->valor) {
      inserta(p->der, v);
   }
}

int main( ) {
   nodo* raiz = nullptr;
   inserta(raiz, 2);
   inserta(raiz, 1);
   inserta(raiz, 3);
}
```

La implementación anterior es recursiva y pasa por referencia el apuntador por el que bajamos. Si el apuntador nos lleva a un espacio libre, creamos un nuevo nodo para el valor a insertar y hacemos que el apuntador lo señale. En caso contrario, bajamos por el subárbol correspondiente y repetimos el proceso.

Para buscar un elemento en un árbol binario de búsqueda, aprovecharemos que, gracias al algoritmo de inserción, existe un único camino a seguir para determinar si un elemento aparece o no en el árbol. Nos pararemos en la raíz y compararemos su valor con el valor buscado v. Si los valores coinciden, la

búsqueda fue exitosa. En caso contrario, bajaremos por la izquierda o por la derecha dependiendo de si v fue menor o mayor. Si al bajar por el árbol encontramos otro nodo, repetiremos el proceso tanto como sea necesario. La búsqueda fracasa cuando bajamos pero ya no encontramos más nodos.

Código

```
nodo*& busca(nodo*& p, int v) {
   if (p == nullptr || p->valor == v) {
      return p;
   } else if (v < p->valor) {
      return busca(p->izq, v);
   } else if (v > p->valor) {
      return busca(p->der, v);
   }
}
```

Eliminar el valor v del árbol se puede llevar a cabo como sigue. Claramente, primero debemos buscar el nodo respectivo. Si lo encontramos, evitaremos en la medida de lo posible borrar el nodo y preferiremos sobreescribir v con el valor de otro nodo que permita mantener la propiedad del árbol binario de búsqueda. Dos reemplazos factibles son el máximo valor del subárbol izquierdo o el mínimo valor del subárbol derecho. Una vez sobreescrito v, eliminaremos recursivamente el nodo del cual provino el valor de reemplazo. Si el nodo es una hoja entonces sí lo borraremos.

Código

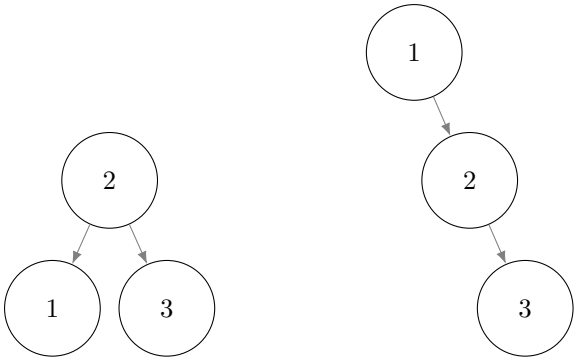
```
nodo*& minimo(nodo*& p) {
   if (p == nullptr || p->izq == nullptr) {
      return p;
   } else {
      return minimo(p->izq);
   }
}

nodo*& maximo(nodo*& p) {
   if (p == nullptr || p->der == nullptr) {
      return p;
   } else {
      return maximo(p->der);
   }
}

void elimina_actual(nodo*& p) {
   auto& reemplazo = (p->izq != nullptr ? maximo(p->izq) : minimo(p->der));
   if (reemplazo != nullptr) {
      p->valor = reemplazo->valor;
      elimina_actual(reemplazo);
   } else {
      delete p;
      p = nullptr;
   }
}

void elimina(nodo*& p, int v) {
   auto& encontrado = busca(p, v);
   if (encontrado != nullptr) {
      elimina_actual(encontrado);
   }
}
```

La altura de un árbol se puede calcular recursivamente como sigue: cada nodo calculará la altura a partir de él. Los nodos que no existen tienen altura 0. Un nodo válido calculará la altura de sus dos subárboles, tomará la mayor y le sumará 1 porque él es el padre de ambos subárboles.

Código

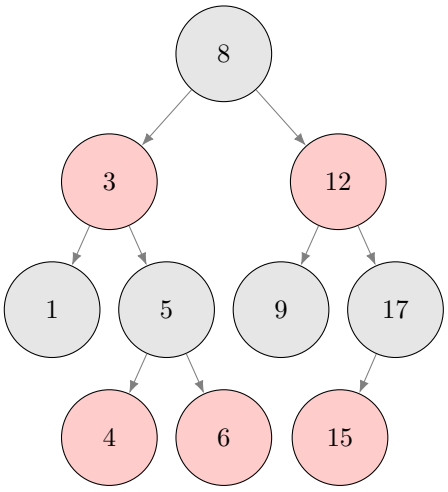
```
int altura(nodo* p) {
   if (p == nullptr) {
      return 0;
   } else {
      return std::max(altura(p->izq), altura(p->der)) + 1;
   }
}
```

En el peor caso, los algoritmos de inserción, búsqueda y eliminación visitan cada nivel del árbol una sola vez. Desafortunadamente, el orden de las inserciones producirá árboles con estructuras distintas aún si el conjunto de elementos el mismo. Por ejemplo:



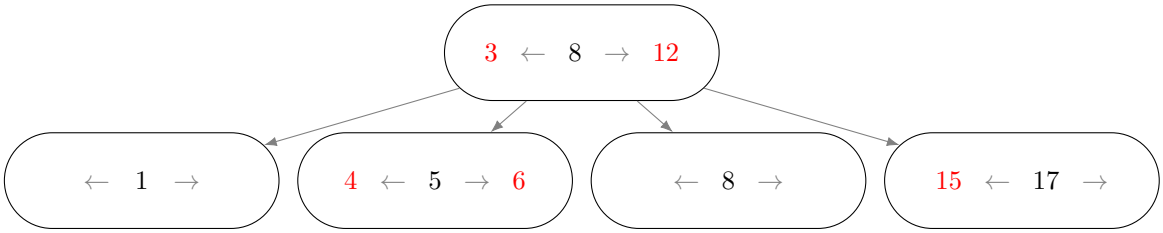
En el árbol de la izquierda, las inserciones fueron 2, 1, 3.
En el árbol de la derecha, las inserciones fueron 1, 2, 3.

Diremos que un algoritmo de inserción construye un árbol binario de búsqueda balanceado si garantiza que la altura del árbol siempre será menor o igual a $(c)(\log_2(n) + 1)$ donde c es una constante. Un árbol rojinegro es un tipo de árbol binario de búsqueda balancedo con $c = 2$ que cumple las siguientes propiedades: cada nodo es negro o rojo, la raíz es negra, los hijos de un nodo rojo son negros y existe la misma cantidad de nodos negros de la raíz a cualquiera de las hojas. Por ejemplo:



Un árbol binario perfectamente balanceado.

En un árbol rojinegro, el camino de la raíz a la hoja de menor altura podría estar formado únicamente por nodos negros, mientras que el camino a la hoja con mayor altura podría estar formado por nodos negros y rojos de forma alternada. Por esta razón, el más largo de estos caminos visitará a lo mucho el doble de nodos que el camino menos largo, lo que garantiza su balance. Implementar un árbol rojinegro es complicado, pero los árboles rojinegros son una contraparte de los árboles 2-3-4, los cuales también son árboles de búsqueda pero son cuaternarios (de cero a cuatro hijos por nodo). El árbol rojinegro del ejemplo anterior es equivalente al siguiente árbol 2-3-4.



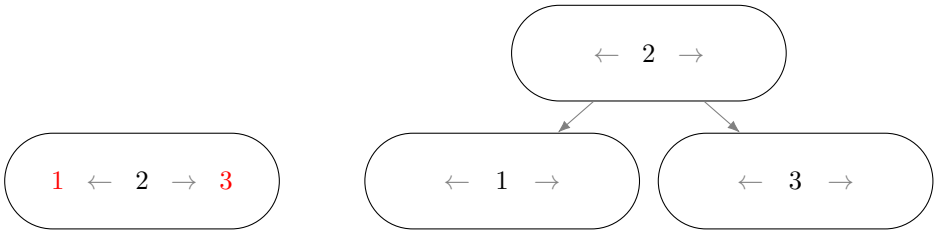
Ejemplo de un árbol 2-3-4. Los cuatro enlaces nos llevan a nodos que guardan valores $v_1 < 3$, $3 \leq v_2 < 8$, $8 \leq v_3 < 12$ y $12 \leq v_4$ respectivamente.

Los nodos de un árbol 2-3-4 pueden almacenar de uno a tres valores, los cuales a su vez denotan intervalos semiabiertos. Cada valor denota el fin de un intervalo semiabierto y el inicio del otro, de tal forma que los valores almacenados en los subárboles hijos de dicho nodo caen en tales intervalos. El tipo `struct` que representa los nodos de un árbol 2-3-4 apartará arreglos para los valores, los apuntadores y un entero que indique la cantidad de valores que realmente están guardados en el nodo.

Código

```
struct nodo {
   int tam;
   int valores[3];
   nodo* hijos[4];
};
```

El balance en un árbol 2-3-4 se lleva a cabo de la siguiente manera: cada vez que un nodo se llene, éste se parte y el elemento de enmedio sube al padre (creando un nuevo nodo arriba si no tiene padre). Lo anterior tiene el siguiente efecto: no importa si los elementos siempre se insertan cargados hacia un lado (por ejemplo, a la derecha); los elementos en exceso serán enviados hacia arriba para que los niveles superiores en encarguen de ellos. Esto provocará que todos los nodos nuevos se creen al nivel de la raíz, pero partir un nodo siempre provoca que baje un nodo de cada lado, lo que consigue el balance.



Un árbol 2-3-4 después de insertar los valores 1, 2, 3. El árbol se muestra antes y después de partir la raíz. Partir un nodo genera un árbol balanceado.

Durante una inserción, podemos partir los nodos llenos que veamos de forma preventiva. Hacer esto simplificará bastante la implementación del algoritmo de inserción, que ya de por sí es algo largo. Se espera que la altura de un árbol 2-3-4 sea menor que la de un árbol binario, justamente porque cada nodo tiene más capacidad. Sin embargo, los iteradores en los árboles 2-3-4 no serían estables porque un valor en específico puede necesitar migrar de un nodo a otro durante el balanceo. Algunas generalizaciones de los árboles 2-3-4 se usan mucho en aplicaciones que guardan datos en disco y no en memoria principal.

```
// partir un nodo; si tenemos padre, el elemento de enmedio subirá a él
nodo* parte_lleno(nodo* p, nodo* padre) {
   // por simplicidad, crearemos nodos para los valores izquierdo y derecho
   nodo* izq = new nodo{1, { p->valores[0] }, { p->hijos[0], p->hijos[1] }};
   nodo* der = new nodo{1, { p->valores[2] }, { p->hijos[2], p->hijos[3] }};
   if (padre == nullptr) {
      // no tenemos padre: nos quedamos con el valor central
      *p = {1, { p->valores[1] }, { izq, der }};
      return p;
   } else {
      // si tenemos padre: insertamos el valor central en él
      // él no estará lleno si es que lo partimos preventivamente al bajar
      auto vini = &padre->valores[0], vfin = vini + padre->tam + 0;
      auto hini = &padre->hijos[0], hfin = hini + padre->tam + 1;
      auto viter = std::lower_bound(vini, vfin, p->valores[1]);
      auto hiter = hini + (viter - vini);
      std::copy_backward(viter, vfin, vfin + 1);   // recorrer hacia adelante
      std::copy_backward(hiter, hfin, hfin + 1);   // recorrer hacia adelante
      padre->tam += 1;
      *viter = p->valores[1];
      *hiter = izq, *(hiter + 1) = der;
      delete p;
      return padre;
   }
}

// insertar el valor y partir todos los nodos llenos que veamos
void inserta(nodo*& p, int v, nodo* padre = nullptr) {
   if (p == nullptr) {           // si no existen hojas, crear el primer nodo
      p = new nodo{1, { v }};
   } else {
      nodo* q = (p->tam == 3 ? parte_lleno(p, padre) : p);
      auto ini = &q->valores[0], fin = ini + q->tam;
      auto iter = std::lower_bound(ini, fin, q);
      if (q->hijos[iter - ini] != nullptr) {    // nodo interno: debemos bajar
         inserta(q->hijos[iter - ini], v, p);
      } else {                                  // nodo hoja: insertar ahí
         std::copy_backward(iter, fin, fin + 1);
         *iter = v, q->tam += 1;
      }
   }
}

nodo*& busca(nodo*& p, int v) {   // debemos buscar primero dentro del nodo
   if (p == nullptr) {            // si no está el valor ahí, intentar bajar
      return p;
   } else {
      auto ini = &p->valores[0], fin = ini + p->tam;
      auto iter = std::lower_bound(ini, fin, v);
      if (iter != fin && *iter == v) {
         return p;
      } else {
         return busca(p->hijos[iter - ini], v);
      }
   }
}
```

La eliminación de un valor en un árbol 2-3-4 es complicada. A su vez, la implementación de un árbol rojinegro (que es binario) también es complicada y no se mostrará. Afortunadamente, la biblioteca estándar ya proporciona una implementación de árboles binarios de búsqueda balanceados en el archivo de biblioteca `<set>`, con la plantilla `std::set`. Por ejemplo:

Código

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

int main( ) {
   int n, m;
   std::cin >> n >> m;
   std::set<int> conjunto;

   // insertar n enteros
   for (int i = 0; i < n; ++i) {
      int t;
      std::cin >> t;
      conjunto.insert(t);
   }

   // buscar m enteros
   for (int i = 0; i < m; ++i) {
      int t;
      std::cin >> t;
      auto p = conjunto.find(t);
      if (p == conjunto.end( )) {
         std::cout << "no\n";
      } else {
         std::cout << *p << "\n";
         // borrar lo encontrado
         conjunto.erase(p);
      }
   }
}
```

Entrada

```
5 4
8 5 9 7 6
2
5
7
5
```

Salida

```
no
5
7
no
```

Se garantiza que las inserciones, búsquedas y eliminaciones en un `std::set` tardan tiempo logarítmico. Más aún, se garantiza que la iteración sobre los elementos de un `std::set` se hace en orden:

Código

```
std::set<int> s = { 3, 1, 4, 8, 4 };
for (auto p = s.begin( );
          p != s.end( ); ++p) {
   std::cout << *p << " ";
}
```

Salida

```
1 3 4 8
```

Un `std::set` no admite duplicados. Si se desea poder insertar duplicados, se puede usar `std::multiset`. Desafortunadamente, ninguna de las implementaciones de la biblioteca de C++ permiten inspeccionar el árbol binario subyacente. Aunque navegar arbitrariamente por un árbol binario no suele ser muy útil, será imposible implementar sobre `std::set` los tres recorridos más usuales sobre árboles binarios:

- Preorden (*"Primero yo, luego mis hijos"*): este recorrido suele emplearse para copiar un árbol.
- Posorden (*"Primero mis hijos, luego yo"*): este recorrido suele emplearse para liberar un árbol.
- Orden (*"Primero mi hijo izquierdo, luego yo, luego mi hijo derecho"*): Este recorrido suele emplearse para imprimir en orden todos los elementos de un árbol binario de búsqueda.

Código

```
void preorden(nodo* p) {
   if (p != nullptr) {
      std::cout << p->valor << "\n";
      preorden(p->izq);
      preorden(p->der);
   }
}

void posorden(nodo* p) {
   if (p != nullptr) {
      posorden(p->izq);
      posorden(p->der);
      std::cout << p->valor << "\n";
   }
}

void orden(nodo* p) {
   if (p != nullptr) {
      orden(p->izq);
      std::cout << p->valor << "\n";
      orden(p->der);
   }
}
```

Un diccionario ordenado es una estructura de datos que permite realizar eficientemente las siguientes operaciones: agregar parejas (clave, valor), buscar parejas por clave, eliminar parejas por clave, e iterar sobre las parejas en orden por clave. Un diccionario puede implementarse fácilmente si modificamos ligeramente la información que guarda un nodo y adecuamos el resto de las funciones:

Código

```
struct nodo {
   int clave;     // la inserción, búsqueda y eliminación se harán por clave
   nodo* izq = nullptr;
   nodo* der = nullptr;
   int valor;     // el valor asociado a la clave
};
```

La biblioteca estándar de C++ ya proporciona una implementación de un diccionario ordenado en `<map>` con la plantilla `std::map<K, T>` donde `K` es el tipo de la clave y `T` es el tipo del valor. Este tipo proporciona el operador `[]` para buscar una pareja por clave (creándola en el proceso si no existe).

Código

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

int main( ) {
   std::map<std::string, int> dicc;   // una cadena como clave
   dicc["juan"] = 10;                 // el valor asociado es un int
   dicc["abril"] = 24;
   dicc["juan"] += 5;
   dicc["pedro"] = 19;

   std::cout <<  dicc["abril"] << "\n";   // 24
   std::cout <<  dicc["juan"] << "\n";    // 15
   std::cout <<  dicc["carlos"] << "\n";  // 0 (la pareja no existía, se creó)
}
```

Con la función miembro `find` es posible buscar una pareja sin crearla. Esta función regresa un iterador de tipo `std::map<K, V>::iterator` que es igual al fin del mapa si la pareja no existía. Cuando la pareja sí existe, la desreferencia del iterador regresa un `std::pair<K, T>` que es tipo `struct` cuyo miembro `first` es de tipo `K` y contiene la clave, mientras que su miembro `second` es de tipo `T` y contiene el valor.

Código	Entrada	Salida

`std::map<std::string, int> dicc = {`
`  { "juan", 15 },`
`  { "abril", 24 },`
`  { "pedro", 19 }`
`};`
 	
`std::string buscar;`
`while (std::cin >> buscar) {`
`  auto p = dicc.find(buscar);`
`  if (p == dicc.end( )) {`
`  std::cout << "no";`
`  } else {`
`  std::cout << p->first << " "`
`  << p->second << "\n";`
`  }`
`}`
	`juan`
	`carlos`
		`juan 15`
		`no`

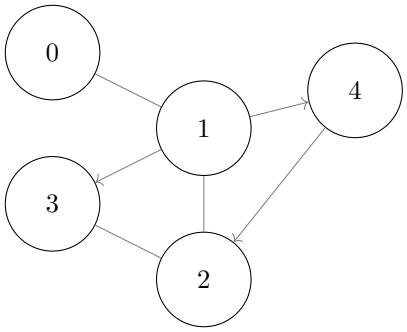
Un `std::map` no admite claves duplicadas. Si se desea poder insertar claves duplicadas, se puede usar `std::multimap`. El algoritmo de ordenamiento por árbol (*tree sort*) consiste simplemente en agregar todos los elementos a un árbol binario de búsqueda, para luego recorrerlo en orden.

18.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Arboles-binarios-sin-balancear`.
2. Resuelve el problema `https://omegaup.com/arena/problem/Encuestando-a-la-gente-de-la-col`.

19. Gráficas y sus aplicaciones

Una gráfica puede verse como una generalización de un árbol:



Una gráfica es una estructura que conecta vértices (anteriormente llamados nodos) mediante arcos (conexiones direccionales) y aristas (conexiones bidireccionales).

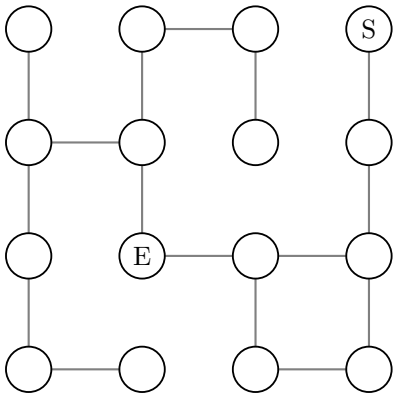
Formalmente, una gráfica dirigida es una pareja (V, A) donde V es un conjunto de vértices y A es un conjunto de arcos que conectan vértices de V. Una gráfica no dirigida es una pareja (V, E) donde E es un conjunto de aristas que conectan vértices de V. Una gráfica mixta contiene arcos y aristas.

Nos enfocaremos en las gráficas no dirigidas. En este contexto, dos vértices son vecinos si están conectados por una arista, un lazo es una arista que conecta a un vértice consigo mismo y una arista es

paralela si conecta dos vértices que ya eran vecinos. Una gráfica simple no tiene lazos ni aristas paralelas. Generalmente trabajaremos con gráficas simples. Una gráfica es conexa si es posible ir de cualquier vértice a otro siguiendo un camino de aristas y un árbol es simplemente una gráfica conexa sin ciclos. Una componente conexa es una subgráfica que incluye a todos los vértices alcanzables desde cualquiera de sus vértices. El grado de un vértice es la cantidad de aristas que inciden en él. El grado de una gráfica es el máximo de los grados de sus vértices. Una gráfica con pesos le asocia un costo a cada arista.

A pesar de lo abstracto que pudiera resultar el concepto de gráfica, éstas sirven para modelar una enorme cantidad de aplicaciones de la vida real. A continuación se listan algunas de estas aplicaciones, aunque sólo se discutirá la implementación de las tres primeras.

- Determinar si es posible escapar de un laberinto: podemos modelar las zonas libres de obstáculos como vértices de una gráfica y la adyacencia entre zonas libres como aristas. Es posible escapar del laberinto si los dos vértices señalados como la entrada y la salida pertenecen a la misma componente conexa.
- Encontrar el camino más corto de una ciudad a otra: podemos modelar las ciudades de un mapa como vértices de una gráfica pesada y las carreteras que las unen como aristas. La longitud de la carretera se suele indicar en el costo de la arista.
- Encontrar la forma óptima de construir una red física: podemos modelar los entes a conectar (por ejemplo, computadoras o centros de datos) como vértices de una gráfica pesada y los posibles lugares por donde se pueden tender cables físicos entre ellos como aristas. La longitud de un cable es el costo de la arista. Un árbol abarcador de costo mínimo de la gráfica es una solución óptima del problema.
- Encontrar un orden factible para completar un conjunto de actividades: podemos modelar las actividades a realizar como vértices de una gráfica y las dependencias entre ellas como arcos. Un orden topológico de la gráfica corresponde con un orden factible para completar las actividades.
- Encontrar un emparejamiento de cardinalidad máxima que asigne tareas a trabajadores, donde las tareas son indivisibles y cada trabajador se dedica exclusivamente a una tarea: podemos modelar las tareas y los trabajadores como vértices de una gráfica y la compatibilidad entre ellos como aristas.



Ejemplo de gráfica que denota un laberinto.

Para implementar cualquiera de las aplicaciones anteriores, primero se debe discutir el cómo representar una gráfica en un programa. Si una gráfica no se modificará una vez construida, entonces una representación de nodos `struct` conectados mediante apuntadores suele ser innecesariamente rebuscada. Las dos representaciones más usadas se describen a continuación y ambas tienen en común que consideran que los vértices de la gráfica están implícitamente numerados de 0 a $n-1$ donde $n = |V|$.

La representación de una gráfica mediante una matriz de adyacencia usa una matriz de $n \times n$ booleanos, donde la entrada (i, j) de la matriz guarda verdadero si y sólo si existe un arco o arista del vértice i al vértice j. En una gráfica no dirigida, dicha matriz es simétrica porque las aristas son bidireccionales. Esta representación tiene la ventaja de que es posible contestar de forma instantánea si los vértices i y j son vecinos, pero tiene la desventaja de que consume memoria tanto para las aristas que sí existen como

para las que no. Es común que una gráfica tenga una cantidad lineal de aristas (por ejemplo, un árbol tiene exactamente $n - 1$ aristas). Por esta razón, una matriz de adyacencia suele implicar un consumo innecesario de memoria. Además, no sería eficiente listar a los vecinos del vértice i porque sería necesaria una búsqueda lineal sobre la i-ésima fila de la matriz.

La representación de una gráfica mediante listas de adyacencia usa un arreglo de n secuencias (por ejemplo, arreglos dinámicos) donde la i-ésima secuencia almacena los índices de los vértices a los que se pueden llegar mediante un arco o arista que sale del vértice i. Esta representación tiene la ventaja de que consume una cantidad de memoria proporcional a $n + m$ donde m es la cantidad de arcos o aristas de la gráfica. Además, es eficiente listar a los vecinos del vértice i y responder si los vértices i y j son vecinos, esto último cuando las secuencias están ordenadas. Cuando la gráfica tiene una cantidad cuadrática de aristas, entonces esta representación consume más memoria que una matriz de adyacencia al guardar un entero por arco o dos enteros por arista (para denotar el recorrido en ambas direcciones).

$i = 0$	No	Sí	No	No	No
$i = 1$	Sí	No	Sí	Sí	Sí
$i = 2$	No	Sí	No	Sí	No
$i = 3$	No	No	Sí	No	No
$i = 4$	No	No	Sí	No	No

$i = 0$	$\{1\}$
$i = 1$	$\{0, 2, 3, 4\}$
$i = 2$	$\{1, 3\}$
$i = 3$	$\{2\}$
$i = 4$	$\{2\}$

La matriz de adyacencia y las listas de adyacencia de la primera gráfica de ejemplo de esta sección.

Normalmente, la entrada de una gráfica se especifica como sigue: primero un entero n que es el número de vértices de la gráfica y después un entero m que es el número de aristas de la misma. Posteriormente m parejas de enteros que denotan los extremos de cada arista. Con este formato de entrada, podemos leer una gráfica representada en una matriz de adyacencia como se muestra a continuación:

Código

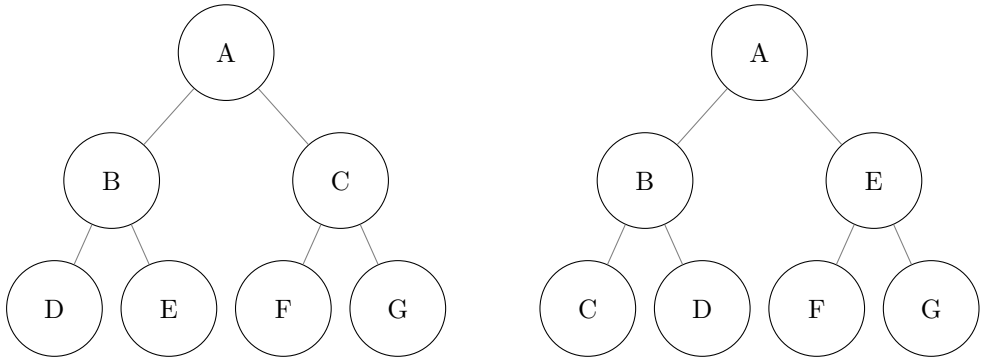
```
int n, m;
std::cin >> n >> m;

std::vector<std::vector<bool>> matriz(n, std::vector<bool>(n, false));
for (int i = 0; i < m; ++i) {
   int v1, v2;
   std::cin >> v1 >> v2;
   matriz[v1][v2] = true;
   matriz[v2][v1] = true;    // las aristas son bidireccionales
}
```

Si queremos representar la gráfica con listas de adyacencia, la lectura se puede hacer como sigue:

Código

```
int n, m;
std::cin >> n >> m;

std::vector<std::vector<int>> adyacencia(n);    // n arreglos vacíos
for (int i = 0; i < m; ++i) {
   int v1, v2;
   std::cin >> v1 >> v2;
   adyacencia[v1].push_back(v2);
   adyacencia[v2].push_back(v1);    // las aristas son bidireccionales
}
```

Ahora presentaremos las dos formas más importantes de recorrer una gráfica, ya que constituyen la base de prácticamente todo lo demás. Los dos recorridos comienzan a partir de un vértice inicial y siempre evitan procesar vértices ya procesados. Un recorrido en amplitud (*BFS* por las siglas en inglés) primero visita el vértice inicial, luego sus vecinos, luego los vecinos de los vecinos, etc. Por el contrario, un recorrido en profundidad (*DFS* por sus siglas en inglés) elige una dirección preferida (por ejemplo, siempre la primera arista) y visita cada vértice en esa dirección. Sólo cambiaremos de dirección cuando ya no sea posible seguir usando la misma dirección.



Los árboles son gráficas y se pueden recorrer como tales. Partiendo del vértice superior, a la izquierda se muestra un recorrido en amplitud y a la derecha un recorrido en profundidad.

A continuación se muestra la implementación del recorrido en amplitud, la cual usa una cola auxiliar. La cola comenzará con el vértice inicial y repetiremos el siguiente proceso mientras la cola no esté vacía: si es la primera vez que sacamos un vértice de la cola, entonces lo marcamos como visitado y meteremos a sus vecinos a la cola; si el vértice ya lo habíamos visitado, entonces lo ignoramos. Este algoritmo toma en consideración cada vértice y cada arista de la gráfica, por lo que tarda $n + m$ pasos. En todas las implementaciones que sigan, siempre representaremos una gráfica mediante listas de adyacencia.

Código

```
std::vector<bool> visitados(n, false);
std::deque<int> cola = { inicial };
do {
   int actual = cola.front( );
   cola.pop_front( );
   if (!visitados[actual]) {
      visitados[actual] = true;
      for (int vecino : adyacencia[actual]) {
         cola.push_back(vecino);
      }
   }
} while (!cola.empty( ));
```

Un recorrido en profundidad se puede obtener simplemente reemplazando la cola auxiliar por una pila, aunque el orden en el que se metan a los vecinos a la pila determinará hacia qué lado estará cargado el recorrido (el recorrido visitará primero el último vecino metido en la pila). También se puede emplear recursión para implementarlo, pero el recorrido en profundidad no resultará muy útil en este curso.

Para determinar si una gráfica es conexa, basta con hacer un recorrido en amplitud o en profundidad desde cualquiera de sus vértices y luego revisar el arreglo de booleanos. Si algún vértice era inalcanzable, entonces su entrada correspondiente sigue en falso. Una manera de encontrar una a una todas las componentes conexas de una gráfica es comenzar un nuevo recorrido a partir de un vértice que permanece inalcanzable después de haber completado los recorridos anteriores.

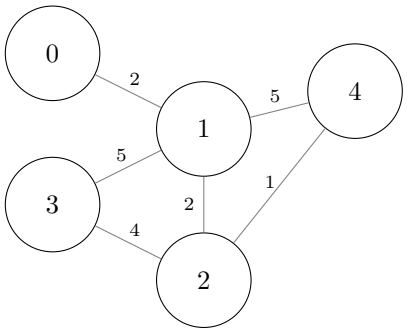
En un laberinto, podemos determinar si hay forma de llegar de la entrada a la salida comenzando un recorrido desde la entrada. Sin embargo, normalmente también nos interesa calcular el camino más corto.

Esto se puede calcular modificando la implementación del recorrido en amplitud (no el de profundidad) de la siguiente forma. En lugar de un arreglo de booleanos, mantendremos un arreglo de enteros inicializados en -1, indicando que no hemos visitado ningún vértice. En la cola ahora almacenaremos parejas: el vértice al que queremos llegar y la distancia que hemos recorrido para llegar a él. Cuando saquemos una pareja de la cola y el vértice correspondiente aún tenga distancia -1, reemplazaremos ese valor por la distancia indicada en la pareja. Cuando metamos a los vecinos del vértice actual en la cola, el camino formado tendrá una arista más que el camino al vértice actual. Aunque el algoritmo puede meter un mismo vértice varias veces en la cola, sólo la primera vez que salga lo registraremos en el arreglo con la distancia dada, que es la más corta. Con esto, el vértice inicial tendrá distancia 0, sus vecinos tendrán distancia 1, los vecinos de sus vecinos tendrán distancia 2, etcétera.

Código

```
struct destino {
   int vertice, distancia;
};

std::vector<int> visitados(n, -1);
std::deque<destino> cola = { {inicial, 0} };
do {
   auto actual = cola.front( );
   cola.pop_front( );
   if (visitados[actual.vertice] == -1) {
      visitados[actual.vertice] = actual.distancia;
      for (int vecino : adyacencia[actual.vertice]) {
         cola.push_back({vecino, actual.distancia + 1});
      }
   }
} while (!cola.empty( ));
```

Desafortunadamente, en la vida real no siempre es cierto que todas las aristas de la gráfica tengan peso unitario. Si los vértices denotan ciudades y las aristas denotan carreteras, entonces es normal pensar que algunas carreteras son más largas que otras (especialmente si éstas no son rectas por culpa del terreno).



Ejemplo de una gráfica con pesos. El camino más corto de 0 a 4 pasa por 1 y por 2.

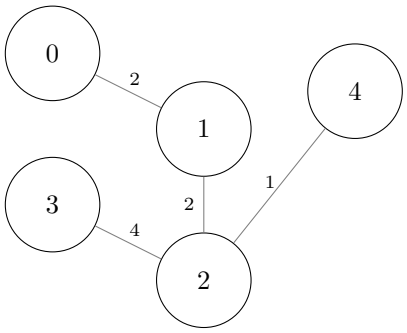
La representación de una gráfica con pesos no negativos en las aristas puede ser la siguiente. En matrices de adyacencia, el valor -1 indica la ausencia de arista y cualquier otro valor corresponde con el peso. En listas de adyacencia, cada vértice destino viene acompañado del peso de la arista.

La pregunta ahora es, ¿cómo determinar el camino más corto de un vértice origen a uno o más vértices destinos ante la presencia de aristas con pesos no negativos? Esto se puede hacer empleando un nuevo tipo de recorrido, llamado recorrido en prioridad. Lo que haremos será reemplazar la cola ordinaria por una cola de prioridad, sacando primero los destinos a los que podemos llegar con menor distancia. Esto funciona porque, por construcción, cualquier camino no descubierto que se forme con elementos que aún están en la cola de prioridad deben tener distancia al menos tan grande como la del elemento recién sacado. Este algoritmo se llama algoritmo de Dijkstra y fue diseñado por Edsger Dijkstra en 1956.

Código

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

struct destino {
   int vertice, distancia;
};

// importa menos el destino más lejano
bool operator<(destino d1, destino d2) {
   return d1.distancia > d2.distancia;
}

int main( ) {
   int n, m;
   std::cin >> n >> m;

   std::vector<std::vector<destino>> adyacencia(n);
   for (int i = 0; i < m; ++i) {
      int v1, v2, d;
      std::cin >> v1 >> v2 >> d;
      adyacencia[v1].push_back({v2, d});
      adyacencia[v2].push_back({v1, d});    // las aristas son bidireccionales
   }

   int inicial;
   std::cin >> inicial;

   std::vector<int> visitados(n, -1);
   std::priority_queue<destino> cola;
   cola.push({inicial, 0});
   do {
      auto actual = cola.top( );
      cola.pop( );
      if (visitados[actual.vertice] == -1) {
         visitados[actual.vertice] = actual.distancia;
         for (auto directa : adyacencia[actual.vertice]) {
            cola.push({directa.vertice, actual.distancia + directa.distancia});
         }
      }
   } while (!cola.empty( ));

   for (int i = 0; i < n; ++i) {
      std::cout << i << ": " << visitados[i] << "\n";
   }
}
```

Como ya se mencionó al inicio de esta sección, una gráfica con pesos también permite modelar el problema de encontrar la forma óptima de conectar los vértices de una gráfica. Nos interesa minimizar la suma de los costos de las aristas que usemos para conectar la gráfica, por lo que buscaremos usar la cantidad mínima de aristas y entonces la solución es un árbol. La razón de esto es que, mientras tengamos ciclos en la gráfica, siempre existirá al menos una arista que podamos quitar sin desconectar la gráfica, lo cual sólo puede provocar que la suma de la solución disminuya si es que todas las aristas tienen costos positivos. A dicho árbol se le conoce como árbol abarcador de costo mínimo y no necesariamente es único.



Un árbol abarcador de costo mínimo de la gráfica anterior,
el cual no incluye algunas aristas

Así como el algoritmo de Dijkstra usa un recorrido en prioridad para calcular la distancia mínima de un origen a cualquier destino, el denominado algoritmo de Prim (diseñado por Robert Prim en 1957) también usará un recorrido en prioridad para calcular un árbol abarcador de costo mínimo. La diferencia del algoritmo de Prim con respecto al algoritmo de Dijkstra es sutil y sorprendente: mientras que el algoritmo de Dijkstra busca procesar el vértice más cercano según la distancia acumulada por cada arista recorrida, el algoritmo de Prim simplemente busca usar las aristas más baratas que vea. Por esta razón, se puede obtener una implementación del algoritmo de Prim modificando una única línea de la implementación del algoritmo de Dijkstra.

Código

```
std::vector<int> visitados(n, -1);
std::priority_queue<destino> cola;
cola.push({inicial, 0});
do {
   auto actual = cola.top( );
   cola.pop( );
   if (visitados[actual.vertice] == -1) {
      visitados[actual.vertice] = actual.distancia;
      for (auto directa : adyacencia[actual.vertice]) {
         cola.push({directa.vertice, directa.distancia});
         // guardar el costo de la arista usada, sin acumular distancia
      }
   }
} while (!cola.empty( ));
```

19.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Invirtiendo-aristas`.

2. Resuelve el problema `https://omegaup.com/arena/problem/Divulgando-las-noticias`.

3. Resuelve el problema `https://omegaup.com/arena/problem/Clausura-de-carreteras`.

A. Soluciones de ejercicios

En general, siempre hay más de una posible respuesta correcta para cada pregunta. Razonen cuidadosamente las respuestas que propongo para que decidan si las suyas son equivalentes.

Soluciones de 1.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Calculo-de-sumatoria`.

Solución: Para resolver este problema, basta con aplicar la estrategia descrita en las notas. Sólo hay que tener cuidado de evaluar la expresión con el tipo `long long`. La forma más fácil de hacerlo es declarar `n` también como `long long`, pero si se desea declararla como `int`, entonces uno de los operandos de la multiplicación debe promoverse a `long long` antes de su evaluación. Esto se puede hacer sumando `1LL` que es de tipo `long long`, en lugar de `1` que es de tipo `int`.

```
#include <stdio.h>

int main( ) {
   long long n;
   scanf("%lld", &n);
   printf("%lld", (n * (n + 1)) / 2);
   return 0;
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/Suma-de-intervalos`.

Solución: Lo primero que se nos ocurre al leer este problema es simplemente leer los extremos de cada intervalo, sumar sus elementos e imprimir la suma, repitiendo el proceso para cada intervalo. Sin embargo, si debemos procesar m intervalos y cada uno tiene aproximadamente n de longitud, entonces nuestro algoritmo realizaría $(n)(m)$ sumas, lo cual es impráctico para valores grandes de m y n. Una mejor estrategia consiste en calcular un arreglo p donde $p[i]$ contenga la suma acumulada del arreglo original, de izquierda a derecha desde la posición 0 hasta la posición i. Para calcular la suma de un intervalo cerrado $[x, y]$, basta tomar $p[y]$ y restarle $p[x-1]$ para descontar la suma de los elementos que están a la izquierda del intervalo. Cuando $x = 0$ no hay que restar nada.

```
#include <stdio.h>

int main( ) {
   int n;
   scanf("%d", &n);

   int acumulado[100000];
   for (int i = 0; i < n; ++i) {
      int actual;
      scanf("%d", &actual);
      acumulado[i] = (i == 0 ? 0 : acumulado[i - 1]) + actual;
   }

   int m;
   scanf("%d", &m);

   for (int i = 0; i < m; ++i) {
```

```
        int ini, ult;
        scanf("%d%d", &ini, &ult);
        printf("%d\n", acumulado[ult] - (ini == 0 ? 0 : acumulado[ini - 1]));
    }

    return 0;
}
```

3. Resuelve el problema `https://omegaup.com/arena/problem/Sumando-todos-los-subarreglos`.

Solución: Para resolver este problema basta darnos cuenta que, dado un subarreglo de tamaño k, el siguiente subarreglo tiene $k-2$ elementos en común con el anterior (el siguiente subarreglo tiene un elemento de más a la derecha y un elemento de menos a la izquierda). Después de calcular manualmente la suma del primer subarreglo, podemos "deslizar la ventana" al sumar elementos a la derecha y restarlos a la izquierda, imprimiendo cada suma en el proceso.

```
#include <iostream>

int main( ) {
    int n, k;
    std::cin >> n >> k;

    int arr[n];
    for (int i = 0; i < n; ++i) {
        std::cin >> arr[i];
    }

    int suma = 0;
    for (int i = 0; i < k; ++i) {
        suma += arr[i];
    }

    std::cout << suma << " ";
    for (int i = k; i < n; ++i) {
        suma -= arr[i - k];
        suma += arr[i];
        std::cout << suma << " ";
    }
}
```

Soluciones de 3.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Forzando-la-caja-fuerte`.

Solución: Para resolver este problema, hay que darnos cuenta que rotar cinco veces en la misma dirección tiene el efecto de dejar los números en su posición original. Por esta razón, basta que rotemos `izq% 5` veces a la izquierda y `der% 5` veces a la derecha. Podemos implementar una rotación a la izquierda generalizando el algoritmo de intercambio e implementándolo como función usando paso por referencia. Una rotación a la derecha se puede implementar en términos de una rotación a la izquierda enviando los argumentos en el sentido contrario.

```
#include <iostream>

void rota_izquierda(int& a, int& b, int& c, int& d, int& e) {
   int temp = a;
   a = b;
   b = c;
   c = d;
   d = e;
   e = temp;
}

int main( ) {
   int izq, der;
   std::cin >> izq >> der;

   int a = 1, b = 2, c = 3, d = 4, e = 5;
   for (int i = 0; i < izq % 5; ++i) {
      rota_izquierda(a, b, c, d, e);
   }
   for (int i = 0; i < der % 5; ++i) {
      rota_izquierda(e, d, c, b, a);
   }
   std::cout << a << " " << b << " " << c << " " << d << " " << e;
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/Ordenando-numeros`.

Solución: En este problema, conviene definir una función que tome dos enteros por referencia y los ordene ascendentemente. Esto se puede hacer mediante un intercambio si es que no están ya ordenados. Para ordenar cuatro enteros, podemos emplear una estrategia tipo torneo: ordenamos la primera pareja, ordenamos la segunda pareja, ordenamos a los más chicos de cada pareja (de modo que ya ubicamos al mínimo de los cuatro), ordenamos a los más grandes de cada pareja (de modo que ya ubicamos al máximo de los cuatro) y finalmente ordenamos la pareja de enmedio.

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

void ordena2(int a, int b) {
   if (a > b) {
      std::swap(a, b);
   }
}

int main( ) {
   int a, b, c, d;
   std::cin >> a >> b >> c >> d;
   ordena2(a, b), ordena2(c, d);
   ordena2(a, c), ordena2(b, d);
   ordena2(b, c);
   std::cout << a << " " << b << " " << c << " " << d;
}
```

Soluciones de 4.1

1. Resuelve el problema `https://omegaup.com/arena/problem/El-ciclo-de-vida-de-las-bacteri2`.

Solución: Para resolver este problema, definiremos una función $f(n)$ que conteste la siguiente pregunta: ¿cuántas bacterias habrían si se simulara el fenómeno descrito en el problema durante `n` días? Claramente, cuando $n = 0$ entonces sólo existe la bacteria original. Cuando $n > 0$ entonces la bacteria original existe pero también existen las bacterias que se generen de repetir el proceso a partir de las dos bacterias descendientes de la bacteria original. Sin embargo, los procesos de las bacterias descendientes sólo dispondrán de $n - 1$ días de la simulación.

```
#include <iostream>

int f(int n) {
   if (n == 0) {
      return 1;
   } else {
      return 1 + 2 * f(n - 1);
   }
}

int main( ) {
   int n;
   std::cin >> n;
   std::cout << f(n);
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/Lavos-el-devorador-de-planetas`.

Solución: Para resolver este problema, definiremos una función $f(n)$ que conteste la siguiente pregunta: ¿cuántos Lavos habrían si se simulara el fenómeno descrito en el problema durante `n` días? Claramente, cuando $n = 0$ entonces sólo existe el Lavos original. Cuando $n > 0$ hay que considerar los siguientes lapsos de tiempo: cuando las larvas de la primera etapa reproductiva aún no llegan a sus destinos, cuando las larvas de la primera etapa reproductiva ya llegaron a sus destinos pero las de la segunda etapa aún no llegan, y cuando las larvas de ambas etapas reproductivas ya llegaron a sus destinos. En realidad, el caso de $n = 0$ coincide con el primero de los lapsos de tiempo anteriormente descritos.

```
#include <iostream>

long long f(int n) {
   if (n < 100000) {
      return 1;
   } else if (n < 500000) {
      return 1 + 3 * f(n - 100000);
   } else {
      return 1 + 3 * f(n - 100000) + 7 * f(n - 500000);
   }
}

int main( ) {
   int n;
```

```
   std::cin >> n;
   std::cout << f(n) << "\n";
}
```

Soluciones de 6.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Invirtiendo-palabras`.

Solución: Para resolver este problema, podemos usar el algoritmo `std::find` para encontrar la primera coma y luego usar `std::reverse` para invertir la secuencia. Una vez hecho esto, nos paramos adelante de la coma recién encontrada y repetimos el proceso.

```
#include <algorithm>
#include <iostream>
#include <string.h>

int main( ) {
   char buffer[1000 + 1];
   std::cin >> buffer;

   char* ini = &buffer[0];
   char* fin = ini + strlen(buffer);
   while (ini < fin) {
      char* p = std::find(ini, fin, ',');
      std::reverse(ini, p);
      ini = p + 1;
   }

   std::cout << buffer;
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/subterraneo-camino-shamash`.

Solución: Lo más fácil en este problema es construir la cadena de derecha a izquierda y luego invertirla e imprimirla. La cadena está dividida en secciones en base 60, por lo que debemos ir descomponiendo `n` en tales dígitos. A su vez, cada dígito en base 60 se descompone en subdígitos en base 10. Sólo debemos recordar escribir un punto entre dos secciones presentes en base 60.

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

int main( ) {
   int n;
   std::cin >> n;

   char buffer[100 + 1] = { };
   char* escribir = &buffer[0];

   for (;;) {
      int r = n % 60;
      std::fill(escribir, escribir + r % 10, 'I');
```

```
      escribir += r % 10;
      std::fill(escribir, escribir + r / 10, 'L');
      escribir += r / 10;
      n /= 60;

      if (n == 0) {
         break;
      } else {
         *escribir++ = '.';
      }
   }

   std::reverse(&buffer[0], escribir);
   std::cout << buffer;
}
```

Soluciones de 8.1

1. Resuelve el problema `https://omegaup.com/arena/problem/cactus_horizonte`.

Solución: Para resolver este problema, basta con ordenar el arreglo una vez y luego posicionarnos correctamente (de derecha a izquierda) cada vez que nos pidan el valor del k-ésimo elemento más grande. Si el arreglo fuera muy grande como para almacenarlo en la pila del proceso, podemos usar la palabra `static` para que el compilador lo almacene en una región de memoria distinta.

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

int main( ) {
   int n;
   std::cin >> n;

   // si el arreglo llegara a ser muy grande para la pila del proceso,
   // declararlo como static int cactus[1000000];
   int cactus[1000000];
   for (int i = 0; i < n; ++i) {
      std::cin >> cactus[i];
   }
   std::sort(&cactus[0], &cactus[0] + n);

   int p;
   std::cin >> p;

   for (int i = 0; i < p; ++i) {
      int pos;
      std::cin >> pos;
      std::cout << cactus[n - pos] << " ";
   }
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/Calculo-de-la-mediana`.

Solución: Para resolver este problema, lo más fácil es hacer una copia del arreglo original y ordenarlo. El elemento de la mitad del arreglo ordenado es la mediana. Con esta información, podemos buscar dicho valor sobre el arreglo original e imprimir su posición.

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

int main( ) {
   int n;
   std::cin >> n;

   int arr[100000];
   for (int i = 0; i < n; ++i) {
      std::cin >> arr[i];
   }

   int ordenado[100000];
   std::copy(&arr[0], &arr[0] + n, ordenado);
   std::sort(&ordenado[0], &ordenado[0] + n);

   int pos = std::find(&arr[0], &arr[0] + n, ordenado[n / 2]) - &arr[0];
   std::cout << ordenado[n / 2] << " " << pos << "\n";
}
```

3. Resuelve el problema `https://omegaup.com/arena/problem/Muletillas-de-robots`.

Solución: La idea clave para resolver este problema es darnos cuenta que, tras ordenar el arreglo, todos los elementos iguales quedan colocados de forma consecutiva. Así podremos barrer el arreglo para ir contando elementos consecutivos iguales y así registrar el elemento que más aparece.

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

int main( ) {
   int n;
   std::cin >> n;

   int arr[100000];
   for (int i = 0; i < n; ++i) {
      std::cin >> arr[i];
   }
   std::sort(&arr[0], &arr[0] + n);

   int mejor = -1, frecuencia_mejor = 0;
   int actual = -1, frecuencia_actual = 0;

   for (int i = 0; i < n; ++i) {
      if (actual == arr[i]) {
         frecuencia_actual += 1;
      } else {
         actual = arr[i];
         frecuencia_actual = 1;
```

```
      }

      if (frecuencia_actual > frecuencia_mejor) {
         mejor = actual;
         frecuencia_mejor = frecuencia_actual;
      }
   }

   std::cout << mejor;
}
```

Soluciones de 9.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Busqueda-binaria-p`.

Solución: Para resolver este problema, basta con ordenar la secuencia una vez y resolver cada búsqueda con el algoritmo de búsqueda binaria.

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

int main( ) {
   int n;
   std::cin >> n;

   int a[100000];
   for (int i = 0; i < n; ++i) {
      std::cin >> a[i];
   }
   std::sort(&a[0], &a[0] + n);

   int m;
   std::cin >> m;

   for (int i = 0; i < m; ++i) {
      int buscar;
      std::cin >> buscar;
      std::cout << std::binary_search(&a[0], &a[0] + n, buscar) << "\n";
   }
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/El-zorro-y-las-uvas`.

Solución: Si ordenamos la secuencia una vez, se vuelve posible encontrar rápidamente todos los elementos que son menores que cierto valor. Podemos usar el algoritmo `std::upper_bound` para posicionarnos en el primer elemento que está fuera de alcance, por lo que todos los de atrás entonces sí lo están. La cantidad de dichos elementos se puede obtener con una resta de apuntadores.

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

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

   int alturas[100000];
   for (int i = 0; i < n; ++i) {
      std::cin >> alturas[i];
   }
   std::sort(&alturas[0], &alturas[0] + n);

   int m;
   std::cin >> m;

   for (int i = 0; i < m; ++i) {
      int salto;
      std::cin >> salto;
      int* p = std::upper_bound(&alturas[0], &alturas[0] + n, salto);
      std::cout << p - &alturas[0] << " ";
   }
}
```

3. Resuelve el problema `https://omegaup.com/arena/problem/El-juego-de-la-silla`.

Solución: En este problema, conviene ordenar la secuencia una vez y resolver cada búsqueda con el algoritmo de búsqueda binaria. Lo mejor que puede ocurrir es que exista una silla justo en la posición que queremos, por lo que esa silla estaría a distancia 0 de nosotros. Sin embargo, si dicha silla no existe entonces debemos revisar la silla más cercana a la derecha (si es que existe) y la silla más cercana a la izquierda (si es que existe) para ver cuál conviene más. El algoritmo `std::lower_bound` nos permite encontrar la silla más cercana ahí o a la derecha. El problema ahora se reduce a revisar en qué situación estamos (no existe silla más cercana a la derecha pero sí a la izquieda, no existe silla más cercana a la izquierda pero sí a la derecha, existen sillas cercanas en ambas direcciones) e imprimir la respuesta que corresponda a la situación.

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

int main( ) {
   int n;
   std::cin >> n;

   int sillas[100000];
   for (int i = 0; i < n; ++i) {
      std::cin >> sillas[i];
   }
   std::sort(&sillas[0], &sillas[0] + n);

   int m;
   std::cin >> m;

   for (int i = 0; i < m; ++i) {
      int pos;
      std::cin >> pos;
```

```
      int* p = std::lower_bound(&sillas[0], &sillas[0] + n, pos);
      if (p == &sillas[0] + n) {
         std::cout << pos - *(p - 1) << "\n";
      } else if (p == &sillas[0]) {
         std::cout << *p - pos << "\n";
      } else {
         std::cout << std::min(*p - pos, pos - *(p - 1)) << "\n";
      }
   }
}
```

Soluciones de 10.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Ordenando-por-magnitud`.

Solución: Para resolver este problema, basta con ordenar la secuencia mediante un predicado que primero compare los valores absolutos de los elementos. Sólo en caso de empate en valor absoluto, se comparan sus valores directamente. Calcular el valor absoluto de un entero es fácil, pero la función `abs` que está declarada en `<stdlib.h>` calcula justamente esto.

```
#include <algorithm>
#include <iostream>
#include <stdlib.h>

int main( ) {
   int n;
   std::cin >> n;

   int arr[100000];
   for (int i = 0; i < n; ++i) {
      std::cin >> arr[i];
   }

   std::sort(&arr[0], &arr[0] + n, [](int a, int b) {
      int ma = abs(a), mb = abs(b);
      return (ma != mb ? ma < mb : a < b);
   });

   for (int i = 0; i < n; ++i) {
      std::cout << arr[i] << " ";
   }
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/Orden-raro-por-divisores`.

Solución: Para resolver este problema, basta con ordenar la secuencia mediante un predicado que primero compare la cantidad de divisores que tiene cada elemento. El ordenamiento por cantidad de divisores es ascendente. Sólo en caso de empate en cantidad de divisores, se comparan sus valores directamente para producir un orden descendente entre ellos. Para calcular la cantidad de divisores de un entero n, es recomendable declarar una función auxiliar que visita todos los enteros en el rango de 1 al n y cuente cuántos de ellos son divisores exactos.

```cpp
#include <algorithm>
#include <iostream>

int divisores(int n) {
   int res = 0;
   for (int i = 1; i <= n; ++i) {
      res += (n % i == 0);
   }
   return res;
}

int main( ) {
   int n;
   std::cin >> n;

   int arr[100000];
   for (int i = 0; i < n; ++i) {
      std::cin >> arr[i];
   }

   std::sort(&arr[0], &arr[0] + n, [](int a, int b) {
      int da = divisores(a), db = divisores(b);
      return (da != db ? da < db : a > b);
   });

   for (int i = 0; i < n; ++i) {
      std::cout << arr[i] << " ";
   }
}
```

3. Resuelve el problema `https://omegaup.com/arena/problem/Ordenando-por-distancia-al-orige`.

Solución: Para resolver este problema, basta con ordenar la secuencia mediante un predicado que primero compare las distancias de los puntos con respecto al origen. Tal distancia se puede calcular con la fórmula de la distancia entre dos puntos y es equivalente a la longitud de la hipotenusa con catetos de longitud x_i y y_i. Sólo en caso de empate en distancia, se comparan las coordenadas x directamente. Sólo en caso de empate tanto en distancia como en x, se comparan las coordenadas y. Calcular el valor de la hipotenusa es fácil, pero la función `hypot` que está declarada en `<math.h>` calcula justamente esto. De todos modos, ni siquiera es necesario comparar la distancia real, sino sólo determinar cuál de dos puntos (x_1, y_1) y (x_2, y_2) está más cerca del origen. Esto se puede hacer comparando $x_1^2 + y_1^2$ contra $x_2^2 + y_2^2$ sin necesidad de calcular la raíz cuadrada.

```cpp
#include <algorithm>
#include <iostream>
#include <math.h>

struct punto {
   int x, y;
};

int main( ) {
```

```
   int n;
   std::cin >> n;

   punto arr[100000];
   for (int i = 0; i < n; ++i) {
      std::cin >> arr[i].x >> arr[i].y;
   }

   std::sort((&arr[0], &arr[0] + n, [](punto p1, punto p2) {
      double d1 = hypot(p1.x, p1.y), d2 = hypot(p2.x, p2.y);
      return (d1 != d2 ? d1 < d2 : (p1.x != p2.x ? p1.x < p2.x : p1.y < p2.y));
   });

   for (int i = 0; i < n; ++i) {
      std::cout << arr[i].x << " " << arr[i].y << "\n";
   }
}
```

4. Resuelve el problema `https://omegaup.com/arena/problem/Contando-fechas-en-intervalos`.

Solución: Este problema se puede resolver restando directamente los apuntadores que calculan `std::lower_bound` y `std::upper_bound` al buscar los extremos de cada intervalo sobre el arreglo de fechas. Desde luego, para que las búsquedas binarias funcionen, el arreglo de fechas debe estar ordenado. Si usamos un tipo `struct` para almacenar las fechas, el predicado de ordenamiento debe comparar los componentes de las fechas en orden de importancia: año, mes y día. Si hay empate en alguna componente, intentamos desempatar con la siguiente componente. El predicado se usa tanto al ordenar con `std::sort` como al buscar con búsqueda binaria.

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

struct fecha {
   int dia, mes, año;
};

bool predicado(const fecha& a, const fecha& b) {
   if (a.año != b.año) {
      return a.año < b.año;
   } else if (a.mes != b.mes) {
      return a.mes < b.mes;
   } else {
      return a.dia < b.dia;
   }
}

int main( ) {
   int n;
   std::cin >> n;

   fecha arr[n];
   for (int i = 0; i < n; ++i) {
      std::cin >> arr[i].dia >> arr[i].mes >> arr[i].año;
   }
```

```
   std::sort(&arr[0], &arr[0] + n, predicado);

   int m;
   std::cin >> m;

   for (int i = 0; i < m; ++i) {
      fecha f1, f2;
      std::cin >> f1.dia >> f1.mes >> f1.año;
      std::cin >> f2.dia >> f2.mes >> f2.año;

      fecha* p1 = std::lower_bound(&arr[0], &arr[0] + n, f1, predicado);
      fecha* p2 = std::upper_bound(&arr[0], &arr[0] + n, f2, predicado);
      std::cout << p2 - p1 << "\n";
   }
}
```

Soluciones de 12.1

1. Resuelve el problema `https://omegaup.com/arena/problem/caracteres_subcadenas`.

Solución: En este problema podemos aplicar una estructura de datos llamada índice invertido: en lugar de trabajar sobre una cadena que mapea posiciones a valores, almacenaremos en un arreglo las posiciones que tienen el mismo valor. Tendremos un arreglo para cada posible valor, pero una cadena alfanumérica tiene menos de 128 caracteres distintos. Crearemos 128 arreglos redimensionables inicialmente vacíos (uno por cada caracter de la tabla ASCII) y con cada caracter visto seleccionaremos un arreglo en donde guardaremos la posición de dicho caracter. Cada posición se almacenará en un único arreglo, por lo que el consumo de memoria será proporcional al tamaño de la cadena. Para determinar si cierto caracter está en un rango de posiciones, haremos búsqueda binaria sobre el arreglo redimensionable que guarda las posiciones de dicho caracter.

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

int main( ) {
   std::string s;
   std::cin >> s;

   std::vector<int> torre[128];
   for (int i = 0; i < s.size( ); ++i) {
      torre[s[i]].push_back(i);
   }

   int n;
   std::cin >> n;

   for(int i = 0; i < n; ++i){
      char c; int ini, tam;
      std::cin >> c >> ini >> tam;
      int* p = std::lower_bound(&torre[c][0], &torre[c][0] + torre[c].size( ),
                                ini);
      std::cout << (p != &torre[c][0] + torre[c].size( ) && *p < ini + tam)
```

```
                << "\n";
   }
}
```

Soluciones de 14.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Dobles-colas`.

Solución: Este problema se presta perfectamente para ejercitar las operaciones sobre dobles colas. Las únicas operaciones que podrían ser complicadas de visualizar son las rotaciones. Sin embargo, una rotación izquierda se puede simular copiando el extremo izquierdo en el extremo derecho y luego quitando el original del extremo izquierdo. De forma similar para una rotación a la derecha.

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

int main( ) {
   int n;
   std::cin >> n;

   std::deque<int> dc;
   for (int i = 0; i < n; ++i) {
      std::string s;
      std::cin >> s;

      if (s == "AGREGA_IZQ") {
         int v;
         std::cin >> v;
         dc.push_front(v);
      } else if (s == "AGREGA_DER") {
         int v;
         std::cin >> v;
         dc.push_back(v);
      } else if (s == "QUITA_IZQ") {
         dc.pop_front( );
      } else if (s == "QUITA_DER") {
         dc.pop_back( );
      } else if (s == "ROTA_IZQ") {
         dc.push_back(dc.front( ));
         dc.pop_front( );
      } else if (s == "ROTA_DER") {
         dc.push_front(dc.back( ));
         dc.pop_back( );
      }
   }

   for (int i = 0; i < dc.size( ); ++i) {
      std::cout << dc[i] << " ";
   }
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/banco-clientes-no-preferentes`.

Solución: En este problema podemos usar dos dobles colas de la siguiente forma. Los clientes siempre llegan por atrás en ambas colas. En la primera doble cola, sacaremos a los clientes por el frente (para que el primero en llegar sea el primero en salir) y en la segunda sacaremos a los clientes por atrás (para que el último en llegar sea el primero en salir).

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

int main( ) {
   int n;
   std::cin >> n;

   std::deque<std::string> colas[2];
   for (int i = 0; i < n; ++i) {
      char evento; int ticket;
      std::cin >> evento >> ticket;
      ticket -= 1;

      if (evento == 'E') {
         std::string s;
         std::cin >> s;
         colas[ticket].push_back(s);
      } else if (evento == 'A') {
         if (ticket == 0) {
            std::cout << colas[0].front( ) << "\n";
            colas[0].pop_front( );
         } else if (ticket == 1) {
            std::cout << colas[1].back( ) << "\n";
            colas[1].pop_back( );
         }
      }
   }
}
```

Soluciones de 15.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Sumando-con-pilas`.

Solución: Este problema se puede resolver fácilmente con una pila (implementada ya sea con `std::vector`, `std::deque` o `std::stack`). Hay que recordar que las funciones `pop_back` y `pop` no regresan el valor que quitan, por lo que hay que consultarlo antes de quitarlo.

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

int main( ) {
   int n;
   std::cin >> n;

   std::vector<int> pila;
```

```
   for (int i = 0; i < n; ++i) {
      std::string s;
      std::cin >> s;

      if (s == "AGREGA") {
         int v;
         std::cin >> v;
         pila.push_back(v);
      } else if (s == "CONSUME") {
         int a = pila.back( );
         pila.pop_back( );
         int b = pila.back( );
         pila.pop_back( );
         pila.push_back(a + b);
      } else if (s == "IMPRIME") {
         std::cout << pila.back( ) << "\n";
      }
   }
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/grupos`.

Solución: Este problema se puede resolver fácilmente con una cola (implementada ya sea con `std::deque` o `std::queue`). Además, podemos usar `std::string` como el tipo de los elementos de la cola. El único problema es que la entrada no nos proporciona numéricamente la cantidad de eventos a procesar. Esto se puede resolver con el valor de retorno de la función `scanf` o también con `std::cin`. Si intentamos leer una variable con `std::cin` dentro de una condición, ésta será verdadera sólo si la lectura fue exitosa. Esto nos permitirá controlar el ciclo de lectura de eventos.

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

int main( ) {
   std::deque<std::string> cola;

   char e;
   while (std::cin >> e) {
      if (e == 'R') {
         std::string s;
         std::cin >> s;
         cola.push_back(s);
      } else if (e == 'P') {
         int k;
         std::cin >> k;

         if (cola.size( ) < k) {
            std::cout << "IMPOSIBLE\n";
         } else {
            for (int i = 0; i < k; ++i) {
               std::cout << cola.front( ) << " ";
               cola.pop_front( );
            }
```

```
            std::cout << "\n";
         }
      }
   }
}
```

Soluciones de 16.1

1. Resuelve el problema `https://omegaup.com/arena/problem/De-una-lista-a-otra`.

Solución: Una vez creadas las n listas enlazadas con el elemento respectivo para cada una, este problema se limita a llamar `splice` de la manera correcta. Una versión de `splice` que no indica una subsecuencia de la lista origen, en realidad migra todos los elementos de la misma.

```
#include <iostream>
#include <list>

int main( ) {
   int n;
   std::cin >> n;

   std::list<int> listas[n + 1];
   for (int i = 1; i <= n; ++i) {
      listas[i].push_back(i);
   }

   int m;
   std::cin >> m;

   for (int i = 0; i < m; ++i) {
      int x, y;
      std::cin >> x >> y;
      listas[x].splice(listas[x].end( ), listas[y]);
   }

   for (int i = 1; i <= n; ++i) {
      for (int v : listas[i]) {
         std::cout << v << " ";
      }
      std::cout << "\n";
   }
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/Reubicando-elementos-de-una-list`.

Solución: En este problema debemos aprovechar que los valores de la lista enlazada se conocen de antemano, son distintos y están en un rango limitado. Esto permite declarar un arreglo donde la posición i almacene el iterador al elemento con valor i de la lista. Conviene aprovechar que la función `insert` de `std::list` regresa un iterador al elemento recién insertado, aunque los iteradores también los podemos obtener tras ejecutar un `push_back` mediante un `std::prev` del fin. Reubicar un elemento se puede simular con una eliminación seguido de una inserción. Sólo

debemos tener cuidado de actualizar el iterador al nuevo nodo que almacena el valor reubicado.

```
#include <iostream>
#include <iterator>
#include <list>

int main( ) {
   int n;
   std::cin >> n;

   std::list<int> lista;
   std::list<int>::iterator iteradores[n];
   for (int i = 0; i < n; ++i) {
      iteradores[i] = lista.insert(lista.end( ), i);
   }

   int m;
   std::cin >> m;

   for (int i = 0; i < m; ++i) {
      int q, p; char c;
      std::cin >> q >> c >> p;

      lista.erase(iteradores[q]);
      if (c == 'A') {
         iteradores[q] = lista.insert(iteradores[p], q);
      } else if (c == 'D') {
         iteradores[q] = lista.insert(std::next(iteradores[p]), q);
      }
   }

   for (int actual : lista) {
      std::cout << actual << " ";
   }
}
```

Soluciones de 17.1

1. Resuelve el problema `https://omegaup.com/arena/problem/La-huida-de-los-jardineros-respo`.

Solución: Este problema es una aplicación trivial de una cola de prioridad. Cada evento del tipo 0 requiere imprimir y quitar el máximo elemento de la cola, mientras que cualquier otro evento requiere insertar en la cola de prioridad.

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

int main( ) {
   int n;
   std::cin >> n;

   std::priority_queue<int> cola;
```

```
   for (int i = 0; i < n; ++i) {
      int actual;
      std::cin >> actual;
      if (actual == 0) {
         std::cout << cola.top( ) << " ";
         cola.pop( );
      } else {
         cola.push(actual);
      }
   }
}
```

Soluciones de 18.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Arboles-binarios-sin-balancear`.

Solución: Este problema debe resolverse creando manualmente el árbol binario de búsqueda. Debemos modificar ligeramente el algoritmo de inserción para llevar la cuenta de cuántas veces hemos bajado por el árbol. Esto puede programarse fácilmente con recursión.

```
#include <iostream>

struct nodo {
   int valor;
   nodo* izq = nullptr;
   nodo* der = nullptr;
};

int inserta(nodo*& p, int v) {
   if (p == nullptr) {
      p = new nodo{v};
      return 0;
   } else if (v < p->valor) {
      return 1 + inserta(p->izq, v);
   } else {
      return 1 + inserta(p->der, v);
   }
}

int main( ) {
   int n;
   std::cin >> n;

   nodo* raiz = nullptr;
   for (int i = 0; i < n; ++i) {
      int actual;
      std::cin >> actual;
      std::cout << inserta(raiz, actual) << "\n";
   }
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/Encuestando-a-la-gente-de-la-col`.

Solución: Para resolver este problema, conviene usar un `std::deque` para agregar a los clientes por nombre y un `std::map` para guardar la frecuencia de cada nombre. Cada vez que un cliente entra, incrementaremos la frecuencia en el `std::map` del nombre dado. De forma similar, cada vez que un cliente sale, decrementaremos la frecuencia. En los eventos de impresión, conviene usar la función miembro `find` para evitar crear entradas innecesarias en el `std::map`.

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

int main( ) {
   int n;
   std::cin >> n;

   std::map<std::string, int> dicc;
   std::deque<std::string> cola;
   for (int i = 0; i < n; ++i) {
      char c;
      std::cin >> c;

      if (c == 'E') {
         std::string s;
         std::cin >> s;
         cola.push_back(s);
         dicc[cola.back( )] += 1;
      } else if (c == 'S') {
         dicc[cola.front( )] -= 1;
         cola.pop_front( );
      } else if (c == 'P') {
         std::string s;
         std::cin >> s;
         auto iter = dicc.find(s);
         std::cout << (iter == dicc.end( ) ? 0 : iter->second) << "\n";
      }
   }
}
```

Soluciones de 19.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Invirtiendo-aristas`.

Solución: Lo más fácil para resolver este problema es usar la representación de matriz de adyacencia para la gráfica. La operación de inversión de arista consistirá simplemente en negar la entrada correspondiente en ambos sentidos. La operación de inversión de un vértice puede implementarse con un ciclo para invertir completas la i-ésima fila y columna de la matriz.

```
#include <iostream>

int main( ) {
   int n, m;
   std::cin >> n >> m;
```

```
   bool matriz[1000][1000] = { };
   for (int i = 0; i < m; ++i) {
      char c;
      std::cin >> c;

      if (c == 'A') {
         int v1, v2;
         std::cin >> v1 >> v2;
         matriz[v1][v2] = !matriz[v1][v2];
         matriz[v2][v1] = !matriz[v2][v1];
      } else if (c == 'V') {
         int v;
         std::cin >> v;
         for (int i = 0; i < n; ++i) {
            matriz[v][i] = !matriz[v][i];
            matriz[i][v] = !matriz[i][v];
         }
      }
   }

   for (int i = 0; i < n; ++i) {
      for (int j = i + 1; j < n; ++j) {
         if (matriz[i][j]) {
            std::cout << i << " " << j << "\n";
         }
      }
   }
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/Divulgando-las-noticias`.

Solución: Para resolver este problema, conviene usar un enfoque distinto al descrito en las notas. Si bastara inspeccionar el tamaño actual de la cola para saber cuántas personas se acaban de enterar de la noticia, podríamos resolver el problema fácilmente. Pero entonces no conviene insertar elementos ya visitados en la cola. Por esta razón, cada vez que insertemos un elemento en la cola, lo marcaremos en el arreglo. También separaremos el procesamiento de los elementos de la cola en etapas, una por día. En cada etapa, sólo procesaremos los elementos que estaban en ese momento.

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

int main( ) {
   int n;
   std::cin >> n;

   std::vector<int> adyacencia[n];
   for (int i = 0; i < n; ++i) {
      int m;
      std::cin >> m;
      for (int j = 0; j < m; ++j) {
         int a;
         std::cin >> a;
```

```
            adyacencia[i].push_back(a);
        }
    }

    int k;
    std::cin >> k;

    std::vector<bool> visitados(n);
    std::deque<int> cola = { k };
    int res = 0; visitados[k] = true;

    do {
        for (int tam = cola.size( ); tam > 0; --tam) {
            int actual = cola.front( );
            cola.pop_front( );
            for (int vecino : adyacencia[actual]) {
                if (!visitados[vecino]) {
                    visitados[vecino] = true;
                    cola.push_back(vecino);
                }
            }
        }
        res = std::max(res, int(cola.size( )));
    } while (!cola.empty( ));

    std::cout << res << "\n";
}
```

3. Resuelve el problema `https://omegaup.com/arena/problem/Clausura-de-carreteras`.

Solución: A pesar de lo complicado que pueda parecer este problema, podemos resolverlo simplemente ejecutando el algoritmo de Dijkstra y desempatando vértices igualmente cercanos por el criterio de costos de Prim. Sólo debemos tener cuidado en usar `long long` para acumular distancias y costos, ya que éstos pueden ser muy grandes.

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

struct destino {
    int vertice;
    long long distancia, costo;
};

bool operator<(destino d1, destino d2) {
    if (d1.distancia != d2.distancia) {
        return d1.distancia > d2.distancia;
    } else {
        return d1.costo > d2.costo;
    }
}
```

```
int main( ) {
   int n, m;
   std::cin >> n >> m;

   std::vector<destino> adyacencia[n];
   for (int i = 0; i < m; ++i) {
      int v1, v2, d, c;
      std::cin >> v1 >> v2 >> d >> c;
      adyacencia[v1 - 1].push_back(destino{v2 - 1, d, c});
      adyacencia[v2 - 1].push_back(destino{v1 - 1, d, c});
   }

   bool vistos[n] = { };
   std::priority_queue<destino> cola;
   cola.push(destino{0, 0, 0});
   long long suma_aristas = 0;

   do {
      auto actual = cola.top( );
      cola.pop( );

      if (!vistos[actual.vertice]) {
         vistos[actual.vertice] = true;
         suma_aristas += actual.costo;
         for (auto directa : adyacencia[actual.vertice]) {
            cola.push(destino{
               directa.vertice,
               actual.distancia + directa.distancia, directa.costo
            });
         }
      }
   } while (!cola.empty( ));

   std::cout << suma_aristas << "\n";
}
```