Notas de curso
Almacenamiento y Estructuras de Archivos

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

Última revisión: 7 de junio de 2023

1. Algoritmia, tecnología y eficiencia

Conforme pasan los años y las computadoras se aplican en más y más actividades humanas, la magnitud de los cálculos que éstas realizan sigue aumentando. Desafortunadamente, el poder de cómputo y almacenamiento disponibles siempre está varios pasos atrás de lo que realmente quisieramos tener. Se ha vuelto muy común que renovemos el hardware de nuevos dispositivos (ya sean móviles o de escritorio) cada cierta cantidad de años, todo con el afán de tener un procesador más rápido o tener más memoria. El poder de cómputo que parezca ser suficiente hoy, casi con seguridad no lo parecerá en diez años.

Un ingenierio en computación debe aprender a usar la computadora de forma *eficiente*, pero no le bastará llegar a la conclusión de que sus programas son eficientes en una computadora teórica: debe ejecutarlos en una computadora real y verificarlo. Muchos cursos de algoritmia se plantean bajo suposiciones aparentemente inocentes, como que cada operación individual dentro de un algoritmo (operación aritmética, comparación y acceso a memoria) tarda una unidad de tiempo. Estas suposiciones son propias de las ciencias de la computación, donde al estudio de los algoritmos no le importa el hardware en el que se ejecutan las implementaciones en software de los mismos. Un ingeniero en computación debe estudiar tanto la eficiencia teórica como la práctica, porque las suposiciones anteriores son falsas en el hardware actual. A su vez, a un ingeniero en computación le importan las limitaciones prácticas que impone el poder de cómputo disponible: para que los usuarios de un videojuego de computadora puedan tener una buena experiencia, los desarrolladores suelen proveer de opciones que permitan configurar algunos aspectos del mismo dependiendo, por ejemplo, de cuánta memoria tiene disponible la computadora.

Los procesadores modernos pueden ejecutar operaciones aritméticas sencillas como sumas o comparaciones en un solo ciclo de reloj del procesador, 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 aún así, en este curso mantendremos la suposición de que todas ellas toman la misma cantidad de tiempo, ya que éstas suelen ser muy rápidas, comparativamente hablando. La suposición que eliminaremos entonces es la de que un acceso a memoria es comparable en tiempo con las demás operaciones. No todos los accesos a memoria son iguales; peor aún, ¡no todos los tipos de memoria son iguales! En este curso estudiaremos los aspectos de eficiencia y de capacidad de los distintos tipos de memoria que están disponibles en una computadora, para así poder diseñar e implementar algoritmos eficientes que usen los recursos del hardware de forma adecuada.

2. El procesador y la memoria principal

Aunque los procesadores se han vuelto muy rápidos en las últimas décadas y las memorias han aumentado su muchísimo su capacidad, las memorias no han aumentado su velocidad al mismo ritmo. Por esta razón, los procesadores actuales frecuentemente deben esperar decenas o cientos de ciclos de reloj a que los valores de los operandos de una instrucción estén disponibles. A la creciente disparidad entre

las velocidades del procesador y la memoria se le conoce como pared de memoria. Para ejemplificar este tema, lo primero que haremos será realizar una prueba de rendimiento entre el procesador y la memoria.

El siguiente programa declara un arreglo de capacidad máxima m y una función f que considerará únicamente los primeros t elementos del arreglo. El algoritmo principal se ejecutará con valores distintos de t. Se desea realizar una cantidad de trabajo igual a w, por lo que el algoritmo principal elegirá una t que divida a w y llamará a f exactamente $\frac{w}{t}$ veces. En principio, la función f se limita a escribir un valor v sobre los primeros t elementos del arreglo, aunque es posible indicarle que sólo escriba sobre uno de cada p elementos. La decisión de variar p es completamente independiente de la elección de t y de w. El programa debe compilarse con optimización (por ejemplo, con la bandera `-O3` de gcc) para que los resultados sean significativos. El valor de v es aleatorio para evitar que el optimizador simplifique de más.

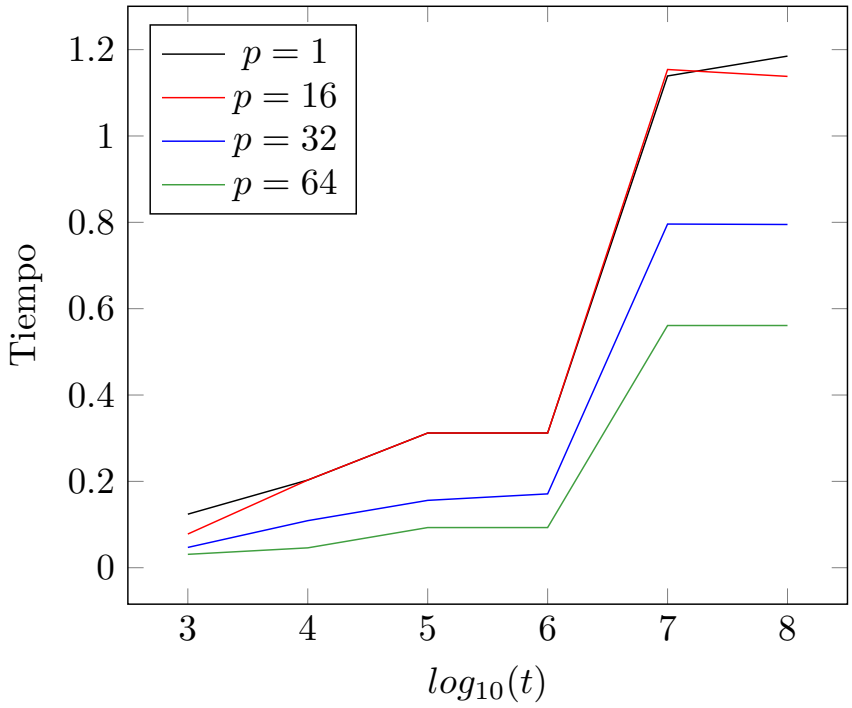
Código

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void f(int arr[], int t, int v, int p = 1) {
   for (int i = 0; i < t; i += p) {
      arr[i] = v;
   }
}

int main( ) {
   int m = 1e8, *arr = new int[m], w = 2e9;
   for (int t = 1000; t <= m; t *= 10) {
      printf("\nEnteros=%d, veces=%d\n", t, w / t);
      for (int p = 1; p <= 256; p *= 2) {
         printf("   Paso %d:\n", p);
         clock_t t0 = clock( );
         for (int i = 0; i < w / t; ++i) {
            f(arr, t, rand( ), p);
         }
         clock_t t1 = clock( );
         printf("      %f\n", double(t1 - t0) / CLOCKS_PER_SEC);
      }
   }
}
```

El tiempo de ejecución del programa muestra un comportamiento extraño, por no decir perturbador.

	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$
$t=10^3$	0.12	0.36	0.20	0.12	0.08	0.05	0.03	0.03
$t=10^4$	0.20	0.31	0.22	0.20	0.20	0.11	0.05	0.03
$t=10^5$	0.31	0.36	0.31	0.31	0.31	0.16	0.09	0.03
$t=10^6$	0.31	0.36	0.31	0.31	0.31	0.17	0.09	0.03
$t=10^7$	1.14	1.14	1.14	1.14	1.15	0.80	0.56	0.16
$t=10^8$	1.19	1.14	1.14	1.14	1.14	0.80	0.56	0.17

Tiempos de ejecución en segundos para $m=10^8$ y $w=10^9$ en un procesador Intel Core i7 5820k.



Gráfica de los tiempos de ejecución en escala logarítmica sobre t

Resulta inicialmente sorprendente que el tiempo de ejecución no se mantenga constante a pesar de que w lo es, ya que el tiempo parece incrementarse súbitamente en $t = 10^4$, $t = 10^5$ y $t = 10^7$. Más sorprendente aún resulta el hecho de que el tiempo de ejecución prácticamente no varía cuando p está en el rango de 1 a 16, a pesar de que la función f sólo escribe sobre $\frac{1}{p}$ de los elementos del arreglo. En otras palabras, escribir sobre cada elemento tarda lo misma que escribir un elemento y luego esquivar los siguientes quince. Debemos resaltar que el procesador *sí* está haciendo sólo un $\frac{1}{p}$ del trabajo. Lo que está provocando ambos fenómenos son los accesos a memoria.

En principio, las computadores actuales van a frecuencias de reloj que van de 2 a 4 GHz aproximadamente. Esos miles de millones de ciclos por segundo idealmente se traducen en miles de millones de instrucciones ariméticas ejecutadas en ese tiempo (con algunos bemoles: una suma suele tomar un ciclo, mientras que una división toma hasta 40 ciclos). Sin embargo, a pesar de que la existencia de operaciones aritméticas más costosas que otras pareciera ser motivo de preocupación, lo anterior es insignificante en comparación con el tiempo que puede tomar un acceso a memoria principal. Actualmente, la memoria principal (la mal nombrada RAM, que es un término mucho más general) tiene una capacidad de decenas de gigabytes pero tiene tiempos de respuesta de la orden de 100 nanosegundos, el cual es un tiempo potencialmente perdido para el procesador. En 1995, cuando los procesadores corrían a 100 MHz, 100 nanosegundos de espera representaban únicamente 10 ciclos de reloj. Sin embargo, para una computadora actual que vaya a 3 GHz, la misma cantidad de tiempo representa 300 ciclos de reloj.

Los diseñadores de procesadores conocen esta situación y proveen de memorias cachés mucho más pequeñas que la memoria principal, pero mucho más rápidas. La mayoría de los procesadores modernos cuentan con tres niveles de caché llamados cachés L1, L2 y L3; cada nivel siendo lento pero más grande el anterior. A su vez, los procesadores cuentan con unas pocas decenas de celdas especiales de memoria llamadas registros, los cuales que pueden accesarse de forma instantánea. La idea es que los operandos de las instrucciones aritméticas estén almacenados en la memoria más rápida, para evitar tener que accesar a la memoria principal en la medida de lo posible. El sitio web `https://www.7-cpu.com` lista datos experimentales de acceso a memoria para varios procesadores y también provee de los programas que se usaron para obtener dichos tiempos. Con base en dichas pruebas de rendimiento y la hoja de datos oficial[1], el procesador Intel Core i7 5820k a 3.6 GHz cuenta con las siguientes características:

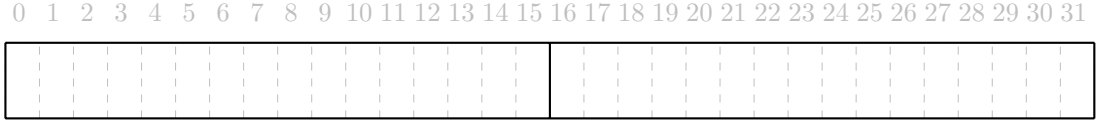
[1] `https://www.intel.com/content/www/us/en/processors/core/core-i7-lga2011-3-datasheet-vol-1.html`

Nivel de memoria	Tamaño	Tiempo de acceso experimental
Registro	De 1 a 32 bytes cada uno (menos de 4 kB en total)	Instantáneo
Caché L1	32 kB por núcleo	5 ciclos
Caché L2	256 kB por núcleo	12 ciclos
Caché L3	15 MB para todos los núcleos (compartida)	43 ciclos
Memoria principal	Externa al procesador (generalmente varios GB)	De 107 a 265 ciclos

Cuando un dato se trae de memoria principal, generalmente se copia en todos los niveles de caché. Si algún nivel de caché ya está lleno, se suele expulsar algún dato que no haya sido usado recientemente para que el dato recién accesado puede ocupar su lugar. Si se accesa a un dato y poco tiempo después se quiere volver a accesarlo, es probable que aún permanezca en alguno de los niveles de caché. Lo siguiente explica parcialmente el comportamiento del programa:

- Cuando el programa trabaja sobre 10^3 elementos, el arreglo se transferirá de la memoria principal sólo la primera vez. Los demás accesos podrán ser satisfechos por la caché L1.
- Cuando el programa trabaja sobre 10^4 elementos, el arreglo se transferirá de la de memoria principal sólo la primera vez. Los demás accesos podrán ser satisfechos por la caché L2. En este caso, la caché L1 irá copiando los primeros elementos cuando son accesados, pero éstos serán expulsados por falta de espacio conforme se itere sobre el resto del arreglo. Cuando se vuelva a iterar sobre el arreglo desde el comienzo, los primeros elementos de arreglo ya no se encontrarán en la caché L1 pero sí en la L2.
- Cuando el programa trabaja sobre 10^5 o 10^6 elementos, el arreglo se traerá de memoria principal sólo la primera vez. Los demás accesos podrán ser satisfechos por la caché L3. En este caso, las cachés L1 y L2 irán copiando los primeros elementos cuando son accesados, pero éstos serán expulsados por falta de espacio conforme se itere sobre el resto del arreglo. Cuando se vuelva a iterar sobre el arreglo desde el comienzo, los primeros elementos ya no se encontrarán en las cachés L1 o L2 pero sí en la L3.
- Cuando el programa trabaja sobre 10^7 elementos o más, el arreglo transferirá de la de memoria principal cada vez que se itere sobre él. En este caso, las cachés L1, L2 y L3 irán copiando los elementos recién accesados, pero éstos serán expulsados por falta de espacio conforme se itere sobre el resto del arreglo. Cuando se vuelva a iterar sobre el arreglo desde el comienzo, los primeros elementos ya no se encontrarán en ningún nivel de caché. Esto se repetirá en cada iteración.

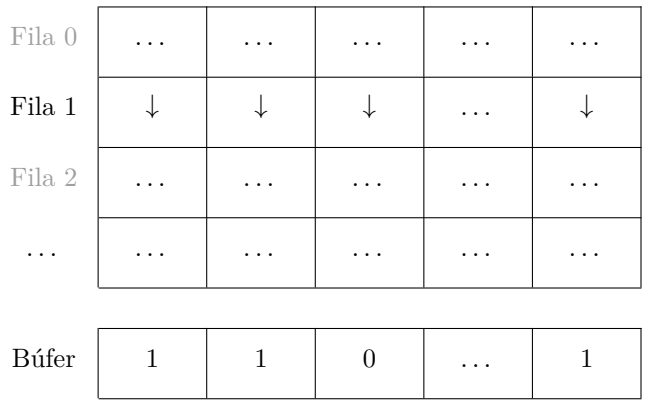
Con todo, lo anterior aún no explica por qué los tiempos reportados para $t \geq 10^4$ y $1 \leq p \leq 16$ son prácticamente los mismos. ¿Qué acaso no se hacen menos accesos a memoria conforme p aumenta? La respuesta es *no*: la cantidad de accesos es la misma y los tiempos de acceso a memoria son suficientemente altos como para éstos dominen el tiempo total de ejecución. La razón es la siguiente: aún si el programador sólo necesita accesar a un único byte, los procesadores actuales manejan los accesos a memoria con una granularidad de línea de caché o línea de coherencia, usualmente de 64 bytes. Esto se hace así porque la mayoría de los programas accesan a secuencias de datos y no a datos individuales. Si un entero ocupa 4 bytes, entonces un único acceso a memoria traerá 16 enteros y no sólo uno; si queremos iterar sobre un arreglo, pagaremos el costo de un acceso a memoria por cada 16 elementos, y no por cada uno.



Una línea de caché ocupa 64 bytes y un entero ocupa 4 bytes.
Un único acceso a memoria transferirá 16 enteros.

El costo total se incurre *aún si no tocamos los demás elementos*. Cuando $1 < p \leq 16$, los elementos que son ignorados de todos modos fueron traídos de memoria, por lo que el tiempo de ejecución no varía (en particular, porque el costo de la aritmética del lado del procesador resultó ser insignificante en comparación con el costo de acceso a memoria). Sólo si $p > 16$ vemos que el tiempo de ejecución baja. Por ejemplo, cuando $p = 32$ entonces una línea de caché de 16 elementos sí se trae de memoria (aunque sea sólo para tocar uno de los elementos) pero la siguiente línea no se traerá. La existencia de las líneas de caché explica por qué, en lenguajes derivados de C, es más eficiente iterar sobre una matriz por filas en lugar de por columnas: en estos lenguajes, los elementos de la misma fila aparecen contiguos en memoria y una línea de caché contiene varios de ellos, maximizando el aprovechamiento de la misma. En cambio, si una matriz es grande y se itera por columnas, los elementos de filas distintas estarán en líneas de caché distintas y visitar todas las filas de la misma columna saturará la caché; al cambiar de una columna a otra, las primeras filas de la matriz ya no estarán en la caché.

Retomando los tiempos de acceso para el procesador Intel Core i7 5820k, podemos notar que existe mucha variación en los tiempos de acceso a la memoria principal y nada de lo dicho anteriormente explica esto todavía. La principal razón es que tecnología de la memoria principal es complicada al deber ser de gran capacidad, aún si es más lenta. Por una parte, se debe cuidar que el tamaño de la memoria no eleve demasiado su consumo de energía, por lo que la tecnología actual usa celdas de memoria que no reciben energía eléctrica de forma permanente pero que, en consecuencia, sólo retienen su carga poco tiempo. Antes de que las celdas pierdan su carga, los circuitos de la memoria principal *refrescan* la carga de las mismas, pero hacer esto tarda un tiempo considerable. Un refresco ocurre aproximadamente cada 10^4 nanosegundos y dura aproximadamente 100 nanosegundos. Mientras se refresca, el chip de memoria se bloquea. Por otra parte, las celdas de la memoria principal están físicamente organizadas en matrices bidimensionales y el controlador de la memoria principal es el encargado de mapear direcciones lógicas en términos de filas y columnas físicas. Para accesar a un byte de la matriz, la fila completa donde éste reside debe cargarse primero en un búfer. Accesos posteriores a datos de la misma fila se atienden directamente desde el búfer. Cuando se necesita accesar a un dato de una fila distinta, el contenido del búfer se escribe de vuelta a la matriz (esto es el *cierre* de la fila) y posteriormente se carga la otra fila.



Cuando el controlador de la memoria principal selecciona una fila, ésta se copia completa en un búfer. Las operaciones sobre la fila se hacen en el búfer.

En las memorias actuales, las operaciones de cerrar una fila, cargar una fila, y cambiar de columna dentro de la fila tardan aproximadamente 15 nanosegundos cada una. Entonces, un acceso que sólo requiera cambiar de columna en una fila ya cargada será más rápido que un acceso que requiera cambiar de fila. Desafortunadamente, en las últimas décadas no han ocurrido reducciones significativas en los tiempos de estas operaciones; a pesar de que existen memorias principales que corren a varios GHz, esta velocidad únicamente es la velocidad a la que la memoria y el procesador pueden comunicarse: que la comunicación sea rápida *no implica* que las operaciones individuales dentro la memoria también lo sean. Además, recordemos que en una computadora que vaya a 3.6 GHz, esperar los 45 nanosegundos del peor caso es equivalente a esperar 162 ciclos de reloj. Por si fuera poco, un acceso a memoria principal también tiene como sobrecarga adicional el tiempo que tarda el procesador en darse cuenta que el dato no está en ningún nivel de caché y que, en efecto, debe ir por él a memoria principal. Esta sobrecarga generalmente

es igual a la suma de los tiempos de acceso de cada nivel de caché, lo que es aproximadamente 60 ciclos en una computadora moderna. Dicho lo anterior, es fácil ver por qué cierto acceso a memoria principal puede tardar poco más de 100 ciclos, mientras que otro puede tardar más de 200 ciclos. Si tenemos mala suerte, incluso podría ocurrir un ciclo de refresco en la memoria que nos obligue a esperar todavía más.

En las últimas décadas, la cantidad de filas por matriz de memoria se ha mantenido en decenas de miles y la cantidad de bytes por fila ha variado considerablemente (por ejemplo, las memorias DDR2 tenían filas de 8 kB cada una, las DDR4 de 512 bytes y las DDR5 de 1 kB). El aumento en la capacidad de la memoria principal se ha dado principalmente por la existencia de múltiples matrices por chip y de múltiples chips por módulo de memoria. La memoria se reparte en las matrices para minimizar la probabilidad de que dos accesos a memoria caigan en la misma matriz pero en distintas filas, lo que obligaría a cerrar una fila para abrir otra. Además, los distintos chips pueden trabajar en paralelo.

Actualmente, la memoria principal provee una tasa de transferencia que va de 10 a 25 GB por segundo, dependiendo de la carga de trabajo (los accesos secuenciales serán cerca de tres veces más rápidos que los accesos arbitrarios, justo porque incurren en menos cambios de filas). Cuando la mayor parte del tiempo de ejecución de un programa es causado por los accesos a memoria, se dice que el cuello de botella del programa es la memoria. Desafortunadamente, cada vez es más frecuente que ocurra justamente esta situación. Los algoritmos y estructuras de datos conscientes de la caché son aquéllos en consideran en sus diseños los tiempos de acceso a memoria, la presencia de las cachés y sus tamaños.

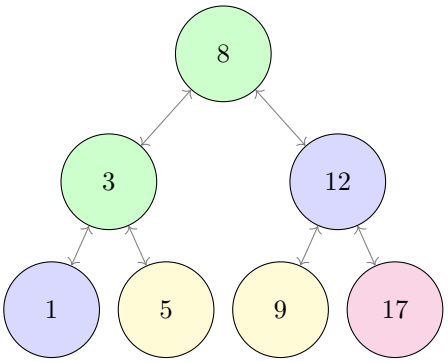
2.1. Ejercicios

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/Invirtiendo-segmentos-de-una-mat`.

3. Conjuntos y diccionarios desordenados (tablas de dispersión)

Un conjunto es una estructura de datos abstracta que permite realizar eficientemente las siguientes operaciones: agregar elementos, buscar elementos y eliminar elementos. Un diccionario permite realizar eficientemente las siguientes operaciones: agregar parejas (clave, valor), buscar parejas por clave y eliminar parejas por clave. Una estructura de datos concreta que permite implementar ambas es el árbol binario de búsqueda balanceado, el cual provee todas las operaciones anteriores en un tiempo proporcional a $\log_2(n)$ donde n es la cantidad de elementos en la estructura, además de que tiene la propiedad de poder iterar sobre los elementos en orden. Si la iteración en orden no se necesita, conviene preguntarnos si esta implementación en verdad es la mejor. En un espacio de memoria direccionada con 64 bits, cada apuntador ocupará 8 bytes y además los algoritmos de iteración y balanceo con frecuencia necesitan almacenar apuntador al nodo padre y no sólo a los hijos. Por esto, sólo cabrán dos nodos por línea de caché y los $\log_2(n)$ pasos serán, con alta probabilidad, accesos a memoria principal.




```
// sizeof(nodo) == 32
struct nodo {
   int valor;
   nodo* izq;
   nodo* der;
   nodo* padre;
};
```

Un árbol binario de búsqueda balanceado tras insertar 8, 3, 12, 1, 5, 9, 17. Nodos del mismo color comparten línea de caché. Un cambio de nivel es un potencial cambio de línea.

Queremos diseñar una estructura de datos concreta que realice sus operaciones de inserción, búsqueda y eliminación en la menor cantidad posible de accesos a memoria. En particular, sería excelente si bastara un solo acceso. El poco margen de maniobra con el que contamos para diseñarla implica que, en cierto modo, ésta debe ser mucho más sencilla que un árbol binario de búsqueda balanceado. De hecho, la estructura de datos que buscamos surgirá de encontrar un nuevo uso para un arreglo ordinario.

Para introducir la idea, la cual desarrollaremos poco a poco, nos concentraremos primero en resolver el siguiente problema: queremos poder insertar, buscar o eliminar elementos de un conjunto de enteros, donde además todos los enteros están en el rango de 0 a K. Lo anterior se puede resolver declarando un arreglo de $K+1$ booleanos, donde el i-ésimo elemento del arreglo almacena verdadero o falso dependiendo si el valor i está o no en el conjunto, respectivamente. Accesar al i-ésimo elemento de un arreglo toma a lo mucho un acceso a memoria principal, sin importar el valor de i.

Conjunto de enteros en el rango de 0 a K	
Operación	**Implementación**
Inicializar al conjunto vacío.	`bool arr[K + 1] = { };`
Agregar el entero v.	`arr[v] = true;`
Eliminar el entero v.	`arr[v] = false;`
Determinar si v aparece en el conjunto.	`bool b = arr[v];`

Por supuesto, una debilidad de este enfoque es que el arreglo consumirá mucha memoria si el valor de K es grande. Por ahora ignoraremos este detalle y nos concentraremos en generalizar la misma idea para conjuntos que no sean de enteros. Por ejemplo, nos gustaría poder insertar, buscar o eliminar cadenas, donde además las cadenas están formadas por 4 letras minúsculas del alfabeto inglés. Para poder emplear la misma estrategia que en el problema anterior, debemos encontrar una forma de asociarle un índice del arreglo a cada posible cadena. Esto lo haremos con lo que se llama una función de dispersión.

Una función de dispersión es una función que toma un valor en un dominio y le asocia (de forma determinista) un entero no negativo o *hash*. Los enteros del codominio generalmente son de ancho fijo, usualmente 32 o 64 bits. El lenguaje C provee del archivo de biblioteca `<stdint.h>`, el cual define los tipos `int32_t`, `int64_t`, `uint32_t` y `uint64_t`, entre otros. Estos tipos tienen un ancho garantizado en bits y además los dos últimos no tienen signo. En el problema que queremos resolver, hay 26 letras en el alfabeto inglés y entonces hay 26^4 cadenas posibles. La función que se muestra a continuación mapeará las cadenas a enteros en el rango de 0 a $26^4 - 1$ simplemente interpretando la cadena como si fuera un entero en base 26, de izquierda a derecha. La letra `a` la consideraremos 0, la `b` será 1, la `c` será 2, etc.

Código

```
// función de dispersión para cadenas alfabéticas de longitud 4
uint64_t hash1(const char s[]) {
   uint64_t res = 0;
   for (int i = 0, p = 1; i < 4; ++i, p *= 26) {
      res += p * (s[i] - 'a');
   }
   return res;
}
```

Conjunto de cadenas alfabéticas de tamaño 4	
Operación	**Implementación**
Inicializar al conjunto vacío.	`bool arr[456976] = { };`
Agregar la cadena s.	`arr[hash1(s)] = true;`
Eliminar la cadena s.	`arr[hash1(s)] = false;`
Determinar si s aparece en el conjunto.	`bool b = arr[hash1(s)];`

La función anterior además tiene la propiedad de ser *perfecta*: una función de dispersión es perfecta si cada valor del dominio tiene asociado un hash distinto. Afortunadamente, en el problema de ejemplo la cantidad de cadenas alfabéticas de tamaño 4 es mucho menor que el rango de un `uint64_t`.

Ahora resolveremos una variante del mismo problema, la cual considera cadenas de entre 0 y 4 letras minúsculas, en lugar de tener exactamente 4. Hay $26^0 + 26^1 + 26^2 + 26^3 + 26^4 = 475255$ de estas cadenas y la estrategia puede ser la misma; sólo necesitamos adecuar la función de dispersión. Para evitar confundir una letra presente con la ausencia de una letra, la asignación de valores a las letras individuales ahora comenzará en 1 (es decir, la letra `a` valdrá 1, la letra `b` valdrá 2, etc).

Código

```
// función de dispersión para cadenas alfabéticas de longitud variable
uint64_t hash2(const char s[], int tam) {
   uint64_t res = 0;
   for (int i = 0, p = 1; i < tam; ++i, p *= 26) {
      res += p * (s[i] - 'a' + 1);
   }
   return res;
}
```

Conjunto de cadenas alfabéticas de tamaño entre 0 y 4	
Operación	**Implementación**
Inicializar al conjunto vacío.	`bool arr[475255] = { };`
Agregar la cadena s.	`arr[hash2(s)] = true;`
Eliminar la cadena s.	`arr[hash2(s)] = false;`
Determinar si s aparece en el conjunto.	`bool b = arr[hash2(s)];`

Aunque hemos tenido éxito resolviendo los problemas anteriores, el verdadero problema aparece si ahora quitamos las restricciones sobre el contenido o la longitud de las cadenas. De entrada, la cantidad de cadenas sin restricciones es infinita y el rango de un `uint64_t` es finito, lo que garantiza que existan colusiones: cadenas distintas mapeadas al mismo entero. Al menos podríamos intentar evitar las colisiones descuidadas si la función de dispersión cumple la siguiente propiedad: ante un valor aleatorio del dominio, todos los valores del codominio deben ser igualmente probables. Lograr lo anterior es complicado y es mejor delegar la tarea de diseñar funciones de dispersión a los especialistas, pero en principio, una función de dispersión relativamente "justa" que suele usarse como ejemplo, es la que se reproduce a continuación y que fue diseñada por Robert Sedgewick, un famoso autor de libros de computación:

Código

```
uint32_t rs_hash(const char arr[], int tam) {
   uint32_t a = 63689, b = 378551, res = 0;
   for (int i = 0; i < tam; ++i, a *= b) {
      res = (a * res) + arr[i];
   }
   return res % 2147483648;
}
```

Algunas funciones de dispersión de mejor calidad (pero casi siempre más lentas) están disponibles en `https://gist.github.com/qxj/1520414` y en `https://github.com/brandondahler/retter`. Algunas de las funciones ahí listadas también tienen propiedades criptográficas, lo que en términos generales quiere decir son sumamente difíciles de invertir; es decir, dado el valor del hash, es difícil producir una cadena con ese hash. Las funciones de dispersión criptográficas se usan mucho en aplicaciones de seguridad, pero no tanto en la implementación de estructuras de datos debido a su alto tiempo de cómputo.

Suponiendo que contamos con una función de dispersión adecuada, la pregunta ahora es cómo emplearla para insertar, buscar y eliminar cadenas si pueden ocurrir colisiones y además no podemos declarar un arreglo ni infinito ni ridículamente grande de booleanos. La respuesta consta de dos partes:

- Ante la presencia de colisiones, necesariamente debemos desambiguar de alguna forma. Lo que haremos será cambiar el arreglo de booleanos por un arreglo de *cubetas*, donde cada cubeta guarde una copia completa de las cadenas insertadas en ella. Al buscar una cadena, usaremos su hash para posicionarnos en la cubeta que le corresponda y la buscaremos ahí (posiblemente con búsqueda lineal).

- Si tendremos un máximo de n cadenas insertadas, podemos declarar un arreglo de n cubetas. Si el hash de una cadena fuera un índice inválido del arreglo por tener un valor demasiado grande, podemos usar la operación módulo n para calcular un índice válido. Si la función de dispersión es de buena calidad y las cadenas a insertar son aleatorias, se esperaría que sólo haya una cadena por cubeta.

Conjunto de n cadenas	
Operación	**Implementación**
Inicializar al conjunto vacío.	`std::set<std::string> arr[n];`
Agregar la cadena s.	`arr[hash(s) % n].insert(s);`
Eliminar la cadena s.	`arr[hash(s) % n].erase(s);`
Determinar si s aparece en el conjunto.	`bool b = arr[hash(s) % n].contains(s);`

A la estructura de datos concreta que resulta de combinar las ideas anteriores se le conoce como tabla de dispersión o tabla hash. En general, pueden usar las tablas de dispersión para almacenar conjuntos de elementos de cualquier tipo que cuente con una función de dispersión adecuada, no solamente cadenas. Un diccionario se puede implementar usando la misma estrategia, simplemente asociándole un valor adicional (que no participa en el cálculo del hash) a cada elemento insertado. Existen dos variantes principales de tablas de dispersión, dependiendo de la estructura de datos usada para las cubetas:

- Si la cubeta es una estructura de datos dinámica (por ejemplo, un arreglo redimensionable, una lista enlazada o incluso un árbol binario de búsqueda) entonces se dice que la tabla usa encadenamiento separado. La memoria de las estructuras dinámicas es ajena a la memoria del arreglo, por lo que las tablas de dispersión con encademiento requieren al menos dos accesos a memoria por búsqueda: el acceso para obtener el enlace a la cubeta y el acceso a la cubeta en sí.
- Si la cubeta es un arreglo de capacidad fija (usualmente del tamaño de una línea de caché), entonces se dice que la tabla usa direccionamiento abierto por la siguiente razón: cuando una de estas cubetas se llena y queremos realizar otra inserción en la misma cubeta, entonces el elemento se insertará en algún otro lado. Lo normal es marcar la cubeta que rechaza al elemento por falta de espacio con una señalización de "sobreflujo" e insertar el elemento rechazado en alguna cubeta que tenga espacio, aún si ésta no corresponde con el hash de el elemento. Hay una gran diversidad de estrategias para elegir esta otra cubeta, pero se suele usar una cercana a la cubeta original. En este tipo de tabla, la memoria de la cubeta pertenece al arreglo y puede bastar un único acceso a memoria por búsqueda.

Tabla de dispersión de cinco cubetas para un conjunto de enteros { 25, 7, 42, 18 }.
El hash de un entero no negativo suele ser él mismo.

El factor de carga de una tabla de dispersión es la cantidad promedio de elementos por cubeta y se calcula como $\frac{n}{c}$ donde n es el número de elementos insertados y c es el número de cubetas. Permitir un factor de carga alto aumenta la probabilidad de que la búsqueda degenere en una búsqueda lineal. Para reducir la probabilidad de que las cubetas crezcan demasiado con direccionamiento separado u ocurra sobreflujo con direccionamiento abierto, se recomienda un factor de carga menor a 1. Si la cantidad de elementos a insertar se desconoce de antemano y el factor de carga aumenta peligrosamente, se recomienda migrar los elementos a una nueva tabla que cuente con más cubetas, usualmente el doble que la anterior.

3.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Busqueda-muy-eficiente-de-elemen`.

4. Archivos, su clasificación y los flujos estándar

Un archivo es, según el sistema operativo UNIX y sus derivados (Linux, Android, MacOS, etc), una secuencia o flujo de bytes. Esta definición amplia permite afirmar que prácticamente *todo es un archivo*:

- Los arreglos de bytes en memoria.
- Los bytes que se transfieren entre dos programas en ejecución.
- Los bytes que se transfieren entre varias computadoras o entre una computadora y sus periféricos.
- Los datos almacenados de forma persistente en algún dispositivo de memoria no volátil (por ejemplo, en discos duros o memorias USB).

Sin embargo, la intuición nos dice que no todos los archivos son iguales. Se vuelve necesario entonces clasificarlos dependiendo de sus propiedades. En general, existen dos tipos de archivos:

- *Archivos de caracter*: son secuencias de bytes cuya lectura o escritura sólo se puede hacer una vez: la lectura no se puede repetir y la escritura no se puede deshacer. Por ejemplo, un dispositivo de red no puede deshacer el envío de los datos que ya transfirió. De forma similar, si un dispositivo de red no está listo para recibir los datos que están llegando en ese momento, dichos datos simplemente se pierden.
- *Archivos de bloque*: se comportan como arreglos de bytes. Es posible posicionarnos arbitrariamente sobre partes del archivo, además de iterar sobre él (ya sea para leer o para escribir) más de una vez.

La intuición también nos dice que archivos de naturaleza tan distinta deben manejarse de formas distintas en código de bajo nivel, y esto es cierto. Por ejemplo, el código del controlador que se encarga de detectar los movimientos de un mouse óptico y convertirlos en una secuencia de bytes es muy distinto al código que se encarga de almacenar datos persistentes en un disco duro magnético. El código de bajo nivel debe entender el archivo desde el punto de vista *físico*.

Sin embargo, los sistemas operativos y los lenguajes de programación modernos se encargan de proveerle al programador una interfaz común y de alto nivel. Haciendo uso de esta interfaz, el programador sólo necesita: abrir el archivo, leer o escribir sobre él y cerrar el archivo. El programador manipulará el archivo desde el punto de vista *lógico*, mientras que el sistema operativo y los controladores se encargarán de traducir los comandos en procesos físicos sobre los datos y su medio.

En los lenguajes C y C++, un manipulador lógico de un archivo es una variable de tipo `FILE*` que está asociada a un archivo que se abrió mediante las utilidades de `<stdio.h>`. Sin embargo, existen tres archivos que se abren automáticamente en cuanto comienza el programa y cuyos manipuladores también están declarados en `<stdio.h>`. A estos archivos se les denomina flujos estándar y son los siguientes:

- La *entrada estándar*: es un archivo de sólo lecutura que está vinculado al flujo de donde provienen los datos que ingresa el usuario. Su manipulador está declarado bajo el nombre `stdin`.
- La *salida estándar*: es un archivo de sólo escritura que está vinculado al flujo a donde se envían los datos que el programa imprime para el usuario. Su manipulador está declarado bajo el nombre `stdout`.
- La *salida estándar de errores*: es un archivo de sólo escritura que está vinculado a un flujo potencialmente distinto al de la salida estándar. La intención de este flujo es que sirva como salida de mensajes de error que no deben mostrársele al usuario. Su manipulador está declarado bajo el nombre `stderr`.

La función `scanf` de C por omisión lee datos desde `stdin`. De forma similar, la función `printf` imprime datos en `stdout`. Las funciones `fscanf` y `fsprintf` son variantes de las funciones anteriores, las cuales tienen un primer parámetro adicional que hace explícito el origen o el destino de los datos. La forma de imprimir en `stderr` es usando `fprintf`.

Código

```
int n;
scanf("%d", &n);            // leer de stdin
fscanf(stdin, "%d", &n);    // equivalente explícito
```

Código

```
int n = 123;
printf("%d\n", n);          // imprimir en stdout
fprintf(stdout, "%d", &n);  // equivalente explícito
```

Código	Salida	Salida de errores

`printf("hola1\n");`
`fprintf(stdout, "hola2\n");`
`fprintf(stderr, "hola3\n");`
	`hola1`
	`hola2`
		`hola3`

El destino de la salida estándar de errores puede configurarse. Por ejemplo, la línea de comandos (también llamada terminal, consola o símbolo del sistema) de un sistema operativo permite ejecutar un programa y vincular o redireccionar los flujos estándar a archivos almacenados en el disco duro. Esto permite elegir destinos distintos para `stdout` y `stderr`:

Comando	Efecto al ejecutar el programa
ruta_ejecutable	Todos los flujos estándar se vinculan a la consola.
ruta_ejecutable < archivo_entrada	La entrada estándar se vincula al archivo.
ruta_ejecutable > archivo_salida	La salida estándar se vincula al archivo.
ruta_ejecutable 2> archivo_salida	La salida estándar de errores se vincula al archivo.
ruta_ejecutable > archivo_salida 2>&1	Las dos salidas estándar se vinculan al mismo archivo.

5. Lectura y escritura de datos en archivos

En un programa de consola escrito en lenguaje C, casi siempre se emplean las funciones `scanf` y `printf` para interactuar con el usuario. Estas funciones usan una representación de datos legible para el humano: si el usuario desea ingresar el número 512 y se usó el especificador de formato `%d` de `scanf`, entonces el usuario debe ingresar el número tal cual. La *entrada y salida con formato* (también llamada entrada y salida con formato de texto) es justamente el proceso de leer o escribir datos en una representación que es legible para el humano. De hecho, la cadena que reciben las funciones `scanf` y `printf` se denomina cadena de formato e implementa muchos especificadores útiles, entre ellos:

Especificador	Comportamiento
`%d`	Leer o imprimir un entero en decimal (por ejemplo, 512).
`%0`*N*`d`	Imprimir un entero en decimal con *N* cifras. Por ejemplo, imprimir 512 con `%05d` imprimiría 00512
`%f`	Leer o imprimir un flotante (por ejemplo, 3.1416).
`%.`*N*`f`	Imprimir un flotante con *N* cifras decimales. Por ejemplo, imprimir 3.1416 con `%.2f` imprimiría 3.14
`%c`	Leer o imprimir un caracter (por ejemplo, `'@'`).
`%s`	Leer o imprimir una cadena (por ejemplo, `"hola"`). La lectura se detiene en el primer espacio.
`%[abc]`	Leer una cadena que sólo puede contener los caracteres `abc`. Por ejemplo, la lectura de `"acalorado"` sólo extraerá la cadena `"aca"`.
`%[^abc]`	Leer una cadena que sólo puede contener caracteres distintos a `abc`. Por ejemplo, la lectura de `"koala"` sólo extraerá la cadena `"ko"`.

Una cadena de formato puede incluir caracteres que no forman parte de ningún especificador. Durante una impresión, estos caracteres simplemente se imprimen. Durante una lectura, la función `scanf` se encargará de extraer e ignorar dichos caracteres de la entrada. Si uno de esos caracteres adicionales es un espacio en blanco, entonces se extraerán todos los espacios en blanco (incluyendo tabuladores y saltos de línea). Por ejemplo, podemos leer dos enteros que estén separados por comas y no por espacios, además de leer el primer caracter posterior que no sea espacio en blanco, de la siguiente forma:

Código	Entrada	Salida

`int a, b;`
`char c;`
`scanf("%d,%d %c", &a, &b, &c);`
`printf("leímos %d %d %c", a, b, c);`
	`5,7`
	`@`
		`leímos 5 7 @`

En un programa de consola, es común suponer que la entrada es correcta y que conocemos el tamaño de la misma. Sin embargo, esto rara vez es cierto en la vida real. El valor de retorno de `scanf` nos permite conocer cuántos de los especificadores de formato de lectura tuvieron éxito.

Código	Entrada	Salida

`int a, b, x, y;`
`int r1 = scanf("%d%d", &a, &b);`
`int r2 = scanf("%d%d", &x, &z);`
`printf("(%d): %d %d\n", r1, a, b);`
`printf("(%d): %d %d\n", r2, x, y);`
	`5 5`
	`5 gatito`
		`(2): 5 5`
		`(1): 5 22092`

La función `scanf` se detiene al primer error que encuentre. Por otra parte, cuando `scanf` falla porque ni siquiera hay qué leer, ésta regresa un valor que casi siempre vale -1 pero que formalmente se llama `EOF`.

Código	Entrada	Salida

`int a, b;`
`int r = scanf("%d%d", &a, &b);`
`printf("%d", r);`
	`gatito 5`	`0`

Código	Entrada	Salida

`int a, b;`
`int r = scanf("%d%d", &a, &b);`
`printf("%d", r);`
		`-1`

La característica anterior nos permite procesar una cantidad desconocida de datos, preguntando en un ciclo si `scanf` pudo leer los datos que le pedimos:

Código	Entrada	Salida

`int n;`
`while (scanf("%d", &n) == 1) {`
`  printf("Leímos %d\n", n);`
`}`
	`1`
	`2`
	`3`
		`Leímos 1`
		`Leímos 2`
		`Leímos 3`

La función `feof` toma un manipulador de archivo de entrada y se usa para determinar si ya se llegó al final de la misma. Desafortunadamente, esta función no detecta el fin de la entrada tras leer su último byte, sino que sólo se da cuenta tras intentar hacer la siguiente lectura, la cual fallará. Por esta razón, suele ser incorrecto usar `feof` para controlar un ciclo que lea un archivo:

Código	Entrada	Salida

`char c;`
`while (!feof(stdin)) {`
`  scanf("%c", %c);`
`  printf("%c", c);`
`}`
	`hola`	`holaa`

En este sentido, es mucho mejor usar el valor de retorno de `scanf`.

En contraste con la entrada y salida con formato, *la entrada y salida sin formato* (también llamada entrada y salida binaria) se emplea para leer o imprimir una secuencia de bytes que denota la representación interna de un valor o variable, tal como ésta almacena en la memoria del procesador. Dicha representación difícilmente es entendible para el usuario común, pero nosotros intentaremos entenderla, lo que hará necesario que primero retomemos algunos conceptos relacionados.

Cada variable tiene un consumo de memoria en bytes (y actualmente se acepta que un byte está formado por 8 bits). 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 usando un entero sin signo de tipo `size_t`, el cual se puede imprimir en decimal con el especificador `%zu`.

Código	Salida

`printf("%zu\n", sizeof(char));`
`int n;`
`printf("%zu\n", sizeof(n));`
`printf("%zu\n", sizeof(short));`
	`1`
	`4`
	`2`

Internamente, los enteros se representan en base binaria, donde cada bit es un dígito binario. De hecho, en C++ es posible expresar una literal entera en base binaria con el prefijo `0b`. Además, el valor máximo sin signo que se puede representar usando 8 bits es el 255 (`0b11111111` en binario).

Código	Salida

`int n1 = 9;`
`int n2 = 0b1001;`
`int n3 = 0b11111111;`
`printf("%d %d %d\n", n1, n2, n3);`
	`9 9 255`

También se pueden escribir literales enteras en base hexadecimal con el prefijo `0x`. En esta base, los números del 10 al 15 se representan con las letras de la `A` a la `F`. Por el rango de valores de la base, cada dígito hexadecimal corresponde con exactamente 4 bits. En consecuencia, cada dos dígitos hexadecimales corresponden con un byte en un entero. El especificador `%x` permite leer o imprimir números en hexadecimal, mientras que las direcciones de memoria generalmente son de 8 bytes y se imprimen con el especificador `%p`, pero se suelen desplegar en hexadecimal.

Código	Salida

`int n1 = 255;`
`int n2 = 0xFF;`
`int n3 = 0x11223344;  // cuatro bytes`
`printf("%d %d %d\n", n1, n2, n3);`
`printf("%x %x %x\n", n1, n2, n3);`
`printf("%p", &n1);`
	`255 255 287454020`
	`ff ff 11223344`
	`0x7ffcd824b8d4`

Una vez explicado lo anterior, ya podemos introducir las funciones de entrada y salida sin formato `fread`, `fwrite`, `fgetc` y `fputc`, donde todas ellas están disponibles en el archivo de biblioteca `<stdio.h>`.

Funciones de entrada y salida sin formato	
Función de biblioteca	**Operación realizada**

`size_t fread(`
`  void* p,`
`  size_t t, size_t n,`
`  FILE* f`
`);`
	Lee del archivo `f` una secuencia de `n` elementos con `t` bytes cada uno y los guarda a partir de `p`. Regresa la cantidad de elementos leídos.

`size_t fwrite(`
`  const void* p,`
`  size_t t, size_t n,`
`  FILE* f`
`);`
	Escribe en el archivo `f` una secuencia de `n` elementos con `t` bytes cada uno almacenados a partir de `p`. Regresa la cantidad de elementos escritos.

`int fgetc(FILE* f);`	Lee un byte del archivo `f`. Regresa `EOF` en caso de error.
`int fputc(int b, FILE* f);`	Escribe un byte en el archivo `f`. Regresa `EOF` en caso de error.

En el siguiente ejemplo, imprimiremos dos enteros sin formato y cuyas secuencias subyacentes de bytes corresponden con caracteres ASCII que tienen representación gráfica. Por ejemplo, la literal hexadecimal `0x40` corresponde con el 64 en decimal y con la `'@'` en la tabla ASCII. Del mismo modo, las literales `0x48`, `0x4F`, `0x4C` y `0x41` corresponden con los caracteres `'H'`, `'O'`, `'L'` y `'A'`, respectivamente.

Código	Salida

`int n1 = 0x40404040; // 1077952576`
`int n2 = 0x484F4C41; // 1213156417`
`fwrite(&n1, sizeof(int), 1, stdout);`
`printf("\n");`
`fwrite(&n2, sizeof(int), 1, stdout);`
	`@@@@`
	`ALOH`

Nótese que el entero con valor `0x484F4C41` imprimió sus bytes del menos significativo al más signiticativo. Anteriormente se dijo que la lectura y escritura sin formato trabaja sobre la representación interna de las variables en el procesador, y esto es verdad. Un procesador con codificación *little-endian* almacena primero en memoria los bytes menos significativos de un entero. Por otra parte, un procesador con codificación *big-endian* almacena primero en memoria los bytes más significativos de un entero. La codificación *big-endian* tiene la ventaja de que los algoritmos de comparaciones de cadenas también sirven para comparar enteros, pero la codificación *little-endian* permite transferir un valor entero entre variables de diferente ancho (por ejemplo, `long long` o `short`) simplemente agregando o quitando bytes por la derecha; los bytes no afectados no necesitan moverse en memoria. Actualmente, la gran mayoría de las computadoras usan codificación *little-endian*.

Leer un dato sin formato es simplemente realizar el proceso inverso. Sin embargo, si usamos una computadora para escribir un entero sin formato en un archivo y luego leemos el entero en una computadora distinta, es imprescindible que ambas usen la misma codificación *endian* para que exista consistencia.

Código	Entrada	Salida

`int n;`
`fread(&n, sizeof(int), 1, stdin);`
`printf("%x %d", n, n);`
	`ALOH`	`0x484F4C41 1213156417`

En un asunto similar, no se recomienda depender de los tamaños de los tipos de datos primitivos `int`, `long`, etc. Por ejemplo, `sizeof(long)` es 4 en Windows y suele ser 8 en Linux, lo cual da a lugar a que ocurra la misma inconsistencia al leer y escribir datos sin formato. Si se necesita escribir un programa portable entre distintos sistemas operativos o compiladores, se recomienda usar tipos de ancho fijo.

5.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Maximo-de-enteros-sin-formato`.

6. Apertura de archivos persistentes

Como ya se mencionó, la memoria principal (también llamado almacenamiento primario) tiene la propiedad de ser volátil: su contenido se pierde si la energía eléctrica se desconecta. En contraste, el almacenamiento secundario es *persistente*: los datos guardados permanecen en el medio aún en ausencia de energía eléctrica. Algunos ejemplos de medios de almacenamiento persistente son los discos duros magnéticos, las memorias de estado sólido, las cintas magnéticas y los discos ópticos. Todos estos medios tienen propiedades de almacenamiento distintas, de las cuales se hablará en su momento. Sin embargo, recordemos que el sistema operativo y los controladores de los dispositivos son los responsables de manejar los detalles de bajo nivel y de transferir los datos entre éstos y el procesador; nosotros como programadores siempre usaremos interfaces de programación de alto nivel para manipularlos.

En primer lugar, un programa denominado sistema de archivos (y que casi siempre forma parte del sistema operativo) es el que encarga de abstraer el almacenamiento persistente en términos de rutas, carpetas, archivos individuales y accesos directos. Por ejemplo, en Windows es común encontrar rutas del estilo `C:\Usuario\usuario\Documentos\notas.txt` mientras que en UNIX las rutas del estilo

`/home/usuario/a.out`. Los sistemas de archivos suelen guardar metainformación como la fecha de creación de los archivos, e incluso son capaces de abstraer archivos especiales o de caracter. Sin embargo, estos detalles realmente forman parte de un curso de sistemas operativos y nosotros nos concentraremos en procesar archivos ya abiertos (como `stdin` y `stdout`), así como en crear o procesar archivos persistentes con rutas conocidas de antemano. Por ejemplo, cuando usemos la ruta `"archivo.txt"` supondremos que queremos accesar a un archivo con ese nombre y que está en la misma carpeta que nuestro ejecutable.

En el lenguaje C, las funciones `fopen` y `fclose` del archivo de biblioteca `<stdio.h>` son las que encargadas de abrir y cerrar archivos. Las declaraciones de estas funciones son:

Funciones de apertura y cierre de archivos	
Función de biblioteca	**Operación realizada**
`FILE* fopen(const char* r, const char* m);`	Abre un archivo con la ruta y el modo de apertura dados. Regresa nulo en caso de error.
`int fclose(FILE* f);`	Cierra un archivo previamente abierto. Regresa 0 en caso de éxito y otro valor en caso de error.

A continuación se describen los modos de apertura disponibles:

Modos de apertura de `fopen`	
Modo	**Operación realizada**
`r`	Abrir el archivo para leer. Si el archivo no existe, se produce un error.
`w`	Crear el archivo para escribir. Si el archivo ya existe, se trunca.
`a`	Crear el archivo para escribir. Si el archivo ya existe, se escribe al final de su contenido previo.
`r+`	Abrir el archivo para leer y escribir. Si el archivo no existe, se produce un error.
`w+`	Crear el archivo para leer y escribir. Si el archivo ya existe, se trunca.
`a+`	Crear el archivo para leer y escribir. Si el archivo ya existe, se escribe al final de su contenido previo.

Si a los modos `w` se les agrega una `x` entonces es un error abrir un archivo que ya existía. Si a cualquier modo se le agrega una `b` entonces se abre el archivo en modo binario, el cual deshabilita conversiones ocultas que algunos sistemas operativos realizan. Por ejemplo, Windows escribe la pareja de caracteres `\r\n` cuando se solicita escribir `\n` (`\r` es el retorno de carro y `\n` es el salto de línea). Del mismo modo, Windows colapsa la pareja `\r\n` en `\n` al leer. Esto no causa problemas con datos con formato, pero las conversiones ocultas suelen ser fatales para datos sin formato y se deben deshabilitar. En los sistemas operativos sin conversiones ocultas (como Linux), el modo binario no tiene efecto. Windows abre los flujos estándar sin modo binario, pero es posible establecerlo (por ejemplo, para `stdin`) incluyendo `<fcntl.h>` y ejecutando `_setmode(_fileno(stdin), _O_BINARY);` al iniciar el programa.
El siguiente programa lee dos enteros de un archivo persistente y escribe su suma en otro.

Código

```
FILE* ent = fopen("entrada.txt", "r");
if (ent != nullptr) {
   FILE* sal = fopen("salida.txt", "w");
   int a, b;
   fscanf(ent, "%d%d", &a, &b);
   fprintf(sal, "%d", a + b);
   fclose(ent), fclose(sal);
}
```

Es común ignorar el cierre de archivos en programas cortos y de poca relevancia, ya que el sistema operativo se encarga de cerrarlos de todos modos al término del programa. Sin embargo, el sistema

operativo suele tener límites en cuanto a la cantidad de archivos abiertos simultáneamente. Si un programa ya desocupó algunos archivos pero seguirá en ejecución durante un largo tiempo, es mejor cerrarlos. El cierre de un archivo válido de escritura puede fallar, pero las razones se explicarán en la siguiente sección.

7. Búferes de la biblioteca estándar y del sistema operativo

En comparación con el almacenamiento primario, el almacenamiento secundario es *muy lento*. Mientras que la memoria principal tiene tiempos de acceso de cientos de ciclos de reloj, los tiempos de accesos del almacenamiento secundario van desde decenas de miles hasta cientos de millones de ciclos, dependiendo de la tecnología. Peor aún, una llamada al sistema operativo tiene una sobrecarga aproximada de mil ciclos, lo cual es relevante porque justamente él es el intermediario entre nuestro programa y los controladores de los dispositivos de almacenamiento secundario. En esta sección se discutirá el uso de búferes, tanto al nivel del sistema operativo como al nivel de la biblioteca de C, los cuales permiten reducir en buena medida la ineficiencia que se produciría si éstos no se usaran.
El proceso que ocurre normalmente durante la escritura de datos es siguiente:

1. Nosotros como programadores llamamos a alguna función de la biblioteca estándar como `fprintf` o `fwrite` para solicitar la escritura de datos.

2. La biblioteca copia esos datos en un búfer interno. El tamaño por omisión de dicho búfer es igual al valor de la constante `BUFSIZ` de `<stdio.h>` y difiere considerablemente entre compiladores y sistemas operativos, pero suele ser del orden de algunos pocos kilobytes. Mientras el búfer no se llene, la biblioteca retiene los datos en espera de más escrituras sobre el mismo archivo.

3. Cuando el búfer de la biblioteca se llena, se transfiere su contenido al sistema operativo.

4. El sistema operativo copia esos datos en un búfer interno de escritura. El tamaño por omisión de dicho búfer depende del sistema operativo, pero es un múltiplo del tamaño de un *bloque* (el equivalente conceptual de línea de caché o línea de coherencia pero a nivel del almacenamiento secundario).

5. Cuando el búfer del sistema operativo se llena, se transfiere su contenido al almacenamiento secundario.

6. El almacenamiento secundario copia esos datos en un búfer interno y eventualmente los escribe en el medio físco de memoria persistente. Los dispositivos incluso pueden contar con una pequeña batería interna para que los datos del búfer no se pierdan si la energía eléctrica se pierde antes de escribirlos.

Durante la escritura secuencial de un archivo y en condiciones normales, sólo pagaremos el costo de interactuar con el sistema operativo una vez cada `BUFSIZ` bytes, además de que sólo pagaremos el costo de interactuar con el almacenamiento secundario una vez por bloque. Actualmente, los dispositivos de almacenamiento secundario suelen tener bloques de 4 kilobytes.

Si ocurre algún error fatal cuando los datos se encuentran en un búfer intermedio, éstos no llegarán al medio persistente y se perderán. Lo anterior lo ilustraremos con el siguiente programa que solicita escribir en un archivo y luego sufre un error en ejecución: el archivo estará vacío cuando el proceso termine.

Código

```
#include <stdio.h>

int main( ) {
   FILE* sal = fopen("salida.txt", "w");
   fprintf(sal, "esto nunca llegará al archivo");
   int* p = nullptr;   // apuntador nulo: no deberíamos desreferenciarlo
   *p = 0xDEADBEEF;    // nos suicidamos; el programa morirá junto con
                       // los datos que aún están en el búfer de la biblioteca
}
```

La función `fflush` toma el manipulador de un archivo de salida y solicita que todos los datos que estén en el búfer sean transferidos en ese momento al sistema operativo. Usando esta función, los datos

sí podrán llegar a escribirse en el archivo aún si nuestro programa sufre un error de ejecución posterior. Sin embargo, se debe recordar que interactuar con el sistema operativo es una operación muy costosa y un programa que abuse de `fflush` será sumamente lento. Solamente se recomienda usar `fflush` cuando es imperativo evitar la pérdida de datos. Algunas situaciones que lo ameritan son bases de datos críticas (como registros bancarios) o cuando se necesita contar con información de depuración de un programa que está terminando abruptamente por culpa de un error en ejecución. Por ejemplo:

Código

```
FILE* sal = fopen("salida.txt", "w");
fprintf(sal, "esto sí podrá llegar al archivo");
fflush(sal);
int* p = nullptr;   // nos suicidamos, pero los datos del búfer
*p = 0xDEADBEEF;    // ya le fueron transferidos al sistema operativo
```

A continuación se muestra un programa que usa un ciclo para escribir cien millones de `'@'` en un archivo. Además, el código también llama a `fflush` en cada iteración. Llamar a `fflush` tantas veces provocará que el programa tarde entre 15 y 30 segundos en ejecutarse, dependiendo de la computadora. Eliminar la llamada a `fflush` hará que el tiempo de ejecución del programa baje a menos de 0.5 segundos.

Código

```
FILE* arch = fopen("arrobas.txt", "w");
for (int i = 0; i < 100000000; ++i) {
   fputc('@', arch);
   fflush(arch);   // ¡lento! enviar cada '@' individual al sistema operativo
}
```

Para que los datos eventualmente le lleguen al sistema operativo, la función `fclose` ejecuta un `fflush` implícito. Además, el sistema operativo escribe su propio búfer en el medio de almacenamiento persistente cada cierto tiempo (por ejemplo, una vez por segundo) aún si su búfer no está lleno. La transferencia de datos de un lugar a otro puede fallar: el sistema operativo puede rechazar una escritura si se da cuenta de que el usuario de una computadora compartida ya alcanzó su límite de almacenamiento, mientras que el medio de almacenamiento persistente puede rechazar la escritura si existe algún daño en el dispositivo. El valor de retorno de `fflush` y `fclose` es un entero distinto de 0 si ocurrió un error.

El estándar del lenguaje C no proporciona una forma de solicitarle al sistema operativo que transfiera los contenidos de su búfer al medio persistente, pero los sistemas operativos sí la proveen. En Linux, la función que realiza esta tarea se llama `fsync` y en Windows se llama `FlushFileBuffers`.

Código

```
#ifdef _WIN32        // detección del s.o. para incluir archivos .h
   #include <io.h>        // para usar _get_osfhandle de Windows
   #include <windows.h>   // para usar FlushFileBuffers de Windows
#else
   #include <unistd.h>    // para usar fsync de Linux
#endif
#include <stdio.h>   // la función fileno (no estándar de C) está aquí

int main( ) {
   FILE* sal = fopen("salida.txt", "w");
   //...
   fflush(sal);      // transferir el búfer de la biblioteca al s.o.
   #ifdef _WIN32     // detección del s.o. para llamar a la función
      HANDLE h = (HANDLE)_get_osfhandle(fileno(sal));  // manipulador del s.o.
      FlushFileBuffers(h);    // transferir el búfer del s.o. al medio
   #else
      int h = fileno(sal);    // manipulador del s.o.
      fsync(h);               // transferir el búfer del s.o. al medio
   #endif
}
```

Si modificamos el programa que imprime cien millones de `'@'` para llamar a `fsync` o `FlushFileBuffers` en cada iteración después de `fflush`, el programa tardará entre 8 y 1000 horas en ejecutarse, dependiendo de la tecnología del medio de almacenamiento (normalmente discos de estado sólido o discos magnéticos). Las características de las tecnologías de almacenamiento persistente se discutirán en secciones posteriores. El proceso que ocurre normalmente durante la lectura de datos es siguiente:

1. Nosotros como programadores llamamos a alguna función de la biblioteca estándar como `fscanf` o `fread` para solicitar la lectura de datos.

2. La biblioteca revisa si los datos solicitados están en su búfer de lectura. En caso afirmativo, los entrega al programa y la lectura termina. En caso contrario, le solicita `BUFSIZ` bytes al sistema operativo y copia lo que le entreguen en su búfer de lectura, eventualmente entregándolo a su vez al programa.

3. Si el sistema operativo recibe una solicitud de lectura por parte de la biblioteca, éste revisa si los datos solicitados están en su búfer. En caso afirmativo, los entrega a la biblioteca. En caso contrario, el sistema operativo solicita un bloque de bytes al almacenamiento persistente y copia lo que le entreguen en su búfer, eventualmente entregándolo a su vez a la biblioteca.

4. Si el dispositivo de almacenamiento secundario recibe una solicitud de lectura por parte del sistema operativo, éste lee el bloque de datos solicitado en un búfer y lo transfiere al sistema operativo.

Durante la lectura secuencial de un archivo, sólo pagaremos el costo de interactuar con el sistema operativo una vez cada `BUFSIZ` bytes, además de que sólo pagaremos el costo de interactuar con el almacenamiento secundario una vez por bloque. Más aún, el sistema operativo suele usar toda la memoria principal libre (es decir, la que no está siendo usada por los programas en ejecución) para mantener una caché de bloques leídos del almacenamiento persistente, jugando así el papel del búfer de lectura. Por esta razón, la lectura de un archivo es mucho más rápida la segunda vez: la primera vez el archivo se lee del almacenamiento persistente, pero la segunda vez se lee de la caché del sistema operativo. Por ejemplo, leer secuencialmente el archivo de cien millones de `'@'` tardará entre 1 y 1.5 segundos la primera vez, pero tardará menos de 0.5 segundos la segunda vez. La opción `"Empty Standby List"` del programa `RAMMap`[2] para Windows permite vaciar la caché del almacenamiento secundario del sistema operativo.

Las funciones de la biblioteca `setbuf` y `setvbuf` permiten especificar la política y el tamaño del búfer de la biblioteca estándar. En particular, `setbuf` solamente permite deshabilitar el búfer o habilitarlo con un arreglo precisamente de `BUFSIZ` bytes. Por otra parte, la función `setvbuf` permite establecer un búfer posiblemente más grande que `BUFSIZ`, así como el modo del búfer para archivos de escritura.

Funciones de establecimiento de búfer	
Función de biblioteca	**Operación realizada**
`void setbuf(FILE* f, char* b);`	Si `b` es nulo, deshabilita el búfer. En caso contrario, usa el arreglo denotado por `b` como un búfer de `BUFSIZ` bytes.

`int setvbuf(`
`  FILE* f, char* b,`
`  int m, size_t n`
`);`
	Si `m == _IONBF`, se deshabilita el búfer y se ignoran los demás parámetros. Si `m == _IOFBF` o `m == _IOLBF` se establece un búfer de `n` bytes. Si `m == _IOLBF` se hace un `fflush` implícito cuando se escribe un salto de línea. Si `b` es nulo, la biblioteca aparta la memoria para el búfer. En caso contrario, se usa el arreglo denotado por `b`.

A continuación se muestra un programa que realiza una prueba de rendimiento. El programa lee secuencialmente un archivo completo (por ejemplo, el que contiene los cien millones de `'@'`) y nos permite especificar el tamaño del búfer de la biblioteca, la cantidad de bytes a leer por llamada a función y, cuando esta cantidad es igual a 1, la función de lectura a emplear (`fgetc` o `fread`). En este programa usaremos la biblioteca `<chrono>` de C++ para medir el tiempo real de ejecución del programa, ya que en Linux la función `clock` del archivo de biblioteca `<time.h>` sólo mide el tiempo de CPU sin incluir el tiempo gastado en interactuar con el sistema operativo y en accesar al almacenamiento secundario.

[2] `https://docs.microsoft.com/en-us/sysinternals/downloads/rammap`

Código

```
#include <chrono>
#include <stdio.h>

long filesize(FILE* f) {    // regresa el tamaño del archivo en bytes
   fseek(f, 0, SEEK_END);   // la implementaciónde de esta función
   long res = ftell(f);     // se explicará en la siguiente sección
   rewind(f);
   return res;
}

int main( ) {
   FILE* ent = fopen("arrobas.txt", "rb");
   long tam = filesize(ent);

   // Determinar el tamaño del búfer

   printf("Elige el tamaño del búfer de la biblioteca (valor >= 0): ");
   int bufer;
   scanf("%d", &bufer);

   if (bufer == 0) {
      setbuf(ent, nullptr);
   } else {
      setvbuf(ent, nullptr, _IOFBF, bufer);
   }

   // Determinar el tamaño de la lectura y la función usada

   printf("Elige la cantidad de bytes a leer por operación (valor > 0): ");
   int cuantos;
   scanf("%d", &cuantos);

   int usar_fgetc = false;
   if (cuantos == 1) {
      printf("Elige si deseas usar fgetc (0 o 1): ");
      scanf("%d", &usar_fgetc);
   }

   // Prueba de rendimiento

   auto t0 = std::chrono::high_resolution_clock::now( );
   if (usar_fgetc) {
      for (int i = 0; i < tam; ++i) {
         fgetc(ent);
      }
   } else {
      char* temp = new char[cuantos];
      for (int i = 0; i < tam / cuantos + bool(tam % cuantos); ++i) {
         fread(&temp[0], sizeof(char), cuantos, ent);
      }
   }
   auto t1 = std::chrono::high_resolution_clock::now( );

   printf("%f", std::chrono::duration<double>(t1 - t0).count( ));
}
```

	$b = 0$	$b = 2^6$	$b = 2^{12}$	$b = 2^{18}$
$r = 1$ / `fgetc`	677.458 (556.277)	30.940 (26.913)	10.268 (9.797)	9.648 (9.615)
$r = 1$ / `fread`	667.572 (564.647)	36.581 (32.490)	15.986 (15.294)	15.118 (14.906)
$r = 2^6$	21.958 (17.429)	21.850 (17.478)	6.018 (0.710)	5.885 (0.361)
$r = 2^{12}$	6.076 (0.449)	6.228 (0.448)	6.163 (0.448)	5.963 (0.106)
$r = 2^{18}$	6.096 (0.093)	5.995 (0.092)	6.178 (0.092)	5.896 (0.093)

Tiempos de ejecución en segundos para leer un archivo de 1 GB con un tamaño de búfer b y r bytes leídos por llamada a función. Se usó un disco WD Surveillance HDD WD20PURZ bajo Windows 10. Entre paréntesis el tiempo de ejecución cuando el archivo ya está en la caché del sistema operativo.

La tabla anterior cuantifica lo explicado anteriormente. La peor combinación ocurre cuando se usa un búfer muy chico y además se leen pocos bytes por llamada a función. Bajo estas condiciones, la cantidad de interacciones con el sistema operativo es exagerada y de poco sirve que el archivo completo esté en la caché del sistema operativo. Por otra parte, leer pocos bytes por llamada a función será un tanto lento (aún con un búfer grande, aún con la caché del sistema operativo) simplemente porque tendremos que hacer muchas llamadas a función y esto representa una sobrecarga importante para el procesador.

Leer muchos elementos por llamada a función es más efectivo que simplemente aumentar el tamaño del búfer, porque una solicitud grande no se parte sino que se le envía completa al sistema operativo. Sólo es cuando el programador lee una cantidad baja de elementos por llamada, que tener un búfer más grande ayuda. Leer un byte individual es más rápido con `fgetc` que hacer lo mismo con `fread`.

Aumentar la cantidad de bytes leídos por llamada y aumentar el tamaño del búfer mejora la situación hasta que el cuello de botella se vuelve el almacenamiento secundario. Cuando esto último ocurre, seguir aumentándolos deja de aportar beneficios. Sin embargo, si el archivo ya está en la caché del sistema operativo, todavía se pueden obtener beneficios visibles al reducir la cantidad de interacciones con éste.

8. Posicionamiento en archivos

Como se dijo anteriormente, los archivos de bloque son aquéllos que denotan arreglos de bytes y entonces admiten operaciones de acceso arbitrario: podemos posicionarnos libremente sobre cualquier parte de ellos. Un archivo de bloque de lectura puede leerse repetidamente y no necesariamente de forma secuencial, mientras que un archivo de bloque de escritura puede sobreescribirse total o parcialmente. Los archivos persistentes almacenados en discos duros convencionales son archivos de bloque.

La biblioteca estándar de C provee las funciones `ftell` y `fseek`, las cuales permiten conocer nuestra posición actual y movernos a una posición del archivo, respectivamente. El valor de retorno de `ftell` es un `long` que denota una posición sobre el arreglo de bytes subyacente del archivo. Del mismo modo, la función `fseek` toma un `long` que puede interpretarse como una posición absoluta sobre el archivo.

Funciones de posicionamiento en archivos	
Función de biblioteca	**Operación realizada**
`long ftell(FILE* f);`	Regresa nuestra posición actual sobre el archivo.
`int fseek(FILE* f, long n, int m);`	Cambia nuestra posición sobre el archivo. Si `m == SEEK_SET`, se interpreta `n` como una posición absoluta. Si `m == SEEK_CUR`, se interpreta `n` como un salto relativo a nuestra posición actual. Si `m == SEEK_END`, se interpreta `n` como un salto relativo al fin de archivo. El valor de `n` puede ser negativo para `SEEK_CUR` y `SEEK_END`.

La función `fseek` puede fallar si se elige una posición inválida, aunque los archivos de escritura suelen admitir posiciones más allá del fin, lo que tiene el efecto de rellenar el archivo. La función `fseek` también puede fallar si se usa sobre un archivo de caracter. Por ejemplo, cuando `stdin` o `stdout` son interactivos, `fseek` fallará porque son archivos de caracter y no de bloque. Sin embargo, basta usar redirigir la entrada y la salida mediante archivos persistentes para que `fseek` funcione correctamente. En los siguientes ejemplos, supondremos que `stdin` y `stdout` están vinculados a archivos de bloque.
El siguiente programa esquiva parte de los caracteres de la entrada antes de leer la cadena.

Código	Entrada	Salida

`char cadena[10 + 1];`
`fseek(stdin, 2, SEEK_SET);`
`scanf("%s", &cadena[0]);`
`printf("%s\n", cadena);`
	`gatito`	`tito`

De forma similar, el siguiente programa imprime una cadena y luego sobreescribe parte de la salida:

Código	Salida

`printf("hola");`
`fseek(stdout, 0, SEEK_SET);`
`printf("b");`
`fseek(stdout, -1, SEEK_END);`
`printf("o");`
	`bolo`

En ambos casos, las posiciones en la entrada y la salida corresponden con las de un arreglo de bytes. Como tal, la posición inicial es la 0. El siguiente ejemplo hace uso de `ftell` e ilustra esto:

Código	Entrada	Salida

`char cadena[10 + 1];`
`printf("actual: %ld\n", ftell(stdin));`
`fseek(stdin, 2, SEEK_SET);`
`printf("actual: %ld\n", ftell(stdin));`
`fseek(stdin, 1, SEEK_CUR);`
`printf("actual: %ld\n", ftell(stdin));`
`scanf("%s", &cadena[0]);`
`printf("%s\n", cadena);`
	`gatito`	`actual: 0`
		`actual: 2`
		`actual: 3`
		`ito`

La función `rewind` toma un `FILE*` y equivale a hacer un `fseek` al inicio del archivo. Además, una forma típica de calcular el tamaño de un archivo es la que se muestra a continuación. El estándar de C no garantiza que funcione siempre, pero sí es correcta prácticamente en cualquier plataforma que sea compatible con POSIX. De todos modos, en Windows los archivos en modo texto son problemáticos y la cantidad de bytes del archivo puede no corresponder con la cantidad de `char` que se pueden leer de él.

Código	Entrada	Salida

`fseek(stdin, 0, SEEK_END);`
`long t = ftell(stdin);`
`printf("%ld bytes\n", t);`
`rewind(stdin);  // regresar al inicio`
`for (int i = 0; i < t; ++i) {`
`  //... (cuidado en Windows)`
`}`
	`gatito`	`6 bytes`

Como `ftell` y `fseek` trabajan con posiciones enteras, podemos usar operaciones aritméticas para calcular posiciones relevantes del archivo y movernos a ellas. Sin embargo, el hecho de que el tipo entero de estas posiciones sea `long` puede ser problemático en las plataformas donde `long` tiene 32 bits al resultar insuficiente para manipular archivos muy grandes. Para solventar este problema, Linux provee las funciones `ftello64` y `fseeko64`, mientras que Windows provee las funciones `_fseeki64` y `_fseeki64`. En Windows, el posicionamiento sobre archivos no binarios no es confiable debido a las conversiones

ocultas que Windows implementa. El estándar de C también provee las funciones `fgetpos` y `fsetpos`, pero éstas tienen la desventaja de no admitir aritmética de posiciones, por lo que sólo se puede saltar mediante `fsetpos` a alguna posición para la que previamente se obtuvo un señalador mediante `fgetpos`.

En un archivo en modo escritura, llamar a `fseek` hace un `fflush` implícito, mientras que en un archivo en modo lectura, llamar a `fseek` invalida el búfer de lectura. Por esta razón, se debe evitar hacer muchos `fseek` innecesariamente. Es buena idea que el usuario use sus propios búferes explícitos si es que este comportamiento de `fseek` llega a impactar seriamente en el rendimiento del programa.

8.1. Ejercicios

Ya que omegaUp vincula los flujos estándar a archivos persistentes, un programa podrá leer la entrada más de una vez y también podrá modificar su salida usando rutinas de posicionamiento en archivos (por ejemplo, `fseek` o `rewind`) sobre `stdin` o `stdout`.

1. Resuelve el problema `https://omegaup.com/arena/problem/El-k-esimo-entero-de-un-archivo`.
2. Resuelve el problema `https://omegaup.com/arena/problem/Invirtiendo-los-enteros-de-un-ar`.

9. Bloques y fragmentación

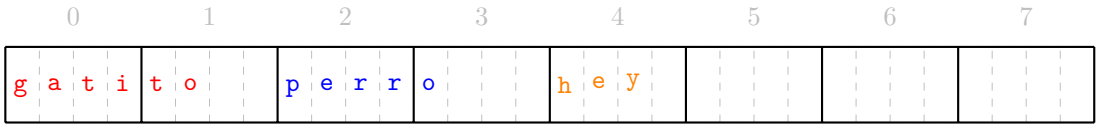
Al igual que la memoria, el almacenamiento secundario es (desde el punto de vista lógico) un arreglo de bytes. Como ya se mencionó anteriormente, las transacciones entre el sistema operativo y el almacenamiento secundario se llevan a cabo con granularidad de un bloque, el cual suele ser de 4096 bytes (aunque existen configuraciones con bloques de medio kilobyte o de varias decenas de megabytes). Además de determinar la cantidad de bytes transferidos por operación de lectura o escritura, el tamaño de un bloque importa porque un archivo siempre ocupa una cantidad entera de bloques: si los bloques son de 4096 bytes y un archivo guarda 10 bytes, en realidad estará ocupando un bloque completo en el almacenamiento secundario. Por esta razón, a este concepto también se le conoce como tamaño de asignación.

El sistema operativo (en concreto, el sistema de archivos) es el responsable de administrar los bloques del almacenamiento secundario. En particular, se debe llegar el registro de cuáles bloques están libres, cuáles están ocupados, qué bloques ocupados pertenecen a cada archivo y qué propiedades tiene cada archivo (ruta con nombre, longitud y posiblemente algunos metadatos como fecha de creación). En esta sección obviaremos el almacenamiento de esto último y nos concentraremos en cómo es que se asignan y liberan los bloques conforme se crean y eliminan archivos.

Para explicar la administración de bloques, supondremos que contamos con ocho bloques de 4 bytes cada uno y que queremos realizar las siguientes tareas en orden:

1. Crear un archivo `A` con la cadena `"gatito"`.
2. Crear un archivo `B` con la cadena `"perro"`.
3. Crear un archivo `C` con la cadena `"hey"`.

Después de realizar las tareas, el uso de los bloques es el siguiente:



Los bloques se numeran del 0 al 7 y cada bloque ocupa 4 bytes. El archivo `A` ocupa los bloques 0 y 1. El archivo `B` ocupa los bloques 2 y 3. El archivo `C` ocupa el bloque 4.

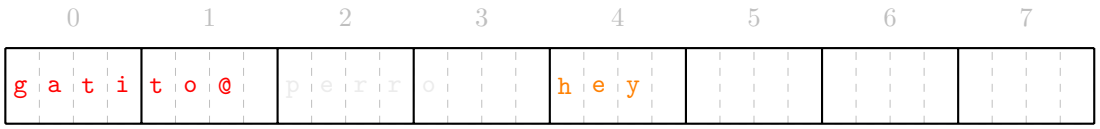
La figura anterior ilustra el fenómeno denominado fragmentación interna: si a un archivo necesariamente se le asigna una cantidad de bytes múltiplo del tamaño de un bloque, entonces es posible que exista espacio desperdiciado en el último bloque del archivo. Una forma de reducir la fragmentación interna consiste en usar bloques más chicos. Sin embargo, recordemos que una tarea del sistema de archivos

es llevar el registro de cuáles bloques están libres y cuáles están ocupados (normalmente con una tabla de booleanos o con una lista enlazada de bloques libres); al reducir el tamaño de un bloque, aumenta el número de ellos y su administración se complica. Conforme transcurren los años y la capacidad de los dispositivos de almacenamiento secundario aumenta, también suele aumentar el tamaño de los bloques para que la administración de los mismos se mantenga manejable.
Ahora supondremos que queremos realizar las siguientes tareas adicionales:

4. Agregar al archivo `A` la cadena `"@"`.

5. Eliminar el archivo `B`.

Agregar la cadena `"@"` al archivo `A` es fácil porque aún hay espacio suficiente en su último bloque. Si esto último no fuera cierto entonces habría problemas, pero esos se ilustrarán en breve. la eliminación de un archivo se suele limitar a marcar sus bloques comos libres y realmente no se hace nada con su contenido. Es por esto que, en determinadas situaciones, es posible recuperar un archivo borrado.



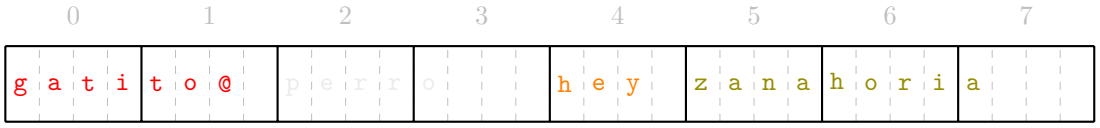
Despúes de realizar las tareas 4 y 5, el archivo `A` ocupa
los bloques 0 y 1, mientras que el archivo `C` ocupa el bloque 4.

Finalmente, supondremos que queremos realizar la siguiente tarea adicional:

6. Crear un archivo `D` con la cadena `"zanahoria"`.

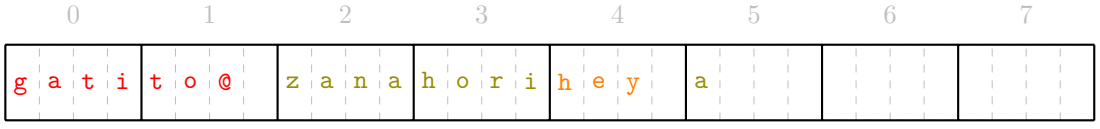
La cadena cabría a partir del último byte del bloque 1. Sin embargo, dos archivos distintos no pueden compartir un bloque. Los bloques 2 y 3 están libres, pero la cadena no cabe ahí. Existen dos alternativas:

- Si fuera obligatorio que los bloques de un archivo sean contiguos, entonces debemos ignorar los bloques 2 y 3 y debemos usar los bloques 5, 6, y 7. Cuando hay bloques libres en el almacenamiento pero el requisito de contigüidad nos impide usarlos, se dice que ocurre fragmentación externa. Éste no suele ser el caso en la administración del almacenamiento secundario, pero sí en la administración de la memoria principal (por ejemplo, al solicitar o liberar arreglos dinámicos con `malloc`, `free`, `new` o `delete`). Cuando hay más de una región contigua de bloques libres y con suficiente capacidad, el algoritmo *best fit* usa la región más justa, el algoritmo *worst fit* usa la región más holgada y el algoritmo *first fit* usa simplemente la primera que encuentre. Se suele preferir el algoritmo *first fit* simplemente por ser el más rápido (*best fit* aprovecha mejor el almacenamiento, pero no suele haber mucha diferencia).



Con fragmentación externa, el archivo `D` ocupa los bloques 5, 6 y 7.

- Si no se necesita contigüidad en los bloques, entonces podemos usar los bloques 2, 3 y 5 para guardar la cadena. A esto se le denomina fragmentación de datos y suele aparecer con mucha frecuencia durante el uso del almacenamiento secundario de una computadora.



Con fragmentación de datos, el archivo `D` ocupa los bloques 2, 3 y 5.

La fragmentación de datos afecta el rendimiento de las operaciones de lectura y escritura, pero para explicar esto se requiere primero discutir cómo funcionan las tecnologías de almacenamiento secundario disponibles en la actualidad, lo cual se verá en la siguiente sección.

9.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/archivos-fragmentados`.

10. Tecnologías de almacenamiento persistente

El almacenamiento persistente se divide en dos clases: el almacenamiento secundario, el cual está disponible en todo momento (por ejemplo, discos duros conectados a la tarjeta madre de la computadora) y el almacenamiento terciario, el cual puede removerse durante el funcionamiento de la computadora (por ejemplo, discos ópticos y cintas magnéticas). En esta sección se discutirán las propiedades de los distintos dispositivos de almacenamiento que se usan en ambas clases.

- Discos duros magnéticos.

Imágenes de un disco duro magnético y de sus partes.
Los platos rotan y la cabeza requiere un movimiento mecánico.

Un disco duro magnético (conocido también como HDD por las siglas de *Hard Drive Disk*) consiste de uno o más platos que rotan sobre su eje, y de un grupo de cabezas mecánicas que leen o escriben datos sobre ellos. El medio físico que almacena los datos está hecho de un material que tiene dos estados magnéticos estables y persistentes, los cuales permiten representar los valores 0 y 1. El estado de un bit puede modificarse aplicando un campo magnético. La distancia entre la cabeza y el medio magnético es de unos pocos nanómetros, por lo que el disco puede dañarse si se mueve bruscamente. Actualmente, la mayoría de los discos duros de computadoras de escritorio son de este tipo y éstos cuentan con una capacidad de entre 1 y 10 terabytes, con un costo aproximado de $20 USD por terabyte.

Los datos están físicamente organizados en pistas que son caminos circulares concéntricos sobre la superficie de cada plato. Se suelen aprovechar ambas caras de un plato para almacenar datos, por lo que un disco con tres platos tendrá seis cabezas mecánicas. Una pista está dividida a su vez en sectores, los cuales son las unidades mínimas de lectura o escritura y corresponden con el concepto de bloque que ya se ha discutido con anterioridad. El tamaño de un sector o bloque físico suele ser de 4096 bytes, aunque el controlador puede simular la presencia de bloques más chicos mediante software. Al conjunto de pistas en la misma posición pero en diferentes platos se le denomina cilindro. Actualmente, el controlador de un disco duro recibe una solicitud de acceso a un bloque y éste calcula la superficie, la pista y el sector que corresponden con dicho bloque.

El tiempo de acceso a un dato se determina con base en dos tipos de retardo: el tiempo de búsqueda (*seek time*) que es el tiempo que necesita la cabeza mecánica en colocarse en la pista requerida, y

el retardo rotacional (*rotational latency*) que es el tiempo que tarda el sector requerido en pasar por debajo de la cabeza. El tiempo de búsqueda varía entre 1 y 10 milisegundos porque depende de la posición actual y la posición deseada de la cabeza (es más rápido cambiarnos a una pista contigua que movernos de la pista más externa a la más interna). El retardo rotacional se puede calcular en términos de las revoluciones por minuto (RPM) a las que giran los platos: a una velocidad de 7200 RPM, una rotación completa requiere 8 milisegundos. En la práctica, el retardo rotacional esperado corresponde con tener que esperar sólo media rotación o 4 milisegundos.

Los discos duros magnéticos están optimizados para acceso secuencial, ya que esto minimiza los movimientos de la cabeza y las rotaciones a esperar. Por ejemplo, el acceso más rápido consiste en leer una pista completa, luego movernos a una pista contigua para también leerla completa, etc. En esta situación ideal, la tasa de transferencia de un disco duro magnético es de aproximadamente 200 MB por segundo. El acceso arbitrario sobre un disco duro magnético puede ser drásticamente más lento. Los discos duros magnéticos cuentan con búferes para intentar mejorar la situación. Por ejemplo, el disco podría anticiparse a leer una pista completa y retenerla en un búfer, de modo que accesos posteriores a esa pista no requieran accesar al disco aún si la cabeza ya está en otra pista. De forma similar, las escrituras sobre el disco podrían retenerse en el búfer para así poder reordenar el orden en el que se ejecutarán y aprovechar al máximo el movimiento mecánico de la cabeza.

- Unidades de estado sólido.

Imagen de una unidad de estado sólido. Los módulos de memoria son electrónicos.

Una unidad de estado sólido (conocida también como SSD por las siglas de *Solid State Drive*) consiste en una colección de módulos de memoria. Todas estas componentes son eléctrónicas y no contiene partes mecánicas. En principio, un bit se puede almacenar con un transistor de compuerta flotante, la cual está aislada eléctricamente y puede retener su carga por largos períodos de tiempo. Aún si la compuerta está aislada, es posible llenarla o vaciarla de electrones mediante el efecto túnel aplicando un voltaje relativamente alto, ya sea positivo o negativo. Dada la naturaleza violenta de la escritura, las compuertas suelen desgastarse después de algunos cientos de miles de estas operaciones. Para maximizar el tiempo de vida útil de la unidad, el controlador evitará escribir repetidamente sobre el mismo bloque y preferirá usar todos los bloques disponibles, aún si los archivos se fragmentan. Una lectura no desgasta el medio y se hace midiendo la conductividad de la compuerta.

El tiempo de acceso a un dato es uniforme y es del orden de 10 microsegundos. Por su velocidad y ausencia de partes mecánicas, ésta es la tecnología que se usa en muchas laptops, en las memorias SD de celular o en las memorias USB. La capacidad usual de estas unidades suele ser de varias decenas o cientos de gigabytes, con un costo aproximado de $80 USD por terabyte. En ellas también existe el concepto de bloque, el cual suele ser de 4096 bytes. Los bloques de un archivo grande se suelen distribuir secuencialmente entre los distintos módulos de memoria, por lo que leer el archivo secuencialmente aprovechará la lectura de bloques en paralelo usando los distintos módulos. El acceso arbitrario será más lento que el acceso secuencial si las operaciones caen repetidamente en el mismo módulo, lo que desperdicia el paralelismo del hardware. Ante un acceso ideal, estas unidades tienen una tasa de transferencia de aproximadamente 2500 MB por segundo.

- Discos ópticos.

Imagen de un disco óptico y su unidad lectora. Los discos ópticos son compactos y removibles.

Un disco óptico es un disco que rota y cuyos datos se graban mediante un láser. Este láser es capaz de crear pequeños hundimientos en la superficie del disco y el proceso de lectura detecta la diferencia que existe en la refracción de un haz de luz ante la presencia o ausencia de los hundimientos. Los discos con múltiples capas permiten crear hundimientos más o menos profundos, lo que permite codificar más de un bit por hundimiento. Además, entre más fino sea el láser, más se podrá aprovechar la superficie del disco. Los llamados discos Blu-ray (cuyo nombre surge del color que tiene el láser que se usa para ellos) pueden llegar a almacenar hasta 100 gigabytes, con un costo aproximado de $15 USD por terabyte.

Los datos se escriben en una única pista continua con forma de espiral, por lo que no hay movimientos mecánicos súbitos. La lectura continua garantizada es deseable cuando lo que se almacena es audio o video, el cual fue el primer uso a gran escala de este tipo de medio. Al igual que en los medios de almacenamiento previos, también existe el concepto de bloque y éste suele ser de 2048 bytes. Los discos ópticos rotan a velocidades que van de 810 a 12960 RPM, pero el dispostivo suele detenerlos mientras no están en uso. Además, el tiempo de acceso a un dato puede llegar a tardar hasta 200 milisegundos, ya que accesar a parter arbitrarias del disco requiere reposicionarnos sobre la espiral que está rotando, lo cual es un proceso relativamente complicado y lento. Actualmente la tasa de transferencia es bastante baja, llegando a ser de 72 MB por segundo en el mejor caso.

- Cintas magnéticas.

Imagen de una cinta magnética. La cinta se debe montar en un dispositivo que la rebobina.

Una cinta magnética es un medio de almacenamiento removible y portátil, el cual además ofrece la mayor capacidad y el menor costo. Actualmente es común encontrar cintas magnéticas de 20 terabytes a un costo aproximado de $6 USD por terabyte. Las cintas se suelen almacenar en cartuchos relativamente resistentes incluso a incendios de baja magnitud, además de que la cinta en sí no contiene partes

mecánicas que puedan dañarse al moverse bruscamente. Por estas razones, las cintas magnéticas se usan con frecuencia para almacenar grandes volúmenes de datos con valor histórico. Al igual que en los otros medios, también existe el concepto de bloque y éste suele ser de 256 kilobytes.

Para leer o escribir datos, las cintas deben colocarse en un dispositivo que la rebobina. Los datos se guardan en una pista tipo serpentina, de modo que podemos comenzar leyendo rebobinando la cinta en una dirección y, cuando la cinta se termine y debamos rebobinarla en la dirección opuesta, podemos continuar leyendo. Las cintas se guardan enrolladas pero suelen medir varios metros desenrolladas, por lo que leer la cinta entera puede tomar incluso minutos. Esto significa que el tiempo de acceso a un dato arbitrario de la cinta es prohibitivamente alto y sólo tiene sentido procesar la cinta de forma secuencial. Accesando de esta última forma, la tasa de transferencia es de aproximadamente 400 MB por segundo.

A continuación se muestra un programa que realiza una prueba de rendimiento. El programa lee un archivo completo, ya sea mediante acceso secuencial o mediante acceso arbitrario. Suponiendo que el archivo es de n bytes y los bloques son de $b = 4096$ bytes, lo primero que el programa hace es construir una lista de índice de bloques $0, 1, 2, \ldots, \lceil \frac{n}{b} \rceil - 1$. Si se desea procesar el archivo secuencialmente, la lista permanece en orden ascendente; en caso contrario, la lista se revuelve. Posteriormente iteraremos sobre ella y nos posicionaremos en cada uno de los bloques con `fseek` para leerlo con `fread`.

Código

```
#include <algorithm>
#include <random>
#include <chrono>
#include <stdio.h>

long filesize(FILE* f) {
   fseek(f, 0, SEEK_END);
   long res = ftell(f);
   rewind(f);
   return res;
}

int main( ) {
   FILE* ent = fopen("arrobas.txt", "rb");
   long tam = filesize(ent);

   // El búfer de la biblioteca se deshabilita y se leerán bloques de 4096 bytes

   constexpr int BLOQUE = 4096;
   setbuf(ent, nullptr);
   int bloques = tam / BLOQUE + bool(tam % BLOQUE);
   long* orden = new long[bloques];
   for (int i = 0; i < bloques; ++i) {
      orden[i] = i;
   }

   // Determinar si revolver o no los accesos

   printf("Desea revolver los accesos? (0 o 1): ");
   int revolver;
   scanf("%d", &revolver);
   if (revolver) {
      std::shuffle(&orden[0], &orden[0] + bloques, std::mt19937_64( ));
   }

   // Prueba de rendimiento (siguiente página)
```

```
    auto t0 = std::chrono::high_resolution_clock::now( );
    char temp[BLOQUE];
    for (int i = 0; i < bloques; ++i) {
       fseek(ent, orden[i] * BLOQUE, SEEK_SET);
       fread(&temp[0], sizeof(char), BLOQUE, ent);
    }
    auto t1 = std::chrono::high_resolution_clock::now( );

    printf("%f", std::chrono::duration<double>(t1 - t0).count( ));
}
```

A continuación se reportan los tiempos de ejecución al usar un disco duro magnético WD Surveillance HDD WD20PURZ[3] y una unidad de estado sólido XPG GAMMIX S41[4] en acceso secuencial y arbitrario. El programa no aprovecha al máximo el acceso secuencial del dispositivo, ya que sólo solicita la lectura del siguiente bloque cuando el anterior ya le fue transferido.

	Acceso secuencial	Acceso arbitrario
HDD	5.933 (0.514)	1531.446 (0.942)
SSD	0.743 (0.545)	17.910 (0.951)

Tiempos de ejecución en segundos para leer los bloques de un archivo de 1 GB bajo Windows 10. Entre paréntesis el tiempo de ejecución cuando el archivo ya está en la caché del sistema operativo.

Como se observa, las unidades de estado sólido son mucho más rápidas que los discos duros magnéticos. El rendimiento de ambos se degrada al usar acceso arbitrario, pero es dramáticamente peor en los discos duros magnéticos debido al movimiento de las partes mecánicas. Esto causa la degradación en el rendimiento de una computadora cuando los archivos sufren fragmentación de datos. El almacenamiento persistente no influye cuando el archivo de hecho está en memoria, en la caché del sistema operativo.

11. Ordenamiento externo

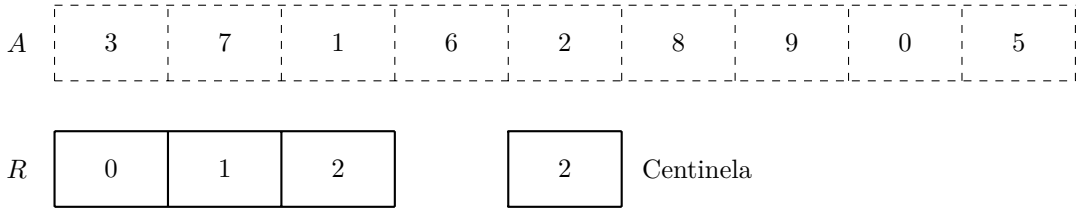
Cuando ordenamos una secuencia de datos que está en la memoria principal, se dice que el ordenamiento es interno. Ordenar un archivo que está en el almacenamiento secundario pero que cabe en la memoria principal no representa un problema: simplemente lo copiamos a memoria, lo ordenamos internamente y luego lo escribimos. Por otra parte, querer ordenar un archivo que es tan grande que no cabe en la memoria principal presenta algunas dificultades. Al ordenamiento que se lleva a cabo en esta situación se le llama ordenamiento externo, porque necesitaremos usar el almacenamiento secundario. En los algoritmos de ordenamiento externo, lo usual es escribir el resultado en un archivo de salida distinto al de entrada. Nosotros lo haremos así y estudiaremos dos algoritmos distintos: uno que sólo opera sobre los archivos de entrada y salida, y otro que emite archivos auxiliares durante el proceso. En ambos casos, supondremos que el archivo a ordenar tiene n elementos y que podemos almacenar un arreglo de m elementos en memoria principal, con $m < n$. Ya que la memoria principal es mucho más rápida que el almacenamiento secundario, ambos algoritmos aprovecharán lo más posible la memoria disponible.

La idea del algoritmo que no emplea archivos auxiliares es la siguiente. Apartaremos un arreglo R de m elementos en memoria y daremos $\lceil \frac{n}{m} \rceil$ pasadas completas por el archivo de entrada A. Durante la i-ésima pasada, calcularemos en R el i-ésimo grupo de elementos más pequeños de A. Al final de cada pasada, ordenamos R y lo escribimos en el archivo de salida. También guardaremos en un centinela el valor del último elemento escrito en esa pasada. La idea es que pasadas posteriores sobre el archivo de entrada completo ignoren valores menores o iguales que ese centinela, ya que se supuestamente ya

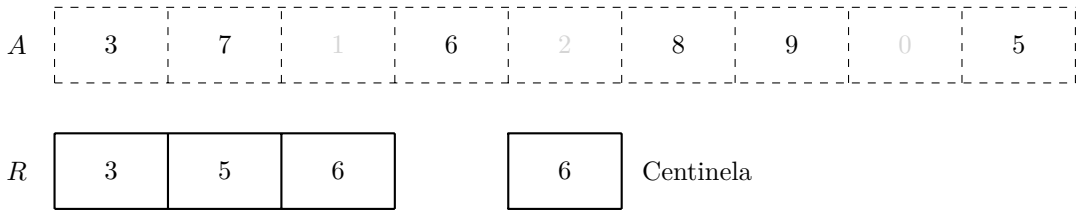
[3] http://products.wdc.com/library/SpecSheet/ENG/2579-810247.pdf

[4] https://www.adata.com/downloadfile_driver.php?file=XPG_GAMMIX%20S41_datasheet_ES_20201014.pdf

fueron escritos en pasadas previas. Si el archivo de entrada contiene valores duplicados, entonces también conviene almacenar en el centinela la posición del valor más grande ya escrito a la salida, para así poder distinguir entre las apariciones que ya fueron o aún no han sido escritas.



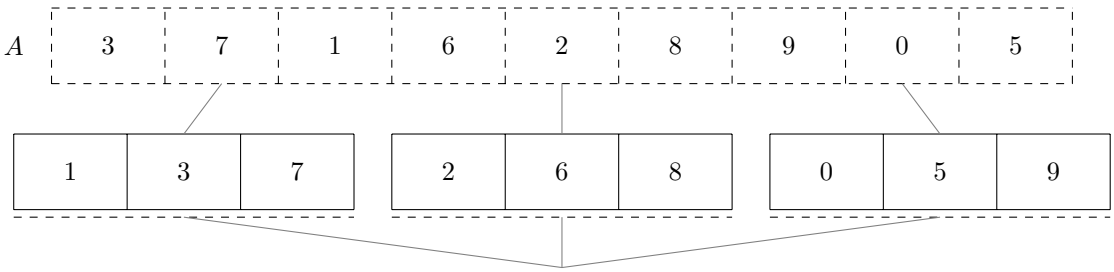
Ejemplo con $n = 9$ y $m = 3$, por lo que se necesitan $\lceil \frac{9}{3} \rceil = 3$ pasadas.
Después de la primera pasada, R tiene los elementos más chicos de A.



La segunda pasada ignora elementos ya escritos, por lo que
R contendrá al siguiente grupo de elementos más chicos.

El cálculo de los elementos más pequeños de A se hace como sigue: copiaremos los elementos de A en R mientras éste último tenga espacio. Cuando R se llene, eliminaremos su máximo actual si encontramos un elemento de A que sea más pequeño que él, el cual pasará a usar el espacio libre. Este proceso debe hacerse eficientemente, lo cual es posible con un montículo binario máximo. Usando esta estructura de datos, la cantidad de pasos que realiza el algoritmo es $\lceil \frac{n}{m} \rceil n \log_2 m$. Si $n = 10^{10}$ y $m = 10^9$, entonces el algoritmo tardará aproximadamente una hora en terminar a un ritmo de 10^9 pasos por segundo, lo cual supone que la lectura secuencial del archivo es lo suficientemente rápido como para que el cuello de botella sea el procesador. Si n aumenta a 10^{12}, entonces el algoritmo tardará aproximadamente un año.

El algoritmo que emplea archivos auxiliares es bastante más rápido y se conoce como mezcla de k-vías. Lo primero que haremos será dividir el archivo en partes que sí quepan en memoria (es decir, de a lo mucho m elementos cada una) y ordenar cada parte. Con esto tendremos $k = \lceil \frac{n}{m} \rceil$ archivos auxiliares ordenados. Ahora necesitaremos calcular la mezcla ordenada de esos archivos. Para hacerlo, abriremos simultáneamente los k archivos ordenados y construiremos un montículo binario mínimo donde cada elemento corresponde con el siguiente elemento a leer de cada uno de los k archivos ordenados (siendo el primer elemento el mínimo del archivo respectivo). La mezcla ordenada se calcula extrayendo repetidamente el mínimo del montículo y avanzando sobre el archivo de donde provino ese mínimo.



Después de dividir el archivo en partes que se puedan ordenar en memoria principal,
los archivos auxiliares resultantes se mezclan de forma ordenada.

A continuación se muestra una implementación del algoritmo de mezcla de k-vías.

Código

```
#include <algorithm>
#include <queue>
#include <stdio.h>

int filesize(FILE* f) {
   fseek(f, 0, SEEK_END);
   int res = ftell(f);
   rewind(f);
   return res;
}

struct archivo_auxiliar {
   FILE* f;
   int actual;    // el valor del último elemento leído del archivo
};

// importa menos el archivo con el siguiente elemento más grande
bool operator<(archivo_auxiliar a, archivo_auxiliar b) {
   return a.actual > b.actual;
}

int main( ) {
   FILE* ent = fopen("entrada.txt", "rb");
   int tam = filesize(ent) / sizeof(int);  // suponer que son enteros

   constexpr int grupo = 1000000;          // límite (artificial) de la memoria
   std::priority_queue<archivo_auxiliar> aux;
   for (int i = 0; i < tam / grupo + bool(tam % grupo); ++i) {
      int bufer[grupo];
      int r = fread(&bufer[0], sizeof(int), grupo, ent);
      std::sort(&bufer[0], &bufer[0] + r);
      // tmpfile crea un archivo temporal en modo "wb+"; escribimos en él
      FILE* temp = tmpfile( );
      fwrite(&bufer[0], sizeof(int), r, temp);
      // nos movemos al segundo elemento del archivo, pero guardamos el primero
      fseek(temp, sizeof(int), SEEK_SET);
      aux.push({ temp, bufer[0] });
   }

   FILE* sal = fopen("salida.txt", "wb");
   do {
      auto elegido = aux.top( ); aux.pop( );
      fwrite(&elegido.actual, sizeof(int), 1, sal);
      if (fread(&elegido.actual, sizeof(int), 1, elegido.f) == 1) {
         aux.push(elegido);
      }
   } while (!aux.empty( ));
}
```

La cantidad de pasos que realiza este algoritmo se calcula como sigue. Durante la primera etapa, se ordenan k archivos de m elementos cada uno, por lo que esta etapa realiza $k(m \log_2 m)$ pasos en total. La segunda etapa procesa n elementos (la suma de los elementos de todos los archivos) pero los compara en un montículo de tamaño k, por lo que realiza $n \log_2 k$ pasos. En total, el algoritmo completo realiza

$k(m \log_2 m) + n \log_2 k$ pasos. Obsérvese que es altamente probable que $k \ll m$ y entonces pareciera que la memoria principal no se aprovecha en la segunda etapa. Sin embargo, hay que recordar que la segunda etapa abre k archivos simultáneamente y que el almacenamiento secundario es bastante lento, por lo que se puede usar la memoria principal para apartar búferes grandes para cada archivo, de modo que se agilice la lectura (secuencial) de los mismos y que el cuello de botella no sea ésta. Si $n = 10^{10}$ y $m = 10^9$, entonces el algoritmo tardará aproximadamente seis minutos en terminar si vamos a un ritmo de 10^9 pasos por segundo. Si n aumenta a 10^{12}, entonces el algoritmo tardará aproximadamente once horas.

El ordenamiento de datos masivos se lleva a cabo frecuentemente con cintas magnéticas, por lo que en la práctica puede no ser posible leer de k cintas simultáneamente. También puede ocurrir (aunque es muy poco probable) que $k \geq m$. En estas situaciones, la mezcla de k-vías se puede replantear en términos de un torneo, con más etapas intermedias pero con menos archivos abiertos simultáneamente.

11.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Ordenando-los-dos-grupos-de-un-a`.

12. Búsqueda externa sobre secuencias

En términos generales, la búsqueda externa sobre secuencias es similar a la búsqueda interna: si la secuencia está desordenada, entonces sólo es posible hacer búsqueda lineal; si la secuencia está ordenada, entonces es posible hacer búsqueda binaria. Sin embargo, el análisis de costos difiere considerablemente según la tecnología de almacenamiento usado. A continuación se muestra una comparación al buscar sobre una secuencia de 10^9 enteros de 32 bits almacenados en ellos. Supondremos que todas las transferencias entre el procesador y el almacenamiento ocurren con la granularidad usual (línea de caché de 64 bytes para memoria principal, bloque de 4096 bytes para el almacenamiento secundario) y que el tiempo de transferencia de datos determina el tiempo total de ejecución, siendo irrelevante el tiempo del procesador. Para cada transferencia hecha, existen dos alternativas que el algoritmo de búsqueda binaria puede tomar: revisar únicamente el elemento que queríamos ver de esa línea o bloque, o revisar también los otros elementos transferidos. Hacer esta distinción es especialmente importante para los dispositivos de almacenamiento secundario, ya que la función `fseek` invalida el búfer de lectura: leer un elemento y eventualmente hacer otro `fseek` para leer otro elemento del mismo bloque es una terrible idea, ya que esto nos obligará a interactuar con el sistema operativo de forma innecesaria.

A continuación se retoman algunas características de distintos medios de almacenamiento. La tasa de transferencia de acceso arbitrario se estimó suponiendo dependencia secuencial entre cada acceso arbitrario. Posteriormente se estima el tiempo requerido para buscar un entero de 32 bits sobre una secuencia de 4 GB si usamos búsqueda lineal o búsqueda binaria y ocurre el peor caso.

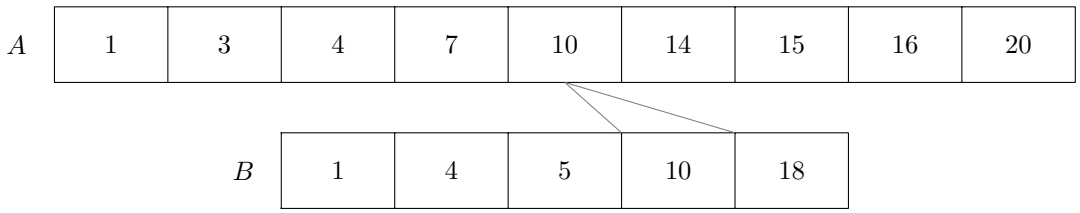
Características de medios de almacenamiento			
Característica	**Memoria principal**	**SSD**	**HDD**
Granularidad de acceso	64 bytes	4096 bytes	4096 bytes
Tiempo de acceso arbitrario	100 ns	10 µs	10 ms
Tasa de transferencia secuencial	30000 MB/s	2500 MB/s	200 MB/s
Tasa de transferencia arbitraria	640 MB/s	410 MB/s	0.41 MB/s

Búsqueda de un entero de 32 bits sobre una secuencia de 4 GB			
Tipo de búsqueda	**Memoria principal**	**SSD**	**HDD**
Búsqueda lineal	0.13 s	1.6 s	20 s
Búsqueda binaria con un acceso arbitrario por elemento	3000 ns	300 µs	300 ms
Búsqueda binaria con un acceso arbitrario por línea o bloque	2600 ns	200 µs	200 ms

Emplear búsqueda binaria es buena idea aún con discos duros magnéticos, donde un `fseek` involucra el movimiento de partes mecánicas. Esto es consecuencia de que la tasa de transferencia secuencial de un disco duro magnético simplemente no es tan grande como para justificar la búsqueda lineal. En estos discos, la búsqueda lineal procesará 4 GB en 20 segundos, pero suponiendo que sólo podemos usar un `fseek`, aprovecharlo para descartar la mitad del archivo resultará en un tiempo de ejecución de $0.01 + 10$ segundos, lo cual claramente es menor a pesar del alto costo de `fseek`. Sin embargo, cuando el archivo en cuestión es pequeño o lo que resta por procesar de él ya es poco, sí es cierto que la búsqueda lineal puede ser mejor que búsqueda binaria: con un archivo de 20 MB, la búsqueda lineal en un disco duro magnético tardará 0.1 segundos, mientras que la búsqueda binaria llamará a `fseek` aproximadamente 13 veces y entonces tardará $(13)(0.01) = 0.13$ segundos, que sí es peor que búsqueda lineal. Un análisis similar se puede hacer con las otras tecnologías de almacenamiento.

Una forma obvia de resolver más de una búsqueda es resolver eficientemente cada búsqueda individual. Sin embargo, esto dista de ser lo óptimo. Usando búsqueda lineal y suponiendo el peor caso, todas las líneas o bloques de la secuencia se transferirán del almacenamiento al procesador, por lo que ejecutar varias búsquedas individuales potencialmente implica transferir varias veces cada línea o bloque. Es mejor aprovechar cada transferencia para resolver tantas búsquedas se pueda con los elementos recién traídos, de modo que se itere sobre la secuencia completa una única vez. De forma similar, todas las ejecuciones individuales de búsqueda binaria siempre comienzan igual: trayendo el elemento o bloque de la mitad de la secuencia. Esa transferencia repetida de los mismos datos es innecesaria y puede evitarse.

El siguiente algoritmo de búsqueda de múltiples valores está inspirado en el algoritmo de mezcla binaria, el cual mezcla dos secuencias ordenadas usando una cantidad óptima de comparaciones[5]. Definiremos A a la secuencia sobre la que buscaremos y como B a la secuencia de los valores buscados. Supondremos que n es el tamaño de A y que m es el tamaño de B, con $n \geq m$. Sea a_c el elemento central de A, comenzaremos leyendo su valor y lo usaremos para particionar B en tres grupos: los elementos que son menores, los que son iguales y los que son mayores que a_c. Los valores iguales representan búsquedas de B que son satisfechas con a_c, mientras que los otros dos grupos deberán buscarse a la izquierda o a la derecha de a_c, dependiendo del sentido de la comparación. Esto se repite recursivamente mientras aún haya elementos que buscar y mientras aún haya elementos en la región candidata de A.



El elemento central de A particiona los elementos de B.

Código

```
void multibusqueda(int* ai, int* af, int* bi, int* bf) {
   if (ai != af && bi != bf) {
      int* ac = ai + (af - ai) / 2;
      int* izq = std::lower_bound(bi, bf, *ac);   // también se puede usar
      int* der = std::upper_bound(bi, bf, *ac);   // std::equal_range de C++
      for (int* p = izq; p != der; ++p) {
         busqueda_exitosa(p);
      }

      multibusqueda(ai, ac, bi, izq);
      multibusqueda(ac + 1, af, der, bf);
   }
}
```

[5]D. Knuth. The Art of Computer Programming, Volumen 2 (Sorting and Searching).

El algoritmo original de mezcla binaria supone que A y B no tienen repetidos y además mantiene la garantia de que $n \geq m$, intercambiando los roles de A y B si fuera necesario. Bajo estas suposiciones, el tiempo total de ejecución es $\log_2 \binom{n+m}{m} \approx m \log_2(\frac{n}{m})$. En contraste, hacer m búsquedas binarias individuales tendría un tiempo total de $m \log_2(n)$, que es mayor. Por ejemplo, si $n = 10^9$ y $m = 10^6$ entonces la multibúsqueda hace tres veces menos comparaciones con elementos de A. En el contexto de búsqueda externa sobre A, no tendremos tanta libertad en intercambiar los roles de A y B pero es altamente probable que $n \gg m$. Además, reducir el número de accesos en A sería equivalente a reducir el número de llamadas a `fseek`, de modo que el beneficio real puede llegar a ser considerable.

Es posible combinar todas las ideas vistas en esta sección (emplear búsqueda lineal para secuencias chicas, aprovechar lo más posible la transferencia de cada bloque y evitar la transferencia repetida de los mismos bloques al buscar múltiples valores) para obtener un algoritmo de búsqueda externa realmente eficiente. Los implementadores de bases de datos persistentes saben que minimizar los accesos al almacenamiento secundario es primordial y, dada una lista de tareas a realizar sobre la base de datos, los gestores de bases de datos invierten millones de ciclos del procesador para diseñar en memoria una estrategia eficiente de acceso al almacenamiento secundario. En parte por esto, es que un gestor de base de datos es más rápido ejecutando una lista de tareas que ejecutando las mismas tareas de forma individual.

12.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Busqueda-simple-sobre-un-archivo`.

13. Representación de caracteres y cadenas

Definir una cadena es sencillo: una cadena es una secuencia de caracteres. La definicion de caracter, así como la representación de cararacteres y cadenas en una computadora, son temas más complicados.

En términos muy generales, un caracter es una unidad de información. Los caracteres pueden estar asociados a símbolos, sonidos o incluso acciones. La interpretación que una computadora haga de un caracter en específico depende del contexto. A lo largo de las décadas y conforme han ido evolucionado las necesidades y las capacidades de cómputo, se han definidos distintos conjuntos de caracteres para usarse en computadoras. Curiosamente, todos ellos emplean una estrategia común: cada caracter del conjunto, además de recibir un nombre y una posible representación gráfica, también recibe un identificador entero único en el rango de 0 a $|\Sigma| - 1$, donde Σ denota al conjunto de caracteres y $|\Sigma|$ denota su cardinalidad. La idea es que los caracteres del conjunto se almacenen internamente como enteros.

Un problema de interoperabilidad podría ocurrir si dos conjuntos de caracteres definen el mismo caracter pero bajo identificadores enteros distintos. Por tal razón, se estandarizaron algunos conjuntos de caracteres, siendo dos de ellos los más relevantes. El primero es el conjunto de caracteres ASCII, el cual fue originalmente publicado en 1963 por el Instituto Nacional Estadounidense de Estándares (mejor conocido como ANSI, por sus siglas en inglés) y actualmente es el mínimo conjunto de caracteres que todo sistema de cómputo debe soportar. En él se definen 128 caracteres, incluyendo las letras del alfabeto inglés, los dígitos, símbolos de puntuación y caracteres de control. El tamaño limitado del conjunto permite almacenar caracteres ASCII usando el tipo entero `char` del lenguaje C, el cual tiene un rango garantizado de 0 a 127. Cabe mencionar que sólo se necesitan 7 bits para representar el rango anterior, mientras que un byte de 8 bits puede representar 256 valores distintos. En este sentido, tanto `char` como `unsigned char` consumen un byte, pero `unsigned char` tiene un rango garantizado de 0 a 255.

Una desventaja del limitado tamaño del conjunto ASCII, es que muchos caracteres útiles no se encuentran definidos en él. Por ejemplo, el conjunto ASCII excluye las vocales con acento, la ñ, el símbolo de negación lógica ¬ y el símbolo de derecho de autor ©, entre muchos otros. Los alfabetos de muchas naciones también se excluyen (letras griegas, rusas o árabes, tan sólo por mencionar algunas). Para solventar parcialmente este problema, muchas regiones del mundo comenzaron a usar conjuntos de caracteres denominados conjuntos ASCII extendidos, los cuales son compatibles con ASCII pero asignan símbolos en el rango de 128 a 255. Por ejemplo, el conjunto de caracteres Windows-1252 le asigna el entero 243 a la ó, mientras que el conjunto IBM-869 le asigna el mismo entero a la letra griega ϕ. A los conjuntos de caracteres orientados a bytes se les denomina páginas de códigos. Sin embargo, el problema de estos conjuntos sigue siendo la interoperatibilidad, ahora en el rango extendido, lo cual es consecuencia de que aún 256 caracteres simplemente son insuficientes para todos los usos.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

La tabla ASCII. Los conjuntos ASCII extendidos complementan el ASCII con 128 caracteres adicionales, pero estos conjuntos usan el rango adicional de formas distintas.

A principios de la década de 1990 surgieron dos esfuerzos inicialmente independientes, aunque ahora coordinados, que buscaban definir y estandarizar un conjunto de caracteres mucho más amplio y ambicioso. De estos esfuerzos sugieron el Conjunto Universal de Caracteres (UCS por sus siglas en inglés) y la codificación Unicode, los cuales son mantenidos por la Organización Internacional de Estándares (ISO por sus siglas en inglés) y por el Consorcio Unicode, respectivamente. Al igual que en otros conjuntos, a cada caracter se le asignó un identificador entero y los primeros 128 caracteres del UCS coinciden con los caracteres ASCII. En su primera edición, el UCS definía poco menos de 35 mil caracteres y un entero de 16 bits era suficiente para poder representarlos. Precisamente el tipo de dato `wchar_t` fue incorporado al lenguaje C para poder representar caracteres de este conjunto ampliado, por lo que en aquel entonces, algunos sistemas operativos (por ejemplo, Windows) definieron este tipo como un entero de 16 bits. Desafortunadamente, el UCS siguió creciendo con el paso de los años y en 2001 su tamaño sobrepasó súbitamente el rango representable en un entero de 16 bits. Actualmente existen casi 150 mil caracteres definidos en el UCS, por lo que el `wchar_t` es un entero de 32 bits en Linux, mientras que sigue siendo de 16 bits en Windows por cuestiones de compatibilidad hacia atrás. Versiones recientes de los lenguajes C y C++ definen los tipos enteros sin signo `char8_t`, `char16_t` y `char32_t`, siendo necesario en C incluir el archivo de biblioteca `<uchar.h>` para poder usarlos.

Desafortunadamente el problema de usar enteros de 32 bits para representar caracteres es que esto cuadruplicaría el consumo de memoria necesario para representar una cadena de caracteres ASCII, caracteres que son (por mucho) lo más comunes y usados. La codificación Unicode contempla el tipo de codificación UTF-8, el cual usa una cantidad variable de bytes por caracter. La principal ventaja de esta codificación es que cada caracter ASCII sigue representándose únicamente con un byte con el valor usual, mientras que el resto de los caracteres del UCS necesitan dos, tres o hasta cuatro bytes. La idea principal es que se usan el o los bits más significativos del byte actual para determinar cómo decodificar el caracter. Por ejemplo, como un caracter ASCII está en el rango de 0 a 127, entonces un byte denota un caracter ASCII si el bit más significativo del byte está apagado.

Rango de caracteres	Primer byte	Segundo byte	Tercer byte	Cuarto byte
0 - 127	0xxxxxxx			
128 - 2047	110xxxxx	10xxxxx		
2048 - 65535	1110xxxx	10xxxxx	10xxxxx	
65536 - 1114111	1110xxx	10xxxxx	10xxxxx	10xxxxx

La codificación UTF-8 para caracteres del UCS. Para codificar un caracter,
el patrón binario de su identificador entero se escribe en los bits señalados con x.

En cierto modo, la codificación UTF-8 es un esquema de compresión de enteros de 32 bits, donde hay preferencia por enteros con valores chicos. El tema de compresión se verá en la siguiente sección.

La representación más simple de una cadena es la de un arreglo de caracteres, aunque si consideramos la existencia de codificaciones de ancho variable, esta conceptualización se complica. Independientemente de lo anterior, existen dos formas comunes de determinar el número de caracteres de una cadena:

- Mediante la búsqueda de un caracter terminador: el tamaño de la cadena se desconoce de antemano, pero en ella aparece un caracter especial que denota su fin (y que no se considera parte de la misma). Normalmente esta representación sólo se usa con cadenas que sólo pueden contener un subconjunto del ASCII (de modo que no haya ambigüedad entre el terminador y un caracter válido de la cadena). El lenguaje C usa cadenas de este estilo y usa el caracter nulo como terminador.

Código

```
int strlen(const char arr[]) {
   int res = 0;
   while (arr[res] != '\0') {
      ++res;
   }
   return res;
}
```

- Acompañando a la cadena con un entero que denota su tamaño: además de almacenar la cadena, se almacena un entero que guarda el tamaño de ésta. Esta es la representación más usada en la actualidad porque no existe la necesidad de definir un caracter terminador ni de recorrer la cadena para conocer su tamaño. Por otra parte, tiene la desventaja de que un entero ocupa un más memoria que el byte usual de un caracter terminador.

Código

```
struct cadena {
   int tam;     // el tamaño de la cadena
   char arr[];  // el arreglo de caracteres con alguna capacidad
};
```

14. Compresión de datos

La compresión de datos consiste en construir una representación alternativa de los datos que ocupe menos espacio que la representación original. El ahorro en el almacenamiento implica que su transferencia de un medio a otro también es más rápida. Si bien no existe un forma general de comprimir datos cualesquiera, se pueden lograr ahorros significativos en casos comunes en la práctica. En general, los

algoritmos que se especializan en comprimir datos que posean alguna propiedad en particular son más eficaces en su tarea, precisamente porque pueden aprovechar dicha estructura.

Existen dos tipos de compresión. Cuando la compresión es *sin pérdida*, es posible reconstruir la representación original de los datos a partir de la representación comprimida. Trabajaremos un ejemplo en el que buscamos comprimir sin pérdida cadenas formadas exclusivamente por letras ASCII, donde además sabemos que existe una alta probabilidad de que cada letra se repita varias veces consecutivamente. La idea es reemplazar una subcadena de varias letras iguales por un preludio de dígitos seguido de la letra que se repite, donde los dígitos denotan la cantidad de veces consecutivas que aparece dicha letra. Por ejemplo, la cadena `gaaaatiittttoo` se puede comprimir a `g4at2i5t2o` siguiendo esta idea.

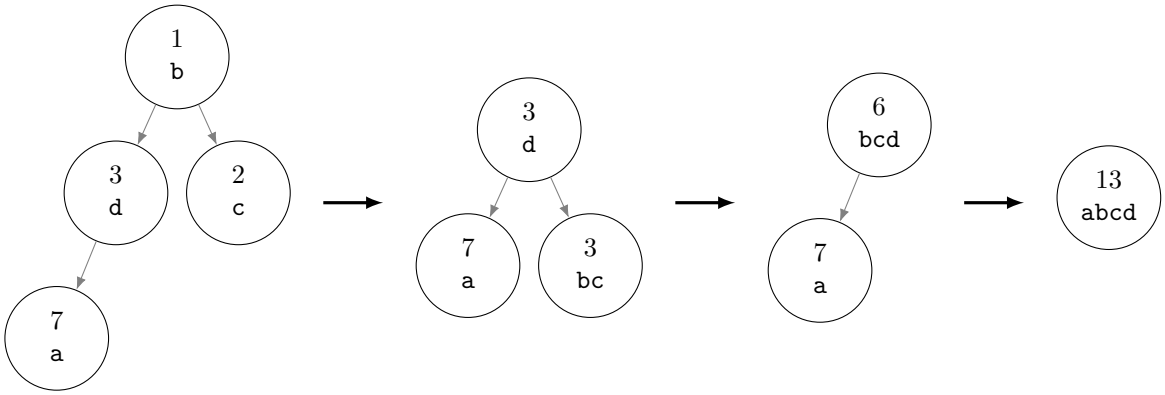
Código

```
char* comprime(const char arr[], int tam, char* w) {
   for (int i = 0; i < tam; ++i) {
      int veces = 1;
      while (i + veces < tam && arr[i] == arr[i + veces]) {
         ++veces;
      }
      if (veces > 1) {
         // sprintf nos permite imprimir en una cadena
         // %n nos permite conocer cuántos bytes se escribieron
         int avance;
         sprintf(w, "%d%n", veces, &avance);
         w += avance;
      }
      *w++ = arr[i];
      i += veces - 1;
   }

   return w;
}

char* descomprime(const char arr[], int tam, char* w) {
   for (int i = 0; i < tam; ++i) {
      int veces = 1;
      if (isdigit(arr[i])) {
         // sscanf nos permite leer de una cadena
         // %n nos permite conocer cuántos bytes se leyeron
         int avance;
         sscanf(&arr[i], "%d%n", &veces, &avance);
         i += avance;
      }
      for (int j = 0; j < veces; ++j) {
         *w++ = arr[i];
      }
   }
   return w;
}
```

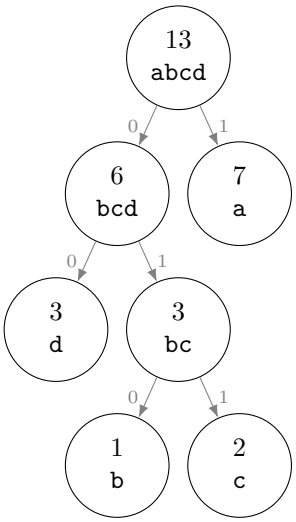
Desafortunadamente, la estrategia anterior no podría hacer nada ante la cadena `abacacadadada` porque no existen caracteres consecutivos iguales. Sin embargo, sí es posible comprimir esta cadena si observamos que la letra `a` aparece bastantes más veces que el resto de los caracteres. La idea será usar secuencias de menos de 8 bits para codificar caracteres que aparezcan muchas veces en la cadena. El algoritmo de Huffman es un algoritmo que calcula patrones bits para cada caracter que aparezca en la cadena, de modo que se garantiza que la longitud total de la cadena es la menor posible.

Lo primero que haremos será contar la cantidad de veces que aparece cada caracter y luego construiremos un montículo binario mínimo con nodos que representan los caracteres y sus frecuencias (el

criterio del montículo usa la frecuencia). Mientras el montículo tenga dos o más nodos, sacaremos un par de ellos y meteremos en su lugar un nodo representante de ambos, el cual tiene anotada la suma de las frecuencias de los nodos que reemplaza. Simultáneamente con esta acción, iremos construyendo un árbol binario donde el nodo representante recién creado es el padre de los dos nodos que reemplazó en el montículo. Como estaremos reemplazando dos nodos por uno, eventualmente el montículo se quedará sólo con un nodo, el cual también es la raíz del árbol binario asociado. Las transiciones izquierdas y derechas del árbol binario resultante se pueden interpretar como bits, de modo que el camino desde la raíz hasta cada hoja (que es un nodo original del montículo) es el patrón binario asociado al caracter de dicha hoja.



El montículo mínimo del algoritmo de Huffman para la cadena `abacacadadada` y la secuencia de operaciones sobre el montículo.



El árbol binario producido por el algoritmo de Huffman. El caracter `a` se codifica con `1`, el caracter `d` se codifica con `00`, el caracter `b` se codifica con `010` y el caracter `c` se codifica con `011`.

Los patrones de bits calculados por el algoritmo de Huffman son libres de prefijos; esto significa que ningún patrón es un prefijo de otro. Esto es importante porque se evitan ambigüedades que ocurrirían de otra forma al descomprimir (¿el patrón actual ya acabó o es parte de otro más grande?). Según lo anterior, el algoritmo de Huffman comprimiría `abacacadadada` en $7(1)+1(3)+2(3)+3(2) = 22$ bits, o poco menos de 3 bytes. Sin embargo, los bits de la cadena comprimida no son suficientes para descomprimirla: es necesario acompañar la cadena de un entero que denote su tamaño en caracteres (especialmente porque el último byte puede tener bits sobrantes) y también es necesaria la tabla que indica el patrón de bits de cada caracter. Por la existencia de esta sobrecarga, no suele valer la pena comprimir cadenas muy chicas, pero el ahorro puede ser significativo para cadenas grandes.

Código

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

struct nodo {
   char simbolo;
   int veces;
   nodo* hijos[2] = { };
};

bool operator<(const nodo& n1, const nodo& n2) {
   return n1.veces > n2.veces;
}

void imprime_tabla(const nodo& actual, const std::string& patron) {
   // los nodos internos siempre tienen dos hijos
   if (actual.hijos[0] == nullptr) {   // ¿es hoja?
      std::cout << actual.simbolo << ": " << patron << "\n";
   } else {
      imprime_tabla(*actual.hijos[0], patron + "0");
      imprime_tabla(*actual.hijos[1], patron + "1");
   }
}

int main( ) {
   // supondremos que la cadena es ASCII
   std::string s;
   std::cin >> s;

   int cuentas[128] = { };
   for (char c : s) {
      cuentas[c] += 1;
   }

   std::priority_queue<nodo> monticulo;
   for (int i = 0; i < 128; ++i) {
      if (cuentas[i] != 0) {
         monticulo.push(nodo{char(i), cuentas[i]});
      }
   }

   while (monticulo.size( ) > 1) {
      nodo n1 = monticulo.top( );
      monticulo.pop( );
      nodo n2 = monticulo.top( );
      monticulo.pop( );

      monticulo.push(nodo{
         char( ), n1.veces + n2.veces, { new nodo(n1), new nodo(n2) }
      });
   }

   imprime_tabla(monticulo.top( ), "");
}
```

Como la mayoría de las computadoras direccionan la memoria en bytes y no en bits, la manipulación de bits es complicada. El lenguaje C provee los siguientes operadores que operan sobre los bits un entero:

Operadores del lenguaje C para manipulación de bits	
Operador	**Descripción**
`a & b`	El resultado es un entero cuyos bits corresponden con la conjunción bit a bit de los operandos de la expresión.
`a | b`	El resultado es un entero cuyos bits corresponden con la disyunción inclusiva bit a bit de los operandos de la expresión.
`a ^ b`	El resultado es un entero cuyos bits corresponden con la disyunción exclusiva bit a bit de los operandos de la expresión.
`~a`	El resultado es un entero cuyos bits corresponden con la negación bit a bit del operando de la expresión
`a >> k`	El resultado es un entero cuyos bits corresponden con el desplazamiento `k` lugares a la derecha de los bits del operando.
`a << k`	El resultado es un entero cuyos bits corresponden con el desplazamiento `k` lugares a la izquierda de los bits del operando.

El lenguaje C también provee operadores compuestos con asignación para los operadores anteriores que sean binarios, es decir, que tengan dos operandos: `&=`, `|=`, `^=`, `>>=` y `<<=`. El lenguaje C++ permite especificar literales enteras en notación binaria con el prefijo `0b`. Por ejemplo, `0b1001` es 9 en decimal.

Para accesar al i-ésimo bit de una secuencia de bytes, necesitamos calcular en qué byte vive ese bit y qué bit de ese byte es el que nos interesa. El archivo `<limits.h>` de la biblioteca de C define la constante `CHAR_BIT`, la cual indica cuántos bits tiene un `char`. Actualmente, esta constante vale 8 en todos los sistemas razonables. A continuación se muestra un programa que ejemplifica cómo leer o escribir bits:

Código

```
#include <limits.h>
#include <stdio.h>

void escribe_bit(char bytes[], int i, bool v) {
   bytes[i / CHAR_BIT] &= ~(1 << (i % CHAR_BIT));
   bytes[i / CHAR_BIT] |= (v << (i % CHAR_BIT));
}

bool lee_bit(const char bytes[], int i) {
   return bytes[i / CHAR_BIT] & (1 << (i % CHAR_BIT));
}

int main( ) {
   char bytes[5];
   for (int i = 0; i < 5 * CHAR_BIT; ++i) {
      escribe_bit(bytes, i, i % 2);
   }
   for (int i = 0; i < 5 * CHAR_BIT; ++i) {
      printf("%d ", lee_bit(bytes, i))
   }
}
```

Un tipo de dato que implementa un arreglo de bits está disponible en `<bitset>` de C++. Los bits se pueden accesar con la notación usual de acceso a arreglo, pero no es posible obtener apuntadores a ellos.

Código

```
std::bitset<8> bits;                  // inicialmente todos apagados
bits[3] = true, bits[6] = true;       // prender algunos bits
printf("%d %lu\n", bool(bits[2]), bits.to_ulong( ));    // 0 y 72 (0b01001000)
```

El segundo tipo de compresión es la compresión *con pérdida*. Cuando una secuencia de bytes tiene un significado asociado, existe la posibilidad de que exista una secuencia más corta que tenga (en términos prácticos) el mismo significado que la original, pero donde no sea posible recuperar la secuencia original a partir de la secuencia comprimida. Por ejemplo, el modelo de color RGB permite expresar un color usando una combinación de tonalidades de rojo, verde y azul. Normalmente se usan 8 bits para especificar una tonalidad de entre 256 posibles para cada color, con lo que se pueden expresar $2^{3\times 8} = 16777216$ combinaciones distintas. Dada una imagen con 24 bits por pixel y que use millones de colores distintos, se podría producir otra que sólo use 12 bits por pixel, aún si esto requiriera aproximar colores que ya no pueden expresarse de forma exacta en el nuevo formato. La nueva imagen ocupará menos especial que la original y es probable que sea igual de útil. Sin embargo, es imposible recuperar la imagen original a partir de la imagen comprimida. Algunos otros ejemplos de compresión con pérdida son:

- En secuencias de puntos: se puede emplear interpolación para describir matemáticamente la secuencia completa, o bien, emplear interpolación por partes para describir grupos de puntos consecutivos.
- En imágenes: el formato GIF restringe la paleta de colores y el formato JPEG comprime con pérdida cada grupo de 8×8 pixeles, simplificando su representación a expensas de perder exactitud.
- En sonido: el formato MP3 simpifica la onda sonora para eliminar frecuencias y variaciones que el oído humano difícilmente percibe.
- En video: el formato MPEG es similar a JPEG en cuanto al manejo de imágenes, pero además describe únicamente los cambios que ocurren de un *frame* al siguiente.

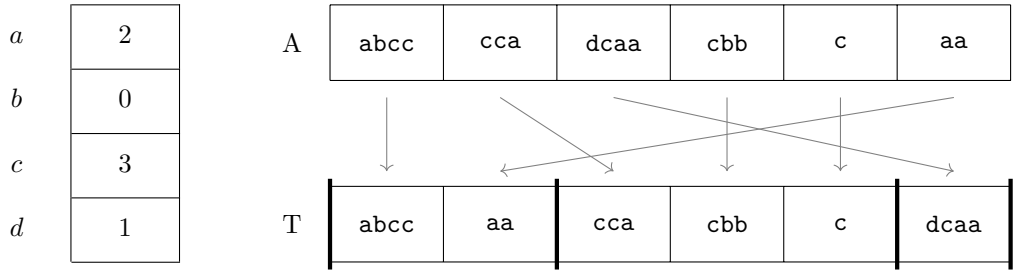
14.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Compresion-de-cadenas-facil`.

15. Ordenamiento y búsqueda de cadenas

Los algoritmos de búsqueda y ordenamiento basados en comparaciones funcionan correctamente sobre cadenas. Es decir, el algoritmo *merge sort* puede ordenar n cadenas usando $n \log_2(n)$ comparaciones, así como el algoritmo de búsqueda binaria puede buscar una cadena usando $\log_2(n)$ comparaciones. Sin embargo, comparar cadenas largas puede ser lento. Si las cadenas involucradas son de longitud L, entonces los algoritmos anteriores de propósito general tardarán $(n \log_2(n))(L)$ y $(\log_2(n))(L)$ pasos para ordenar y buscar, respectivamente. Existen algoritmos especializados para cadenas que son más rápidos.

El algoritmo de ordenamiento *radix sort* ordena n cadenas de longitud L como sigue. Inicialmente se ordenarán las cadenas considerando únicamente su primer caracter (el k-ésimo caracter con $k = 0$). Si los caracteres de las cadenas pertenecen a un conjunto Σ, lo anterior en realidad es equivalente a particionar las n cadenas en $|\Sigma|$ grupos disjuntos, lo cual se puede hacer en tiempo $n + |\Sigma|$ usando una idea similar al algoritmo de ordenamiento *counting sort*: se cuenta la frecuencia de aparición de los caracteres en la primera posición de las cadenas y luego se apartan subarreglos de tamaño justo dentro de un arreglo temporal; las cadenas se colocarán en los subarreglos según su primer caracter. Cada grupo se ordenará recursivamente pero ahora considerando el caracter $k + 1$ de las cadenas.



La primera etapa de *radix sort* para ordenar la secuencia A de cadenas con $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$.
En los algoritmos de cadenas, al conjunto Σ también se le conoce como alfabeto.

Para evitar copiar las cadenas durante el ordenamiento, normalmente se ordenan apuntadores a las mismas. Cada llamada recursiva de *radix sort* con $n > 1$ hace una cantidad de trabajo proporcional a $n + |\Sigma|$. Si todas las cadenas son iguales, entonces las cadenas siempre están juntas y la recursión tendrá una profundidad de tamaño L, por lo que el algoritmo total realizará $(n + |\Sigma|)(L) = nL + |\Sigma|L$ pasos en este caso. Si las cadenas no son iguales y sí se separan en grupos, entonces en análisis es más complicado. Si en una llamada llegara a ocurrir que $1 < n < |\Sigma|$, entonces el trabajo local de tamaño $|\Sigma|$ podría ser significativo. Una buena implementación de *radix sort* sería híbrida y cambiaría de algoritmo en este caso especial. De todos modos, el trabajo total no puede ser peor que $nL + n|\Sigma|$. Como $|\Sigma|$ es una constante, se dice que *radix sort* simplemente toma tiempo proporcional a nL, lo cual es óptimo.

Código

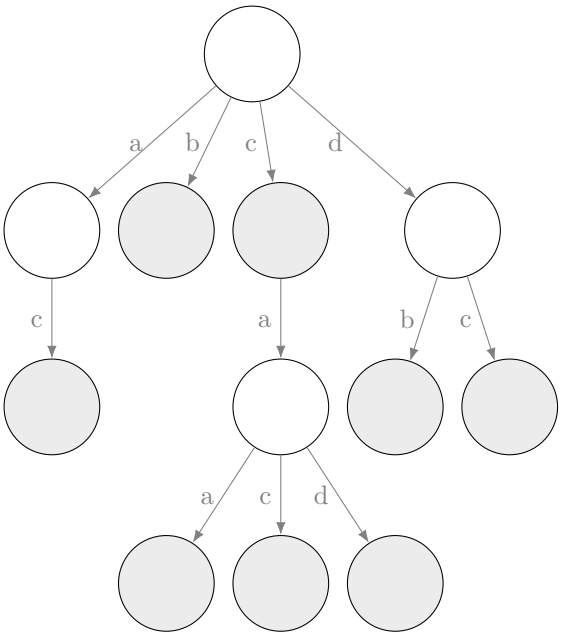
```
// ordenamiento de cadenas ASCII terminadas en el caracter nulo
void radix_sort(const char** ini, const char** fin, int k) {
   if (fin - ini <= 1) {
      return;
   }

   int cuenta[128] = { };
   for (auto i = ini; i != fin; ++i) {
      cuenta[(*i)[k]] += 1;
   }

   int inicios[128] = { };
   for (int i = 1; i < 128; ++i) {
      inicios[i] = inicios[i - 1] + cuenta[i - 1];
   }

   const char* temp[fin - ini];
   for (auto i = ini; i != fin; ++i) {
      temp[inicios[(*i)[k]]++] = *i;
   }

   std::copy(&temp[0], &temp[0] + (fin - ini), ini);
   for (int i = 1; i < 128; ++i) {
      radix_sort(ini + inicios[i] - cuenta[i], ini + inicios[i], k + 1);
   }
}
```

Poder ordenar una colección A de cadenas es importante, pero lo anterior no resuelve el detalle de que realizar una búsqueda binaria sobre esas cadenas tomaría tiempo $(\log_2(n))(L)$. Además, existen otras operaciones que nos gustaría poder llevar a cabo eficientemente. Las operaciones que queremos son:

- Determinar si una cadena aparece aparece en A.
- Determinar cuántas cadenas de A empiezan con cierto prefijo.
- Agregar o eliminar cadenas en A.

El último punto sugiere una estructura de datos dinámica. Más aún, deseamos reducir el tiempo por operación a L pasos, lo cual sería óptimo e independiente de la cantidad de cadenas de A.

La estructura de datos conocida como árbol de prefijos o *trie* permite lograr lo anterior. Un trie se puede ver como una representación en memoria del árbol de recursión que realiza *radix sort* durante el ordenamiento. Cada nodo del árbol será un nodo $|\Sigma|$-ario, donde cada transición a un nodo hijo corresponderá con un caracter del alfabeto. Las transiciones del nodo raíz particionan las cadenas de A según el primer caracter, mientras que los niveles inferiores del árbol se encargan de los caracteres sucesivos. No es necesario guardar copias explícitas de las cadenas almacenadas, porque las transiciones del árbol las denotan implícitamente. El fin de una cadena debe tener una representación especial.



Ejemplo de trie para las cadenas `ac`, `b`, `c`, `caa`, `cac`, `cad`, `db`, `dc`. Los nodos sombreados son terminales de cadena. Todas las hojas son terminales, pero también puede haber terminales internos.

Para buscar una cadena en un trie, simplemente intentaremos bajar por el árbol usando las transiciones que correspondan con la secuencia de caracteres de la cadena. La búsqueda fracasa si es imposible bajar por la ausencia de una transición, o si el último nodo visitado no es terminal. La inserción de una cadena es similar a la búsqueda, con la diferencia de que ésta última crea las transiciones y nodos que falten al momento de bajar. El último nodo visitado durante una inserción se marca como terminal. La eliminación de una cadena del trie podría implementarse simplemente buscando la cadena y haciendo que el último nodo visitado deje de ser terminal, pero esto tiene el potencial de desperdiciar memoria. Sin embargo, cadenas con un mismo prefijo comparten una parte del trie, por lo que no debemos eliminar nodos o transiciones impulsivamente. Si en cada nodo anotamos la cantidad de cadenas que hay en el subárbol que comienza a partir de dicho nodo, entonces podremos distinguir cuando un nodo ya no es usado por ninguna cadena. Esta misma información es útil para poder contestar rápidamente cuántas cadenas empiezan con cierto prefijo. Para mantener actualizada dicha información, basta con incrementar la cuenta de los nodos visitados durante una inserción exitosa y decrementar la misma durante una eliminación exitosa. Normalmente un trie no admite cadenas duplicadas, por lo que la inserción puede fallar.

A continuación se muestra una implementación simple de los algoritmos de inserción y búsqueda para un trie de cadenas ASCII terminadas en nulo. En esta implementación, se interpreta la existencia de una transición válida para el caracter nulo como la terminación de una cadena.

Código

```
struct nodo {
   nodo* hijos[128] = { };
};

void inserta(nodo* actual, const char* p) {
   do {
      nodo*& bajar = actual->hijos[*p];
      if (bajar == nullptr) {
         bajar = new nodo;
      }
      actual = bajar;
   } while (*p++ != '\0');
}
```

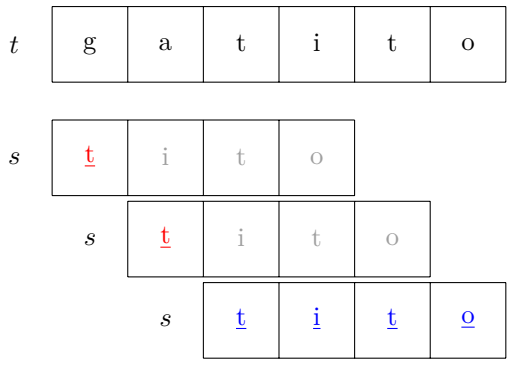
```
bool busca(const nodo* actual, const char* p) {
   do {
      const nodo* bajar = actual->hijos[*p];
      if (bajar == nullptr) {
         return false;
      }
      actual = bajar;
   } while (*p++ != '\0');

   return true;
}
```

En la implementación anterior, cada nodo consumirá mucha memoria aún si la mayoría de sus apuntadores son nulos. Es difícil implementar un trie rápido y que consuma poca memoria, pero es posible[6].

Un problema relacionado, pero computacionalmente muy distinto, es el de determinar si una cadena o patrón de longitud aparece como subcadena de una cadena posiblemente más grande llamada texto. Éste es el problema que se resuelve cuando usamos la típica opción de "Buscar" dentro de un editor de texto. Sea s un patrón de longitud m y t un texto de longitud $n \geq m$, podemos resolver este problema comparando s contra posiblemente todas las subcadenas de tamaño m de t.

Código

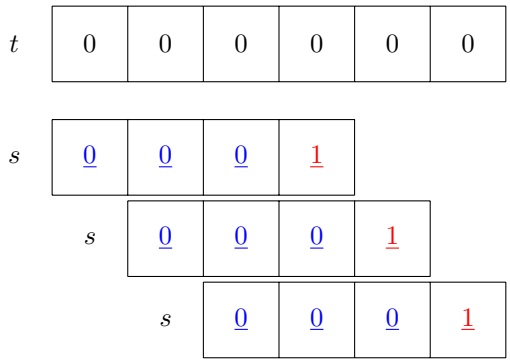
```
int busca(const char s[], int m, const char[] t, int n) {
   for (int i = 0; i < n - m + 1; ++i) {
      // std::equal de <algorithm> determina si dos secuencias son iguales y
      // regresa falso en cuanto encuentra la primera diferencia
      if (std::equal(&s[0], &s[0] + m, &t[i], &t[i] + m)) {
         return i;   // regresamos el punto de alineación
      }
   }
   return -1;
}
```



El algoritmo trivial de búsqueda de subcadenas compara el patrón contra las subcadenas del mismo tamaño en el texto. Cada comparación representa una alineación o deslizamiento del patrón en el texto.

El algoritmo anterior termina en cuanto encuentra la primera coincidencia del patrón en el texto. Además, en la práctica y con casos de entrada realistas, lo usual es que `std::equal` termine rápido si las cadenas comparadas no son iguales. Sin embargo, un ejemplo de caso patológico es buscar el patrón `0001` en el texto `000000`, ya que el algoritmo hará $n - m + 1$ llamadas a `std::equal` (porque ninguna tiene éxito) y además cada llamada a `std::equal` tendrá que comparar m caracteres porque el patrón difiere hasta el último caracter. El algoritmo tardará en total $nm - m^2 + m$ pasos, donde nm domina la suma. La posibilidad de que esto ocurra es inaceptable en muchos contextos.

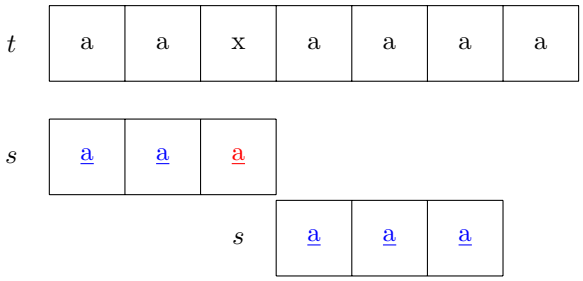
[6] `http://judy.sourceforge.net/index.html`



Ejemplo de caso patológico del algoritmo trivial de búsqueda de subcadenas.

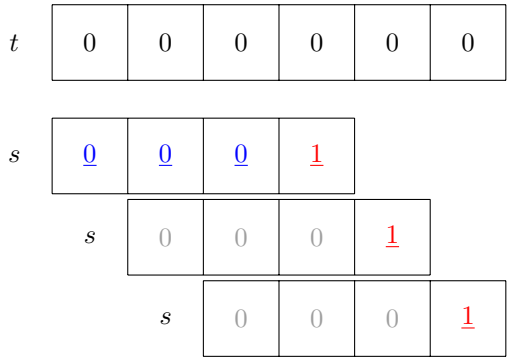
Un algoritmo mucho más eficiente para este problema es el algoritmo de Knuth-Morris-Pratt. Este algoritmo primero analiza el patrón antes de buscarlo en el texto, de modo que el resultado de dicho análisis lo usa para evitar comparaciones redundantes al momento de buscarlo y deslizarlo por el texto.

Para comprender la idea detrás del algoritmo de Knuth-Morris-Pratt, usaremos un ejemplo en el que la primera diferencia del patrón con el texto se da por culpa de un caracter del texto que no aparece en el patrón. Ante ese caracter problemático, lo único que tiene sentido es deslizar el patrón lo suficiente como para poder esquivar dicho caracter completamente.



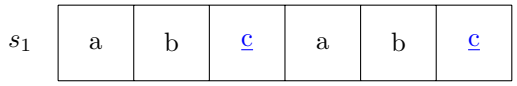
Buscando el patrón `aaa` en el texto `aaxaaaa`. La `x` del texto no coincidirá con ningún caracter del patrón. Lo esquivaremos por completo.

Sin embargo, no siempre es posible (o buena idea) deslizar el patrón tan bruscamente. Supongamos que los primeros k caracteres del patrón coincidieron antes de encontrar la primera diferencia. Diremos que el k-ésimo borde del patrón es la mayor subcadena de tamaño $u < k$ que es simultáneamente prefijo y sufijo de los primeros k caracteres del patrón. Al buscar y encontrar la primera diferencia, simularemos que primero deslizamos el patrón bruscamente a la derecha hasta esquivar la diferencia completamente, pero que luego retrocedemos el patrón u posiciones. La búsqueda reinicia tras hacer esto, pero es innecesario volver a revisar los primeros u caracteres del patrón, porque también eran los últimos que coincidían.

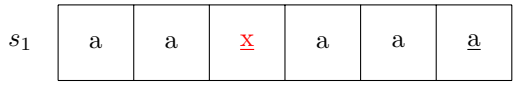


En el algoritmo Knuth-Morris-Pratt, es innecesario volver a comparar las secuencias de ceros.

El tamaño del k-ésimo borde identifica al borde de manera única, por lo que basta calcular y almacenar ese entero. Ahora calcularemos eficientemente los bordes del patrón para $0 \leq k \leq m$. Diremos que el 0-ésimo borde está indefinido, pero que el borde actual es de tamaño 0. A partir de ahí, intentaremos agrandar por la derecha el borde actual. Cada vez que procesemos un caracter a la derecha del sufijo actual, revisaremos si ese caracter también aparece a la derecha del prefijo respectivo. En caso afirmativo, hemos agrandado el borde actual. En caso contrario, tomaremos el borde del borde actual (el cual, si es que existe, es prefijo y sufijo del mismo) e intentaremos agrandar ése. Si volvemos a fallar, repetiremos el proceso hasta quedarnos con un borde actual vacío. Como ya se conocían los bordes de prefijos de menor tamaño y como todo borde es un prefijo, la información ya había sido calculada.



Antes de ver la última `c`, el borde actual era `ab` de tamaño 2. Al procesar la `c` a la derecha del sufijo y verificar que también aparece a la derecha del prefijo, agrandamos el borde a `abc`.



Antes de ver la última `a`, el borde actual era `aa` de tamaño 2. Con la última `a` es imposible agrandar el borde actual. El borde de `aa` es `a` y sí es posible agrandarlo usando la última `a`, por lo que el borde de `aaxaaa` es `aa` de tamaño 2.

A continuación se muestra una implementación del algoritmo que preprocesa el patrón para calcular los bordes y del algoritmo de búsqueda que emplea los bordes para evitar comparaciones innecesarias. El algoritmo de búsqueda es extremadamente similar al algoritmo de preprocesamiento, pero obsérvese que este algoritmo siempre usa el índice `i` para iterar sobre el texto y el índice `j` para iterar sobre el patrón. La implementación no simula deslizar el patrón de más para luego retrocederlo, sino que sólo lo desliza a la derecha la cantidad adecuada de posiciones. En términos de alinear el patrón, mantener `i` pero disminuir `j` quiere decir que el patrón se desliza hacia la derecha.

Código

```
void preprocesa(const char s[], int m, int bordes[]) {
   bordes[0] = -1;
   for (int i = 0, j = -1; i < m; ++i) {
      while (j != -1 && s[i] != s[j]) {
         j = bordes[j];
      }
      bordes[i + 1] = ++j;
   }
}

int busca(const char s[], int m, const char t[], int n, const int bordes[]) {
   for (int i = 0, j = 0; i < n; ++i) {
      while (j != -1 && t[i] != s[j]) {
         j = bordes[j];
      }
      if (++j == m) {
         return i + 1 - m;
      }
   }
   return -1;
}
```

El algoritmo de preprocesamiento toma $2m$ pasos porque el ciclo `while` que maneja el fallo de agrandar un borde sólo puede disminuir `j` tantas veces cómo ésta se haya incrementado, pero sólo se incrementa `j` una vez por iteración del ciclo externo, el cual se hace m veces. El mismo argumento se puede usar para el algoritmo de búsqueda, el cual entonces requerirá $2n$ pasos en el peor caso. Esto es mucho mejor que los nm pasos del algoritmo trivial. La tabla de bordes calculada por el preprocesamiento del algoritmo de Knuth-Morris-Pratt puede reutilizarse si se va a buscar el mismo patrón en diferentes textos.

15.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Cuenta-de-prefijos`.

16. Serialización de registros

El uso de archivos no está limitado a almacenar y procesar secuencias de enteros, caracteres u otros tipos básicos. Es usual querer almacenar datos compuestos que representan, por ejemplo, las entradas de una agenda telefónica, la información de los alumnos de una universidad, las transacciones bancarias del día, etc. El término *serialización* se emplea para referirnos al proceso de escribir la representación de un dato no trivial en un archivo. Del mismo modo, el término *deserialización* se emplea para reconstruir el dato en memoria a partir de su representación en el archivo. Las rutinas de escritura y lectura sin formato de C y C++ son las únicas en estos lenguajes que podrían intentar usarse para serializar y deserializar datos cualesquiera, pero desafortunadamente son inadecuadas en muchas situaciones. En esta sección se verá cuáles son esas situaciones problemáticas y cómo se pueden resolver.

Un *registro* es la representación serializada de un dato no trivial en un archivo. Normalmente, un programador usa tipos `struct` para representar datos sofisticados que residan en memoria. En principio, es posible emplear `fwrite` para volcar la representación en memoria de una variable `struct` en un archivo:

Código

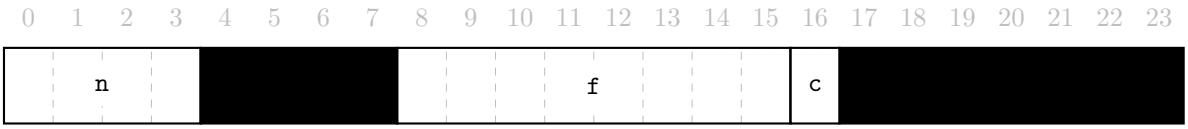
```
struct tupla {
   int n;
   double f;
   char c;
};

int main( ) {
   FILE* arch = fopen("temp.txt", "w+b");   // archivo de lectura y escritura
   tupla t1 = { 7, 3.14, '@' };
   fwrite(&t1, sizeof(tupla), 1, arch);

   rewind(arch);

   tupla t2;
   fread(&t2, sizeof(tupla), 1, arch);
   printf("%d %f %c\n", t2.n, t2.f, t2.c);
}
```

El código anterior funcionará como se espera. Sin embargo, el tamaño del archivo resultante será de 24 bytes, lo cual si bien coincide con el valor de `sizeof(tupla)`, es mucho mayor que la suma de los `sizeof` de los miembros del `struct`. Lo anterior se debe a que las variables en memoria están *alineadas*.

La alineación natural de un tipo de dato es un entero positivo a tal que el compilador procurará que todas las variables en memoria de ese tipo estén asignadas en direcciones múltiplos de a. El valor de a se puede obtener con el operador `_Alignof` de C y `alignof` de C++.

Código	Salida
`printf("%zu\n", alignof(char));`	`1`
`printf("%zu\n", alignof(double));`	`8`

Para los tipos de datos nativos, lo normal es que `sizeof` y `alignof` coincidan. El compilador también buscará honrar la alineación de las variables que sean miembros de un `struct`, aún si fuera necesario rellenar `struct` con bytes basura. La alineación de un `struct` es igual a la alineación más estricta de sus miembros, lo que le permite al compilador calcular una disposición estática para ellos. Además, el `sizeof` de un `struct` será un múltiplo de su propia alineación. Esta última regla existe para garantizar la alineación de miembros en los elementos de un arreglo de variables `struct`.



La disposición de los elementos dentro del `struct tupla`. Los bytes de relleno aparecen en negro.

La posición relativa en bytes de cada miembro de un `struct` se puede obtener con la utilidad `offsetof`, la cual se encuentra definida en el archivo `<stddef.h>` de la biblioteca de C.

Código	Salida
`printf("%zu\n", offsetof(tupla, n));`	`0`
`printf("%zu\n", offsetof(tupla, f));`	`8`
`printf("%zu\n", offsetof(tupla, c));`	`16`

Históricamente, los datos se alineaban en memoria por requerimientos del hardware. Actualmente es menos común encontrar procesadores que no puedan manejar datos desalineados, pero incluso entre los que sí pueden, hay pérdidas de rendimiento al hacerlo. El detalle aquí es que no hay requerimientos de alineación para datos almacenados en archivos, por lo que escribir la representación interna de un `struct` en un archivo escribiría también los bytes de relleno, desperdiciando espacio por culpa de un fenómeno que sólo ocurre en la memoria. La alternativa es serializar cada miembro manualmente, lo cual es engorroso. Desafortunadamente, aún no existe una buena forma de automatizar esto último en C o C++.

Código

```
void serializa(const tupla& t, FILE* arch) {
   fwrite(&t.n, sizeof(t.n), 1, arch);
   fwrite(&t.f, sizeof(t.f), 1, arch);
   fwrite(&t.c, sizeof(t.c), 1, arch);
}

void deserializa(tupla& t, FILE* arch) {
   fread(&t.n, sizeof(t.n), 1, arch);
   fread(&t.f, sizeof(t.f), 1, arch);
   fread(&t.c, sizeof(t.c), 1, arch);
}
```

Es muy usual que un tipo de dato compuesto contenga cadenas, pero la serialización de éstas es motivo de discusión. En general, las cadenas de un programa pertenecen a una de las siguientes tres categorías:

- Cadenas de longitud fija: En los sistemas de la vida real, algunas cadenas importantes tienen una longitud conocida de antemano. Por ejemplo, en México cada habitante cuenta con una CURP (Clave Única de Registro de Población), la cual es una cadena de 18 caracteres. La serialización y deserialización de estas cadenas puede implementarse transfiriendo exactamente la cantidad esperada de caracteres.

Código

```
void serializa(const char curp[], FILE* arch) {
   fwrite(&curp[0], sizeof(char), 18, arch);
}

void deserializa(char curp[], FILE* arch) {
   fread(&curp[0], sizeof(char), 18, arch);
}
```

- Cadenas de longitud variable con terminador: Con frecuencia se tienen cadenas con una longitud que no es fija, pero donde los caracteres válidos de la cadena pertenecen a un conjunto restringido. Bajo estas condiciones, la aparición de cualquier caracter fuera de dicho conjunto se puede interpretar como un terminador de la cadena. Las cadenas al estilo C usan el caracter nulo `\0` como terminador y, dependiendo del contexto, otros caracteres pueden jugar el mismo rol. Por ejemplo, los caracteres `@` o `|` tampoco suelen ser válidos dentro del nombre de una persona y tienen la ventaja de que se despliegan correctamente con cualquier editor de texto. Si decidimos usar la misma representación en un archivo, debemos serializar la cadena escribiéndola junto con el terminador (salvo que el archivo finalice con esa cadena, en cuyo caso el terminador podría omitirse), mientras que la deserialización buscaría el terminador o el fin de archivo para terminar la lectura de la cadena. Un caracter terminador generalmente ocupa sólo un byte, siendo éste suficiente para inferir el tamaño de la cadena sin importar qué tan larga sea. Sin embargo, esta representación no se puede usar cuando no existe ningún caracter que se pueda considerar inválido dentro de la cadena. Las cadenas al estilo C son incapaces de manipular datos binarios porque un caracter nulo que sea parte válida de la cadena se interpretará incorrectamente como el terminador de la misma. Cuando el formato elegido para las cadenas es incapaz de almacenar caracteres arbitrarios, se dice que el formato es binariamente inseguro.

Código

```
void serializa(const char nombre[], int tam, FILE* arch) {
   fwrite(&nombre[0], sizeof(char), tam, arch);
   fputc('|', arch);
}

int deserializa(char nombre[], FILE* arch) {
   int tam = 0, c;
   while ((c = fgetc(arch)) != EOF && c != '|') {
      nombre[tam++] = c;
   }
   return tam;
}
```

- Cadenas de longitud variable con tamaño almacenado: Cuando se tienen cadenas con longitudes que no son fijas, lo más fácil es simplemente almacenar su tamaño en un entero que la acompañe. Esto nos ahorra la necesidad de definir un caracter terminador, con lo que además se vuelve innecesario iterar sobre la cadena para calcular su tamaño. Al no existir un caracter terminador, la cadena también es binariamente segura. Si decidimos usar esta representación en un archivo, primero debemos serializar el tamaño de la cadena y después su contenido. La deserialización simplemente lee primero el tamaño y luego el contenido. Una desventaja es que, salvo que se desee usar un esquema relativamente complicado, el tamaño de la cadena se serializaría con un entero de ancho fijo, el cual podría ocupar más bytes que el contenido de la cadena en sí. En un sistema real suelen haber límites prácticos en el tamaño de una cadena (por ejemplo, la longitud de un nombre no debería superar jamás el valor máximo de un entero de 16 bits) y se podría usar el entero de ancho fijo más justo posible. Por ejemplo, el gestor de base de datos MySQL ofrece varios tipos distintos de cadenas de longitud variable (`VARCHAR`, `TEXT`, `MEDIUMTEXT` y `LONGTEXT`), las cuales difieren en la capacidad máxima de la cadena y, por consiguiente, en el ancho del entero que necesitan almacenar para indicar el tamaño de la misma.

Código

```
void serializa(const char s[], int tam, FILE* arch) {
   fwrite(&tam, sizeof(int), 1, arch);
   fwrite(&s[0], sizeof(char), tam, arch);
}

void deserializa(char s[], int& tam, FILE* arch) {
   fread(&tam, sizeof(int), 1, arch);
   fread(&s[0], sizeof(char), tam, arch);
}
```

En memoria, las secuencias de longitud variable se suelen implementar usando memoria dinámica (`malloc` de C o `new` de C++). Entonces, un `struct` que represente una de estas secuencias en el programa en realidad almacena apuntadores a memoria asignada en el almacenamiento libre.

Código

```
struct cadena {
   char* memoria;
   int tam;
};
```

Es un error fatal creer que si serializamos los apuntadores en un archivo, podremos deserializarlos en otro proceso (por ejemplo, una nueva ejecución del mismo programa) para recuperar la secuencia que estaba en memoria en la ejecución anterior. *Lo anterior no ocurrirá.* La memoria de un proceso se libera cuando éste termina, así que cualquier apuntador serializado carecerá de sentido al deserializarlo. Las estructuras de datos dinámicas que estén en memoria se tendrán que serializar guardando su contenido completo en el archivo, sin depender de apuntadores. En todo caso, la contraparte en un archivo de un apuntador es una posición entera sobre ése u otro archivo.

17. Mantenimiento de registros

Así como es usual tener archivos sólo para fines de consulta, también es usual tener archivos que están siendo actualizados constantemente. En particular, abordaremos un ejemplo sencillo que ilustre cómo implementar eficientemente una pequeña base de datos con operaciones de inserción, modificación, búsqueda y borrado de registros almacenados en archivos. Los archivos generados deben persistir para poder ser usados en otras ejecuciones del programa que manipule la base de datos. En este ejemplo, supondremos que los registros almacenan datos de alumnos, donde cada alumno cuenta con una matrícula que es un entero único no negativo, un nombre que es una cadena de longitud variable, y una edad que se representa en años enteros. Antes de programarlo, estableceremos las siguientes pautas de diseño:

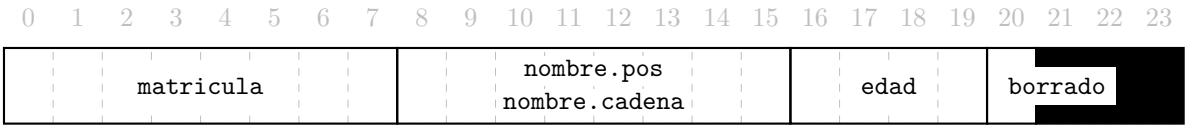
- La matrícula de cada alumno se asignará de forma automática e incremental.
- Las matrículas de los alumnos borrados no se reutilizarán.
- Los alumnos deberán buscarse por matrícula. No permitiremos buscarlos por ningún otro campo.
- Debemos poder buscar alumnos eficientemente (por ejemplo, con búsqueda binaria). Esto implica que:
  - Los alumnos se guardarán en el archivo ordenados por matrícula.
  - El tamaño de los registros de los alumnos debe ser predecible, de modo que podamos usar aritmética de posiciones para colocarnos de forma arbitraria en los registros de los alumnos. Como los nombres de los alumnos son de longitud variable, éstos se almacenarán en un archivo asociado.
- La operación de borrar un alumno debe ser eficiente. Para evitar reacomodar los registros del archivo, al alumno borrado simplemente lo marcaremos como tal, aunque siga ocupando espacio en el archivo.
- Se implementará una operación especial (y potencialmente lenta) que permita recuperar el espacio ocupado por alumnos borrados. Dicha operación especial recibe el nombre de compactación.

En el programa que implementaremos, representaremos a un alumno como sigue:

Código

```
struct alumno {
   size_t matricula;
   union {
      size_t pos;
      std::string* cadena;
   } nombre;
   int edad;
   bool activo;
};
```

Una `union` es una construcción del lenguaje C que permite declarar una región de memoria que tiene más de un tipo. El tipo que el compilador usará para interpretar la región de memoria dependerá del miembro que usemos de la `union`. El `sizeof` de una `union` no es la suma sino el máximo de los `sizeof` de sus miembros (es decir, sólo la memoria suficiente para usar la región con cualquiera de los tipos de los miembros). Nosotros usaremos una `union` para representar el nombre del alumno en dos contextos distintos: cuando usemos `pos` nos referiremos a la posición donde está almacenado el nombre del alumno en el archivo auxiliar; cuando usemos `cadena` nos referiremos a una cadena en memoria en donde está almacenado el nombre (ya sea mientras se lo pedimos al usuario o tras deserializarlo). La razón por la que usaremos un `std::string*` y no un `std::string` es porque, desafortunadamente, el valor de `sizeof(std::string)` es bastante grande en la mayoria de las implementaciones, mientras que el tamaño de un apuntador coincide con el tamaño de un `size_t`. Como queremos usar el mismo `struct` tanto en memoria como en el archivo, queremos evitar inflar el tamaño de la `union` innecesariamente. Por simplicidad, no nos importarán los bytes de relleno al serializar un alumno.



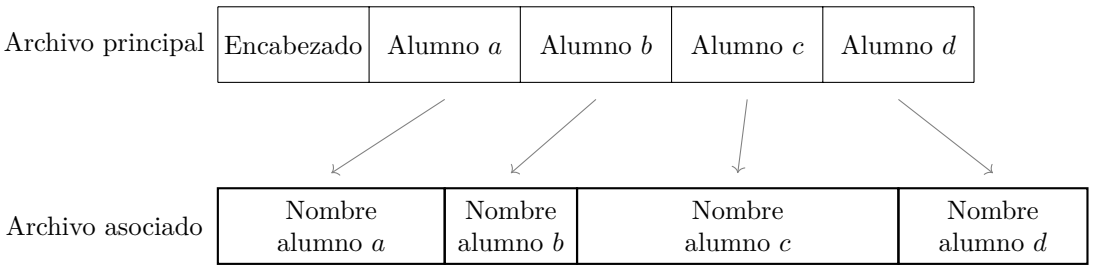
La disposición de los elementos dentro del `struct alumno`. En una computadora de 64 bits, tanto las variables `size_t` como los apuntadores son de 8 bytes. La `union` es una región de memoria del `struct` con dos tipos disponibles.

En la implementación a desarrollar, las matrículas se asignarán a partir de la matrícula 0. Como no debemos reutilizar matrículas de alumnos borrados, es necesario mantener en todo momento la información de qué matrículas ya se usaron, o bien, cuál es la próxima matrícula que se puede asignar. Este dato debe persistir y es independiente de qué alumnos están presentes en el archivo (si se borran todos los alumnos activos y después se compacta el archivo, ya no hay registros de alumnos pero de igual modo debemos evitar reutilizar sus matrículas). En consecuencia, el primer dato que contendrá el archivo principal será el número de la siguiente matrícula disponible, siendo éste un pequeño ejemplo de encabezado o *header* de un archivo, una parte del mismo que contiene información suplementaria a la información incluída en el cuerpo (en nuestro ejemplo, los datos de los alumnos en sí). El encabezado puede ampliarse para contener también metadatos útiles, por ejemplo, la cantidad de alumnos no borrados del archivo. En ausencia de este metadato, el diseño no permitiría calcular dicha cantidad eficientemente porque un alumno borrado seguirá ocupando espacio antes de que se compacte el archivo, así que la única alternativa sería iterar sobre todos los registros y contar los alumnos no borrados. Incluiremos este metadato en el encabezado.

Código

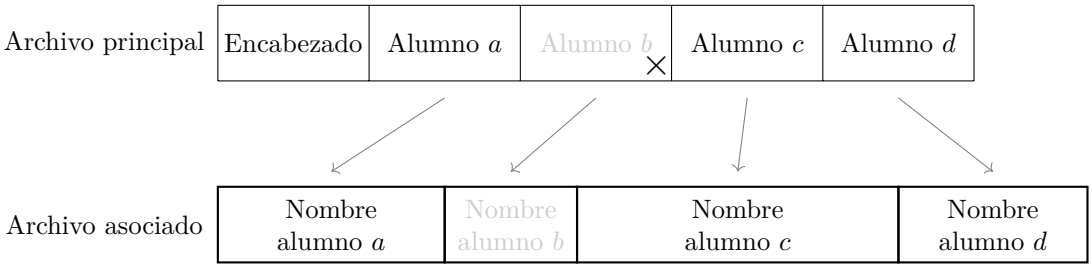
```
struct header {
   size_t matricula = 0;
   size_t activos = 0;
};
```

La base de datos estará conformada por dos archivos. El archivo principal contendrá el encabezado y los registros de los alumnos, mientras que el archivo asociado contendrá los nombres de los alumnos.



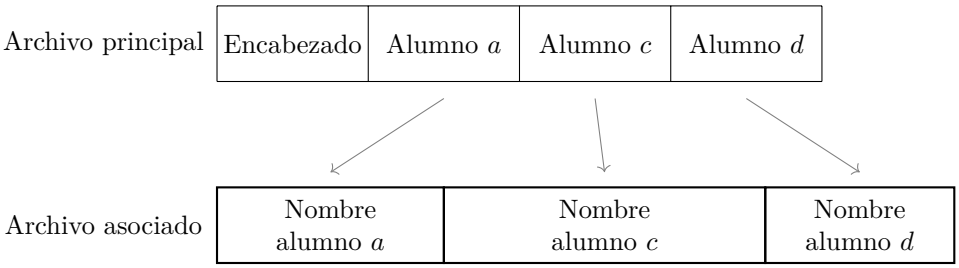
Vista general de los archivos de la base de datos. Los alumnos guardan posiciones del archivo de nombres.

Los nombres de los alumnos son cadenas de longitud variable. Serializaremos cada cadena almacenando explícitamente su tamaño (por simplicidad con `size_t`) y después su contenido. Con respecto al borrado de alumnos, en principio simplemente marcaremos el registro como borrado (poniendo en falso el miembro `activo` del `struct alumno`). El espacio usado por el alumno en ambos archivos permanecerá ocupado.



Vista de los archivos después de borrar el alumno b pero antes de compactar.

Sólo después de una compactación es que el espacio ocupado por registros borrados se recupera. La compactación se puede implementar iterando sobre los alumnos activos y reconstruyendo la base de datos en archivos nuevos, liberando los archivos viejos al terminar. La compactación es lenta y la implementación sugerida requerirá que el medio tenga espacio suficiente para almacenar temporalmente tanto los archivos viejos como los nuevos. Los gestores de bases de datos suelen ofrecer esta operación. Por ejemplo, el gestor de bases de datos MySQL ofrece la sentencia `OPTIMIZE TABLE` justamente para esto.



Vista de los archivos después de la compactación.

Como ya se dijo, la implementación sugerida de compactación reconstruye la base de datos en archivos nuevos. El último paso sería eliminar los archivos viejos y mover los nuevos a las rutas usadas por los archivos originales. En `<stdio.h>` de la biblioteca de C se encuentran dos funciones útiles para esto.

Funciones de gestión de archivos	
Función de biblioteca	**Operación realizada**
`int remove(const char s[]);`	Elimina el archivo con ruta `s`.
`int rename(const char s1[], const char s2[]);`	Mueve el archivo con ruta `s1` a la ruta `s2`.

No hay funciones equivalentes que tomen `FILE*`, por lo que las rutas de los archivos abiertos deben poder conocerse durante la ejecución del programa completo.

Las operaciones que manipulan la base de datos pueden ser relativamente complicadas de implementar, por lo que es mejor separarlas en funciones. Por otra parte, ua vez abiertos los archivos principal y asociado, lo mejor es siempre tener acceso a ambos, pero puede resultar engorroso pasar demasiados argumentos de función. Por lo mismo, agruparemos los manipuladores de ambos archivos en un `struct`.

Código

```
struct archivos {
   FILE* alumnos;
   FILE* nombres;
};
```

La función `main` implementará toda la interacción del programa con el usuario. El usuario podrá agregar un nuevo alumno, buscar un alumno por matrícula, listar a todos los alumnos activos, borrar un alumno, compactar la base de datos y salir del programa. Desde `main` llamaremos a las funciones individuales que implementan las operaciones, las cuales a su vez se auxiliarán de otro grupo de funciones con el prefijo `impl`. El programa no aprovechará la memoria principal disponible porque deseamos ver cómo implementar las operaciones bajo el supuesto de que la base de datos no cabe en memoria.

El suponer que tenemos poca memoria, el desear que `main` implemente toda la interacción con el usuario y el desear que las operaciones estén en funciones separadas, complica ligeramente el diseño. Por ejemplo, la función de listar todos los alumnos no debe decidir cómo imprimirlos y tampoco puede regresarle a `main` un arreglo en memoria con toda la información. Para resolver esto, `main` le enviará una función de retrollamada o *callback* a la función de listado: cada vez que la función de listado cargue un alumno en memoria, ésta ejecutará la retrollamada para procesar alumno leído; posteriormente, la función de listado reutilizará la misma memoria para leer el siguiente alumno y volverá a ejecutar la retrollamada. Se usará el mismo truco para implementar la operación de búsqueda e inserción de un alumno, de modo que las únicas funciones responsables de construir o liberar alumnos en memoria sean las funciones que implementan las operaciones sobre la base de datos; la función `main` sólo los procesará.

A continuación se describen las funciones que conformarán el programa. Posteriormente se listará el programa completo. El manejo de errores será mínimo (por ejemplo, se supondrá que abrir un archivo con modo `w+b` siempre tiene éxito), además de que algunos `fseek` podrían evitarse. Sin embargo, el programa estará razonablemente bien escrito para los fines de este curso y para los de esta sección en particular.

Funciones del programa de base de datos de alumnos	
Función	**Documentación**
`header impl_lee_header(const archivos&);`	Lee el encabezado del archivo principal.

`void impl_escribe_header(`
`  archivos&, const header&`
`);`
	Sobreescribe el encabezado del archivo principal.

`void impl_seek_end(const archivos&);`	Se posiciona al final de ambos archivos.

`bool impl_ubica_alumno(`
`  const archivos&, size_t`
`);`
	Si el alumno con la matrícula dada está activo, deja posicionado el archivo principal al inicio de su información y regresa verdadero.

`void impl_serializa_alumno(`
`  archivos&, const alumno&`
`);`
	Escribe al alumno en la posición actual del archivo principal (ignorando su nombre).

`void impl_serializa_alumno(`
`  archivos&, alumno, const std::string&`
`);`
	Escribe al alumno en la posición actual del archivo principal y guarda su nombre en la posición actual del archivo asociado.

`bool impl_deserializa_alumno(`
`  const archivos&, alumno&, std::string&`
`);`
	Lee el alumno que está en la posición actual del archivo principal, se posiciona en el archivo asociado y lee el nombre del alumno.

`archivos abre_archivos(bool = false);`	Si el booleano es falso, abre una base de datos existente o la crea si no existe. Si el booleano es verdadero, crea y abre archivos temporales.
`void inserta_alumno(archivos&, auto);`	Construye un alumno en memoria, ejecuta la retrollamada para llenar los datos distintos a la matrícula y lo inserta en la base de datos.

`bool busca_alumno(`
`  const archivos&, size_t, auto`
`);`
	Si el alumno con la matrícula dada está activo, lo carga en memoria, ejecuta la retrollamada y regresa verdadero.

`void lista_alumnos(`
`  const archivos&, auto, auto`
`);`
	Ejecuta la primera retrollamada con la cantidad de alumnos activos y la segunda con cada alumno activo traído a memoria.

`bool borra_alumno(archivos&, size_t);`	Borra el alumno activo con la matrícula dada.
`void compacta_archivos(archivos&);`	Compacta la base de datos.

Código

```
#include <stddef.h>
#include <stdio.h>
#include <iostream>
#include <string>

constexpr const char* ruta_alumnos = "alumnos.dat";
constexpr const char* ruta_nombres = "nombres.dat";
constexpr const char* temp_alumnos = "alumnos.tmp";
constexpr const char* temp_nombres = "nombres.tmp";

struct header {
   size_t matricula = 0;
   size_t activos = 0;
};

struct alumno {
   size_t matricula;
   union {
      size_t pos;
      std::string* cadena;
   } nombre;
   int edad;
   bool activo;
};

struct archivos {
   FILE* alumnos;
   FILE* nombres;
};

header impl_lee_header(const archivos& arch) {
   rewind(arch.alumnos);
   header h;
   fread(&h, sizeof(header), 1, arch.alumnos);
   return h;
}

void impl_escribe_header(archivos& arch, const header& h) {
   rewind(arch.alumnos);
   fwrite(&h, sizeof(header), 1, arch.alumnos);
}

void impl_seek_end(const archivos& arch) {
   fseek(arch.alumnos, 0, SEEK_END);
   fseek(arch.nombres, 0, SEEK_END);
}

bool impl_ubica_alumno(const archivos& arch, size_t matricula) {
   fseek(arch.alumnos, 0, SEEK_END);
   size_t ini = 0, fin = (ftell(arch.alumnos) - sizeof(header)) / sizeof(alumno);
   for (alumno a; ini != fin;) {
      size_t mitad = ini + (fin - ini) / 2;
      fseek(arch.alumnos, sizeof(header) + sizeof(alumno) * mitad, SEEK_SET);
      fread(&a, sizeof(alumno), 1, arch.alumnos);
```

```
      if (matricula == a.matricula) {
         fseek(arch.alumnos, -int(sizeof(alumno)), SEEK_CUR);
         return a.activo;
      } else if (matricula < a.matricula) {
         fin = mitad;
      } else if (matricula > a.matricula) {
         ini = mitad + 1;
      }
   }
   return false;
}

void impl_serializa_alumno(archivos& arch, const alumno& a) {
   fwrite(&a, sizeof(alumno), 1, arch.alumnos);
}

void impl_serializa_alumno(archivos& arch, alumno a, const std::string& cad) {
   a.nombre.pos = ftell(arch.nombres);
   fwrite(&a, sizeof(alumno), 1, arch.alumnos);
   size_t tam = cad.size( );
   fwrite(&tam, sizeof(tam), 1, arch.nombres);
   fwrite(&cad[0], sizeof(char), tam, arch.nombres);
}

bool impl_deserializa_alumno(const archivos& arch, alumno& a, std::string& cad) {
   if (fread(&a, sizeof(alumno), 1, arch.alumnos) == 1) {
      fseek(arch.nombres, a.nombre.pos, SEEK_SET);
      size_t tam;
      fread(&tam, sizeof(tam), 1, arch.nombres);
      cad.resize(tam);
      fread(&cad[0], sizeof(char), tam, arch.nombres);
      a.nombre.cadena = &cad;
      return true;
   }
   return false;
}

archivos abre_archivos(bool temp = false) {
   if (temp) {
      return { fopen(temp_alumnos, "w+b"), fopen(temp_nombres, "w+b") };
   }

   archivos arch = { fopen(ruta_alumnos, "r+b"), fopen(ruta_nombres, "r+b") };
   if (arch.alumnos == nullptr || arch.nombres == nullptr) {
      if (arch.alumnos != nullptr) {
         fclose(arch.alumnos);
      }
      if (arch.nombres != nullptr) {
         fclose(arch.nombres);
      }
      arch = { fopen(ruta_alumnos, "r+b"), fopen(ruta_nombres, "w+b") };
      impl_escribe_header(arch, header{0, 0});
   }
   return arch;
}
```

```
void inserta_alumno(archivos& arch, auto cb) {
   header h = impl_lee_header(arch);
   impl_escribe_header(arch, header{h.matricula + 1, h.activos + 1});
   impl_seek_end(arch);

   std::string cadena;
   alumno a = { h.matricula, { .cadena = &cadena }, -1, true };
   cb(a);
   impl_serializa_alumno(arch, a, cadena);
}

bool busca_alumno(const archivos& arch, size_t matricula, auto cb) {
   if (impl_ubica_alumno(arch, matricula)) {
      alumno a; std::string cadena;
      impl_deserializa_alumno(arch, a, cadena);
      cb(a);
      return true;
   }
   return false;
}

void lista_alumnos(const archivos& arch, auto cb_cuantos, auto cb) {
   cb_cuantos(impl_lee_header(arch).activos);
   alumno a; std::string cadena;
   while (impl_deserializa_alumno(arch, a, cadena)) {
      if (a.activo) {
         cb(a);
      }
   }
}

bool borra_alumno(archivos& arch, size_t matricula) {
   if (impl_ubica_alumno(arch, matricula)) {
      impl_serializa_alumno(arch, alumno{matricula});
      header h = impl_lee_header(arch);
      impl_escribe_header(arch, header{h.matricula, h.activos - 1});
      return true;
   }
   return false;
}

void compacta_archivos(archivos& arch) {
   archivos nuevos = abre_archivos(true);
   impl_escribe_header(nuevos, impl_lee_header(arch));
   lista_alumnos(arch, [](size_t){ }, [&](const alumno& a) {
      impl_serializa_alumno(nuevos, a, *a.nombre.cadena);
   });

   fclose(arch.alumnos), fclose(arch.nombres);
   fclose(nuevos.alumnos), fclose(nuevos.nombres);
   remove(ruta_alumnos), remove(ruta_nombres);
   rename(temp_alumnos, ruta_alumnos);
   rename(temp_nombres, ruta_nombres);
   arch = abre_archivos( );
}
```

```
int main( ) {
   archivos arch = abre_archivos( );

   for (;; std::cout << "\n") {
      std::cout << "Elige una opción válida:\n";
      std::cout << "1. Agregar alumno.\n";
      std::cout << "2. Buscar un alumno por matricula.\n";
      std::cout << "3. Listar los alumnos.\n";
      std::cout << "4. Borrar un alumno.\n";
      std::cout << "5. Compactar archivos.\n";
      std::cout << "6. Salir.\n";

      int opcion;
      std::cin >> opcion;

      if (opcion == 1) {
         inserta_alumno(arch, [](alumno& a) {
            std::cout << "Ingresa el nombre y la edad del alumno con matrícula "
                      << a.matricula << " (autocalculada): ";
            std::cin >> *a.nombre.cadena >> a.edad;
         });
      } else if (opcion == 2) {
         std::cout << "Ingresa la matrícula del alumno a buscar: ";
         size_t matricula;
         std::cin >> matricula;

         if (!busca_alumno(arch, matricula, [](const alumno& a) {
            std::cout << "Alumno encontrado: " << a.matricula << " "
                      << *a.nombre.cadena << " " << a.edad << "\n";
         })) {
            std::cout << "No se encontró a alumno. Verifica que existe.\n";
         }
      } else if (opcion == 3) {
         lista_alumnos(arch, [](size_t activos) {
            std::cout << "Hay " << activos << " alumnos activos\n";
         }, [](const alumno& a) {
            std::cout << a.matricula << " " << *a.nombre.cadena << " "
                      << a.edad << "\n";
         });
      } else if (opcion == 4) {
         std::cout << "Dame la matricula del alumno a borrar: ";
         size_t matricula;
         std::cin >> matricula;
         if (borra_alumno(arch, matricula)) {
            std::cout << "Alumno borrado exitosamente.\n";
         } else {
            std::cout << "No se encontró a alumno. Verifica que existe.\n";
         }
      } else if (opcion == 5) {
         compacta_archivos(arch);
      } else if (opcion == 6) {
         break;
      }
   }
}
```

17.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Sobreescribiendo-cadenas-de-long`.

18. Índices primarios y secundarios en archivos

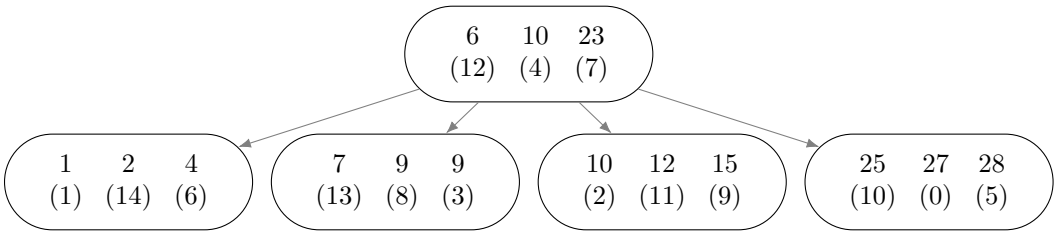
El manejo eficaz de una colección de registros normalmente requiere la existencia de un campo identificador de registro o *clave primaria*: los alumnos tienen matrícula, los pedidos de una tienda en línea tienen número de pedido. Como se vio en la sección anterior, si ordenamos los registros por clave primaria entonces es posible implementar eficientemente las operaciones de inserción, eliminación y búsqueda de registros por clave primaria. En el ejemplo de la base de datos de alumnos, la estructura de datos usada fue un arreglo ordenado sobre un archivo. Precisamente, un *índice primario* es una estructura de datos que nos permite buscar registros eficientemente con base en la clave primaria.

Desafortunadamente, hay situaciones donde la clave primaria no es un identificador numérico incremental. Por ejemplo, la CURP es una clave alfanumérica donde la primera parte se construye con base en los apellidos de la persona (la CURP incluye fecha de nacimiento, pero ni siquiera ésta es incremental porque no necesariamente es cierto que las personas se registran de forma ordenada). También hay situaciones donde puede ser necesario buscar registros por una *clave secundaria*: un campo distinto a la clave primaria que requiere búsqueda eficiente (un *índice secundario* es una estructura de datos que nos permite buscar registros eficientemente por clave secundaria). Por ejemplo, es común buscar pedidos por número de cliente, notando además que un cliente puede tener varios pedidos. En las situaciones anteriormente descritas no es viable usar un arreglo ordenado, pues las inserciones pueden ocurrir desordenadas en el campo de la clave. Cuando el orden de inserción no está garantizado pero la búsqueda debe ser eficientemente de todos modos, en memoria se emplean ya sea los árboles binarios de búsqueda balanceados o tablas de dispersión. A continuación se muestra un ejemplo en memoria, pero las estructuras de datos mencionadas anteriormente tendrán contrapartes en archivos persistentes.

Código

```
using id_pedido = size_t;    // alias de un tipo entero
using id_cliente = size_t;   // alias de un tipo entero

struct pedido {
   id_pedido id;             // clave primaria: única e incremental
   id_cliente cliente;       // clave secundaria: no ordenada y con duplicados
   std::string producto;
   //... otros datos
};

// se usan globales sólo para simplificar el ejemplo, pero no se recomiendan
std::vector<pedido> pedidos;                 // primario: ordenado por pedido
std::multimap<id_cliente, id_pedido> arbol;  // secundario: buscar por cliente

void agrega_pedido(pedido p) {
   p.id = pedidos.size( );    // suponer que el id está bien calculado porque
   pedidos.push_back(p);      // los pedidos se pueden cancelar pero no borrar
   arbol.insert({ p.cliente, p.id });    // guardar la relación cliente-pedido
}

// imprimir todos los productos comprados por el cliente
void imprime_productos(int cliente) {
   auto it = arbol.find(cliente);      // iterador al primer pedido del cliente
   while (it != arbol.end( ) && it->first == cliente) {
      std::cout << pedidos[it++->second].producto << "\n";
   }
}
```

Los árboles B y B++ son dos tipos de árboles de búsqueda que se emplean en archivos y son generalizaciones de los árboles binarios de búsqueda. Introduciremos primero el árbol B, pero para hacerlo debemos recordar lo que ocurre con un árbol binario de búsqueda en memoria. Estando en memoria, saltar de un nodo del árbol a otro es un potencial cambio de línea de caché y un potencial acceso a memoria principal. En los medios de almacenamiento persistente, la granularidad de los accesos es un bloque (generalmente 4096 bytes), pero saltar de un nodo a otro es un `fseek`. En este contexto, los árboles B son árboles k-arios de búsqueda balanceados con un valor muy grande de k (normalmente $k > 100$), de modo que cada nodo ocupe y aproveche lo más posible de un bloque y además la altura del árbol sea ridículamente pequeña, al ser proporcional a $\log_k(n)$ donde n es la cantidad de elementos del árbol. Así como los nodos binarios tienen dos hijos y guardan un elemento, los nodos de los árboles B tienen k hijos y guardan $k-1$ elementos. Los elementos en un mismo nodo deben estar ordenados.

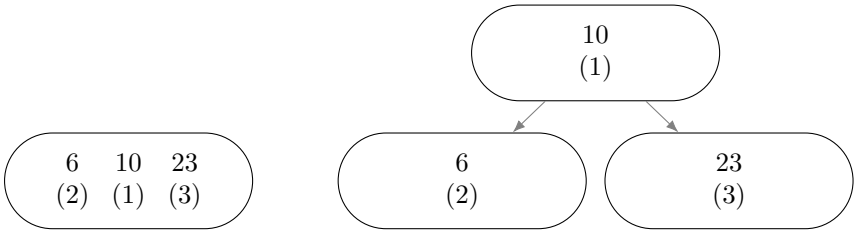


Ejemplo de un árbol B con $k = 4$. Cada nodo guarda claves de búsqueda y valores asociados. Los hijos de la raíz guardan claves $c_1 < 6$, $6 \leq c_2 < 10$, $10 \leq c_3 < 23$ y $23 \leq c_4$ respectivamente.

Código

```
template<int K>              // nodo parametrizado por número de hijos
struct nodo_k {
   struct elemento {         // en nuestro ejemplo, la relación cliente-pedido
      id_cliente cliente;    // el índice secundario estará ordenado por cliente
      id_pedido pedido;
   };
   elemento relaciones[K - 1];
   nodo_b* hijos[K];
   int tam;                  // con nodos tan grandes, es necesario llevar
};                           // la cuenta de cuántas entradas se están usando

using nodo = nodo_k<171>;    // el nodo de un árbol B que ocupa 4096 bytes
```

Las claves de búsqueda de un árbol B denotan intervalos semiabiertos (una clave es el fin de un intervalo y el inicio del otro), de modo que las claves almacenadas en los subárboles hijos caen en tales intervalos. El balance en un árbol B 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.

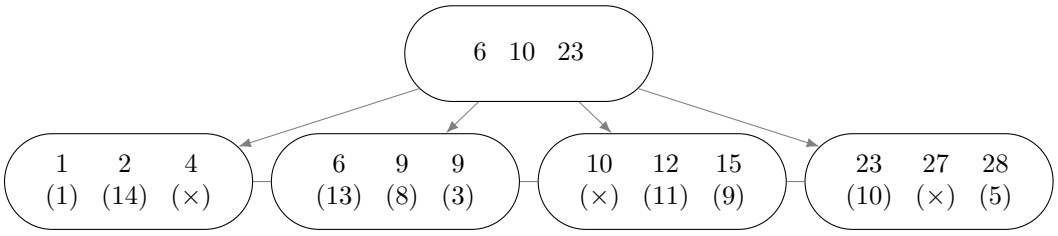


Un árbol 2-3-4 después de insertar las claves 6, 10, 23. El árbol se muestra antes y después de partir la raíz. Partir un nodo genera un árbol balanceado.

Durante una inserción, podemos partir los nodos llenos que veamos de forma preventiva. Hacer esto simplificará bastante la inserción, porque si un nodo tiene padre, entonces dicho padre tendrá espacio para recibir claves de hijos que se partan. Los nodos que se generan tras partir un nodo están "medio llenos" porque cada uno recibe aproximadamente la mitad de las claves del original. Desafortunadamente, es probable que permanezcan así a largo plazo. Por esta razón, la altura de un árbol B será aproximadamente de $\log_{\frac{k}{2}}(n)$. Esto no malo, ya que un árbol de 10^9 claves tendrá una altura máxima de 5 para $k = 171$.

Los árboles B tienen dos inconvenientes importantes en la práctica. Por una parte, la eliminación de claves es sumamente complicada y reconfigurar un árbol que realmente está guardado en el almacenamiento persistente no será una operación rápida. Por otra parte, la búsqueda de un intervalo de claves es una operación que puede ser común en algunas situaciones (por ejemplo, buscar todas las personas en un rango de edad) y los árboles B no ofrecen ninguna facilidad para acelerar dichas búsquedas, más allá del uso obvio del árbol. Un árbol B+ es una variante de árbol B con las siguientes diferencias:

- Cuando un nodo se parte, un árbol B+ retiene en uno de los nodos inferiores una copia de la clave que sube. Esto tiene como consecuencia que todas las claves del árbol aparecen en las hojas, mientras que los nodos internos tienen copias de sólo algunas claves.
- El borrado de una clave se hace únicamente en la hoja en la que ésta aparece. Lo anterior implica que toda búsqueda debe acabar en una hoja, pero también implica que el valor asociado a una clave sólo necesita almacenarse en la hoja y no en los nodos internos.
- Las hojas del árbol B+ están conectadas mediante una lista enlazada. Una búsqueda de intervalo puede buscar la clave inferior y luego visitar todas las claves de interés siguiendo la lista.



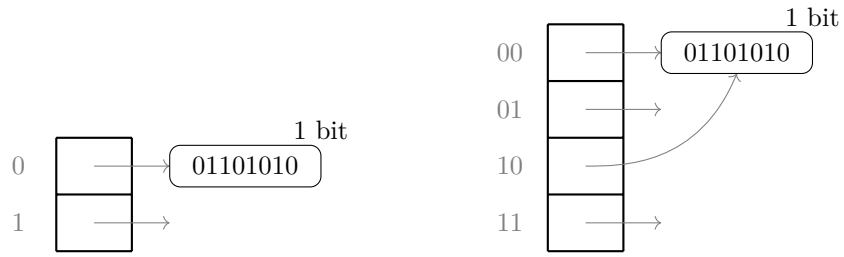
Ejemplo de un árbol B+ con $k = 4$. Los nodos internos sólo guardan claves, mientras que las hojas guardan claves y valores. El borrado se hace en las hojas y hay una lista enlazada que las conecta.

El borrado en un árbol B+ se hace en las hojas. Cuando se borran más de la mitad de las claves de una hoja, se intentarán redistribuir sus claves junto con las de sus hermanas, de modo de que ninguna hoja quede a menos de la mitad de su capacidad. Cuando la redistribución fracasa porque todas las hojas tienen pocas claves, dos hojas se fusionan. Tanto la redistribución de claves como la fusión de hojas requieren actualizar el contenido del padre (en particular, la fusión reduce la cantidad de claves que aparecen en el padre). Cuando el padre queda con pocas claves, vuelve ocurrir una redistribución con sus nodos hermanos, o bien, ocurre una fusión de nodos internos. Este proceso podría continuar en cascada hacia arriba hasta que todos los hijos inmediatos de la raíz se fusionen, con lo que la vieja raíz puede desaparecer y la altura del árbol disminuye. En general, tanto la búsqueda como la inserción y la eliminación en un árbol B+ tardarán tiempo proporcional a $\log_{\frac{k}{2}}(n)$.

Los valores asociados a las claves secundarias pueden ser copias de las claves primarias (como en el ejemplo anterior donde se almacenan parejas de identificadores cliente-pedido) o incluso pueden ser posiciones sobre el archivo principal donde están almacenados los registros asociados a las claves secundarias. Guardar las posiciones de los registros es más eficiente al momento de buscar, pero cualquier movimiento de los registros en el archivo principal provocará que los índices secundarios deban actualizarse también. Los árboles B+ que sean índices primarios de claves problemáticas (por ejemplo, cadenas) almacenan ya sea los registros completos o posiciones de archivos. Las claves que sean cadenas de longitud variable son un gran problema para los árboles B o B+, ya que si las cadenas se almacenaran directo en los nodos, entonces no se podría calcular un valor de k fijo para los mismos. La mayoría de las implementaciones obligan al programador a usar claves que sean prefijos de longitud fija de las cadenas de longitud variable.

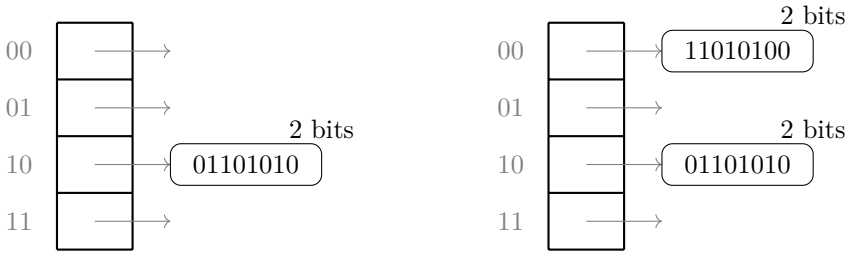
La última estructura para índices que se verá es la contraparte de las tablas de dispersión, pero en archivos. La adaptación recibe el nombre de dispersión extendida y funciona como se describe a continuación. Como en cualquier tabla de dispersión, el acceso a un arreglo de cubetas se hace en función del valor de dispersión h de la clave c. Supondremos que h es de w bits. En dispersión extendida, el arreglo principal guarda las posiciones de las cubetas en el archivo y éstas pueden ser nulas (es decir, una entrada del arreglo puede no tener cubeta asignada). Por su parte, las cubetas son de capacidad fija t. El arreglo tendrá 2^k entradas con $k \leq w$ e inicialmente tendremos que $k = 1$.

En principio, la inserción de una clave en la tabla de dispersión sólo considerará los últimos k bits de la clave. Si la entrada del arreglo no tiene cubeta, entonces se crea una. Si la entrada del arreglo sí tiene cubeta y ésta tiene espacio, entonces la clave se inserta. El problema ocurre cuando la cubeta sí existe pero está llena. En este caso, el arreglo principal tendrá que duplicar su capacidad (es decir, incrementaremos k), lo cual significa que ahora consideraremos más bits de la clave durante la inserción. Sin embargo, para evitar tener que hacer muchas operaciones sobre el archivo, las claves previas no se reacomodarán en las nuevas cubetas, además de que la recién inaugurada porción inferior de la tabla comenzará siendo una copia de la porción superior. Esto implica que una entrada del arreglo puede estar apuntando a una cubeta que ya no corresponda con el patrón de bits asignado a la cubeta, además de que esa cubeta puede tener claves que tampoco correspondan con el patrón de bits.



Ejemplo de tabla de dispersión extendida con cubetas de capacidad 1. Por simplicidad sólo se muestran los hashes. A la izquierda, la tabla tras insertar la clave con hash 01101010. A la derecha, la expansión que se realizaría como primer paso si queremos insertar en la cubeta llena.

Recordemos que el problema real ocurría al querer insertar en una cubeta llena. Una vez expandida la tabla, sólo esa cubeta sí se analizará a profundidad para redistribuir sus claves en las cubetas que en verdad correspondan. El resto de las cubetas seguirán en el estado previo, pero para evitar ambigüedades futuras, también se registran cuántos bits del patrón de dispersión se tomaron en cuenta para construirlas.



Al intentar insertar la clave con hash 11010100, se analiza la cubeta 00 y las claves que no pertenezcan ahí se migran a la cubeta correcta. Si esto bastó para generar un espacio libre, se inserta la nueva clave.

Desafortunadamente, en dispersión extendida basta que una cubeta que use los k bits se llene para que el arreglo completo deba duplicar su tamaño. Claramente es mala idea usar cubetas muy pequeñas o tener una función de dispersión de mala calidad, pero en general no podemos predecir lo que va a pasar. Dicho esto, la dispersión extendida no se usa mucho en la práctica y se prefieren los árboles B y B+.

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 2.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Invirtiendo-segmentos-de-una-mat`.

Solución: Para resolver este problema, debemos recordar que en C y C++ las matrices se almacenan en memoria por filas y no por columnas. Sin embargo, el problema especifica que se deben realizar distintas operaciones sobre las columnas de la matriz, lo cual será ineficiente en términos de accesos a memoria y aprovechamiento de las líneas de caché. Para resolver este problema, basta con trabajar sobre la matriz transpuesta, procesando por filas y no por columnas. Por supuesto, se debe cuidar que los datos impresos correspondan con la solución esperada.

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

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

   int matriz[n][n];
   for (int i = 0; i < n; ++i) {
      for (int j = 0, id = i + 1; j < n; ++j, id += n) {
         matriz[i][j] = id;
      }
   }

   int q;
   std::cin >> q;

   for (int i = 0; i < q; ++i) {
      char c;
      std::cin >> c;
      if (c == 'R') {
         int col, fi, fu;
         std::cin >> col >> fi >> fu;
         while (fi < fu) {
            std::swap(matriz[col][fi++], matriz[col][fu--]);
         }
      } else if (c == 'P') {
         int x, y;
         std::cin >> x >> y;
         std::cout << matriz[y][x] << "\n";
      }
   }
}
```

Soluciones de 3.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Busqueda-muy-eficiente-de-elemen`.

Solución: En este problema, los datos a procesar son tantos que incluso el tiempo de lectura y escritura llega a notarse en el tiempo total de ejecución. Las funciones `scanf` y `printf` suelen ser más rápidas que `std::cin` y `std::cout`, por lo que conviene usarlas. Aún así, búsqueda binaria será demasiado lenta, pero una tabla de dispersión con direccionamiento abierto funcionará.

Apartaremos `n` cubetas y, para no usar una cantidad excesiva de memoria, cada cubeta tendrá una capacidad de 3 enteros. Ya que los enteros a almacenar son aleatorios, se espera que el sobreflujo de cubetas ocurra rara vez. De todos modos, en caso de que ocurra podemos buscar hacia abajo la primera cubeta que tenga espacio disponible para insertar el elemento ahí. Durante la búsqueda de un elemento deberemos considerar que, si una cubeta está llena, puede ser que el elemento buscado sí esté pero se encuentre en una cubeta de abajo. En caso de que ya no haya cubetas libres hacia abajo, procederemos circularmente al inicio del arreglo.

```
#include <algorithm>
#include <stdio.h>

struct cubeta {
   int cuenta = 0;
   int elementos[3];
};

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

   auto arr = new cubeta[n];
   for (int i = 0; i < n; ++i) {
      int actual;
      scanf("%d", &actual);

      int elegir = actual % n;
      while (arr[elegir].cuenta == 3) {
         elegir = (elegir + 1) % n;
      }
      arr[elegir].elementos[arr[elegir].cuenta++] = actual;
   }

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

   for (int i = 0; i < m; ++i) {
      int actual;
      scanf("%d", &actual);

      int elegir = actual % n;
      bool encontrado = false;
      for (;;) {
         auto ini = &arr[elegir].elementos[0], fin = ini + arr[elegir].cuenta;
         encontrado = (std::find(ini, fin, actual) != fin);
         if (encontrado || arr[elegir].cuenta != 3) {
```

```
            break;
         }
         elegir = (elegir + 1) % n;
      }
      printf("%d\n", encontrado);
   }
}
```

Soluciones de 5.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Maximo-de-enteros-sin-formato`.

Solución: Para resolver este problema, se deben emplear correctamente las funciones `fread` y `fwrite` sobre `stdin` y `stdout` respectivamente, además de que se debe tener cuidado en quitar el salto de línea después de leer el entero `k`. La lectura de una secuencia con un tamaño desconocido se puede hacer usando el valor de retorno de `fread` para controlar un ciclo, ya que dicho valor corresponde con la cantidad de elementos leídos exitosamente.

```
#include <algorithm>
#include <stdio.h>
#include <stdint.h>

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

   int32_t res = 0, actual;
   if (k >= 0) {
      for (int i = 0; i < k; ++i) {
         fread(&actual, sizeof(actual), 1, stdin);
         res = std::max(res, actual);
      }
   } else {
      while (fread(&actual, sizeof(actual), 1, stdin) == 1) {
         res = std::max(res, actual);
      }
   }

   fwrite(&res, sizeof(res), 1, stdout);
}
```

Soluciones de 8.1

1. Resuelve el problema `https://omegaup.com/arena/problem/El-k-esimo-entero-de-un-archivo`.

Solución: Para resolver este problema, basta aplicar correctamente las funciones `fseek` y `ftell`. Se puede usar `ftell` para calcular la posición del inicio de la secuencia de enteros sin formato. Con este dato, es fácil calcular la posición del entero `k` y luego regresar a la posición del `k`-ésimo entero.

```
#include <stdio.h>

int main( ) {
   int n;
   scanf("%d", &n);
   getchar( );
   int offset = ftell(stdin);

   fseek(stdin, 4 * n + 1, SEEK_CUR);
   int k;
   scanf("%d", &k);

   fseek(stdin, 4 * k + offset, SEEK_SET);
   int v;
   fread(&v, sizeof(int), 1, stdin);
   printf("%d", v);
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/Invirtiendo-los-enteros-de-un-ar`.

Solución: Para resolver este problema, debemos recordar que llamar a `fseek` invalida el búfer de lectura, además de que leer elementos individuales es lento incluso en la presencia de un búfer. Por esta razón, conviene leer muchos elementos en la misma operación, por ejemplo $g = 10^5$ elementos. Por el orden de impresión solicitado en el problema, conviene usar `fseek` para posicionarnos al inicio de los últimos g elementos, leerlos e imprimirlos al revés, repitiendo el proceso hacia atrás.

```
#include <algorithm>
#include <stdio.h>

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

   constexpr int grupo = 100000;
   int offset = ftell(stdin);
   int bufer[grupo];

   for (int fin = n; fin > 0; fin -= grupo) {
      int leer = std::min(fin, grupo);
      fseek(stdin, offset + (fin - leer) * sizeof(int), SEEK_SET);
      fread(&bufer[0], sizeof(int), leer, stdin);
      std::reverse(&bufer[0], &bufer[0] + leer);
      fwrite(&bufer[0], sizeof(int), leer, stdout);
   }
}
```

Soluciones de 9.1

1. Resuelve el problema `https://omegaup.com/arena/problem/archivos-fragmentados`.

Solución: El límite de memoria de este problema nos permite leer la cadena del disco en memoria, pero no nos permite reconstruir en memoria cada archivo: los accesos los tendremos que satisfacer directamente de la cadena. Afortunadamente, lo anterior se puede resolver con aritmética: con una división podemos calcular qué bloque del archivo es el que necesitamos (¿el primero del archivo?, ¿el segundo?, etc) y con un módulo podemos calcular el índice local del caracter k en dicho bloque. Sólo hay que usar esa información para calcular una posición absoluta sobre la cadena.

```
#include <stdio.h>

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

   char memoria[p * b + 1];
   scanf("%s", &memoria[0]);

   int n;
   scanf("%d", &n);
   int* entradas[n];

   for (int i = 0; i < n; ++i) {
      int id, c;
      scanf("%d%d", &id, &c);
      entradas[id] = new int[c];

      for (int j = 0; j < c; ++j) {
         scanf("%d", &entradas[id][j]);
      }
   }

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

   for (int i = 0; i < m; ++i) {
      int id, k;
      scanf("%d%d", &id, &k);
      printf("%c\n", memoria[b * entradas[id][k / b] + (k % b)]);
   }
}
```

Soluciones de 11.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Ordenando-los-dos-grupos-de-un-a`.

Solución: Para resolver este problema, basta notar que la memoria límite es suficiente para almacenar todos los elementos con la misma paridad. Entonces, podemos leer el archivo completo para guardar, ordenar y escribir los elementos pares, posicionarnos otra vez al inicio de la secuencia de enteros y repetir el proceso pero ahora para los elementos impares.

```
#include <algorithm>
#include <stdio.h>
```

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

   int offset = ftell(stdin);
   for (int p = 0; p < 2; ++p) {
      int grupo[n / 2], tam = 0, actual;
      for (int i = 0; i < n; ++i) {
         fread(&actual, sizeof(int), 1, stdin);
         if (actual % 2 == p) {
            grupo[tam++] = actual;
         }
      }
      std::sort(&grupo[0], &grupo[0] + tam);
      fwrite(&grupo[0], sizeof(int), tam, stdout);
      fseek(stdin, offset, SEEK_SET);
   }
}
```

Soluciones de 12.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Busqueda-simple-sobre-un-archivo`.

Solución: En este problema, el tiempo límite es demasiado bajo como para poder resolverlo usando búsqueda lineal. Sin embargo, como sólo hay que realizar una búsqueda, basta emplear búsqueda binaria sobre el archivo aún si no aplicamos las optimizaciones descritas en las notas. En lugar de manejar apuntadores sobre la secuencia, manejaremos índices de elementos.

```
#include <stdio.h>

int busqueda_binaria(int ai, int af, int b, int offset) {
   for (;;) {
      int am = ai + (af - ai) / 2, actual;
      fseek(stdin, offset + am * sizeof(int), SEEK_SET);
      fread(&actual, sizeof(int), 1, stdin);
      if (b == actual) {
         return am;
      } else if (b < actual) {
         af = am;
      } else if (b > actual) {
         ai = am + 1;
      }
   }
}

int main( ) {
   int n, b;
   scanf("%d", &n);
   getchar( );
   fread(&b, sizeof(int), 1, stdin);
```

```
   getchar( );
   printf("%d\n", busqueda_binaria(0, n, b, ftell(stdin)));
}
```

Soluciones de 14.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Compresion-de-cadenas-facil`.

Solución: Para resolver este problema, basta con emplear los algoritmos de compresión y descompresión listados en las notas y que son específicos para este problema. Aunque el código se puede reducir si el resultado se imprime conforme se calcula (en lugar de guardarlo en una cadena y luego imprimir la cadena), se usarán las funciones tal como aparecen en las notas para ilustrar su uso. Haciendo esto último, sólo se debe recordar que las funciones no terminan la cadena resultante con un caracter nulo, el cual es necesario para poder imprimir una cadena con `printf`.

```
#include <ctype.h>
#include <stdio.h>
#include <string.h>

char* comprime(const char arr[], int tam, char* w) {
   for (int i = 0; i < tam; ++i) {
      int veces = 1;
      while (i + veces < tam && arr[i] == arr[i + veces]) {
         ++veces;
      }
      if (veces > 1) {
         int avance;
         sprintf(w, "%d%n", veces, &avance);
         w += avance;
      }
      *w++ = arr[i];
      i += veces - 1;
   }

   return w;
}

char* descomprime(const char arr[], int tam, char* w) {
   for (int i = 0; i < tam; ++i) {
      int veces = 1;
      if (isdigit(arr[i])) {
         int avance;
         sscanf(&arr[i], "%d%n", &veces, &avance);
         i += avance;
      }
      for (int j = 0; j < veces; ++j) {
         *w++ = arr[i];
      }
   }
   return w;
}

int main( ) {
```

```
   char c, cadena[10000 + 1], res[10000 + 1];
   scanf("%c%s", &c, &cadena[0]);
   auto rutina = (c == 'C' ? comprime : descomprime);
   *rutina(cadena, strlen(cadena), &res[0]) = '\0';
   printf("%s\n", res);
}
```

Soluciones de 15.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Cuenta-de-prefijos`.

Solución: Una forma de resolver este problema es adaptar la implementación de tries provista en las notas. Por una parte, hay que agregar el entero contador a cada nodo del trie; incrementaremos el contador de un nodo cada vez visitemos éste durante una inserción. Por otra parte, el problema sólo trata con cadenas alfabéticas no vacías, por lo que la cantidad máxima de hijos de un nodo es de 26 (+1, si es que también se usará el caracter nulo para marcar el fin de una cadena, lo cual realmente es innecesario en este problema). La búsqueda de un prefijo es similar a la búsqueda ordinaria, excepto que no debemos procesar el fin de cadena del prefijo.

```
#include <stdio.h>

struct nodo {
   nodo* hijos[26 + 1] = { };
   int cuenta = 0;
};

int mapa(char c) {
   return (c == '\0' ? 0 : c - 'a' + 1);
}

void inserta(nodo* actual, const char* p) {
   actual->cuenta += 1;    // necesario para la raíz si p pudiera ser vacía
   do {
      nodo*& bajar = actual->hijos[mapa(*p)];
      if (bajar == nullptr) {
         bajar = new nodo;
      }
      actual = bajar;
      actual->cuenta += 1;
   } while (*p++ != '\0');
}

int cuenta(const nodo* actual, const char* p) {
   while (*p != '\0') {
      const nodo* bajar = actual->hijos[mapa(*p++)];
      if (bajar == nullptr) {
         return 0;
      }
      actual = bajar;
   }
   return actual->cuenta;
}
```

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

   nodo raiz;
   for (int i = 0; i < n; ++i) {
      char cadena[10 + 1];
      scanf("%s", &cadena[0]);
      inserta(&raiz, cadena);
   }

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

   for (int i = 0; i < n; ++i) {
      char cadena[10 + 1];
      scanf("%s", &cadena[0]);
      printf("%d\n", cuenta(&raiz, cadena));
   }
}
```

Soluciones de 17.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Sobreescribiendo-cadenas-de-long`.

Solución: Este problema tiene dos objetivos: practicar el posicionamiento sobre archivos de escritura e ilustrar lo problemático que es trabajar con registros de longitud variable. Por las restricciones del problema, no podremos leer en memoria la base de datos completa (lo cual descarta de inmediato el poder modificarla en memoria). Deberemos leer la base de datos poco a poco para imprimirla en la salida y posteriormente deberemos modificar nuestra propia salida. Una dificultad del problema consiste en determinar la longitud y la posición inicial de cada cadena en la salida, pero una vez hecho esto, podemos usar dichas posiciones para usar `fseek` sobre `stdout`.

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

int main( ) {
   int inicios[25 + 1], cuantas = 0;
   for (char anterior = '|'; anterior != '\n';) {
      if (anterior == '|') {
         inicios[cuantas++] = ftell(stdin);
      }
      putchar(anterior = getchar( ));
   }

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

   for (int i = 0; i < n; ++i) {
      int id; char cadena[1000000 + 1];
      scanf("%d%s", &id, &cadena[0]);
```

```
      int tam = strlen(cadena);

      int tam_vieja = inicios[id + 1] - inicios[id] - 1;
      if (tam_vieja < tam) {
         std::fill(&cadena[0], &cadena[0] + tam_vieja, 'X');
      } else {
         std::fill(&cadena[0] + tam, &cadena[0] + tam_vieja, ' ');
      }
      fseek(stdout, inicios[id], SEEK_SET);
      fwrite(&cadena[0], sizeof(char), tam_vieja, stdout);
   }
}
```