Algoritmos y Programación II – Curso Buchwald
Guía de Ejercicios

Todas las guías cuentan con un ejercicio resuelto y una tanda de ejercicios propuestos. Los ejercicios propuestos están puntuados del 1 a 5 estrellas ($\star$) en dificultad (no de forma exacta, sino para que usen de guía).

Los ejercicios de 1 estrellas son ejercicios muy básicos o introductorios que no entrarían en un examen, pero sirven para iniciarse en el tema.

La mayoría de los ejercicios de entre 2 y 4 estrellas han sido sacado de parcialitos de cuatrimestres anteriores.

Los ejercicios de 5 estrellas sólo serían tomados en exámenes finales.

Se recomienda utilizar la versión web de esta guía, puesto que el formato está pensado para verse mejor allí.

Resumen de temas vistos en la materia:

Ejercicio resuelto

Implementar la *primitiva* de la pila `void** pila_multitop(const pila_t* pila, size_t n)`, que devuelve un arreglo de tamaño n con los n topes de la pila (los primeros n elementos si estos fueran desapilados), sin utilizar estructuras auxiliares. Completar el arreglo a devolver con NULL cuando el n recibido por parámetro sea mayor a la cantidad de elementos de la pila.

Indicar el orden de complejidad de la primitiva.

Solución

Algo que está implícito en el ejercicio es que se asume que la implementación del TDA Pila es sobre un arreglo dinámico, tal cual fue visto en clase. Siempre es válido asumir que la implementación es la misma a la que debieron implementar, salvo que el enunciado especifique lo contrario.

Dos cosas que es necesario entender desde el inicio del ejercicio:

1. En este ejercicio *no se puede* modificar la pila, pues el puntero recibido es de tipo *const*; ello quiere decir que la pila a la que apunta es de sólo lectura, y por tanto sería erróneo modificar cualquiera de sus campos. *Además, no se debe* modificarla pues, al ser una primitiva y tener acceso a los miembros internos de la estructura, no es necesario desapilar para acceder a los datos.
2. Se pide devolver un arreglo, pero es muy importante entender que debe ser un arreglo dinámico (creado con *malloc*). Sería un error **muy grave** devolver un arreglo estático, pues al terminar la ejecución de la función, la memoria de ese arreglo pasaría a ser inválida. Un error de este tipo anula el ejercicio por completo.

Una vez esto quede claro, la solución es bastante directa: si queremos que los elementos sean aquellos que desapilaríamos, simplemente tendríamos que iterar el arreglo en ese orden. Dada la implementación de la pila, deberíamos hacerlo de atrás hacia delante. Si fuera una pila enlazada, simplemente sería iterar por los nodos.

```c
void** pila_multitop(const pila_t* pila, size_t n) {
    void** topes = malloc(sizeof(void*) * n);
    if (!topes) {
        return NULL;
    }
    // Se podria hacer que i comience en k o pila->cantidad - 1, pero hay que
    // tener cuidado con esto, y con la condicion de corte: un size_t nunca
    // va a ser menor a 0
    for (size_t i = 0; i < n; i++) {
        if (i < pila->cantidad) {
            topes[i] = pila->datos[pila->cantidad - 1 - i];
        } else {
            topes[i] = NULL;
        }
```

```
    }
    return topes;
}
```

También sería válido hacerlo en dos iteraciones: una para llenar con los topes, y otra para rellenar con `NULL`s el resto. Así como también lo sería iterar hasta el más pequeño entre `n` y `pila->cantidad`, habiendo inicializado antes el arreglo con `NULL`s (ya sea iterando, o usando `calloc`). Otras tantas variantes también serían aceptadas, no hay una única forma de resolver este ejercicio.

Sobre la complejidad, sea cual sea el caso, vamos a estar llenando siempre el arreglo de n elementos con algo (datos de la pila, o `NULL`). Acceder a cada elemento de la pila, siendo que accedemos directamente, es $\mathcal{O}(1)$, y nunca vamos a ver más elementos de la pila si son más de n, por ende la primitiva es $\mathcal{O}(n)$. Es importante denotar que n en este caso no es la cantidad de elementos de la pila, sino la cantidad de elementos pedidos. Si quisiéramos hacer más clara la distinción, podríamos haber llamado a dicho parámetro con otro nombre. Tener cuidado con esto, porque si el parámetro tuviera otro nombre no sería correcto decir que es $\mathcal{O}(n)$, salvo que se aclare qué signifique n.

Ejercicios propuestos

1. ($\star$) Implementar el TDA Fracción. Dicho TDA debe tener las siguientes primitivas, cuya documentación puede encontrarse aquí: `fraccion_t* fraccion_crear(int numerador, int denominador);` `fraccion_t* fraccion_sumar(fraccion_t* f1, fraccion_t* f2);` `fraccion_t* fraccion_mul(fraccion_t* f1, fraccion_t* f2);` `char* fraccion_representacion(fraccion_t* fraccion);` `int fraccion_parte_entera(fraccion_t* fraccion);` `void fraccion_destruir(fraccion_t* fraccion);`

   Nota: considerar que se puede utilizar la función `sprintf` para generar la representación de la fracción. Por ejemplo: `sprintf(buffer, "%d/%d", num1, num2)`. Puede encontrarse la resolución de este ejercicio aquí.

2. ($\star$) Implementar el TDA NumeroComplejo. Dicho TDA debe tener las siguientes primitivas, cuya documentación puede encontrarse aquí: `complejo_t* complejo_crear(double real, double img);` `void complejo_multuplicar(complejo_t* c1, complejo_t* c2);` `void complejo_sumar(complejo_t* c1, complejo_t* c2);` `double complejo_obtener_imaginaria(const complejo_t* complejo);` `double complejo_obtener_real(const complejo_t* complejo);` `double complejo_obtener_modulo(const complejo_t* complejo);` `double complejo_obtener_angulo(const complejo_t* complejo);` `void complejo_destruir(complejo_t* complejo);`

   Nota: considerar que se puede utilizar la función `sprintf` para generar las representaciones.

3. ($\star$) Implementar una función que reciba un arreglo de `void*` e invierta su

orden, utilizando los TDAs vistos. Indicar y justificar el orden de ejecución.

4. (⋆) Mismo a lo anterior, pero que el arreglo sea de `int` (no de `int*`), utilizando los TDAs tal cual se los implementa en clase.

5. (⋆⋆) Implementar en C el TDA `ComposiciónFunciones` que emula la composición de funciones (i.e.`f(g(h(x)))`). Se debe definir la estructura del TDA, y las siguientes primitivas: `composicion_t* composicion_crear();` `void composicion_destruir(composicion_t*);` `bool composicion_agregar_funcion(comp double (*f)(double));` `double composicion_aplicar(composicion_t*, double);` Considerar que primero se irán agregando las funciones como se leen, pero tener en cuenta el correcto orden de aplicación. Por ejemplo: para emular `f(g(x))`, se debe hacer:

   ```
   composicion_agregar_funcion(composicion, f);
   composicion_agregar_funcion(composicion, g);
   composicion_aplicar(composicion, x);
   ```

   Indicar el orden de las primitivas.

6. (⋆⋆⋆) Dada una lista enlazada implementada con las siguientes estructuras:

   ```
   typedef struct nodo_lista {
       struct nodo_lista* prox;
       void* dato;
   } nodo_lista_t;

   typedef struct lista {
       nodo_lista_t* prim;
   } lista_t;
   ```

   Escribir una primitiva que reciba una lista y devuelva el elemento que esté a k posiciones del final (el ante-k-último), recorriendo la lista una sola vez y sin usar estructuras auxiliares. Considerar que k es siempre menor al largo de la lista. Por ejemplo, si se recibe la lista `[ 1, 5, 10, 3, 6, 8 ]`, y `k = 4`, debe devolver 10. Indicar el orden de complejidad de la primitiva.

7. (⋆⋆⋆) Dada una pila de enteros, escribir una función que determine si sus elementos están ordenados de manera ascendente. Una pila de enteros está ordenada de manera ascendente si, en el sentido que va desde el tope de la pila hacia el resto de elementos, cada elemento es menor al elemento que le sigue. La pila debe quedar en el mismo estado que al invocarse la función. Indicar y justificar el orden del algoritmo propuesto.

8. (⋆⋆) Implementar la primitiva `void** cola_multiprimeros(const cola_t* cola, size_t k)` que dada una cola y un número k, devuelva los primeros k elementos de la cola, en el mismo orden en el que habrían salido de la cola. En caso que la cola tenga menos de k elementos, rellenar con NULL. Indicar y justificar el orden de ejecución del algoritmo.

9. (⋆⋆) Implementar la función `void** cola_multiprimeros(cola_t* cola,`

`size_t k)` con el mismo comportamiento de la primitiva anterior.

10. (⋆⋆) Implementar en C una primitiva `void lista_invertir(lista_t* lista)` que invierta la lista recibida por parámetro, sin utilizar estructuras auxiliares. Indicar y justificar el orden de la primitiva.

11. (⋆⋆) Se quiere implementar un TDA ColaAcotada sobre un arreglo. Dicho TDA tiene un espacio para k elementos (que se recibe por parámetro al crear la estructura). Explicar cómo deberían implementarse las primitivas encolar y desencolar de tal manera que siempre sean operaciones de tiempo constante.

12. (⋆⋆⋆⋆) Implementar una función que ordene de manera ascendente una pila de enteros sin conocer su estructura interna y utilizando como estructura auxiliar sólo otra pila auxiliar. Por ejemplo, la pila `[ 4, 1, 5, 2, 3 ]` debe quedar como `[ 1, 2, 3, 4, 5 ]` (siendo el último elemento el tope de la pila, en ambos casos). Indicar y justificar el orden de la función.

13. (⋆⋆) Implementar una función `void cola_filtrar(cola_t* cola, bool (*filtro)(void*))`, que elimine los elementos encolados para los cuales la función *filtro* devuelve `false`. Aquellos elementos que no son eliminados deben permanecer en el mismo orden en el que estaban antes de invocar a la función. No es necesario destruir los elementos que sí fueron eliminados. Se pueden utilizar las estructuras auxiliares que se consideren necesarias y no está permitido acceder a la estructura interna de la cola (es una función). ¿Cuál es el orden del algoritmo implementado?

14. (⋆⋆⋆) Sabiendo que la firma del iterador interno de la lista enlazada es:

    ```
    void lista_iterar(lista_t* lista,
                      bool (*visitar)(void* dato, void* extra),
                      void* extra);
    ```

    Se tiene una lista en donde todos los elementos son punteros a números enteros. Implementar la función *visitar* para que calcule la suma de todos los números pares. Mostrar, además, una invocación completa a `lista_iterar()` que haga uso del *visitar* implementado.

15. (⋆⋆⋆⋆⋆) Diseñar un TDA `PilaConMáximo`, que tenga las mismas primitivas de la pila convencional (en este caso, sólo para números), y además permita obtener el máximo de la pila. **Todas** las primitivas deben funcionar en $\mathcal{O}(1)$. Explicar cómo implementarías el TDA para que cumpla con todas las restricciones.

Ejercicio resuelto

Implementar un algoritmo en C que reciba un arreglo de enteros de tamaño n, ordenado ascendentemente y sin elementos repetidos, y determine en $\mathcal{O}(\log n)$ si es mágico. Un arreglo es mágico si existe algún valor i tal que $0 \leq i < n$ y `arr[i] = i`. Justificar el orden del algoritmo.

Ejemplos:

- `A = [ -3, 0, 1, 3, 7, 9 ]` es mágico porque `A[3] = 3`.
- `B = [ 1, 2, 4, 6, 7, 9 ]` no es mágico porque `B[i] != i` para todo `i`.

Solución

Como es mencionado en clase, al ver que se nos pide:

- Un algoritmo de división y conquista,
- Un orden $\mathcal{O}(\log n)$

Si no se nos pidiera que sea de división y conquista, y más aún que su orden sea logarítimico, podríamos simplemente ir elemento por elemento chequeando si se cumple la condición:

```
bool arreglo_es_magico_lineal(int arr[], size_t n) {
    for (size_t i = 0; i < n; i++) {
        if (arr[i] == i)
            return true;
    }
    return false;
}
```

Por supuesto, esta sería la solución *trivial*, casi con nulo esfuerzo de pensar y aprovechar la cualidad de que el arreglo se encuentre ordenado y sin repetidos. Se podría hasta implementar con división y conquista, pero no dejará de ser $\mathcal{O}(n)$. Como en otros problemas, vamos a buscar aprovechar las precondiciones que nos dan para el arreglo.

Lo primero que tenemos que pensar es en el algoritmo *estrella* de división y conquista que tiene ese orden: Búsqueda Binaria. Al ver que se nos pide eso, seguramente nuestro algoritmo no sea muy distinto al de búsqueda binaria (o bien, que la forma de la función va a ser similar). Pero para poder aplicar un algoritmo así, necesariamente tenemos que poder desechar toda una proporción del problema original en cada iteración. Ya vemos que tenemos la condición de que el arreglo se encuentra ordenado ascendentemente, y no cuenta con repetidos. Veremos si esa última restricción es realmente necesaria, pero al menos con la primera ya contamos con una restricción bastante fuerte.

Lo primero a pensar es el caso base: si se nos da vuelta el inicio y fin, significa que nunca nos topamos con un índice que cumpla la condición (no siempre va a ser este nuestro caso base, ojo).

Ahora, lo crucial: pensar la condición de éxito. Acá es donde analizamos qué resuelve nuestro algoritmo. Vamos al medio, y lo que tenemos que verificar es si `arr[medio] == medio`. Si eso sucede, ¡éxito! Entonces, por ahora tenemos:

```
bool arreglo_es_magico(int arr[], size_t n) {
    return _arreglo_es_magico(arr, 0, n - 1);
}

bool _arreglo_es_magico(int arr[], size_t inicio, size_t fin) {
    if (inicio > fin) {
        return false;
    }

    size_t medio = (inicio + fin) / 2;
    if (arr[medio] == medio) {
        return true;
    }
    // nos falta el caso sin exito
}
```

Ahora, pensemos que eso no sucede. Necesitamos quedarnos con una sola de las mitades, lo cual implica descartar la otra. Pero ¿cómo descartamos una mitad? Bueno, podemos ver qué pasó con `arr[medio]` que no cumple con la condición de éxito. ¿Qué sucede si `arr[medio] < medio`? ¿Puede suceder que algún elemento anterior sí cumpla la condición? **¡NO!**, porque al no poder haber repetidos, si `arr[medio] < medio`, entonces `arr[medio - 1] < medio - 1`, y tambien para todos los anteriores. Por eso, podemos simplemente ver de la mitad en adelante, descartando la primera mitad. Ahí vemos que la condición extra era necesaria; si no, no podríamos descartar la primera mitad. Podemos hacer el mismo análisis al revés, vamos a ver que si `arr[medio] > medio`, no puede ser que se cumpla la condición para los elementos siguientes.

Luego de este análisis, podemos escribir el código:

```
bool arreglo_es_magico(int arr[], size_t n) {
    return _arreglo_es_magico(arr, 0, n - 1);
}

bool _arreglo_es_magico(int arr[], size_t inicio, size_t fin) {
    if (inicio > fin) {
        return false;
    }

    size_t medio = (inicio + fin) / 2;
    if (arr[medio] == medio) {
        return true;
    }
    if (arr[medio] < medio) {
```

```
        return _arreglo_es_magico(arr, medio + 1, fin);
    } else {
        return _arreglo_es_magico(arr, inicio, medio - 1);
    }
}
```

Nota: El algoritmo puede, en vez de resolverse con una función auxiliar que haga la recursión, usando directamente `arreglo_es_magico`, usando aritmética de punteros. Quien prefiera hacerlo así, bienvenido es, aunque nos parece más claro para la explicación (y más directo) el código propuesto.

Demostración del orden Veamos la ecuación de recurrencia. Hay escritos dos llamados recursivos, cada uno se invoca con la mitad del problema (mitad izquierda o derecha), y todo lo que cuesta *partir y juntar* no son más que algunas operaciones básicas de tiempo constante. Lo importante es que si bien hay escritos dos llamados recursivos, nunca se llamará a ambos. Siempre se llamará bien para el lado izquierdo, bien para el derecho. Por lo tanto, la ecuación de recurrencia quedará:

$\mathcal{T}(n) = \mathcal{T}\left(\frac{n}{2}\right) + \mathcal{O}(1)$

Es, evidentemente, igual a la de Búsqueda Binaria, por lo que tendrá el mismo orden, pero aplicamos el Teorema Maestro para corroborar:

$A = 1; B = 2; C = 0 \rightarrow \log_B(A) = \log_2(1) = 0 = C$

Caemos en el caso de $\log_B(A) = C$, por lo que el orden del algoritmo será $\mathcal{O}(n^C \log n) = \mathcal{O}(\log n)$

Ejercicios propuestos

1. ($\star$) Explicar por qué el siguiente siguiente código **no** es de división y conquista.

```
// Algoritmo ¿por D&C? para obtener el máximo de un arreglo
int maximo(int* arreglo, size_t n) {
    if (n == 1) {
        return arreglo[0];
    }
    int max_restante = maximo(arreglo, n - 1);
    return arreglo[n - 1] > max_restante ? arreglo[n - 1] : max_restante;
}
```

2. ($\star$) Explicar por qué el siguiente siguiente código **no** es de división y conquista.

```
// Algoritmo ¿por D&C? para obtener el máximo de un arreglo
int maximo(int* arreglo, size_t n) {
    size_t medio = n / 2;
    int max_izquierda = _maximo(arreglo, 0, medio);
    int max_derecha = _maximo(arreglo, medio + 1, n - 1);
    return max_izquierda > max_derecha ? max_izquierda : max_derecha;
}
```

```c
int _maximo(int* arreglo, size_t inicio, size_t fin) {
    int max = arreglo[inicio];
    for (size_t i = inicio + 1; i <= fin; i++) {
        if (max < arreglo[i]) {
            max = arreglo[i];
        }
    }
    return max;
}
```

3. (⋆⋆) Indicar la complejidad del siguiente algoritmo, utilizando el teorema maestro:

```c
// Busca un elemento usando D&C. El arreglo se encuentra ordenado.
bool elemento_esta(int* arreglo, size_t inicio, size_t fin, int elem) {
    if (inicio > fin) return false;
    size_t medio = (fin + inicio) / 2;
    if (arreglo[medio] == elem) {
        return true;
    }
    if (arreglo[medio] < elem) {
        return elemento_esta(arreglo, medio + 1, fin, elem);
    }

    for (size_t i = medio - 1; i > inicio - 1; i--) {
        if (arreglo[i] == elem) return true;
    }
    return false;
}
```

4. (⋆) Hacerle el seguimiento al siguiente algoritmo:

```c
void imprimir_dyc(int m) {
    if (m < 4) return;
    printf("%d\n", m);
    imprimir_dyc(m / 4);
    imprimir_dyc(m - (m / 4));
}
```

5. (⋆) Indicar, utilizando el Teorema Maestro, la complejidad del ejercicio anterior.

6. (⋆⋆) Indicar cuál es la complejidad de un algoritmo cuya ecuación de recurrencia es: $\mathcal{T} = 2\mathcal{T}\left(\frac{2}{3}n\right) + \mathcal{O}\left(\sqrt{n}\right)$.

7. (⋆⋆) Nos dan para elegir entre los siguientes 3 algoritmos para solucionar el mismo problema. ¿Cuál elegirías? Justificar calculando el orden de los algoritmos.

    1. El algoritmo `A` resuelve el problema dividiéndolo en 5 subproblemas de la

mitad del tamaño, resolviendo cada subproblema de forma recursiva, y combinando las soluciones en tiempo lineal.
2. El algoritmo `B` resuelve el problema (de tamaño n) dividiéndolo en 9 subproblemas de tamaño $\frac{n}{3}$, resolviendo cada subproblema de forma recursiva y combinando las soluciones en tiempo cuadrático de n.
3. El algoritmo `C` resuelve el problema (de tamaño n) eligiendo un subproblema de tamaño $n-1$ en tiempo $\mathcal{O}(n)$, y luego resolviendo recursivamente ese subproblema.

8. ($\star\star$) Implementar, por división y conquista, una función que determine el mínimo de un arreglo. Indicar y justificar el orden.

9. ($\star\star$) Implementar, por división y conquista, una función que dado un arreglo y su largo, determine si el mismo se encuentra ordenado. Indicar y justificar el orden.

10. ($\star\star$) Implementar, por división y conquista, una función que dado un arreglo sin elementos repetidos y *casi ordenado* (todos los elementos se encuentran ordenados, salvo uno), obtenga el elemento fuera de lugar. Indicar y justificar el orden.

11. ($\star\star\star$) Se tiene un arreglo tal que `[1, 1, 1, ..., 0, 0, ...]` (es decir, *unos seguidos de ceros*). Se pide:

    1. una función de orden $\mathcal{O}(\log n)$ que encuentre el índice del primer 0. Si no hay ningún 0 (solo hay unos), debe devolver -1.
    2. demostrar con el Teorema Maestro que la función es, en efecto, $\mathcal{O}(\log n)$.

    Ejemplos:

    ```
    [1, 1, 0, 0, 0] → 2
    [0, 0, 0, 0, 0] → 0
    [1, 1, 1, 1, 1] → -1
    ```

12. ($\star\star\star\star$) Implementar un algoritmo que, por división y conquista, permita obtener la parte entera de la raíz cuadrada de un número n, en tiempo $\mathcal{O}(\log n)$. Por ejemplo, para $n = 10$ debe devolver 3, y para $n = 25$ debe devolver 5.

13. ($\star\star\star$) Se tiene un arreglo de $N >= 3$ elementos en forma de pico, esto es: estrictamente creciente hasta una determinada posición p, y estrictamente decreciente a partir de ella (con $0 < p < N-1$). Por ejemplo, en el arreglo `[1, 2, 3, 1, 0, -2]` la posición del pico es $p = 2$. Se pide:

    1. Implementar un algoritmo de división y conquista de orden $\mathcal{O}(\log n)$ que encuentre la posición p del pico: `size_t posicion_pico(int v[], size_t ini, size_t fin);`. La función será invocará inicialmente como: `posicion_pico(v, 0, n-1)`, y tiene como pre-condición que el arreglo tenga forma de pico.
    2. Justificar el orden del algoritmo mediante el teorema maestro.

14. ($\star\star$) Se quiere implementar MergeSort pero, en vez de dividir en dos partes el arreglo, dividirlo en tres, llamando recursivamente al algoritmo para cada una

de las partes y luego uniéndolas.

1. Suponiendo que el merge de las tres partes se realiza en tiempo lineal, calcular el orden del algoritmo.
2. Si en vez de dividir en tres partes, se dividiera el arreglo en n, siendo n la cantidad de elementos del arreglo ¿a qué otro algoritmo de ordenamiento se asemeja esta implementación? ¿Cuál es el orden de dicho algoritmo?
3. Dado lo obtenido en los puntos anteriores ¿tiene sentido implementar MergeSort con k separaciones, para $k > 2$?

15. ($\star\star\star$) Un algoritmo sencillo para multiplicar matrices de $n \times n$ demora $\mathcal{O}(n^3)$. El algoritmo de Strassen (que utiliza División y Conquista) lo hace en $\mathcal{O}(n^{\log_2 7})$. La profesora Manterola quiere implementar un algoritmo de División y Conquista que sea aún más veloz, donde divida al problema en `A` subproblemas de tamaño de $\frac{n}{4}$, y que juntar las soluciones parciales sea $\mathcal{O}(n^2)$. ¿Cuál es el máximo `A` para que el orden del algoritmo sea menor que el del algoritmo de Strassen? Justificar.

16. ($\star\star\star\star$) En Bytelandia tienen un sistema monetario extraño: su divisa es el bytelandés, y existen denominaciones en moneda física... ¡para todos los valores enteros! (Esto es, se emiten monedas de 1, 2, 3, ..., 14, 15, ..., 28, 29, 30, ... bytelandeses). No solo eso, *cualquier* moneda de valor n puede cambiarse en el banco por 3 monedas de menor denominación, de valores $\lfloor \frac{n}{2} \rfloor$, $\lfloor \frac{n}{3} \rfloor$ y $\lfloor \frac{n}{4} \rfloor$ respectivamente (no existe denominación de 0 pesos, por lo cual es posible recibir menos de 3 monedas en el cambio); una vez hecho el cambio, sin embargo, no pueden cambiarse de vuelta por la moneda mayor (e.g., no pueden cambiase una moneda de 2 y otra de 3 por una de 5). Finalmente, también es posible intercambiar bytelandeses por pesos argentinos, a una tasa de 1 a 1. Se pide implementar un algoritmo que, utilizando **división y conquista**, reciba el valor n de una moneda en bytelandés y devuelva la cantidad máxima de pesos argentinos que se podría obtener realizando los intercambios arriba mencionados. Por ejemplo, si $n = 12$, la cantidad máxima de pesos que se puede obtener es \$13, ya que podemos dividir la moneda de 12 bytelandeses en monedas de 6, 4 y 3; y \$6 + \$4 + \$3 = \$13 (se podría seguir dividiendo, pero en este caso no conviene dividir ninguna de esas monedas resultantes). Justificar el orden del algoritmo implementado.

17. ($\star\star\star\star\star$) Implementar una función (que utilice división y conquista) de orden $\mathcal{O}(n \log n)$ que dado un arreglo de n números enteros devuelva `true` o `false` según si existe algún elemento que aparezca más de la mitad de las veces. Justificar el orden de la solución. Ejemplos:

```
[1, 2, 1, 2, 3] -> false
[1, 1, 2, 3] -> false
[1, 2, 3, 1, 1, 1] -> true
[1] -> true
```

Aclaración: Este ejercicio puede resolverse, casi trivialmente, ordenando el arreglo con un algoritmo eficiente, o incluso se puede realizar más rápido

utilizando una tabla de hash. Para hacer interesante el ejercicio, resolver **sin ordenar el arreglo**, sino puramente división y conquista.

Ejercicio resuelto

Se tiene un arreglo de cadenas que representan fechas de la forma `YYYYMMDD` (ej: `20110626` representa el 26/06/2011). Implementar un algoritmo lineal que las ordene de forma creciente.

Solución

Dado que todos los elementos van a ser fechas en un formato fijo, de mismo largo para cada uno, y bien definido qué es cada dígito, podemos aplicar Radix Sort. Ahora bien, tenemos dos formas:

1. Vamos de cifra menos significativa a más significativa, así como está definido, aplicando el ordenamiento interno. Esto sería, aplicar en cada una de las posiciones según el orden: 7, 6, 5, 4, 3, 2, 1, 0. No hay muchas más vueltas.
2. Considerar que no es necesario trabajar en base 10. Sabemos que si o si los días están en el rango 1-31. Los meses en el rango 1-12. Luego, en los años podríamos no tener esto acotado, y no conviene trabajar con un rango 1-9999 porque puede ser más grande que la cantidad de elementos a ordenar. Pero cuanto menos, nos ahorramos dos pasadas.

Algunas preguntas:

- ¿Cuál es la mejor opción? La segunda, claramente. No sólo por hacerlo más rápido, sino por mostrar un entendimiento tanto del problema como del algoritmo.
- ¿La otra opción está mal? No.Pero considerar que si nos piden un seguimiento, si no consideramos estas cosas vamos a demorar más. Y es tiempo que se nos consume del parcialito. Conviene más pensar primero, y luego ponerse a aplicar, que aplicar de entrada. Se termina ganando más tiempo. Ahora bien, si además es una cuestión de implementación, podría restar un poco no haberlo pensado de la mejor forma (esta es una aclaración general que puede depender de lo evidente que sea la mejora según el caso particular).

Aclaración: en general este tipo de ejercicios pueden directamente implementarse en pseudo código, pero aquí proponemos una solución en C para que puedan ver una resolución completa en dicho lenguaje.

Aplicamos la primer solución, que es más corta:

```
void ordenar_fechas(char** fechas, size_t n) {
    for (size_t i = 0; i < 8; i++) {
        ordenar_por_digito(fechas, 7 - i, n);
    }
}

void ordenar_por_digito(char** fechas, size_t digito, size_t n) {
    lista_t* digito[10];
    for (size_t i = 0; i < 10; i++) digito[i] = lista_crear();
```

```
    for (size_t i = 0; i < n; i++) {
        int valor = char_a_int(fecha[i][digito]);
        lista_insertar_ultimo(digito[valor], fecha);
    }

    size_t indice = 0;
    for (size_t i = 0; i < 31; i++) {
        while(!lista_esta_vacia(digito[i])) {
            fechas[indice++] = lista_borrar_primero(digito[i]);
        }
        lista_destruir(digito[i], NULL);
    }
}
```

Aplicamos la segunda solución, que es un tanto más larga, pero más rápida:

```
void ordenar_fechas(char** fechas, size_t n) {
    ordenar_por_dia(fechas, n);
    ordenar_por_mes(fechas, n);
    ordenar_por_anio(fechas, n);
}

int char_a_int(char v) {
    return a - '0';
}

void ordenar_por_dia(char** fechas, size_t n) {
    lista_t* dias[31];
    for (size_t i = 0; i < 31; i++) dias[i] = lista_crear();

    for (size_t i = 0; i < n; i++) {
        int dia = char_a_int(fecha[i][6]) * 10 + char_a_int(fecha[i][7]) - 1;
        lista_insertar_ultimo(dias[dia], fecha);
    }

    size_t indice = 0;
    for (size_t i = 0; i < 31; i++) {
        while(!lista_esta_vacia(dias[i])) {
            fechas[indice++] = lista_borrar_primero(dias[i]);
        }
        lista_destruir(dias[i], NULL);
    }
}

void ordenar_por_mes(char** fechas, size_t n) {
    lista_t* meses[12];
```

```
    for (size_t i = 0; i < 12; i++) meses[i] = lista_crear();

    for (size_t i = 0; i < n; i++) {
        int mes = char_a_int(fecha[i][4]) * 10 + char_a_int(fecha[i][5]) - 1;
        lista_insertar_ultimo(meses[mes], fecha);
    }

    size_t indice = 0;
    for (size_t i = 0; i < 31; i++) {
        while(!lista_esta_vacia(meses[i])) {
            fechas[indice++] = lista_borrar_primero(meses[i]);
        }
        lista_destruir(meses[i], NULL);
    }
}

void ordenar_por_anio(char** fechas, size_t n) {
    for (size_t i = 0; i < 4; i++) {
        ordenar_por_digito(fechas, 3 - i, n);
    }
}
```

Ahora vemos de mejorar un poco el código:

```
void ordenar_fechas(char** fechas, size_t n) {
    ordenar_por_dia(fechas, n);
    ordenar_por_mes(fechas, n);
    ordenar_por_anio(fechas, n);
}

int char_a_int(char v) {
    return a - '0';
}

void ordenar_por_dos_digitos(char** fechas, size_t n, size_t digito_ini,
                             size_t max) {
    lista_t* dias[max];
    for (size_t i = 0; i < max; i++) dias[i] = lista_crear();

    for (size_t i = 0; i < n; i++) {
        int dia = char_a_int(fecha[i][digito_ini]) * 10
                + char_a_int(fecha[i][digito_ini + 1]) - 1;
        lista_insertar_ultimo(dias[dia], fecha);
    }

    size_t indice = 0;
    for (size_t i = 0; i < max; i++) {
```

```
        while(!lista_esta_vacia(dias[i])) {
            fechas[indice++] = lista_borrar_primero(dias[i]);
        }
        lista_destruir(dias[i], NULL);
    }
}


void ordenar_por_dia(char** fechas, size_t n) {
    ordenar_por_dos_digitos(fechas, n, 6, 31);
}

void ordenar_por_mes(char** fechas, size_t n) {
    ordenar_por_dos_digitos(fechas, n, 4, 12);
}
```

Ejercicios propuestos

1. (⋆) Realizar un seguimiento de ordenar el siguiente arreglo utilizando Inserción, Selección, MergeSort, QuickSort y HeapSort. Contar la cantidad de operaciones (aproximadamente) para validar empíricamente la diferencia en el orden de cada uno, y poder comparar aquellos que sean iguales: `[1, 7, 5, 8, 2, 4, 9, 6, 5]`.

2. (⋆) Implementar un algoritmo que permita ordenar de menor a mayor en $\mathcal{O}(n)$ un arreglo *casi* ordenado de mayor a menor.

3. (⋆) Se tiene un arreglo de números, como el del primer ejercicio (no necesariamente ese). Indicar cuál sería un algoritmo eficiente para ordenar dicho arreglo. Aquí la solución del ejercicio.

4. (⋆⋆) Se tiene un arreglo de estructuras de la forma `struct {long anio, char* evento}`, que indica el año y evento de un hecho definido a lo largo de la historia de la Tierra. Indicar y justificar cuál sería un algoritmo de ordenamiento apropiado para utilizar para ordenar dicho arreglo por año. Indicar también, si en vez de ordenar por año se decide ordenar por evento (lexicográficamente). Si se quiere ordenar por año y dentro de cada año, por evento: ¿Deben utilizarse para ambos campos el mismo algoritmo de ordenamiento? ¿Que característica/s deben cumplir dicho o dichos algoritmos para que quede ordenado como se desea? ¿En qué orden deben aplicarse los ordenamientos?

5. (⋆⋆) Hacer el seguimiento de counting sort para ordenar por año las siguientes obras:

   ```
   1988 - Crónicas del Ángel Gris
   2000 - Los Días del Venado
   1995 - Alta Fidelidad
   1987 - Tokio Blues
   2005 - En Picada
   ```

```
1995 - Crónica del Pájaro que Da Cuerda al Mundo
1995 - Ensayo Sobre la Ceguera
2005 - Los Hombres que No Amaban a las Mujeres
```

¿Cuál es el orden del algoritmo? ¿Qué sucede con el orden de los elementos de un mismo año, respecto al orden inicial, luego de finalizado el algoritmo? Justificar brevemente.

6. (⋆⋆) Realizar el seguimiento de ordenar por Radix Sort el siguiente arreglo de cadenas que representan versiones. Cada versión tiene el formato `a.b.c`, donde cada valor `a`, `b` y `c` puede tener un valor entre 0 y 99. Considerar que se quiere que quede ordenado primero por la primera componente (`a`), luego por la segunda (`b`) y finalmente por la tercera (`c`). Tener en cuenta que, por ejemplo `1.1.3 < 1.1.20`, $2.20.8 < 3.0.0$.

   ```
   ["4.3.2", "5.1.2", "10.1.4", "2.1.20", "2.2.1", "4.2.3",
   "2.1.5", "8.1.2", "5.30.1", "10.0.23"]
   ```

7. (⋆⋆) Indicar Verdadero o Falso, justificando de forma concisa en cualquier caso.

   a. Podríamos mejorar el orden de complejidad de QuickSort si contaramos con más información sobre cómo son los datos a ordenar.

   b. No siempre conviene utilizar Counting Sort para ordenar un arreglo de números enteros.

   c. Que un algoritmo de ordenamientos sea estable implica que el algoritmo ordena sobre el arreglo original (sin utilizar otro adicional). Por ejemplo, Counting Sort no es estable.

8. (⋆⋆⋆⋆) Se quiere ordenar un arreglo de películas por su género. No se conoce cuántos, ni cuáles son estos géneros, pero se sabe que son muy pocos comparando con la cantidad de películas a ordenar. Diseñar un algoritmo que permita ordenar las películas en tiempo lineal de la cantidad de películas y explique ćomo funcionaría sobre las siguientes películas:

   - Donnie Darko (2001): Thriller psicológico
   - Juno (2007): Coming of age
   - The Shining (1980): Thriller psicológico
   - Labyrinth (1986): Fantasía
   - Ferris Bueller's Day Off (1986): Coming of age

9. (⋆⋆⋆) Se tiene una larga lista de números de tres cifras `abc` que representan números en notación científica de la forma: $a,b \cdot 10^c$. Por ejemplo 123 representaría el número $1,2 \cdot 10^3$.

   1. Diseñe un algoritmo para ordenar los números según su valor en notación científica. ¿De qué orden es?

   2. Muestre cómo se ordena la siguiente lista de números con el algoritmo que diseñó:

      $[122, 369, 332, 180, 486, 349, 326, 101]$

que representan

$[1,2 \cdot 10^2; 3,6 \cdot 10^9; 3,3 \cdot 10^2; 1,8 \cdot 10^0; 4,8 \cdot 10^6; 3,4 \cdot 10^9; 3,2 \cdot 10^6; 1,0 \cdot 10^1]$,

y equivalen a $[120; 3600000000; 330; 1,8; 4800000; 3400000000; 3200000; 10]$.

10. ($\star\star\star\star$) Suponer que se tiene un arreglo de n elementos ordenados, seguido de $f(n)$ elementos desordenados. Cómo ordenarías el arreglo según si $f(n)$ es:
    a. $f(n) = \mathcal{O}(1)$
    b. $f(n) = \mathcal{O}(\log n)$
    c. $f(n) = \mathcal{O}\left(\sqrt{n}\right)$
11. ($\star\star\star\star\star$) Implementar un algoritmo que, dado un arreglo de n números enteros cuyos valores van de 0 a K (constante conocida), procese dichos números en tiempo $\mathcal{O}(n + K)$, devuelva alguna estructura que permita consultar cuántos valores ingresados están en el intervalo `(A, B)`, en tiempo $\mathcal{O}(1)$. Explicar cómo se usaría dicha estructura para poder realizar tales consultas.

Ejercicio resuelto

1. Para un hash cerrado, implementar una primitiva `lista_t*hash_claves(const hash_t*)` que reciba un hash y devuelva una lista con sus claves.
2. Repetir lo mismo para un hash abierto.
3. Implementar de nuevo la operación, en este caso como una función.

Solución

Partiendo del punto *a)*, es importante notar que no podemos utilizar el iterador externo, ya que se trata de una primitiva para el hash (y es una mala práctica que, dado que el iterador dependa del hash, ahora hagamos que el hash dependa del iterador).

Hash Cerrado Para este punto, sólo es necesario iterar campo por campo, considerando únicamente aquellos que están ocupados.

```
lista_t* hash_claves(const hash_t* hash) {
    lista_t* claves = lista_crear();
    if (!claves) {
        return NULL;
    }
    for (size_t i = 0; i < hash->tam; i++) {
        // valor de un enum definido para el hash
        if (hash->tabla[i].estado == OCUPADO) {
            lista_insertar_ultimo(claves, hash->tabla[i].clave);
        }
    }
    return claves;
}
```

Es importante notar que en este ejercicio se está evaluando que sabemos trabajar internamente con el hash cerrado. Los puntos importantes:

- Sabemos cuáles son los campos de la estructura.
- Sabemos que la tabla es de tipo `campo_t*`, y no `campo_t**`, puesto que es completamente innecesario un segundo grado de indirección (como se analiza en la respectiva clase práctica).
- Entendemos que el estado correcto a considerar es el de `OCUPADO`, que es un enumerativo. Cuanto mucho, una constante. Definitivamente no un `"ocupado"`.

Hash Abierto Para este caso, consideramos todas las listas, las cuales podemos ir iterando utilizando el iterador externo o interno. Aquí mostramos una implementación utilizando el iterador interno, no porque sea mejor implementación, sino para que tengan un ejemplo de uso.

```c
lista_t* hash_claves(const hash_t* hash) {
   lista_t* claves = lista_crear();
    if (!claves) {
        return NULL;
    }
    for (size_t i = 0; i < hash->tam; i++) {
        lista_iterar(hash->tabla[i], insertar_clave, claves);
    }
    return claves;
}

bool insertar_clave(void* dato, void* extra) {
    lista_t* claves = extra;
    par_clave_valor_t* par = dato;
    lista_insertar_ultimo(claves, par->clave);
    return true;
}
```

Aclaración: Hay quienes deciden implementar el Hash Abierto de tal forma que no tenga listas vacías (esto es, si una posición aún no ha sido utilizada, entonces se guarda `NULL` y se crea la lista cuando sea necesaria). Esto es totalmente válido, y en todo caso con aclararlo en algún lado es suficiente (pero debe hacerse la validación contra `NULL` en ese caso).

Función En este caso, no sólo sí está permitido utilizar el iterador externo del hash, sino que **no nos queda otra opción**, dado que no es posible acceder a los campos internos de la estructura. Además, en particular no se nos dice cuál es la implementación, cosa que no es necesario conocer.

Entonces, simplemente iteramos utilizando el iterador externo y guardamos en una lista.

```c
lista_t* hash_claves(const hash_t* hash) {
   lista_t* claves = lista_crear();
    if (!claves) {
        return NULL;
    }
    hash_iter_t* iter = hash_iter_crear(iter);
    if (!iter) {
        lista_destruir(claves);
        return NULL;
    }
    while (!hash_iter_al_final(iter)) {
        char *clave = hash_iter_ver_actual(iter);
        lista_insertar_ultimo(claves, clave);
        hash_iter_avanzar(iter);
    }
    hash_iter_destruir(iter);
```

```
    return claves;
}
```

Ejercicios propuestos

1. (⋆) Suponer que se tiene un hash cerrado que se redimensiona cuando el factor de carga llega a 0.75, y que no se tienen en cuenta los elementos borrados a la hora de calcular el factor de carga.

    1. Describir, en términos generales, el peor escenario posible para esta implementación.

    2. Dado un hash de estas características, con capacidad inicial 100, calcular el número máximo de casillas que podría llegar a visitar `hash_obtener()` si la cantidad actual de elementos en el hash es 1, y no se realizó ningúna redimensión, pero sí se insertaron y borraron elementos. (En otras palabras, poner una cota superior al caso peor de este hash.)

2. (⋆) ¿Para qué casos la función `hash_obtener()` tiene una complejidad peor que $\mathcal{O}(1)$? Explicar tanto para el hash abierto, como el cerrado.

3. (⋆) Justificar si la siguiente función de hashing es correcta o no: `cpp size_t calcular_hash(char *clave, size_t largo) { // rand() devuelve un numero entero positivo aleatorio return rand() % largo; }`

4. (⋆⋆) a. Mostrar el resultado de las siguientes operaciones tanto para un hash cerrado como para un hash abierto, ambos de capacidad 9 e inicialmente vacíos (los números son también el resultado de la función de hashing): insertar 17, insertar 22, insertar 35, borrar 17, insertar 52, insertar 54.

    b. Tras estas inserciones ¿qué pasos hay que seguir para verificar si el 70 pertenece al hash?

    c. Posteriormente se realizan más inserciones. ¿Cuándo redimensionaría cada hash? ¿Qué pasos hay que seguir para hacerlo?

5. (⋆⋆) Implementar una función de orden $\mathcal{O}(n)$ que dado un arreglo de n números enteros devuelva `true` o `false` según si existe algún elemento que aparezca más de la mitad de las veces. Justificar el orden de la solución. Ejemplos:

    ```
    [1, 2, 1, 2, 3] -> false
    [1, 1, 2, 3] -> false
    [1, 2, 3, 1, 1, 1] -> true
    [1] -> true
    ```

6. (⋆⋆) Asumiendo que se tiene disponible una implementación completa del TDA Hash, se desea implementar una función que decida si dos Hash dados representan o no el mismo Diccionario. Considere para la solución que es de interés la mejor eficiencia temporal posible. Indique, para su solución, eficiencia en tiempo y espacio. Nota: Dos tablas de hash representan el mismo diccionario

si tienen la misma cantidad de elementos; todas las claves del primero están en el segundo; todas las del segundo, en el primero; y los datos asociados a cada una de esas claves son iguales (se pueden comparar los valores con "==").

7. (⋆⋆) Implementar el TDA MultiConjunto. Este es un Conjunto que permite más de una aparición de un elemento, por lo que eliminando una aparición, el elemento puede seguir perteneciendo. Dicho TDA debe tener como primitivas:

   - `multiconj_t* multiconj_crear()`: crea un multiconjunto. A fines del parcialito, no es necesario implementar la primitiva de destruir.
   - `bool multiconj_guardar(multiconj_t* multiconj, char* elem)`: guarda un elemento en el multiconjunto. Devuelve `true` si se pudo guardar el elemento correctamente, `false` en caso contrario.
   - `bool multiconj_pertenece(multiconj_t* multiconj, char* elem)`: devuelve `true` si el elemento aparece al menos una vez en el conjunto.
   - `bool multiconj_borrar(multiconj_t* multiconj, char* elem)`: elimina *una aparición* del elemento dentro del conjunto. Devuelve `true` si se eliminó una aparición del elemento.

   Dar la estructura del TDA y la implementación de las 4 primitivas marcadas, de forma tal que todas sean $\mathcal{O}(1)$.

8. (⋆⋆⋆) Se tiene un hash que cuenta con una función de hashing que, recibida una clave, devuelve la posición de su inicial en el abecedario. La capacidad inicial del hash es 26. Para los puntos B, C y D indicar y justificar si las afirmaciones son verdaderas o falsas. Se puede considerar que todas las claves serán palabras (sólo se usan letras para las claves).

   a. Mostrar cómo quedan un hash abierto y un hash cerrado (sólo el resultado final) tras guardar las siguientes claves: Ambulancia (0), Gato (6), Manzana (12), Ananá (0), Girasol (6), Zapato (25), Zapallo (25), Manzana (12), Bolso (1). *Aclaración*: No es necesario hacer una tabla de 26 posiciones, lo importante es que quede claro en cuál posición está cada elemento.

   b. En un hash **abierto** con dicha función de hashing, se decide redimensionar cuando la cantidad alcanza la capacidad (factor de carga = 1). El rendimiento de `hash_obtener()` es mejor en este caso respecto a si se redimensionara al alcanzar un factor de carga 2.

   c. En un hash **cerrado** con dicha función de hashing, si se insertan `n + 1` claves diferentes (considerar que se haya redimensionado acordemente), `n` con la misma letra inicial, y 1 con otra distinta, en el primer caso `obtener()` es $\mathcal{O}(n)$ y en el segundo siempre $\mathcal{O}(1)$.

   d. En un hash **abierto** con dicha función de hashing, si se insertan `n + 1` claves diferentes (considerar que se haya redimensionado acordemente), `n` con la misma letra inicial, y 1 con otra distinta, en el primer caso `obtener()` es $\mathcal{O}(n)$ y en el segundo siempre $\mathcal{O}(1)$.

9. (⋆⋆⋆) El Ing. Musumeci quiere implementar un hash abierto, pero en el que

las listas de cada posición se encuentren ordenadas por clave (usando `strcmp`). Explicar cómo mejora o empeora respecto a la versión que vemos en clase para el caso de inserciones, borrados, búsquedas con éxito (el elemento se encuentra en el hash) y sin éxito (no se encuentra).

10. (⋆⋆⋆) Hacer un seguimiento e indicar cómo queda un hash que aplica hashing perfecto (con Hash & Displace) con las siguientes claves (los valores de las funciones de hashing se encuentran a continuación de las claves).

```
"casa": 1; 3; 2
"perro": 3; 3; 3
"cobayo": 6; 2; 6
"conejo": 3; 2; 1
"hamster": 2; 0; 1
"gato": 3; 0; 5
"hurón": 2; 3; 5
"pejelagarto": 6; 0; 7
```

11. (⋆⋆) Del ejercicio anterior, explicar qué sucedería si los resultados de aplicar las funciones de hashing sobre *"pejelagarto"* fueran (6, 0, 3), y cómo debería resolverse.

12. (⋆⋆⋆) a. Realizar las siguientes operaciones sobre un hash con Hopscotch hashing, de largo 9 y $K = 2$ (los números son también el resultado de la función de hashing): Guardar 1, Guardar 0, Guardar 9, Borrar 1, Buscar 9, Guardar 5, Guardar 7, Guardar 17, Borrar 7, Guardar 16, Guardar 82 Guardar 81, Guardar 2.

    b. Indicar qué sucede si guardamos el valor 89, y cómo se debe resolver.

13. (⋆⋆⋆) a. Realizar el seguimiento de guardar las siguientes claves dentro de un hash con Cuckoo Hashing, con 2 funciones de hashing (los valores de aplicar las funciones de hashing se indican a continuación de las claves), y tamaño 10:

```
gato: 2; 3
perro: 6; 9
ornitorrinco: 2, 5
oso: 8; 3
delfín: 8; 4
canario: 6; 3
```

    b. Explicar qué sucede al buscar la clave *"perro"*.

    c. Explicar cómo hacer para eliminar la clave *"perro"* del hash, y cómo se debe modificar el hash para que siga funcionando correctamente.

14. (⋆⋆⋆) Supongamos que queremos crear una estructura *Diccionario inmutable*. Dicho TDA es un Diccionario que no será modificado. No tendrá altas, ni bajas. Los elementos se los pasan al crearlo (por ejemplo, a través de otra implementación de diccionario, o con un par de arreglos con las claves y valores). Explicar cómo harías para implementar esto de forma eficiente, considerando

que su mayor utilidad es realizar búsquedas lo más rápido posible.

15. (★★★★) Dar una implementación en C de cómo podría ser la primitiva `hash_guardar` para el caso de un hash cerrado con Cuckoo Hashing con dos funciones de hashing (suponer que se llaman `h1` y `h2`). Si la clave ya se encontraba en el hash simplemente devolver `false` (no es necesario hacer un reemplazo del dato). La estructura del Hash es:

```cpp
typedef struct hash {
campo_hash_t* tabla;
size_t cantidad;
size_t tam_tabla;
} campo_hash_t;

typedef struct campo_hash {
char* clave;
void* dato;
int num_fhash;
} hash_t;
```

16. (★★★★★) Se quiere implementar un TDA Diccionario con las siguientes primitivas: `obtener(x)` devuelve el valor de `x` en el diccionario; `insertar(x, y)` inserta en el diccionario la clave `x` con el valor `y` (entero); `borrar(x)` borra la entrada de `x`; `add(x, n)` le suma `n` al contenido de `x`; `add_all(m)` le suma `m` a **todos** los valores.

    Proponer una implementación donde **todas** las operaciones sean $\mathcal{O}(1)$. Justificar el orden de las operaciones.

Ejercicio resuelto

Se tiene un árbol binario de búsqueda con cadenas como claves y función de comparación `strcmp`. Implementar una primitiva `lista_t* abb_mayores(const abb_t* abb, const char* cadena)` que, dados un ABB y una cadena, devuelva una lista ordenada con las claves del árbol estrictamente mayores a la cadena recibida por parámetro (que no necesariamente está en el árbol).

Suponer que la estructura del TDA es:

```
typedef struct abb {
    const char* clave;
    struct abb* izq;
    struct abb* der;
} abb_t;
```

Aclaración: se debe realizar la menor cantidad posible de comparaciones.

Solución

La aclaración hace mención a que utilicemos la propiedad de ABB: sabemos que los nodos a izquierda son menores al actual, y los que estén a derecha son mayores. En particular, si estamos en un nodo cuya clave es menor (o igual) a la buscada, entonces es innecesario revisar a izquierda: todos esos nodos también serán menores. Sí tendremos siempre que revisar a derecha, porque no es posible descartar. Esto es similar a una búsqueda por rango, solo que sin un límite superior.

```
lista_t* abb_mayores(const abb_t* abb, const char* clave) {
    lista_t* mayores = lista_crear();
    if (!mayores) {
        return NULL;
    }
    _abb_mayores(abb, clave, mayores);
    return claves;
}

void _abb_mayores(const abb_t* abb, const char* clave, lista_t* claves) {
    // caso base SIEMPRE
    if (abb == NULL) {
        return;
    }
    // si la actual es mayor, llamamos a la izquierda y guardamos la actual
    if (strcmp(abb->clave, clave) > 0) {
        _abb_mayores(abb->izq, clave, claves);
        lista_insertar_primero(claves, abb->clave);
    }
    _abb_mayores(abb->der, clave, claves);
}
```

Ya que estamos, vemos la complejidad: En el peor de los casos, se pasa una clave que

es menor a la mínima, por lo que se ven todos los nodos, así que será $\mathcal{O}(n)$. Por si nos interesara el mejor caso, esto sería con una clave mayor o igual a la máxima clave del árbol, por lo que recorreríamos la rama derecha, haciéndose en $\Omega(\log n)$ (recordar, la notación Ω es una cota inferior, a diferencia de $\mathcal{O}$ que es una cota superior).

Ejercicios propuestos

1. ($\star$) Dado un árbol binario, escribir una primitiva recursiva que determine la altura del mismo. Indicar y justificar el orden de la primitiva.

2. ($\star$) Implementar una primitiva que devuelva la suma de todos los datos (números) de un árbol binario. Indicar y justificar el orden de la primitiva.

3. ($\star$) Se tiene un AB con números enteros como datos, y se quiere reemplazar cada dato por el resultado de multiplicarlo con los datos de los hijos. Hacer un seguimiento de hacer dicho procesamiento con un preorder, inorder o postorder. A continuación se deja la implementación mediante cada recorrido:

```
void multiplicar_con_hijos_pre(ab_t* arbol) {
    if (!arbol) return;
    int valor_izq = arbol->izq != NULL ? arbol->izq->dato : 1;
    int valor_der = arbol->der != NULL ? arbol->der->dato : 1;
    arbol->dato *= valor_izq * valor_der;
    multiplicar_con_hijos_pre(arbol->izq);
    multiplicar_con_hijos_pre(arbol->der);
}

void multiplicar_con_hijos_in(ab_t* arbol) {
    if (!arbol) return;
    multiplicar_con_hijos_in(arbol->izq);
    int valor_izq = arbol->izq != NULL ? arbol->izq->dato : 1;
    int valor_der = arbol->der != NULL ? arbol->der->dato : 1;
    arbol->dato *= valor_izq * valor_der;
    multiplicar_con_hijos_in(arbol->der);
}

void multiplicar_con_hijos_post(ab_t* arbol) {
    if (!arbol) return;
    multiplicar_con_hijos_post(arbol->izq);
    multiplicar_con_hijos_post(arbol->der);
    int valor_izq = arbol->izq != NULL ? arbol->izq->dato : 1;
    int valor_der = arbol->der != NULL ? arbol->der->dato : 1;
    arbol->dato *= valor_izq * valor_der;
}
```

4. ($\star\star$) Dado un árbol binario, escriba una *primitiva* recursiva que cuente la cantidad de nodos que tienen exactamente dos hijos directos. ¿Qué orden de complejidad tiene la función implementada?

5. (⋆⋆) Escribir una *primitiva* con la firma `void arbol_invertir(arbol_t* arbol)` que invierta el árbol binario pasado por parámetro, de manera tal que los hijos izquierdos de cada nodo se conviertan en hijos derechos.

   La estructura `arbol_t` respeta la siguiente definición:

   ```
   typedef struct arbol {
       struct arbol* izq;
       struct arbol* der;
   } arbol_t;
   ```

   Indicar el orden de complejidad de la función implementada.

6. (⋆⋆) Suponer que se tiene un ABB A con una función de comparación `cmp1` con n claves. También, se tiene otro ABB vacío B con función de comparación `cmp2` (con `cmp1` y `cmp2` diferentes). ¿Es posible insertar en algún orden todas las claves de A en B de tal forma que ambos tengan exactamente la misma estructura? Si es posible, describir un algoritmo que permita lograr esto; si no lo es, razonar por qué. (Considerar que la lógica a emplear debe funcionar para cualquier valor de n y cualquier estructura que tenga el ABB A.)

7. (⋆⋆⋆) Se tiene un AVL con números enteros como claves (su función de comparación simplemente compara dichos valores de la forma tradicional). Su estado inicial puede reconstruirse a partir del preorder: `15 - 6 - 4 - 7 - 50 - 23`. Hacer el seguimiento de las siguientes inserciones, incluyendo rotaciones intermedias: `71 - 27 - 38 - 19 - 11 - 21 - 24 - 25`.

8. (⋆⋆⋆) Mostrar cómo se modifica la estructura de un árbol B (incluyendo los pasos intermedios) con tamaño para 3 claves por nodo que inicialmente se encuentra vacío, al aplicar las siguientes operaciones: insertar 14, insertar 2, insertar 10, insertar 6, insertar 7, insertar 1, insertar 4, insertar 8, insertar 11, insertar 19, insertar 9, insertar 5, insertar 15, insertar 3.

9. (⋆⋆⋆) Definimos como *quiebre en un árbol binario* cuando ocurre que:

   - un hijo derecho tiene un solo hijo, y es el izquierdo
   - un hijo izquierdo tiene un solo hijo, y es el derecho

   Implementar una *primitiva* para el árbol binario `size_t ab_quiebres(const ab_t*)` que, dado un árbol binario, nos devuelva la cantidad de quiebres que tiene. La primitiva no debe modificar el árbol. La estructura del tipo `ab_t` es:

   ```
   typedef struct arbol {
       struct arbol* izq;
       struct arbol* der;
   } ab_t;
   ```

   Indicar y justificar el orden de la primitiva, e indicar el tipo de recorrido implementado.

10. (⋆⋆) Indicar si las siguientes afirmaciones son verdaderas o falsas. En caso de ser verdaderas, justificar, en caso de ser falsas poner un contraejemplo:

1. Si dos árboles binarios tienen el mismo recorrido inorder, entonces tienen la misma estructura.
2. Si dos árboles binarios tienen el mismo recorrido preorder, entonces tienen la misma estructura.
3. Si dos árboles binarios de búsqueda (e idéntica función de comparación) tienen el mismo recorrido preorder, entonces tienen la misma estructura.

11. (⋆⋆) Implementar una primitiva para el ABB, que reciba el ABB y devuelva una lista con las claves del mismo, ordenadas tal que si insertáramos las claves en un ABB vacío (con la misma función de comparación), dicho ABB tendría la misma estructura que el árbol original. ¿Qué tipo de recorrido utilizaste? Indicar y justificar el orden de la primitiva.

12. (⋆⋆⋆⋆) Implementar una primitiva para el AB que reciba dos arreglos (o listas) de cadenas. El primer arreglo corresponde al preorder de un árbol binario. El segundo al inorder del mismo árbol (ambos arreglos tienen los mismos elementos, sin repetidos). La función debe devolver un árbol binario que tenga dicho preorder e inorder. Indicar y justificar el orden de la primitiva (tener cuidado con este punto). Considerar que la estructura del árbol binario es:

```
typedef struct arbol {
    struct arbol* izq;
    struct arbol* der;
    char* dato;
} ab_t;
```

13. (⋆⋆⋆⋆) Implementar una función que reciba un arreglo ordenado y devuelva un arreglo o lista con los elementos en orden para ser insertados en un ABB, de tal forma que al insertarlos en dicho orden se asegure que el ABB quede balanceado. ¿Cómo cambiarías tu resolución si en vez de querer guardarlos en un ABB se fueran a insertar en un AVL?

14. (⋆⋆⋆) Determinar cómo es el Árbol cuyo pre order es EURMAONDVSZT, e in order es MRAUOZSVDNET, e indicar su recorrido post order.

Ejercicio resuelto

Implementar en C una primitiva para el heap (siendo este un max-heap) que reciba un heap y una función de comparación y lo reordene de manera tal que se se comporte como max-heap para la nueva función de comparación (se cambia la función de prioridad). El orden de dicha primitiva debe ser $\mathcal{O}(n)$.

Solución

La única dificultad de este ejercicio radica en entender verdaderamente qué es lo que se nos pide: darle a un arreglo ya existente (el interno del heap) forma de heap, dada por una función de comparación (la nueva función de comparación del heap). Entonces, esto no es más que invocar a `heapify`, y ya. Eso es todo. También, es la única forma de darle ese orden a la primitiva.

```
void heap_cambiar_prioridad(heap_t* heap, heap_cmp_t nueva_cmp) {
    heap->cmp = nueva_cmp;
    heapify(heap->datos, heap->cantidad, nueva_cmp);
}
```

Obviamente, la primitiva es $\mathcal{O}(n)$ dado que `heapify` (bien implementado) tiene dicha complejidad. Por supuesto, este ejercicio involucra más pensar bien cuáles son las operaciones que se pueden hacer con el heap (y, especialmente, cuál es la única con el orden pedido).

Ejercicios propuestos

1. ($\star$) Implementar en lenguaje C una función *recursiva* con la firma `bool es_heap(int arr[], size_t n)`. Esta función debe devolver true o false de acuerdo a si el arreglo que recibe como parámetro cumple la propiedad de heap (de mínimos).

   Hacer el seguimiento de la función para el arreglo `[ 1, 7, 2, 8, 7, 6, 3, 3, 9, 10 ]`.

2. ($\star$) Implementar una primitiva para el heap (de máximos) que obtenga los 3 elementos más grandes del heap en $\mathcal{O}(1)$.

3. ($\star\star$) Si en el ejercicio anterior vez de quererse los 3 elementos más grandes se quisieran los K elementos más grandes ¿cómo se debería proceder? ¿Cuál terminaría siendo la complejidad del algoritmo?

4. ($\star\star$) En un heap de máximos ¿cuáles son las posibles posiciones del arreglo donde podría encontrarse el mínimo?

5. ($\star\star$) Realizar el seguimiento del algoritmo *heapsort* para ordenar el siguiente arreglo: `[ 4, 7, 8, 14, 10, 9, 16, 2, 3, 1 ]`.

6. ($\star\star\star$) ¿Puede utilizarse un Heap para implementar el TDA cola (en el que se extraen los elementos en el orden en que fueron insertados)? ¿Y para implementar el TDA pila?

7. (⋆⋆) Hacer el seguimiento de las siguientes operaciones sobre un heap (de mínimos), mostrando el estado de la estructura después de cada modificación:

   1. Crear un heap de mínimos desde el arreglo `[8, 2, 1, 5, 10, 6, 14, 4]`.

   2. Sobre el heap resultante del punto anterior, realizar las siguientes operaciones: `encolar(6)`, `encolar(3)`, `encolar(17)`, `desencolar()`, `encolar(7)`, `desencolar()`.

8. (⋆⋆⋆) Escribir una función en C que, dado un arreglo de n cadenas y un entero positivo k, devuelva una lista con las k cadenas más largas. Se espera que el orden del algoritmo sea $\mathcal{O}(n \log k)$. Justificar el orden.

9. (⋆⋆⋆) Para implementar un TDA Cola de prioridad, nos proponen la siguiente solución: usar un arreglo desordenado (`arr`) para insertar los datos, y una variable (`maximo`) para poder obtener el máximo en $\mathcal{O}(1)$. Se mantiene actualizada la variable `maximo` cada vez que se encola o desencola. ¿Es una buena solución en el caso general? Justificar (recomendación: comparar contra la implementación de colas de prioridad vista en clase).

10. (⋆⋆⋆⋆) Se tienen k arreglos de enteros previamente ordenados y se quiere obtener un arreglo ordenado que contenga a todos los elementos de los k arreglos. Sabiendo que cada arreglo tiene tamaño h, definimos como n a la sumatoria de la cantidad de elementos de todos los arreglos, es decir, $n = k \times h$.

    Escribir en C una función `int* k_merge(int** arr, size_t k, size_t h)` que reciba los k arreglos y devuelva uno nuevo con los n elementos ordenados entre sí. La función debe ser de orden $\mathcal{O}(n \log k)$. Justificar el orden del algoritmo propuesto.

11. (⋆⋆⋆⋆⋆) Diseñar el TDA Mediana. Dicho TDA debe poder recibir un flujo de números y, en cualquier momento, debe poder consultársele cuál es la mediana de **todos** los elementos vistos hasta ese momento. La primitiva para agregar un nuevo número debe poder hacerse en $\mathcal{O}(\log n)$ mientras que la operación de consultar la mediana debe ser $\mathcal{O}(1)$. Recordar que la mediana de una secuencia de números es el elemento que se encontraría a la mitad si la secuencia se encontrara ordenada (en caso de ser una cantidad par, se puede definir como el promedio entre ambos valores adyacentes del medio, o como uno de los dos de ellos de forma arbitraria).

Grafos: Usos, implementaciones, recorridos

Ejercicio resuelto

Implementar un algoritmo que, dado un grafo no dirigido, nos devuelva un ciclo dentro del mismo, si es que los tiene. Indicar el orden del algoritmo.

Solución

Antes que nada, debemos entender que el ejercicio en sí es el mismo, se trate de un grafo dirigido o uno no dirigido. La única diferencia se encuentra en que un ciclo por defínción necesita contar con al menos dos aristas. Esta definición en sí no nos importaría mucho en el caso de un grafo dirigido, pero si en el de uno no dirigido. Si no lo tuviéramos en cuenta, todo grafo no dirigido, con al menos una arista, va a tener un ciclo, lo cual no es cierto.

Para resolver este problema, podemos pensar en simplemente recorrer el grafo no dirigido, sea con un recorrido *BFS* o *DFS*. Una vez que nos topemos con un vértice ya visitado, ahí tenemos un posible ciclo. Esto es, si estoy viendo los adyacentes a un vértice dado, y dicho vértice está visitado, uno se apresuraría a decir que ahí se cierra un ciclo. Esto es cierto, salvo un caso: que dicho vértice visitado sea el antecesor a nuestro vértice en el recorrido (BFS o DFS). Recordar que se trata de un grafo no dirigido, por ende si v tiene de adyacente a w, entonces también vale la recíproca, y no por ello se crea un ciclo. El problema se da con la arista de la que vengo. Básicamente, deberíamos obviar al vértice del que vengo en el recorrido, que justamente es el padre. Si nosotros ya tenemos que `padre[W] = V`, entonces simplemente tenemos que saltearnos a V cuando veamos a los adyacentes ya visitados.

Lo resolvemos con ambos recorridos. Por *BFS*:

```
def obtener_ciclo_bfs(grafo):
  visitados = {}
  for v in grafo:
    if v not in visitados:
      ciclo = bfs_ciclo(grafo, v, visitados)
      if ciclo is not None:
        return ciclo
  return None

def bfs_ciclo(grafo, v, visitados):
  q = Cola()
  q.encolar(v)
  visitados[v] = True
  padre = {}  # Para poder reconstruir el ciclo
  padre[v] = None

  while not q.esta_vacia():
    v = q.desencolar()
```

```
    for w in grafo.adyacentes(v):
      if w in visitados:
        # Si w fue visitado y es padre de v, entonces es la arista
        # de donde vengo (no es ciclo).
        # Si no es su padre, esta arista (v, w) cierra un ciclo que
        # empieza en w.
        if w != padre[v]:
          return reconstruir_ciclo(padre, w, v)
      else:
        q.encolar(w)
        visitados[v] = True
        padre[w] = v

  # Si llegamos hasta acá es porque no encontramos ningún ciclo.
  return None
```

La solución por *DFS* sería:

```
def obtener_ciclo_dfs(grafo):
  visitados = {}
  padre = {}
  for v in grafo:
    if v not in visitados:
      ciclo = dfs_ciclo(grafo, v, visitados, padre)
      if ciclo is not None:
        return ciclo
  return None

def dfs_ciclo(grafo, v, visitados, padre):
  visitados[v] = True
  for w in grafo.adyacentes(v):
    if w in visitados:
      # Si w fue visitado y es padre de v, entonces es la arista de donde
      # vengo (no es ciclo).
      # Si no es su padre, esta arista (v, w) cierra un ciclo que empieza
      # en w.
      if w != padre[v]:
        return reconstruir_ciclo(padre, w, v)
    else:
      padre[w] = v
      ciclo = dfs_ciclo(grafo, w, visitados, padre)
      if ciclo is not None:
        return ciclo

  # Si llegamos hasta acá es porque no encontramos ningún ciclo.
  return None
```

Ambas técnicas usan la función para reconstruir el ciclo:

```
def reconstruir_ciclo(padre, inicio, fin):
  v = fin
  camino = []
  while v != inicio:
    camino.append(v)
    v = padre[v]
  camino.append(inicio)
  return camino.invertir()
```

Ahora bien, para ver el orden, podemos ver que en el caso feliz, vamos a encontrar un ciclo muy rapido. Pero claramente eso no nos cambia mucho. Pensemos el caso de, a lo sumo, encontrar el ciclo muy tarde en el recorrido (también veremos el caso de no haber ciclo). En ese caso, en cualquiera de los dos recorridos vamos a pasar por cada vértice una vez, y solo una vez (a fin de cuentas, no volvemos a estar sobre un vértice ya visitado). Por cada vértice vemos *sus* aristas. Recordar que es muy importante no caer en la tentación de decir que entonces el algoritmo es $\mathcal{O}(V \times E)$, porque si bien es cierto, es una muy mala cota. Por cada vértice pasamos por *sus* arisas, que distan de ser las totales del grafo. Si por cada vértice vemos sus aristas (y no las de todo el grafo), en total estamos viendo todas las aristas del grafo, dos veces (una por cada extremo). Entonces, el orden será $\mathcal{O}(V + 2E) = \mathcal{O}(V + E)$. Todo esto, considerando que la implementación es con listas de adyacencias (implementadas con diccionarios, o bien siendo los vértices valores numéricos para indexar en un arreglo). Si fuera otra la implementación, obtener los adyacentes a un vértice dado nos costará más ($\mathcal{O}(V)$, en el caso de una matriz de adyacencia, u $\mathcal{O}(E)$ en el caso de una matriz de incidencia).

Haciendo un poco más de análisis: ¿es acaso el caso de no tener ciclos nuestro peor caso? Supongamos que el grafo es conexo, por simplifcación. Si el grafo no tiene ciclos, y es conexo, necesariamente se trata de un árbol. Para este caso, $|E| = |V| - 1$, por ende nuestro orden a fin de cuentas terminaría siendo $\mathcal{O}(V)$. ¿Esto implica que entonces nuestro algoritmo es en realidad $\mathcal{O}(V)$? No, significa que ese, que a priori podíamos pensar que era nuestro peor caso, en realidad no lo es. Nuestro peor caso implica que haya un ciclo, pero tener la mala suerte de tardar en encontrarlo. Ya sea porque el vértice en el que se empieza el recorrido está lejos del ciclo, o por el orden aleatorio de las cosas.

¿Cuáles serían las diferencias si en vez de trabajar con un grafo no dirigido, lo hiciéramos sobre un grafo dirigido? Notar que la solución no debería ser demasiado distinta a estas propuestas, pero tienen sus pequeñas diferencias que dejamos para resolver en los ejercicios propuestos.

Ejercicios propuestos

1. ($\star$) a. Dibujar un grafo no dirigido que:
   - Tenga 6 vértices
   - Tenga un ciclo que incluya 4 de dichos vértices
   - Sea conexo

- Haya un vértice de grado 1
- Haya un vértice de grado 4

b. Escribir la representación de matriz de incidencia y matriz de adyacencia del grafo resultante del punto anterior.

2. (⋆) a. Dibujar un grafo dirigido que:

   - Tenga 7 vértices
   - Tenga un ciclo que incluya 3 de dichos vértices
   - Tenga un ciclo que incluya 2 de dichos vértices
   - Haya un vértice de grado de entrada 0
   - Haya un vértice de grado de salida 4

   b. Escribir la representación de matriz de incidencia y matriz de adyacencia del grafo resultante del punto anterior.

3. (⋆) Implementar una función que determine el:

   a. El grado de todos los vértices de un grafo no dirigido.

   b. El grado de salida de todos los vértices de un grafo dirigido.

   c. El grado de entrada de todos los vértices de un grafo dirigido.

4. (⋆) Implementar un algoritmo que determine si un grafo no dirigido es conexo o no. Indicar la complejidad del algoritmo si el grafo está implementado con una matriz de adyacencia.

5. (⋆⋆) Implementar un algoritmo que, dado un grafo dirigido, nos devuelva un ciclo dentro del mismo, si es que lo tiene. Indicar el orden del algoritmo.

6. (⋆⋆) Un árbol es un grafo no dirigido que cumple con las siguientes propiedades:

   a. $\|E\| = \|V\| - 1$

   b. Es acíclico

   c. Es conexo

   Por teorema, si un grafo cumple dos de estas tres condiciones, será árbol (y por consiguiente, cumplirá la tercera). Haciendo uso de ésto (y únicamente de ésto), se pide implementar una función que reciba un grafo no dirigido y determine si se trata de un árbol, o no. Indicar el orden de la función implementada.

7. (⋆⋆) Proponer una función para calcular el grafo traspuesto G^T de un grafo dirigido G. El grafo traspuesto G^T posee los mismos vértices que G, pero con todas sus aristas invertidas (por cada arista *(v, w)* en G, existe una arista *(w, v)* en G^T). Indicar la complejidad para un grafo implementado con:

   a. lista de adyancencias

   b. matriz de adyacencias

8. (⋆⋆⋆) La teoría de los 6 grados de separación dice que cualquiera en la Tierra puede estar conectado a cualquier otra persona del planeta a través de una

cadena de conocidos que no tiene más de cinco intermediarios (conectando a ambas personas con solo seis enlaces). Suponiendo que se tiene un grafo G en el que cada vértice es una persona y cada arista conecta gente que se conoce (el grafo es no dirigido):

   a. Implementar un algoritmo para comprobar si se cumple tal teoría para todo el conjunto de personas representadas en el grafo G. Indicar el orden del algoritmo.

   b. Suponiendo que en el grafo G no habrán altas ni bajas de vértices, pero podrían haberla de aristas (la gente se va conociendo), explicar las ventajas y desventajas que tendría implementar al grafo G con una matriz de adyacencia.

9. (⋆⋆⋆) Matías está en Barcelona y quiere recorrer un museo. Su idea es hacer un recorrido bastante lógico: empezar en una sala (al azar), luego ir a una adyacente a ésta, luego a una adyancente a la segunda (si no fue visitada aún), y así hasta recorrer todas las salas. Cuando no tiene más salas adyacentes para visitar (porque ya fueron todas visitadas), simplemente vuelve por donde vino buscando otras salas adyacentes. Teniendo un grafo no dirigido, que representa el mapa del museo (donde los vértices son salas, y las aristas (v, w) indican que las salas v y w se encuentran conectadas), implementar un algoritmo que nos devuelva una lista con un recorrido posible de la idea de Matías para visitar las salas del museo. Indicar el recorrido utilizado y el orden del algoritmo. Justificar.

10. (⋆⋆) Escribir una función `bool es_bipartito(grafo)` que dado un grafo no dirigido devuelva true o false de acuerdo a si es bipartito o no. Indicar y justificar el orden del algoritmo. ¿Qué tipo de recorrido utiliza?

11. (⋆⋆⋆) Implementar un algoritmo que reciba un grafo dirigido, un vértice V y un número N, y devuelva una lista con todos los vértices que se encuentren a exactamente N aristas de distancia del vértice V. Indicar el tipo de recorrido utilizado y el orden del algoritmo. Justificar.

12. (⋆⋆) Implementar una función que permita determinar si un grafo *puede ser* no dirigido. Determinar el orden del algoritmo implementado.

13. (⋆⋆) a. Dada la siguiente matriz de adyacencias, escribir la representación del grafo como una lista de adyacencias:

	A	B	C	D	E	F
A	0	7	5	0	3	8
B	7	0	0	0	1	3
C	5	0	0	5	3	2
D	0	0	5	0	2	0
E	3	1	3	2	0	0
F	8	3	2	0	0	0

   b. ¿Qué complejidad tiene hacer esta traducción?

c. ¿Qué ventajas y desventajas encuentra en cada representación? Explicar teniendo en cuenta cuestiones de espacio y tiempo según las operaciones que admite un grafo y considerar que las listas de adyacencias es un diccionario de listas.

14. (⋆⋆) Explicar las ventajas y desventajas de un grafo implementado con listas de adyacencia (lista de listas) por sobre una implementación de matriz de adyacencia, sabiendo que su densidad es $\frac{|E|}{max\ |E|} > 075$ o superior.

15. (⋆⋆⋆) Un autor decidió escribir un libro con varias tramas que se puede leer de forma no lineal. Es decir, por ejemplo, después del capítulo 1 puede leer el 2 o el 73; pero la historia no tiene sentido si se abordan estos últimos antes que el 1.

    Siendo un aficionado de la computación, el autor ahora necesita un orden para publicar su obra, y decidió modelar este problema como un grafo dirigido, en dónde los capítulos son los vértices y sus dependencias las aristas. Así existen, por ejemplo, las aristas `(v1, v2)` y `(v1, v73)`.

    Escribir un algoritmo que devuelva un orden en el que se puede leer la historia sin obviar ningún capítulo. Indicar la complejidad del algoritmo.

16. (⋆⋆) Implementar una función que reciba un grafo no dirigido y no pesado implementado con listas de adyacencia (diccionario de diccionarios) y devuelva una matriz que sea equivalente a la representación de matriz de adyacencia del mismo grafo. Indicar y justificar el orden del algoritmo implementado.

17. (⋆⋆) Implementar una función que reciba un grafo no dirigido, y que compruebe la siguiente afirmación: *"La cantidad de vértices de grado IMPAR es PAR"*. Indicar y justificar el orden del algoritmo si el grafo está implementado como matriz de adyacencia.

18. (⋆⋆⋆⋆) Dado un número inicial `X` se pueden realizar dos tipos de operaciones sobre el número:

    - Multiplicar por 2
    - Restarle 1.

    Implementar un algoritmo que encuentra la menor cantidad de operaciones a realizar para convertir el número `X` en el número `Y`, con tan solo las operaciones mencionadas arriba (podemos aplicarlas la cantidad de veces que querramos).

19. (⋆⋆⋆⋆⋆) Se tiene un arreglo de palabras de un lenguaje alienigena. Dicho arreglo se encuentra ordenado para dicho idioma (no conocemos el orden de su abecedario). Implementar un algoritmo que reciba dicho arreglo y determine un orden posible para las letras del abecedario en dicho idioma. Por ejemplo: `{"caa", "acbd", "acba", "bac", "bad"} --> ['c', 'd', 'a', 'b']`

Ejercicio resuelto

El diámetro de una red es el máximo de las distancias mínimas entre todos los vértices de la misma. Implementar un algoritmo que permita obtener el diámetro de una red, para el caso de un grafo no dirigido y no pesado. Indicar el orden del algoritmo propuesto.

Solución

Vamos desglosando lo que nos dice el enunciado para entender cómo resolverlo: El diámetro de una red (grafo, etc.) es la distancia más grande entre todos los caminos mínimos que hayan. Por lo tanto, será necesario obtener todas las distancias mínimas y quedarnos con el máximo valor. En esta definición no contamos el infinito, puesto que no tiene mayor sentido. Particularmente vamos a asumir que el grafo es conexo, si bien puede resolverse también para otros casos (incluyendo que sea dirigido). Sería tan solo despreciar el caso en el que no se haya encontrado camino (ej, que camino mínimo devuelva `None`).

Una primera solución podría ser:

```
def diametro(grafo):
    max_min_dist = 0
    for v in grafo:
        for w in grafo:
            distancia = camino_minimo(grafo, v, w)
            # aca podria validar si la distancia no es None o infinito
            if distancia > max_min_dist:
                max_min_dist = distancia
    return max_min_dist
```

El problema aquí es que el costo de todo esto va a ser $\mathcal{O}(V^2(V+E))$ (considerando que la forma eficiente de obtener los caminos mínimos en un grafo no pesado es utilizando un recorrido BFS), ya que buscamos cada camino mínimo para cada origen y destino posible.

Una optimización (bastante evidente) es aprovechar que BFS nos puede devolver un diccionario con todos los caminos mínimos (en particular, sólo nos interesan las distancias, no las forma de reconstruirlos), y luego solo quedarnos con la distancia más grande:

```
def diametro(grafo):
    max_min_dist = 0
    for v in grafo:
        distancias = caminos_minimos(grafo, v)
        for w in distancias:
            if distancias[w] > max_min_dist:
                max_min_dist = distancias[w]
    return max_min_dist
```

Con esto, la solución pasará a ser $\mathcal{O}(V(V+E))$. Ya que estamos, también escribimos

cómo sería una implementación para obtener esos caminos mínimos:

```
def caminos_minimos(grafo, origen)
    q = Cola()
    visitados = set()
    distancia = {}
    distancia[origen] = 0
    visitados.add(origen)
    q.encolar(origen)

    while not q.esta_vacia():
        v = q.desencolar()
        for w in grafo.adyacentes(v):
            if w not in visitados:
                distancia[w] = distancia[v] + 1
                q.encolar(w)
                visitados.add(w)
    return distancia
```

Ejercicios propuestos

1. (⋆⋆) a. Obtener una representación del camino mínimo desde el vértice A en el siguiente grafo (representado con una matriz de adyacencias), hacia todos los demás vértices, utilizando el algoritmo de Dijkstra:

	A	B	C	D	E	F
A	0	7	5	0	3	8
B	7	0	0	0	1	3
C	5	0	0	5	3	2
D	0	0	5	0	2	0
E	3	1	3	2	0	0
F	8	3	2	0	0	0

   b. ¿Qué condiciones debe cumplir el grafo para poder aplicar el algoritmo de Dijkstra? ¿Qué característica tiene el grafo si al finalizar la ejecución del algoritmo, uno o más vértices quedan a *distancia infinita*?

2. (⋆⋆⋆) a. Obtener una representación del camino mínimo desde el vértice A en el siguiente grafo (representado con una matriz de adyacencias), hacia todos los demás vértices, utilizando el algoritmo de Bellman-Ford.

   b. Volver a realizar, suponiendo que la arista de `B` a `A` ahora tiene un peso de 1.

3. (⋆⋆) Obtener el Árbol de Tendido Mínimo del siguiente grafo:

   a. Utilizando el Algoritmo de Kruskal.

   b. Utilizando el Algoritmo de Prim.

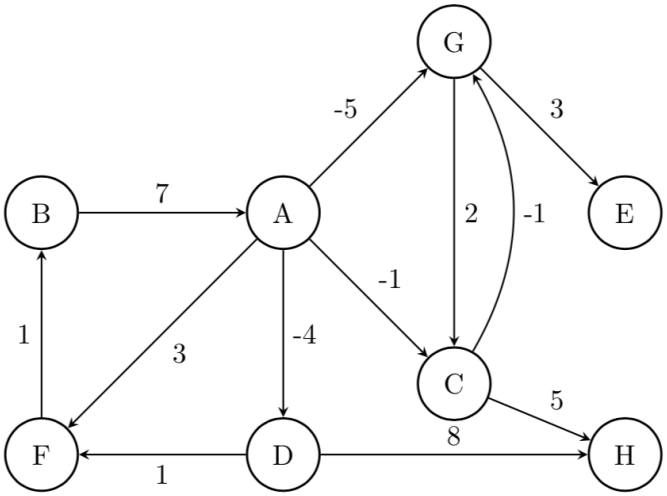

Figura 1: grafo bf

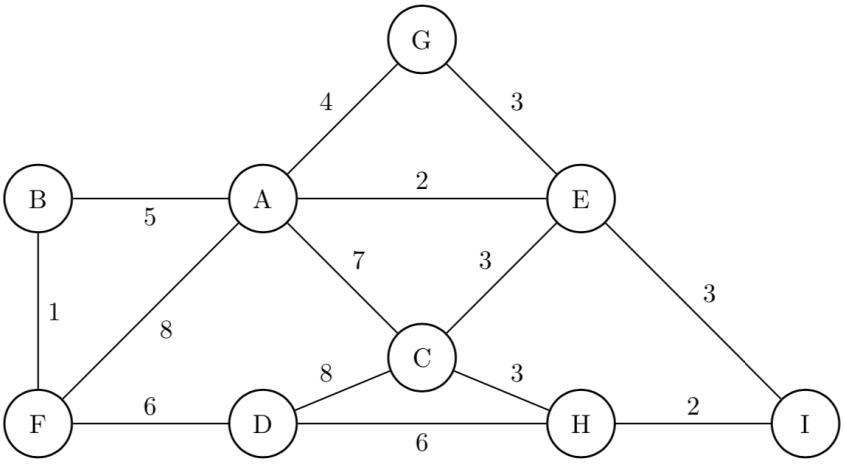

Figura 2: grafo mst

4. (⋆⋆⋆) Dadas las matrices de adyacencia `M1`, `M2` y `M3`, responder las siguientes preguntas (recomendamos pasar los grafos a una representación visual para mayor facilidad):

   a. ¿Puede ser el grafo definido por `M2` un árbol de tendido mínimo de `M1`? Justificar.

   b. Realizar un seguimiento de aplicar el algoritmo de Kruskal para obtener un árbol de tendido mínimo del grafo definido por `M3`.

M1	A	B	C	D	E	F	G
A	0	3	4	0	0	0	0
B	3	0	5	3	3	0	0
C	4	5	0	2	0	0	6
D	0	3	2	0	4	2	1
E	0	3	0	4	0	6	0
F	0	0	0	2	6	0	0
G	0	0	6	1	0	0	0

M2	A	B	C	D	E	F	G
A	0	3	4	0	0	0	0
B	3	0	0	0	3	0	0
C	4	0	0	2	0	0	6
D	0	0	2	0	0	2	0
E	0	3	0	0	0	0	0
F	0	0	0	2	0	0	0
G	0	0	6	0	0	0	0

M3	A	B	C	D	E	F	G
A	0	6	0	4	0	0	0
B	6	0	7	8	6	0	0
C	0	7	0	0	4	0	0
D	4	8	0	0	14	5	0
E	0	6	4	14	0	7	8
F	0	0	0	5	7	0	10
G	0	0	0	0	8	10	0

5. (⋆⋆) Analizar la complejidad del algoritmo de Prim según si el grafo está implementado con listas de adyacencia o matriz de adyacencia.

6. (⋆⋆) Responder las siguientes preguntas, **justificando** la respuesta:

   a. Al aplicar sobre un grafo el algoritmo de Dijkstra para encontrar caminos mínimos desde un vértice v cualquiera, se obtiene un árbol definido por el diccionario de padres (que permite reconstruir dichos caminos mínimos). Dicho árbol, ¿es siempre de tendido mínimo?

b. Al obtener un árbol de tendido mínimo de un grafo, se asegura que la suma de los pesos de las aristas sean mínimos. ¿Es posible utilizar el árbol de tendido mínimo para encontrar el camíno mínimo entre dos pares de vértices cualesquiera?

c. Si un grafo es no pesado, ¿Se puede utilizar el Algoritmo de Dijkstra para obtener los caminos mínimos en dicho grafo?

7. (⋆⋆⋆) Realizar un seguimiento de aplicar el Algoritmo de Ford-Fulkerson para obtener el Flujo máximo a través de la red definida por el siguiente grafo.

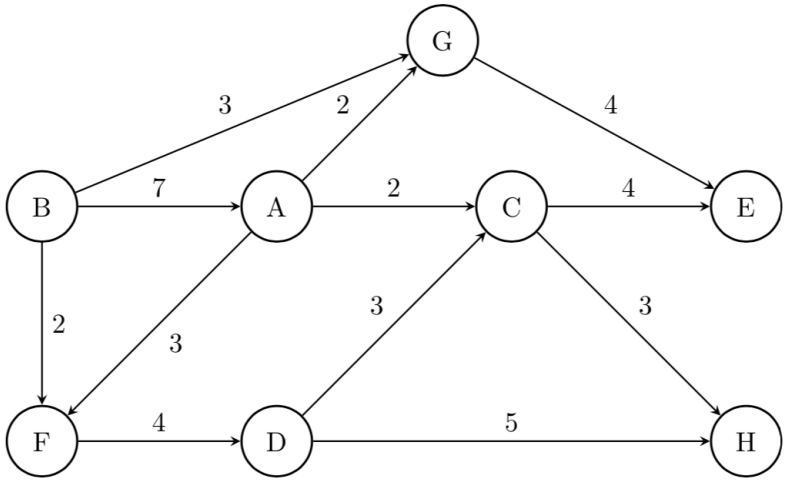

Figura 3: grafo flujo

8. (⋆⋆⋆) Implementar por **backtracking** un algoritmo que, dado un grafo no dirigido y un número $n < |\mathcal{V}|$, nos permita obtener un subconjunto de n vértices tal que ningún par de dichos vértices sean adyacentes entre sí. Se puede suponer que los vértices están identificados con números de 0 a $|\mathcal{V}| - 1$.

9. (⋆⋆⋆) Implementar un algoritmo que reciba un grafo y un número n que, utilizando **backtracking**, indique si es posible pintar cada vértice con n colores de tal forma que no hayan dos vértices adyacentes con el mismo color.

10. (⋆⋆⋆⋆) Problema del viajante (TSP): Dada una lista de ciudades y la distancia entre cada par de ellas, ¿Cuál es la ruta más corta posible que visita cada ciudad exactamente una vez y al finalizar regresa a la ciudad de origen?

11. (⋆⋆⋆) Implementar un algoritmo que reciba un grafo, un vértice v y otro w y, utilizando **backtracking**, obtenga **todos** los caminos simples de v a w en el grafo.

12. (⋆⋆⋆⋆) Se cuenta con un grafo en el que sus aristas tiene peso 1 o 2, únicamente. Implementar un algoritmo que permita obtener el camino mínimo de un vértice hacia todos los demás, en tiempo $\mathcal{O}(V + E)$.

13. (⋆⋆⋆⋆⋆) Implementar un algoritmo que, dado un grafo dirigido, un vértice s y otro t determine la cantidad mínima de aristas que deberían cambiar de sentido en el grafo para que exista un camino de s a t.

14. (⋆⋆⋆⋆⋆) Supongamos que tenemos un sistema de una facultad en el que cada alumno puede pedir hasta 10 libros de la biblioteca. La biblioteca tiene 3 copias de cada libro. Cada alumno desea pedir libros diferentes. Implementar un algoritmo que nos permita obtener la forma de asignar libros a alumnos de tal forma que la cantidad de préstamos sea máxima.