Apunte 1 - Introducción y Repaso

Índice general

Índice de figuras

Algoritmos

Índice de ejercicios

Apunte 1

Introducción

Actualizado: 14 de marzo de 2024

En este apunte nos enfocaremos en definiciones que serán centrales a lo largo de esta materia y repasaremos algunos conceptos vistos en materias anteriores.

1.1. ¿Qué es una estructura de datos?

Antes de comenzar a estudiar las estructuras de datos, es necesario que definamos claramente la diferencia entre "tipo de dato", "estructura de datos" y "tipo de dato abstracto".

De acuerdo a lo definido por Aho-1991:

- El *tipo de dato de una variable* "es el conjunto de valores que la variable puede tomar". Por ejemplo, una variable de tipo boolean sólo puede tomar un valor verdadero (true) o falso (false).
- Una *estructura de datos* "es una colección de variables, posiblemente de diferentes tipos de datos, conectados de varias formas". El contenido básico de una estructura de datos es la "celda" o "nodo", que es capaz de almacenar un dato de un tipo determinado.
- Un *tipo de datos abstracto (TDA)* "es un modelo matemático, junto con varias operaciones definidas sobre ese modelo". Es decir, un TDA define un modelo de un tipo de datos, indicando su posible contenido y funcionalidad (operaciones), pero no dice cómo debe implementarse. Como programadores, lo ideal es diseñar algoritmos en términos de TDAs. Luego, al momento de implementar un algoritmo en un lenguaje de programación particular, será necesario encontrar la manera de representar los TDAs con los tipos de datos y operadores de dicho lenguaje de programación.

La estructura básica en la mayoría de los lenguajes de programación es el arreglo unidimensional (array), que es una sucesión de celdas que almacenan un dato cada una, donde todos los datos son del mismo tipo. Además, es posible acceder a cada celda indicando su posición dentro de la sucesión (por ejemplo, en Java se accede con la sintaxis *a[i]*). La posibilidad de acceder a una celda particular, para observar o modificar su contenido, es parte de la funcionalidad que la estructura *array* permite. Cuando se ha utilizado un arreglo en algún lenguaje, logramos entender cómo funciona en otros lenguajes, dado que en realidad hemos aprendido los principios del TDA arreglo. Al usar otro lenguaje de programación sólo necesitamos saber con qué sentencias se crea un arreglo y qué operaciones tiene disponibles, pero "qué es un arrelgo" ya lo hemos aprendido.

Entonces podemos decir que *una estructura de datos define y organiza la interrelación de los datos y las operaciones que se pueden realizar sobre ellos.* Las operaciones sobre las estructuras pueden ser básicas como alta (agregar un nuevo valor a la estructura), baja (eliminar un valor de la estructura), y búsqueda (encontrar un valor determinado dentro de la estructura); u operaciones más complejas como ordenar los valores, crear una copia, etc.

1.2. Clasificación de las estructuras de datos

Como hemos dicho, las estructuras de datos son modelos para organizar y acceder a un conjunto de datos del mismo tipo[1].

Casi todos los programas de software guardan mucha información, y para ello se utilizan las estructuras de datos. Existen distintos tipos de estructuras de datos (lineales, jerárquicas, de grafos, etc.). Cada estructura

[1] Aunque algunas estructuras pueden mantener datos de diferentes tipos, en esta materia nos independizaremos de ese enfoque.

tiene propósitos diferentes y la elección de las estructuras de datos adecuadas a un problema es uno de los puntos clave de un buen desarrollo de software.

- *Estructuras lineales*: Estas son las estructuras de datos más sencillas. Guardan los datos (a los que llamaremos elementos) como si estuvieran en una fila (uno detrás de otro). Visualmente las estructuras lineales tienen la forma que se muestra en la Figura 1.1. En general pueden almacenar elementos repetidos. Tienen operaciones para añadir y quitar elementos de la estructura y también para preguntar por algún elemento en particular. El arreglo es una estructura lineal que tiene un tamaño fijo (una vez creado no puede crecer ni decrecer). En la materia veremos tres estructuras lineales: Pila, Cola y Lista.

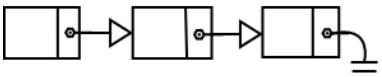


Figura 1.1: Representación visual de una estructura lineal

- *Estructuras jerárquicas*: Los árboles son estructuras de datos jerarquizadas. Cada árbol está constituido por una colección de elementos llamados nodos, uno de los cuales se denomina raíz. Cada nodo tiene una relación de parentesco dentro de la estructura jerárquica (de padres a hijos). Visualmente los árboles tienen la forma que se muestra en la Figura 1.2. Hay distintos tipos de árboles según la cantidad máxima de hijos que puede tener un nodo, si sus elementos se mantienen ordenados o no, etc. En la materia veremos varios tipos de árboles: binarios, genéricos, heap, binario de búsqueda y AVL.

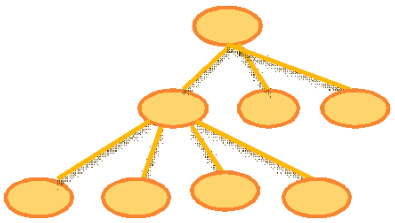


Figura 1.2: Representación visual de una estructura jerárquica o árbol

- *Grafos*: son estructuras de datos en las que se pueden expresar relaciones de conexión sin restricción entre los diversos elementos. Estos elementos son denominados vértices o nodos. Los vértices están unidos por líneas llamadas aristas o arcos. Visualmente los grafos tienen una forma similar a la que se muestra en la Figura 1.3. Los grafos son muy utilizados en problemas reales de conectividad (circuitos analógicos y digitales, caminos o rutas entre localidades o aeropuertos, redes de computadoras, redes sociales, etc). Uno de los problemas más comunes en los grafos es encontrar el camino de costo mínimo o verificar si dos nodos tienen un camino entre sí (por ej., en una web que vende pasajes de avión, un grafo puede ser apropiado para buscar maneras de viajar entre dos ciudades que no tienen vuelos directos).

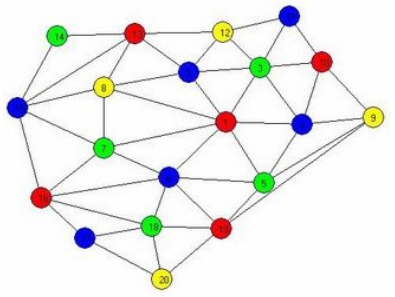


Figura 1.3: Representación visual de una estructura de grafo

1.3. Definición e implementación de un TDA

Al trabajar con estructuras de datos utilizaremos el concepto de TDA tanto para definir a la estructura en sí como para crear y manipular a los datos almacenados en ellas.

Un Tipo de Dato Abstracto (TDA en castellano o ADT en inglés, de Abstract Data Type), representa un conjunto de datos que cumplen ciertas condiciones especificadas por el mismo TDA, más un conjunto de operaciones primitivas que representan el comportamiento del mismo. Es decir que el TDA tiene una identidad (información) y un comportamiento (operaciones), con el cual permite la interacción o comunicación con sí mismo.

Algo muy importante es que todo TDA es independiente del lenguaje en el que se pueda implementar: una vez definido puede implementarse en PASCAL, C, C ++, C#, JAVA, etc. y debería funcionar de igual manera. El TDA facilita el trabajo con tipos de datos, haciendo abstracción de la implementación al momento de definirlo y también cuando éste es utilizado:

- *Abstracción al utilizar un TDA*: Al usar un TDA, sólo se debe saber de qué se trata dicho TDA, es decir qué tipos de datos involucra y cuáles son las operaciones que ofrece. La parte que se da a conocer se llama la INTERFAZ, y está conformada por los nombres, parámetros y resultados de todas sus operaciones o funciones. Cómo dichas operaciones han sido construidas y con qué variables, constantes o estructuras internas cuenta, conforman la IMPLEMENTACIÓN del TDA y siempre son transparentes al usuario. Por ejemplo: la clase `String` de Java es un TDA que se sabe que representa a una cadena de caracteres, que tiene un conjunto de rutinas (funciones/procedimientos) que permiten manipularla (como `length()` que devuelve un `int` que representa la longitud, `charAt(int)` que devuelve un char que está en la posición dada, `indexOf(String)` que devuelve un int que es la posición del substring dado, etc). Sin embargo, los atributos, constantes e implementaciones de los métodos públicos se mantienen ocultos y de forma transparente al usuario final.

- *Abstracción al definir e implementar un TDA*: Un TDA es ante todo un producto software, cuyo usuario será otro programador. Al igual que un programa común, al archivo o archivos necesarios para que el TDA funcione (IMPLEMENTACION) debe adjuntarse la documentación apropiada (INTERFAZ + comentarios de uso). Además, es muy importante que al construir un TDA se comience por tener un buen diseño previo. Para ello se recomienda lo siguiente:

  1. Analizar el problema a plantear (obtener una lista de sus características principales)
  2. Hacer un diagrama analizando la identidad (valores variables y/o constantes, tipos de datos de cada una) y sus operaciones (funciones, procedimientos)
  3. Implementarlo en el lenguaje deseado, haciendo uso de buenas pautas de programación y documentación
  4. Testear el TDA una vez terminado
  5. Volver a rever y ajustar el producto de los pasos anteriores mientras sea necesario

Recomendaciones para diseño e implementación de TDA

- Un TDA tiene que modelar o estar enfocado en resolver un sólo problema cada uno, nunca varios a la vez.

- Las operaciones de un TDA (primitivas) deben ser implementadas con códigos cortos. Siempre se debe modularizar con operaciones ocultas (privadas del TDA).

- Todo TDA tiene que tener una primitiva para su "construcción" y, si el lenguaje lo requiere, otra para su "destrucción"

  - Todo TDA que tenga clave unívoca no puede tener el constructor vacío. Debe existir al menos un constructor a partir de su clave. Luego pueden agregarse otros constructores con la clave y otros datos. Ejemplo: Alumno es identificado por su clave legajo. Debe existir un constructor Alumno(legajo) y opcionalmente pueden haber otros constructores que además del legajo agreguen datos básicos como nombre, apellido, tipo y nro de DNI, fecha de nacimiento, etc.
  - En Java no es necesaria una operación de destrucción dado que utiliza un recolector de basura (Garbage Collector) que se dispara automáticamente y no tiene primitivas para devolver el espacio liberado a la memoria heap [2].

- La información de la entidad (datos) debe estar siempre oculta, excepto por medio de las operaciones que correspondan. Para ello, al implementar en Java, los atributos deben ser siempre privados (private). Además, las operaciones de recuperación (get) y modificación (set) de dichos atributos deben ser diseñadas sin ir en contra de la encapsulación y de la naturaleza del TDA que representan:

[2] Reference: https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html

- *No debe haber método set para los campos clave de un TDA*: Por ejemplo, una instancia de Alumno puede cambiar su nombre y apellido, género, domicilio o teléfono, pero nunca se puede cambiar su número de legajo porque podría ocurrir que en una estructura que los mantiene ordenados por clave, nunca se lo volviera a encontrar. En caso que un campo clave sea erróneo, será necesario quitarlo de la estructura y volver a insertar el dato correcto.
- De forma similar, *no debería haber métodos set para cualquier campo considerado inmutable* (es decir aquellos que no debería ser posible modificar libremente) aún cuando no sean clave. Por ejemplo, el campo número de DNI no debería ser modificado libremente, aún cuando no sea la clave del dato Alumno. Esto debe ser analizado para cada clase del dominio particular.
- *Hay campos para los cuales la operación obtener (get) debería estar vedada.* Por ejemplo, el campo contraseña de un usuario de Facebook o cualquier sistema similar nunca debería ser recuperado mediante un método get. En cambio, podría existir una operación en el TDA llamada `esPasswordCorrecto(clave)` que devolviera true o false según corresponda.

- Las primitivas del TDA tienen que permitirle al usuario manipularlo sin dar información sobre la implementación del TDA (ocultamiento de la Información). El ocultamiento de información al implementar las operaciones primitivas, se denomina encapsulamiento y es fundamental.
- El testeo del TDA tiene que ser 100 % exitoso.

Ejercicio 1.1: Diseño, especificación e implementación de TDA

1. TDA Alumno
   a) Diseñar el TDA considerando campo clave legajo y como información asociada los datos nombre, apellido, tipo y número de DNI, domicilio (calle, número y ciudad), teléfono, usuario y clave en el sistema SIU. Especifique las operaciones primitivas de acuerdo a las recomendaciones anteriores.
   b) Implemente el TDA Alumno para el lenguaje Java, de acuerdo a la especificación anterior.
2. TDA Fecha
   a) Diseñe el TDA Fecha con clave compuesta día, mes y año. Analice y especifique las operaciones básicas del TDA.
   b) Implemente el TDA Fecha especificado en 2a) con tres atributos enteros que representen dia, mes y año por separado.
   c) Implemente el TDA Fecha especificado en 2a) con un único atributo interno String con el formato "AAAAMMDD" donde AAAA es el año expresado con cuatro dígitos, MM y DD son mes y día expresados con dos dígitos cada uno.

1.4. Eficiencia de algoritmos

Uno de los temas principales al estudiar las estructuras de datos es ser capaces de calcular la eficiencia de sus operaciones. Para ello, utilizaremos los conceptos de *eficiencia de algoritmos*, vistos en la materia Desarrollo de Algoritmos. En dicha materia, se ha visto que el análisis de la eficiencia de algoritmos es una estimación teórica de los recursos de cómputo que necesita un algoritmo para resolver un problema computacional (Knuth). Aunque puede haber muchos parámetros, los más usuales (y los que utilizaremos en la materia) son el *tiempo de ejecución* y la *cantidad de memoria utilizada*. El cálculo de la cantidad de memoria es bastante intuitivo (por ejemplo, un arreglo de 2000 elementos ocupará más espacio que uno de 100). Por otro lado, el tiempo de ejecución puede no ser simple de calcular, pero se trata de un aspecto crucial de la eficiencia del software. Si, por ejemplo, un problema se resuelve en un tiempo exponencial (k^n), dicho algoritmo en una computadora con un ejemplar de tamaño n=10 puede tardar una décima de segundo. El mismo algoritmo necesitará casi 10 minutos para resolver un ejemplar de tamaño 20. Un día entero no bastaría para resolver uno de tamaño 30 y en un año de cálculo ininterrumpido, a duras penas resolvería uno de tamaño 38. Esto nos da una idea de la importancia de conocer este tema al programar.

A continuación repasaremos los conceptos necesarios para medir el tiempo de ejecución.

1.4.1. Análisis del tiempo de ejecución

Para calcular el tiempo de ejecución siempre partimos de la base que el algoritmo es correcto, es decir, que produce el resultado deseado en un tiempo finito. Para comparar dos algoritmos correctos que resuelven el mismo problema, se determina su eficiencia de acuerdo a una medida de tamaño N, que representa la cantidad de datos que manipulan dichos algoritmos.

Para comparar los algoritmos se puede utilizar una estrategia empírica o teórica:

- La *estrategia empírica* consiste en programar los algoritmos y ejecutarlos en una computadora sobre varios casos de prueba, y comparar los tiempos reales de ejecución.
- La *estrategia teórica* consiste en determinar matemáticamente la cantidad de recursos (tiempo, espacio, etc) que necesitará el algoritmo en función de la cantidad de datos manipulados por el algoritmo. Esta estrategia tiene la ventaja de que no depende de la computadora ni del lenguaje de programación utilizado. Además, evita el esfuerzo de programar algoritmos ineficientes y de desperdiciar tiempo de máquina para ejecutarlos. También permite conocer la eficiencia de un algoritmo para cualquier tamaño de entrada.

La estrategia que utilizaremos en esta materia es la teórica. Para ello, el tiempo de ejecución de un algoritmo se expresará como una función de la longitud de entrada en relación con un número de pasos a realizar. Así, el tiempo de ejecución de un programa será una función de una cantidad n de elementos, que se la denomina *T(n)*. Estas estimaciones se analizan para un tamaño n muy grande, generalizando decimos que estudiamos a n cuando tiende a infinito.

En las sentencias condicionales (alternativas y repetitivas de tipo mientras o hasta) los algoritmos dependen de valores concretos, y generalmente se pueden diferenciar el "mejor caso", el "peor caso" y el "caso promedio".

- *Mejor caso*: $T_{mejor}(n)$, es el menor de los tiempos de ejecución sobre todas las entradas de tamaño n. Puede dar lugar a errores en algoritmos lentos que trabajan rápido sobre unas pocas entradas. Por ejemplo, en el método de ordenamiento burbuja mejorado, el mejor caso es cuando el arreglo ya está ordenado y en una sola pasada termina sin hacer ningún intercambio.
- *Caso promedio*: $T_{promedio}(n)$, es el promedio de los tiempos de ejecución sobre todas las entradas de tamaño n. Es el más fiel, pero puede ser muy difícil de determinar. En el ejemplo del método burbuja mejorado, el caso promedio es cuando el arreglo está aleatoriamente desordenado.
- *Peor caso*: $T_{peor}(n)$, es el máximo de los tiempos de ejecución sobre todas las entradas de tamaño n, puede no ser muy fiel, pero consideramos que es una cota superior del tiempo promedio. En el mismo ejemplo del método burbuja mejorado, el peor caso es cuando el arreglo está ordenado de manera inversa a la deseada. En este caso se produce la mayor cantidad de comparaciones e intercambios.

Como el análisis debe realizarse con independencia de los valores que determinan las sentencias condicionales, en general se prefiere el cálculo más pesimista, es decir el "peor caso". Para ello se evalúa la estructura del algoritmo bajo la suposición de una memoria infinita (es decir que no habrá demoras por intercambio de información entre memoria primaria y secundaria).

Análisis del tiempo de ejecución según el tipo de sentencia

- *Instrucciones simples*: Consideraremos que las operaciones de asignación, impresión en pantalla, operaciones lógicas (and, or, not, $<$, $>$, $=$), operaciones matemáticas simples ($+$,-,*,/, %) y los accesos a un elemento de un arreglo se cumplen en un tiempo 1. Este tiempo es independiente de las máquinas y de los lenguajes utilizados.

Ejemplo:

```
(1) a = 1
(2) b = 2*y + 3
(3) c = arr[i] * 9
```

$T_1 = 1\, asig = 1$
$T_2 = 1\, asig + 2\, op.mát = 3$
$T_3 = 1\, asig + 1\, op.mát + 1\, acc.arreglo = 3$

- *Secuencias*: Sea un algoritmo donde hay una sentencia S1 seguida por una o más sentencias S2, el tiempo del bloque completo es la suma de los tiempos de cada bloque que lo compone.

Ejemplo

```
Algoritmo x
 (1) a = 1
 (2) b = 2*y + 3
 (3) c = arr[i] * 9
Fin algoritmo
```

$T_{secuencia} = T_1 + T_2 + T_3$

T_1, T_2 y T_3 ya fueron calculados en el ejemplo anterior, luego:

$T_{secuencia} = T_1 + T_2 + T_3 = 1 + 3 + 3 = 7$

- *Alternativas*: El tiempo de ejecución de la alternativa nunca es más grande que el tiempo empleado por la evaluación de la condición más el mayor de los tiempos de S1 y S2.

Definición

```
si cond hacer
   S1
sino
   S2
fin si
```

$$T_{alternativa} = T_{cond} + max(T_1, T_2)$$

Ejemplo

```
si i+1 <= 2*p
   a[i+1] = 2*i+5
sino
   exito = falso
fin si
```

$$T_{cond} = 2\,op.mát + 1\,op.lóg = 3$$
$$T_1 = 1\,asig + 3\,op.mát + 1\,acc.arreglo = 5$$
$$T_2 = 1\,asig = 1$$
$$T_{alternativa} = T_{cond} + max(T_1, T_2) = 3 + max(5, 1) = 3 + 5 = 8$$

- *Repetitivas*: Al evaluar el tiempo de una repetitiva tendremos en cuenta el tipo de la misma:

  - Repetitiva *PARA*: se tiene en cuenta el tiempo de la inicialización de la variable de control (T_{ini}), luego por cada repetición del ciclo ($\sum_{min}^{max}$), donde min es el valor de inicialización y max es el último valor para el que la condición da true; sumamos el tiempo para evaluar la condición (T_{cond}), el tiempo del cuerpo de sentencias interno del bucle (T_{int}), y el tiempo del incremento de la variable de control (T_{incr}). Además consideramos que la evaluación de la condición (T_{cond}) se ejecuta una vez más que las que se ejecuta el bloque interno (cuando la condición es falsa).

Definición

```
para var=min hasta max
   bloque interno
fin para
```

$$T_{for} = T_{ini} + \sum_{min}^{max}(T_{cond} + T_{int} + T_{incr}) + T_{cond}$$
$$T_{for} = T_{ini} + (max - min + 1) * (T_{cond} + T_{int} + T_{incr}) + T_{cond}$$

Ejemplo en Java

```
for (i=3; i<9; i++){
    a[i]=a[i+1];
}
```

$$T_{ini} = 1\,asig = 1$$
$$T_{cond} = 1\,op.lóg = 1$$
$$T_{incr} = 1\,op.mát = 1$$
$$T_{int} = 1\,asig + 2\,acc.arr + 1\,op.mát = 4$$
$$T_{for} = T_{ini} + \sum_{min}^{max}(T_{cond} + T_{int} + T_{incr}) + T_{cond} = 1 + \sum_{3}^{8}(1+4+1) + 1 =$$
$$= 1 + (8 - 3 + 1) * (1 + 4 + 1) + 1 = 1 + 6 * 6 + 1 = 38$$

  - Repetitiva MIENTRAS: consideramos la cantidad de repeticiones (las veces que la condición da true); por cada repetición sumamos el tiempo para evaluar la condición (T_{cond}) y el tiempo del cuerpo de sentencias interno del bucle (T_{int}), y consideramos que la evaluación de la condición (T_{cond}) se ejecuta una vez más que las que se ejecuta el bloque interno (cuando la condición es falsa).

Definición

```
mientras cond
   bloque interno
fin mientras
```

$$T_{mientras} = cant_{rep}(T_{cond} + T_{int}) + T_{cond}$$

Ejemplo en Java

```
while (i<n){
    a[i]=a[i+1];
    i=i+1;
}
```

$$T_{cond} = 1op.lóg = 1$$
$$T_{int1} = 1asig + 2acc.arr + 1op.mát = 4$$
$$T_{int2} = 1asig + 1op.mát = 2$$
$$T_{int} = T_{int1} + T_{int2} = 4 + 2 = 6$$
$$T_{mientras} = cant_{rep}(T_{cond} + T_{int}) + T_{cond} = n * (1 + 6) + 1 = 7n + 1$$

- Repetitiva HASTA: consideramos la cantidad de repeticiones (las veces que la condición da true); por cada repetición sumamos el tiempo para evaluar la condición (T_{cond}) y el tiempo del cuerpo de sentencias interno del bucle (T_{int}). A diferencia del *mientras* y el *para*, esta sentencia evalúa la condición sólo la cantidad de veces que se ejecuta el cuerpo interno.

Definición

```
repetir
   bloque interno
hasta cond
```

$T_{hasta} = cant_{rep}(T_{cond} + T_{int})$

Ejemplo en Java

```
do {
    a[i]=a[i+1];
    i=i+1;
} while i<=n;
```

T_{cond}= 1 op.lóg = 1
$T_{int} = T_{int1} + T_{int2} = 4 + 2 = 6$
$T_{hasta} = cant_{rep}(T_{cond} + T_{int}) = n * (1 + 6) = 7n$

Orden de eficiencia (notación asintótica)

Los algoritmos sobre cantidades de datos pequeñas se pueden resolver sin mucho cuidado pero las limitaciones aparecen al trabajar con cantidades de datos grandes. Matemáticamente hablando, cuando N tiende a infinito. Es decir, lo que nos interesa conocer es el *comportamiento asintótico* de la función del tiempo de ejecución. El conjunto de funciones que comparten un mismo comportamiento asintótico comparten un *orden de complejidad*.

El orden de complejidad lo escribimos O(f(n)) donde f(n) representa la velocidad de crecimiento de la función del tiempo de ejecución.

Para calcularlo podemos hacerlo de dos maneras:

- *A partir de la función de tiempo de ejecución*: cuando se conoce la función del tiempo de ejecución, se aísla el término que crece más rápido, despreciando las constantes que lo multiplican. Ejemplo, si $T(n) = 4n^2 + 5n + 8$ su orden es $O(n^2)$.
- A partir del código (o pseudocódigo), simplificando las sentencias de tiempo constante (orden O(1)) y aplicando las siguientes reglas: Sean $T_1(n)$ y $T_2(n)$ las funciones que expresan los tiempos de ejecución de dos fragmentos de un programa, y sus órdenes $T_1(n) \epsilon O(f_1(n))$ y $T_2(n) \epsilon O(f_2(n))$
  - *Regla de la suma*: El orden de la suma de dos bloques de código secuenciales es el máximo de sus órdenes.

$$O(f_1(n)) + O(f_2(n)) = O(máx(f_1(n), f_2(n)))$$

  - *Regla del producto*: El orden del producto de ambos bloques es el producto de sus órdenes.

$$O(f_1(n)) * O(f_2(n)) = O(f(n) * f_2(n))$$

Jerarquía de ordenes de complejidad

Mencionamos distintos órdenes de complejidad O(f(n)) de acuerdo a una jerarquía. Si un algoritmo A se puede demostrar de un cierto orden O, es cierto que también pertenece a todos los órdenes superiores; pero en la práctica lo útil es encontrar la "menor cota superior", es decir el menor orden de complejidad que lo incluya. En orden de mayor a menor eficiencia, son:

- $O(1)$: orden constante
- $O(logn)$: orden logarítmico
- $O(n)$: orden lineal
- $O(n * logn)$
- $O(n^2)$: orden cuadrático
- $O(n^a)$: orden polinomial (a >2)

- $O(a^n)$: orden exponencial (a >2)
- $O(n!)$: orden factorial

Los algoritmos de complejidad $O(n)$ y $O(n * log\, n)$ son los que muestran un comportamiento más natural, prácticamente a doble cantidad de datos procesados, doble de tiempo necesario. Los algoritmos de complejidad logarítmica son excelentes pues para resolver un problema el doble de grande sólo hace falta un poco más de tiempo. Sobre los algoritmos de tipo polinómico, mientras complejidades del orden $O(n^2)$ y $O(n^3)$ suelen ser aceptables, nadie acepta algoritmos de orden mayor. Finalmente, cualquier algoritmo por encima de una complejidad polinómica se dice "intratable" y sólo será aplicable a problemas muy pequeños.

Conclusión

Antes de encarar la codificación de un programa se debe analizar y elegir un buen algoritmo. La elección de las estructuras de datos también influyen en la eficiencia de los programas que desarrollamos, por ello debemos ser capaces de elegir la estructura más adecuada para cada problema. Por bueno o adecuado entendemos que utilicen el mínimo de recursos; siendo los más importantes el tiempo de ejecución y la cantidad de memoria necesaria. En el análisis de algoritmos consideraremos siempre el peor caso o trataremos de estimar un caso promedio. En el caso de las estructuras analizaremos el tipo de acceso que proveen y la cantidad de accesos necesarios para cada tarea, así como la memoria que utilizan. En general, analizaremos en forma conjunta tanto la estructura de datos a utilizar como los algoritmos relacionados. Para independizarse de factores tales como el lenguaje de programación, la habilidad del codificador, la máquina soporte, etc. trabajaremos con el cálculo asintótico que indica como se comporta el algoritmo para datos muy grandes o mucha cantidad de información. Para problemas pequeños es cierto que casi todos los algoritmos son "más o menos iguales", primando otros aspectos como esfuerzo de codificación, legibilidad, etc.

Ejercicio 1.2: Cálculo de tiempo y orden de ejecución

1. Calcular el tiempo de ejecución de los métodos listados a continuación:

```
private static int sumaSm3(int[] a){
     int n, estaSuma, sumaMax, j;
     n = a.length;
     sumaMax = 0;
     estaSuma = 0;

     for (j= 0; j<n; j++){
             estaSuma = estaSuma + a[j];
             if (estaSuma > sumaMax)
                    sumaMax = estaSuma;
             else
                    if (estaSuma < 0)
                           estaSuma = 0;
     }

     return sumaMax;
}

 private static int sumaSm1 (int[] a){
      int n, estaSuma, sumaMax, i, j;
      n = a.length;

      sumaMax= 0;
      for (i = 0; i < n; i++){
              estaSuma = 0;
              for (j = i;  j < n; j++){
                     estaSuma = estaSuma + a[j];
                     if (estaSuma > sumaMax)
                            sumaMax = estaSuma;
              }
      }
      return sumaMax;
 }

private static void misterioII( int n, int[] a){
        //n es la dimensión del arreglo a
        int i;

        i= 1;
        while (i <= n){
               if (a[i] >= a[n])
                      a[n] = a[i];
               i= i * 2;
        }
}
```

2. Calcular el orden de ejecución de los mismos métodos, utilizando las reglas de la suma y del producto, y verificar que concuerdan con el cálculo a partir de la función del tiempo de ejecución.

1.5. Recursividad

La recursividad es un método usual de simplificación de un problema complejo, mediante la división de este en subproblemas del mismo tipo. Esta técnica de programación, conocida como *divide y vencerás*, es el núcleo del diseño de algoritmos muy eficientes, como el de búsqueda binaria, y los métodos de ordenamiento quicksort y mergesort, entre otros.

En las estructuras de datos jerárquicas y de tipo grafo, la manera más adecuada de recorrerlas (y a veces la única manera posible) es utilizando procedimientos recursivos.

A continuación repasaremos los conceptos más importantes:

- Un *algoritmo recursivo* es un algoritmo que expresa la solución de un problema en términos de *una llamada a sí mismo*. La llamada a sí mismo se conoce como llamada recursiva o recurrente.
- Generalmente, si la primera llamada al subprograma se plantea sobre un problema de tamaño N, cada nueva ejecución recurrente se planteará sobre problemas de igual naturaleza que el original, pero de un

tamaño menor que N. De esta forma, al ir reduciendo progresivamente la complejidad del problema a resolver, llega un momento en que su resolución es más o menos trivial (o, al menos, suficientemente manejable como para resolverlo de forma no recursiva). En esa situación diremos que estamos ante un *caso base* de la recursividad.

- A veces, los algoritmos recursivos pueden ser más ineficientes en cuanto al tiempo de ejecución que los iterativos, pero por lo general son mucho más sencillos (cortos en líneas de código).

Las claves para construir un algoritmo recursivo son:

- *Descomposición*: Cada llamada recursiva se debe definir sobre un problema de menor complejidad (más fácil de resolver).
- *Composición*: Definir cómo se combinan las soluciones de él o los problemas más pequeños, para lograr la solución del problema original.
- *Caso base*: Debe existir al menos un caso base para evitar que la recurrencia sea infinita.

El ejemplo del cálculo recursivo del *factorial* de un número es el ejemplo más habitual de resolución recursiva:

- Definición del factorial de un numero n: n! = producto de los n primeros números naturales.
- Solución iterativa: n! = 1 * 2 * ... * (n-1) * n
- Solución recursiva: n! = n * (n-1)!

El algoritmo recursivo podemos expresarlo de la siguiente forma:

- *Si n <=1 devolver 1*
- *sino devolver n*(n-1)!*

En la Figura 1.4 se muestra el algoritmo codificado en Java, indicando las partes claves del mismo: en el caso recursivo detectamos la descomposición del problema en un problema más pequeño, que se combina al multiplicarlo por n a la vuelta de la recursión. Luego, es preciso que exista el caso base que permitirá terminar la sucesión de llamadas recursivas.

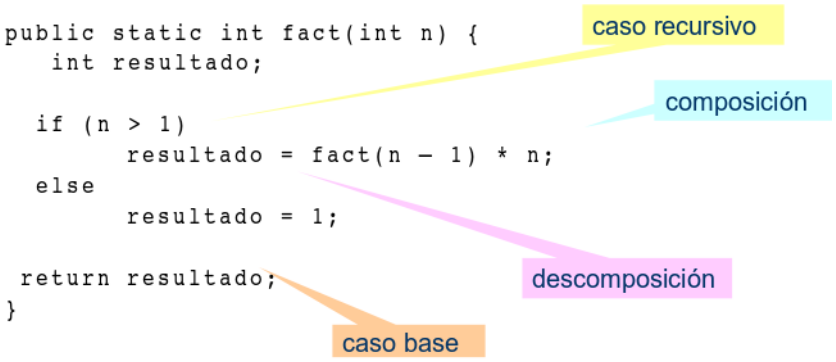


Figura 1.4: Análisis del algoritmo recursivo para cálculo del factorial(n)

1.5.1. Traza de algoritmos recursivos con una sola invocación

Un aspecto fundamental al programar algoritmos recursivos, es entender paso a paso la ejecución de los mismos y ser capaces de graficar la pila de ejecución. En la Figura 1.5 se observa la traza para la invocación fact(5). En la parte superior de la figura se observa el crecimiento de la pila de ejecución, indicando en un piso nuevo cada llamada recursiva. Con el símbolo "?" se indican los puntos donde ha quedado pendiente la combinación del resultado esperado. En la parte inferior de la figura se observa la acción "a la vuelta" de la recursión, donde el valor retornado por cada piso de la pila es utilizado en el piso inferior para resolver la combinación pendiente y emitir el nuevo resultado.

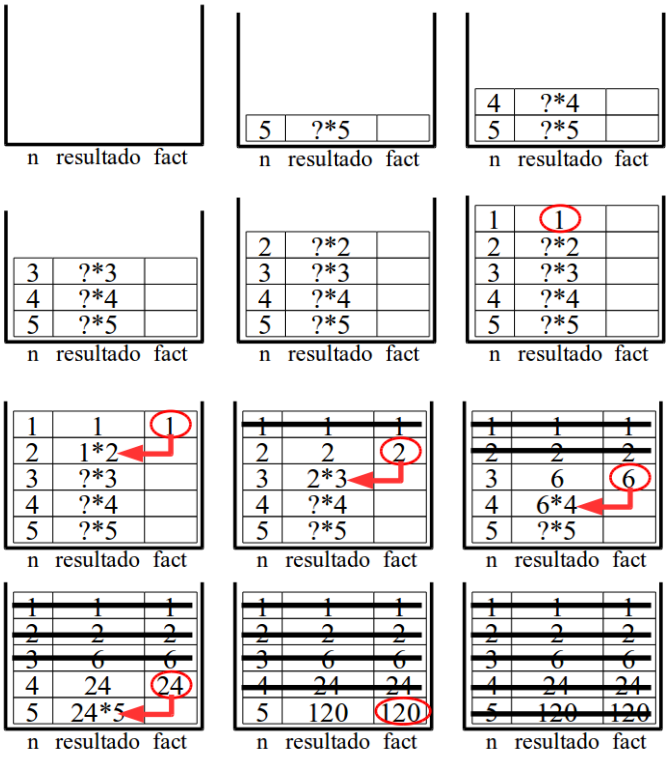


Figura 1.5: Traza de pila recursiva para fact(n)

Para realizar la traza con pila de ejecución, se dibuja un nuevo piso al comenzar la ejecución de la llamada recursiva, luego se agrega una columna para cada parámetro y cada variable local de tipo simple. Para las variables de tipo objeto (como arreglo en Java) se lo dibuja fuera de la pila, dado que el contenido de un objeto es visible de igual manera en todos los pisos que lo referencian.

Como ejemplo, veamos en la Figura 1.6 un método recursivo que busca un elemento en un arreglo y devuelve su posición. Si no lo encuentra, devuelve -1 (que es una posición inválida). Si analizamos el método podemos reconocer dos casos base: (1) cuando encuentra el elemento en el arreglo y (2) cuando buscándolo llega más allá del límite, es decir, cuando el elemento no está; luego existe una llamada recursiva cuando el elemento visitado no es el buscado. La composición consiste en la asignación del valor de la llamada recursiva a la variable de salida.

```
public static int buscarElem(String buscado, String[] arr, int pos){
        //busca el elemento desde la posicion pos en adelante
        int salida;

        if (pos>=arr.length)
                // caso base 2: se termino el arreglo
                salida = -1;
        else
                if (arr[pos].compareTo(buscado) == 0)
                        // caso base 1: encontro el elemento
                        salida = pos;
                else
                        // caso recursivo: buscar en la siguiente posicion
                        salida = buscarElem(buscado,arr,pos+1);
        return salida;
}
```

Figura 1.6: Algoritmo recursivo que busca un elemento en un arreglo

En la Figura 1.7 se muestra la traza para la invocación buscarElem(“tres”, arr, 0). En la parte superior de la figura se observa paso a paso el crecimiento de la pila de ejecución. Como se puede ver, se ha dibujado una columna para cada parámetro (buscado, arr, pos) y variable local (salida). Para el arreglo, al ser de tipo objeto, se indica que la variable apunta o referencia al objeto *arr* que todos los pisos comparten. De esta manera, un cambio del contenido de *arr* en cualquiera de los pisos de la pila, será visible en todos los otros que referencien al mismo objeto. En la parte inferior de la figura se observa la acción “a la vuelta” de la recursión, donde el

valor retornado por cada piso de la pila es utilizado en el piso inferior para actualizar la propia variable local salida y luego emitir el nuevo resultado.

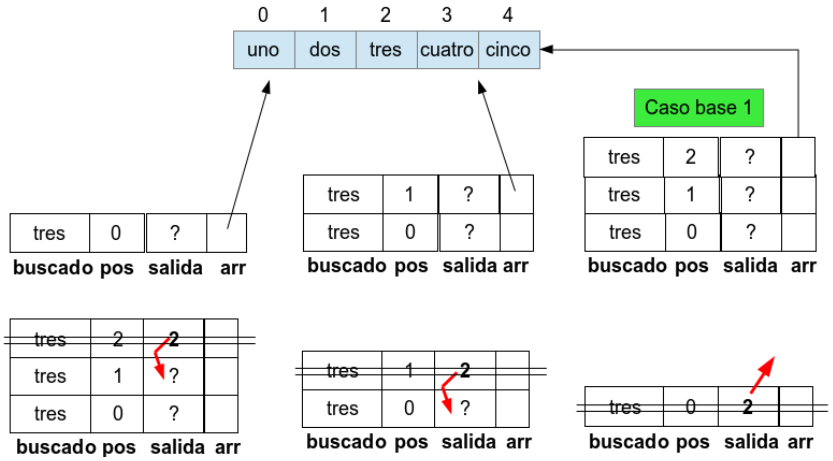


Figura 1.7: Traza del algoritmo recursivo que busca un elemento en un arreglo

1.5.2. Traza de algoritmos recursivos con dos invocaciones

En matemáticas, la sucesión o serie de Fibonacci es la siguiente sucesión infinita de números naturales: 0,1,1,2,3,5,8,13,21,34,55, ...

La sucesión comienza con los números 0 y 1, y luego se calcula cada término como "la suma de los dos términos anteriores". La fórmula de la sucesión se define recursivamente de la siguiente manera:

- f(0) = 0
- f(1) = 1
- f(n) = f(n-1) + f(n-2) cuando n >1

Generar la sucesión de Fibonacci es un típico ejemplo de recursividad con más de una invocación. El código en Java se muestra en la Figura 1.8. Este ejemplo tiene la particularidad que para realizar la traza es necesario saber claramente desde qué punto del módulo se ha invocado cada vez. Para ello se ha nombrado a la primera invocación con la letra A y a la segunda con la letra B.

```
public static int fib(int n) {
  int res;
  if (n <= 1)
     res = n;
  else
     res = fib(n-1) + fib(n-2);
           A        B
  return res;
}
```

Figura 1.8: Código de la función fib(n)

En la Figura 1.9 se observa la traza para la invocación fib(4). De manera similar, en la fila (1) se observa el crecimiento de la pila de ejecución, indicando en un piso nuevo cada llamada recursiva. Con el símbolo "?" se indican los puntos donde ha quedado esperando, pendiente del resultado de la invocación. Las letras A y B se ponen a la izuierda de cada piso de la pila para marcar desde dónde se realizó la llamada, para volver exáctamente al punto del programa que debe resolverse. En el caso del piso de más abajo, que representa la invocación original, no se le pone letra porque se sabe que es desde un módulo externo. A partir de la fila (2) en adelante, cuando termina de calcular la primera invocación (A), llama desde la segunda (B), produciendo que la pila crezca y decrezca hasta cumplir con todas las llamadas necesarias.

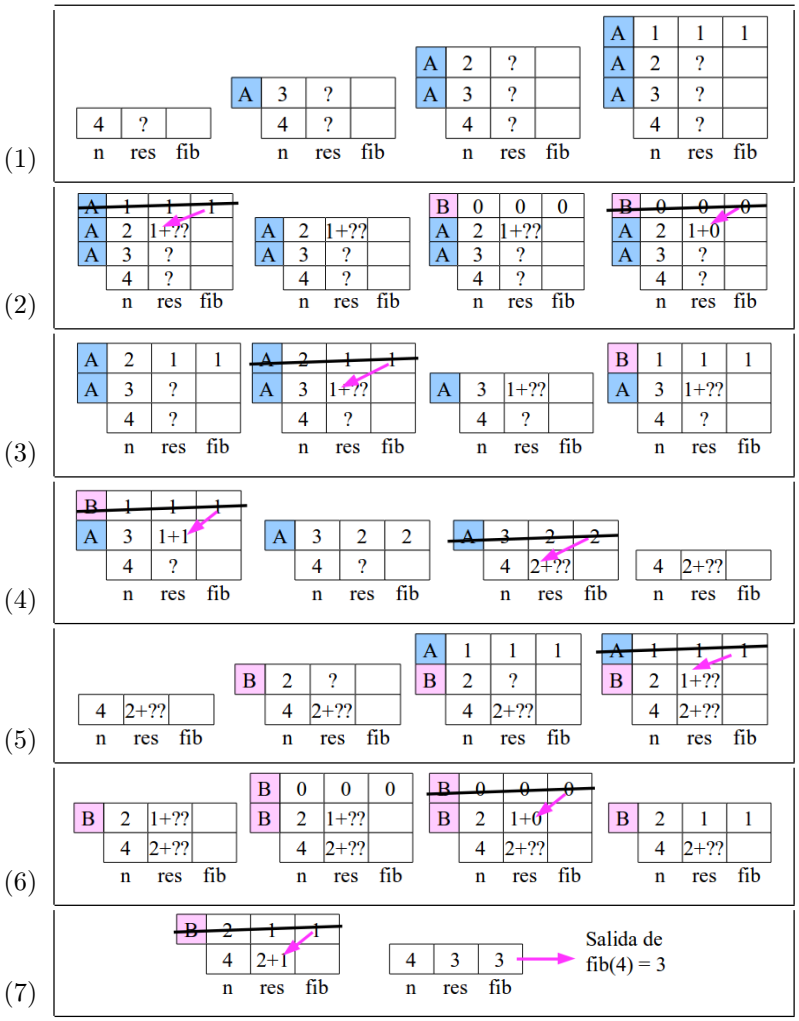
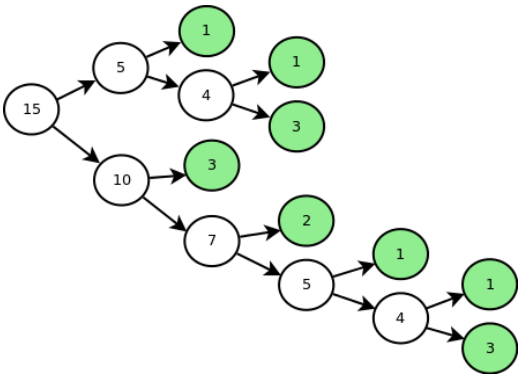


Figura 1.9: Traza de pila recursiva para fib(4)

Ejercicio 1.3: Recursividad

1. Siguiendo el código de la Figura 1.4 realizar una traza con pila para calcular el valor del factorial de 4.
2. Siguiendo el código de la Figura 1.6 realizar una traza con pila para buscar el elemento "seis" en el arreglo.
3. Siguiendo el código de la Figura 1.8 realizar una traza con pila para calcular el valor del término 5 de la sucesión de Fibonacci.
4. Escriba una subrutina recursiva para cada inciso siguiente, considerando que todas reciben un String por parámetro:
   a) Devuelva la primera posición en la que aparece una letra dada.
   b) Devuelva la última posición en la que aparece una letra dada.
   c) Devuelva un String que sea la cadena de caracteres original seguida de su inversa.
5. Diseñe un lote de pruebas apropiado y haga trazas con pilas para asegurar la correctitud de cada subrutina recursiva del ejercicio anterior.
6. Dado un arreglo de números enteros, implementar un método recursivo que en un único recorrido imprima los elementos menores que el promedio de todos los números cargados. *Por ejemplo:* dado el arreglo [10,5,3,7,11,1] el promedio es (10+5+3+7+11+1)/6=37/6=6.166, luego debe imprimir "1, 3, 5"
   (*) Pista: calcular el promedio a la ida de la recursión e imprimir los elementos menores a la vuelta.
7. Diseñe un lote de pruebas apropiado y haga trazas con pila para asegurar la correctitud del ejercicio anterior.
8. **Ejercicio adicional***: El problema de los números explosivos*:
   Dados dos números n y b, tales que b <n, podemos hacer que n explote usando a b como bomba. Cuando n explota se parte en dos números n1 = (n div b) y n2 = n - (n div b). Pero b produce una reacción en cadena: si n1 (y/o n2) es mayor que b, también explota y se parte nuevamente en dos pedazos, según el criterio anterior. Esto se repite hasta que todos los pedazos resultantes a partir de n sean menores o iguales que b. Ejemplo: si n = 15, y la bomba b = 3, el número n se parte inicialmente en dos: n1=15 div 3, y n2=15-(15 div 3), es decir, 5 y 10. Como ambos son mayores que la bomba, deben estallar. El proceso se repite según lo muestra la figura.



   a) Escribir un método recursivo *explotar*, que dado un número n y un número bomba b, imprima los pedazos que quedan al explotar n usando b. (En el caso que muestra la figura debe imprimir 1,1,3,3,2,1,1,3)
   b) Realice la traza con pila para el ejemplo de la figura.