

Guía de Lectura

Análisis de Complejidad

Capítulos 1, 2, 3 y 4, y Apéndice A de [Cormen et al. 01].

Los Caps. 1 y 2 presentan una introducción al área de diseño de algoritmos, que incluye el análisis de complejidad de éstos. Para ejemplificar complejidad presenta los algoritmos de ordenamiento mediante inserción y mediante mezcla.

Los Caps. 3 y 4 contienen, respectivamente, notación asintótica y manejo de recurrencias. Para hacer buen análisis de complejidad, además de estos dos tópicos es muy conveniente manejar cómodamente muchas propiedades de sumatorias, que están en el Apéndice A. Hay bastante información, que puede resultar un poco pesada, pero todo computista que se precie de serlo debe pasar por una buena lectura de estos tres tópicos.

Búsqueda

Capítulo 4 de [Hume et al. 99], a excepción de las secciones 4.6 y 4.7.

En este libro, como su nombre claramente lo indica, los programas son presentados directamente en Java (lo cual puede resultar agradable/desagradable para algunos lectores).

Este capítulo introduce además al uso de especificaciones de problemas algorítmicos, empezando por versiones informales y terminando en las formales. También se explica brevemente cómo demostrar correctitud. Todo esto puede resultar un buen repaso de especificaciones y correctitud, siendo además positivo el ver presentaciones alternas de conceptos ya conocidos.

Ordenamiento

Capítulos 4 y 15 de [Hume et al. 99], a excepción de las secciones 15.6 y 15.7, y capítulos 6 y 7 de [Cormen et al. 01].

Empezamos con los algoritmos sencillos de orden cuadrático: Inserción, Selección y Burbuja. Las secciones 4.6, 4.7 y 4.8 de [Hume et al. 99] desarrollan Burbuja y analizan su complejidad. Luego, en las secciones 15.1 y 15.2, se presentan Inserción y Selección, y se repasa Burbuja. Inserción ya lo habíamos conocido, además, en la introducción a análisis de complejidad (Cap. 2 de [Cormen et al. 01]).

Seguimos con los algoritmos lineal-logarítmicos: Mezcla, “HeapSort” (ordenamiento mediante montículos) y “QuickSort” (ordenamiento “rápido”). Al inicio del trimestre sólo veremos Mezcla, y dejaremos “HeapSort” y “QuickSort” para el final.

Mezcla es presentado en [Hume et al. 99], sección 15.5, y ya lo conocíamos de la introducción a complejidad (Cap. 2 de [Cormen et al. 01]).

“HeapSort” y “QuickSort” son presentados tanto en [Hume et al. 99], secciones 15.3 y 15.4, como en [Cormen et al. 01], capítulos 6 y 7. La presentación en pseudo-lenguaje de [Cormen et al. 01] resulta más legible que la de [Hume et al. 99], en la que los detalles de Java distraen.

Para todos los algoritmos de ordenamiento, la sección 15.8 de [Hume et al. 99] agrega herramientas de visualización. Esto puede ser del agrado de quienes disfrutan de la programación y el uso de objetos gráficos.

Abstracción de Datos

Capítulos 1, 2 y 4 de [Mitchell 92], y capítulos 1, 3, 5 y 9 de [Liskov & Guttag 01].

Empezando con [Mitchell 92], su Cap. 1 introduce al uso de TADs (Tipos Abstractos de Datos) como herramienta de modularización de programas y, más aún, como herramienta de diseño OO (Orientado a Objetos). El Cap. 2 introduce especificaciones *algebraicas* de TADs, y el Cap. 4 explica cómo llevar estas especificaciones a realizaciones concretas mediante *módulos*.

[Mitchell 92] utiliza como lenguaje de especificación algebraica a OBJ, y como lenguaje de programación para construir módulos a Modula-2.

En cambio, [Liskov & Guttag 01] no utiliza especificaciones algebraicas para los TADs, sino especificaciones *basadas en modelos* (aunque modelos informales; en las clases de teoría utilizaremos modelos formales). Estas especificaciones las lleva a realizaciones concretas en el lenguaje Java, no en módulos, sino en *clases* (aunque éstas pueden ser vistas en realidad como una especie particular de módulos).

En [Liskov & Guttag 01], el Cap. 1 también se refiere a modularización de programas, no sólo mediante abstracción de datos, sino también mediante abstracción procedimental (tema ya conocido desde Algoritmos y Estructuras I). El Cap. 3 se refiere a abstracción procedimental, introduciendo el estilo de especificación utilizado por el libro. Se presentan muchas consideraciones interesantes sobre especificación e implementación de procedimientos, que deben ser revisadas adecuadamente aunque el tema suene a conocido y “cuento viejo” de cursos previos.

El Cap. 5 se dedica ya específicamente a abstracción de datos. Además de introducir al uso de TADs y a la implementación de TADs (siempre todo en Java), presenta también las nociones de función de abstracción e invariante de representación (que en las clases de teoría analizaremos aún más formalmente bajo el tema de Refinamiento de Datos).

Por último, el Cap. 9 diserta sobre la importancia de las especificaciones en general, tanto para abstracción procedimental como para abstracción de datos.

Intermezzo: Java

Capítulos 2 y 4 de [Liskov & Guttag 01].

En vista de que Java será utilizado (aunque sólo parcialmente) en el laboratorio de este curso, y dado que [Liskov & Guttag 01] presenta todos los conceptos apoyándose en ejemplos construidos en Java, vale la pena aprender más sobre este lenguaje.

El Cap. 2 toca aspectos generales importantes de Java, como por ejemplo el manejo interno de memoria que se realiza para almacenar variables y objetos. El Cap. 4 presenta el importantísimo tema de manejo de excepciones, que permite considerar y controlar la aparición de casos de error durante la ejecución de un programa.

Modelos Abstractos de Representación

Capítulo 9 de [Morgan 94].

La construcción de especificaciones de TADs *basadas en modelos* requiere el uso de algunas estructuras matemáticas que sirvan como modelos formales de representación. Para esto utilizaremos: conjuntos, multiconjuntos, secuencias, funciones y relaciones.

El estudio, con mayor o menor profundidad, de estas cinco nociones es objetivo de los cursos de Estructuras Discretas. En nuestro caso, sólo nos interesa utilizarlas como modelos abstractos de representación de TADs y, por lo tanto, nos basta una presentación resumida como la ofrecida por el Cap. 9 de [Morgan 94].

Tipos Concretos de Datos (o Modelos Concretos de Representación)

Capítulos 1 y 4 de [Wirth 76], específicamente las secciones 1.7 y 1.8, y las secciones 4.1 y 4.2.

Como tipos concretos de datos, primero contamos con los arreglos y los registros. Los primeros son ya viejos conocidos de Algoritmos y Estructuras I. Los segundos son presentados en la sección 1.7. En la siguiente sección 1.8 se introducen además los *registros variantes*; más adelante en el curso veremos que tales tipos se pueden construir mediante *herencia de clases* en un lenguaje de programación orientado a objetos (como Java).

Luego de arreglos y registros, tenemos las referencias, también llamadas apuntadores. El principal uso que daremos a las referencias será el de construir tipos recursivos, aunque éstas pueden ser utilizadas con muchos otros fines. La sección 4.1 explica qué es un tipo recursivo. Luego, la sección 4.2 explica qué es una referencia, o apuntador, e indica cómo usar éstas para construir tipos recursivos.

Por último, como tipos concretos de datos también utilizaremos los llamados tipos algebraicos libres. Mediante éstos construiremos de una manera más limpia a los tipos recursivos. La referencia de estudio de tipos algebraicos libres se presenta más adelante.

El resto del capítulo 4 de [Wirth 76] elabora con detalles algunos tipos recursivos de particular interés, como lo son, por ejemplo, listas y árboles. Para el desarrollo de estos temas, que sí serán tratados en este curso, nos basaremos en otras referencias, tal como indicaremos más adelante en esta guía.

Ejemplos de Implementación de TADs con Arreglos: Pilas y Colas

Sección 10.1 de [Cormen et al. 01].

Esta sección presenta una implementación sencilla del TAD Pila y una implementación sencilla del TAD Cola, ambas mediante arreglos.

Tipos Algebraicos Libres

Capítulo 10 de [Thompson 96].

Las tres primeras secciones introducen al uso de tipos algebraicos libres en el lenguaje de programación funcional Haskell. (En un lenguaje funcional, toda la programación se realiza definiendo funciones al estilo matemático.)

La sección 10.1 se inicia con ejemplos de tipos algebraicos libres en los que no hay recursión. Sin recursión se puede, por ejemplo, definir tipos en los que se enumeran todos sus posibles valores constantes, como un tipo para definir los días de la semana o un tipo para manejar nombres de colores. La subsección final en 10.1 puede ser ignorada, pues, salvo que el lector interesado revise capítulos previos de este libro, no tenemos información sobre el manejo de clases en el lenguaje Haskell.

La sección 10.2 introduce recursión en los tipos, permitiendo así la construcción de, por ejemplo, árboles. Esta sección sólo maneja tipos recursivos monomórficos o específicos, como lo son, por ejemplo, árboles de enteros, árboles de strings, o árboles de booleanos. La sección 10.3 introduce el uso de polimorfismo, lo cual permite definir tipos parametrizados, como, por ejemplo, árboles de T , donde T puede ser un tipo cualquiera.

En la sección 10.4 se presenta un ejemplo muy interesante bajo la subsección “Error types”; la subsección anterior, que se refiere a la función predefinida *error* de Haskell, puede ser ignorada debido a que maneja detalles internos específicos de ese lenguaje que no son de nuestro interés. En la subsección que sí nos interesa, se trabaja con el tipo *Err t*, que equivale al tipo *Opcional(T)* al que se hizo referencia en el proyecto del laboratorio. La sección 10.5 también presenta otro ejemplo de tipo, haciendo hincapié en el cómo diseñar su estructura.

La sección 10.6 puede ser ignorada, pues también tiene que ver con el manejo de clases en Haskell, de lo cual de nuevo no tenemos información.

La sección 10.7 trabaja el importante tópico de demostraciones inductivas sobre la estructura de un tipo algebraico libre cualquiera. Se presentan ejemplos con árboles binarios y con expresiones aritméticas; también se ejemplifica el caso no recursivo del antes referido tipo *Err t*.

Muy lamentablemente, no disponemos bibliografía para un importantísimo tópico dentro de este tema: el manejo de tipos algebraicos libres en programas imperativos (esto es, en programas escritos en lenguajes de programación imperativa, al estilo de Java o GCL). Para esto se les ruega sobrevivir con la definición y ejemplos vistos en las clases de teoría del predicado *is* y la instrucción *match*, que nos permiten manejar variables de estos tipos en nuestros programas imperativos.

Apuntadores / Referencias

Capítulo 10 de [Cormen et al. 01], a excepción de la sección 10.1 (ya utilizada en un tema anterior), y capítulo 4 de [Wirth 76], sólo parte de las secciones 4.3 y 4.4.

Las secciones 10.2 y 10.4 de [Cormen et al. 01] tratan el manejo de los tipos recursivos lista, árbol binario, y árbol general, por medio de apuntadores. El aspecto negativo de esta presentación es que sólo muestra cómo manipular estas estructuras con apuntadores una vez que están creadas; no se muestra la creación de las estructuras. Notar por ejemplo que las operaciones de inserción siempre asumen que ya se dispone del espacio en el que está almacenado el nuevo elemento.

La sección 10.3 de [Cormen et al. 01] muestra cómo simular el manejo de apuntadores por medio de arreglos. Esto corresponde realmente al manejo interno que los lenguajes de programación terminan realizando con los apuntadores (donde el “gran arreglo” es la memoria del computador).

En [Cormen et al. 01], al igual que en Java, el manejo de apuntadores es implícito (que es a lo que algunos autores prefieren llamar referencias en lugar de apuntadores; las primeras son implícitas mientras que los segundos son explícitos). Un buen manejo explícito de apuntadores se encuentra en [Wirth 76].

En la sección 4.3 de [Wirth 76] se presenta el manejo del tipo recursivo lista por medio de apuntadores. La sección 4.4 hace lo mismo para el tipo recursivo de árboles binarios (con información en los nodos internos y hojas vacías). Sus subsecciones 4.4.1 y 4.4.2 proveen suficiente información para el manejo básico de este tipo recursivo; el resto de la sección profundiza en tópicos más especializados como árboles de búsqueda y árboles balanceados del tipo AVL.

Árboles Binarios de Búsqueda

Capítulo 12 de [Cormen et al. 01].

La presentación es buena, en términos de apuntadores, mostrando los procedimientos imperativos correspondientes a cada operación (búsqueda, inserción, eliminación, y varias otras).

En vista de que en un árbol binario de búsqueda la complejidad de las operaciones depende de la altura del árbol, la sección 12.4 presenta un análisis probabilístico de la altura esperada de un árbol cualquiera. Sin conocimientos de teoría de probabilidades no vale la pena profundizar en esta sección. En cualquier caso, sabemos que el problema de la altura “aleatoria” se resuelve al entrar al tema de los árboles rojo-negros.

Árboles Rojo-Negros

Capítulo 13 de [Cormen et al. 01], y capítulo 3 de [Okasaki 98], específicamente la sección 3.3.

De nuevo, [Cormen et al. 01] ofrece una buena presentación, en términos de apuntadores, con los procedimientos imperativos de todas las operaciones.

Sin embargo, el manejo de los apuntadores (con ¡doble enlace!) tiende a oscurecer la lógica de las operaciones. Una presentación en términos de tipos algebraicos libres mejora este aspecto. [Okasaki 98] ofrece tal tipo de presentación. Como en esta referencia el lenguaje de programación utilizado es funcional (el lenguaje en cuestión es ML, el cual es pariente cercano de Haskell), la consideración de casos de balanceo termina siendo ligeramente diferente a la presentada en [Cormen et al. 01].

Implementación de TADs con Tipos Recursivos

Ver introducción a Parte III de [Cormen et al. 01].

Esta introducción indica que en todos los capítulos de esta tercera parte del libro (capítulos 10 al 14) se presentan estructuras de datos para manipular “conjuntos dinámicos”. Leyendo con detenimiento esta introducción, puede concluirse que los capítulos en cuestión presentan *modelos concretos* de implementación de los *TADs Conjunto* y *Diccionario*.

Herencia

Capítulos 7 y 8 de [Liskov & Guttag 01].

En estos dos capítulos se presenta el concepto de herencia, su manejo dentro del lenguaje Java, y muchas aplicaciones de reutilización de software por medio de ella. El capítulo 7 es la gran introducción al tema, mientras que el capítulo 8 se concentra en la implementación de polimorfismo por medio de herencia.

Tablas de “Hash”

Capítulo 11 de [Cormen et al. 01].

Aquí se presenta todo lo relevante de estas estructuras de implementación de diccionarios y conjuntos. Incluye manejo de colisiones por medio de listas y por medio de funciones de “re-hash”. Inicialmente se presenta incluso el caso extremo en el que se dispone de una tabla con un índice por posible clave (llamadas tablas de acceso-directo).

Por la “aleatoriedad” con que las funciones de “hash” distribuyen los elementos en las tablas, el análisis de complejidad en este caso requiere de conocimientos de teoría de probabilidades.

Otros temas: Iteradores y “Testing”

Capítulos 6 y 10 de [Liskov & Guttag 01].

Estos temas no son exigidos en el programa de esta asignatura, pero están muy relacionados con el resto del contenido, por lo que se recomienda su estudio.

Los iteradores son objetos que permiten *iterar*, o recorrer, una estructura de datos (sea ésta un árbol, una lista, un arreglo, o cualquier otra). Éstos son presentados en el capítulo 6 de [Liskov & Guttag 01] en el contexto particular del lenguaje Java. Tarde o temprano, Ud. necesitará un iterador; ésta podría ser su única oportunidad de aprender sobre ellos.

“Testing” se refiere al proceso de *probar* un programa, pero no en términos de demostración matemática de su correctitud, sino de alimentar la ejecución del programa con varios ejemplos de entrada que nos convenzan de que el programa “parece” funcionar bien siempre. Hacer buen “testing”, mediante la preparación de un buen conjunto de ejemplos de entrada, es una actividad mucho más compleja que lo que puede lucir en un principio. Todo el capítulo 10 de [Liskov & Guttag 01] está dedicado a este tema. Afortunadamente, este tópico sí es contenido obligatorio de otras asignaturas de la carrera, como Sistemas de Programas y la cadena de Ingeniería de Software; en el futuro, es bueno que recuerde que este tópico está tratado en [Liskov & Guttag 01].

Cierre

Fue un placer haber construido esta guía de lectura. Esperamos que les haya sido útil.