

Algoritmos Discretos: Análisis y Diseño

Diego Arroyuelo

12 de marzo de 2022

Índice general

Índice general	III
1 Introducción	1
1.1 Introducción	1
1.2 ¿Qué es un Algoritmo?	2
1.3 Tiempo de Ejecución de un Algoritmo	2
1.4 La Importancia de un Algoritmo Eficiente	3
1.5 ¿Cuál es el Origen de los Algoritmos?	3
1.6 ¿Por Qué Estudiar Algoritmos?	7
1.7 Nuestro Enfoque para el Estudio de Algoritmos	8
1.8 Cómo Usar este Libro	9
1.9 Algoritmos en la Industria	9
1.10 Algoritmos en Competencias de Programación	10
 Conceptos Preliminares	 11
2 Conceptos Matemáticos Preliminares	13
2.1 Conjuntos	13
2.2 Funciones y Relaciones Binarias	16
2.3 La Función Factorial	26
2.4 Sucesiones, Strings, Permutaciones, Combinaciones	27
2.5 Glosario de Conceptos Matemáticos Básicos	35
 3 Conceptos Algorítmicos Preliminares	 39
3.1 Modelos de Computación	39
3.2 Problemas Abstractos	42
3.3 Tipos de Problemas Abstractos	44
3.4 Algoritmos: Definición Formal	46
3.5 El Rol de la Algoritmia en la Resolución de Problemas	47
3.6 Tiempo de Ejecución de un Algoritmo	48
3.7 Análisis de Mejor y Peor Caso	50
3.8 Notaciones Asintóticas	51
 4 Inducción y Algoritmos Recursivos	 65
4.1 Uso 1: La Inducción como Técnica de Demostración	65
4.2 Uso 2: Definición Inductiva de Conjuntos	68
4.3 Uso 3: Ecuaciones de Recurrencia	69

4.4	Uso 4: Algoritmos Recursivos	69
4.5	Resolución de Ecuaciones de Recurrencia	76
5	Estructuras de Datos Fundamentales	85
5.1	Pilas	85
5.2	Colas	88
5.3	Listas	91
5.4	Árboles	100
5.5	Grafos	113
	Técnicas de Análisis de Algoritmos	121
6	Análisis de Algoritmos	123
6.1	Análisis de Caso Promedio	123
6.2	Análisis Amortizado	133
7	Cotas Inferiores y Complejidad	139
7.1	La Complejidad de un Problema	139
7.2	Árboles de Decisión	141
7.3	Argumentos de Adversario	147
	Técnicas de Diseño de Algoritmos	151
8	Fuerza Bruta y Búsqueda Exhaustiva	153
8.1	Introducción	153
8.2	El Problema de Selección	154
8.3	El Problema de Ordenamiento	156
8.4	La Fila de Monedas	162
8.5	Búsqueda en Texto	163
9	Programación Dinámica	171
9.1	Introducción	171
9.2	Números de Fibonacci	171
9.3	Elementos Generales de la Programación Dinámica	175
9.4	La Fila de Monedas	175
9.5	Viajes en Canoa por el Río	178
9.6	Coeficientes Binomiales	179
9.7	Distancia de Edición	181
9.8	Multiplicación de Secuencias de Matrices	186
10	Backtracking Recursivo	193
10.1	Introducción	193
10.2	El Problema de las N Reinas	195
11	Dividir y Conquistar	203
11.1	Introducción	203
11.2	El Problema de Ordenamiento	203
11.3	El Teorema Maestro	216
11.4	El Problema de Determinar si Existe Mayoría Absoluta	218
11.5	El Problema de Intercalar Elementos de dos Conjuntos	221

11.6 El Problema de Multiplicar Números Enteros	223
11.7 El Problema de Multiplicar Matrices	226
11.8 Convex Hull de un Conjunto de Puntos	228
12 Decrecer y Conquistar	233
12.1 Introducción	233
12.2 El Problema de Ordenamiento: <code>InsertionSort</code>	234
12.3 El problema de Búsqueda en un Conjunto Ordenado	236
12.4 El Problema de Selección	248
13 Estructurar y Conquistar	255
13.1 Introducción	255
13.2 El Problema de Ordenamiento: <code>HeapSort</code>	255
14 Algoritmos Greedy	263
14.1 Introducción	263
14.2 El Problema de Dar Cambio	264
14.3 División de un Texto en Líneas	265
14.4 Árboles Recubridores Mínimos en Grafos	265

1.1. Introducción

La *algoritmia* es una rama fundamental de ciencias de la computación, dedicada al estudio del diseño y análisis de algoritmos, y a la cual dedicamos este libro.

Antes de avanzar en la discusión más profunda del tema, y a fin de ganar algo de intuición, es conveniente hacerse las siguientes preguntas:

¿Cuál fue el primer algoritmo que conoció?

¿Cuál fue el último algoritmo con el que interactuó y hace cuánto?

Quizás no lo note, pero el hecho de representar cantidades usando números arábigos ¹ introdujo avances impensados para la civilización. Uno de esos avances es que con este sistema se pueden escribir grandes cantidades usando pocos dígitos ². Otro avance es que las operaciones sobre esas cantidades —como la suma, resta, multiplicación, y división, entre otras— ahora podían hacerse mediante procesos mecánicos —o *algoritmos*— simples de aprender para todos. Quizás uno de los primeros algoritmos que el lector aprendió es el que permite sumar dos (o más) números enteros.

Tal como lo ilustra este ejemplo, los algoritmos juegan un papel más importante en nuestras vidas de lo que creemos. De hecho, estamos interactuando con algoritmos continuamente, por ejemplo cuando un buscador web nos genera la respuesta a nuestra búsqueda, cuando nos sugieren amigos o personas a las que seguir en medios sociales, cuando nos sugieren películas, series, o videos en algún servicio de video por streaming, cuando nos muestran publicidad online mientras usamos servicios en la web, cuando validamos nuestra entrada a un servicio de transporte público, o cuando necesitamos saber cómo llegar a una dirección particular de la ciudad.

Muchas de las aplicaciones más importantes en la actualidad necesitan resolver de forma eficiente problemas computacionales no triviales, como por ejemplo:

- Ordenar un conjunto de datos de gran tamaño para posteriormente ser capaces de procesarlos eficientemente.
- Obtener la mediana de un gran conjunto de valores numéricos, algo común en aplicaciones estadísticas.
- Obtener el par de puntos más cercados en un gran conjunto de puntos en el plano.
- Obtener todas las ciudades, carreteras, y accidentes geográficos que se ubican en un área rectangular de un mapa.
- Obtener todas las ocurrencias de un string de caracteres dentro de un texto.
- Generar una lista de recomendaciones de amistad o seguimiento para un usuario de un medio social.

1: Es decir, números escritos como secuencias de los dígitos $0, 1, \dots, 9$. Casi lo olvido: (ab)usaré notas al margen para aclarar conceptos o agregar comentarios a la exposición.

2: De hecho, se necesitan sólo $\lceil \log_{10} n \rceil$ dígitos decimales para representar una cantidad n . Imagine tener que escribir un número grande usando palitos que representen las unidades, e incluso usando números romanos.

- Obtener los anuncios que atraigan el interés de un usuario de la web, un motor de búsqueda, o medio social.
- Obtener las comunidades de personas que conforman una red social.
- Obtener el camino más rápido para movernos desde un punto de una ciudad a otro.

Aunque estos problemas son desafiantes, resolverlos eficientemente no es trivial y requiere de entrenamiento y estudio. Este documento estudia la algoritmia en suficiente profundidad como para resolver muchos de los problemas antes mencionados, y muchos más que encontraremos en el camino. Para ello se requiere entrenamiento y estudio, que trataremos de acompañar a lo largo del desarrollo del libro.

1.2. ¿Qué es un Algoritmo?

3: Más tarde formalizaremos la noción de algoritmo.

La siguiente es una definición informal de algoritmo ³.

Definición 1.2.1 *Un **algoritmo** es una secuencia finita, ordenada, y no ambigua de instrucciones (o pasos a seguir) que permiten resolver un problema.*

Que la secuencia de instrucciones sea finita implica que todo algoritmo debe detener su ejecución y producir un resultado que resuelva el problema subyacente. Que la secuencia sea ordenada significa que el orden en que se ejecutan las instrucciones es importante. En un ejemplo doméstico, cuando se prepara una receta en la cocina, generalmente llevar los ingredientes al horno suele ocurrir después de hacer la preparación básica, y no antes. El orden es importante. Por último, que la secuencia sea no ambigua significa que cada una de las instrucciones tiene una única interpretación posible, sin ambigüedad. Para solucionar esto último, en general, es preferible usar un lenguaje formal —por ejemplo, un lenguaje de programación— para describir un algoritmo.

1.3. Tiempo de Ejecución de un Algoritmo

Definición 1.3.1 *El tiempo de ejecución de un algoritmo es la cantidad de pasos que éste necesita para transformar una entrada en su correspondiente salida.*

Generalmente, ese tiempo será modelado como una función del tamaño de la entrada —es decir, la cantidad de datos que un algoritmo recibe como entrada. Este concepto será una de las principales formas de analizar un algoritmo, con el fin de compararlo con otros que resuelven el mismo problema: un algoritmo con menor tiempo de ejecución significa que es más rápido y eficiente.

¿Qué pasa con los algoritmos que (por su naturaleza) nunca finalizan?

Tal como lo indica la Definición 1.2.1, un algoritmo **siempre finaliza y entrega una respuesta correcta**. De esa forma, es posible medir su tiempo de ejecución. Sin embargo, hay problemas de naturaleza infinita cuyos algoritmos asociados que ejecutan, en principio, infinitamente. Por ejemplo, el problema de computar los infinitos dígitos decimales de π . Para comparar el tiempo de ejecución de dos algoritmos distintos que resuelven este problema, note que el proceso para generar cada uno de los dígitos decimales del número debe ejecutar una cantidad finita de instrucciones. En ese caso, se podría comparar algoritmos que generan los dígitos de π en términos del tiempo que tardan en escribir cada dígito. No tendría sentido compararlos a lo largo del proceso completo, infinito.

1.4. La Importancia de un Algoritmo Eficiente

El diseño de algoritmos eficientes es uno de los principales objetivos de este libro, ya que tiene notables consecuencias en la práctica:

- La primera es que un tiempo de ejecución elevado puede resultar prohibitivo al manipular grandes cantidades de datos.
- Un algoritmo eficiente dejará satisfechos a usuarios que necesitan respuestas rápidas (piense, por ejemplo, en los servicios online que existen hoy en día).
- Un algoritmo eficiente hará uso más eficiente de los recursos computacionales necesarios para la ejecución.
- Un algoritmo con tiempo de ejecución grande implica una mayor cantidad de ciclos de procesador necesarios para producir un resultado, lo cual deriva en un mayor uso de energía. Si ese algoritmo ineficiente está ejecutando sobre un dispositivo móvil, consumirá más rápidamente su batería. Si, en cambio, está ejecutando en un gran centro de cómputo, consumirá más energía no sólo porque ejecuta una mayor cantidad de ciclos para producir un resultado, sino también porque producirá más calor, lo cual incrementa el costo de refrigeración del centro. En este último caso, si además el algoritmo ineficiente da servicio a usuarios a lo largo de mucho tiempo, el consumo adicional de energía (y el costo monetario asociado) por ser ineficiente puede ser muy grande.

1.5. ¿Cuál es el Origen de los Algoritmos?

Como ya lo hemos dicho anteriormente, hoy en día la *algoritmia* es una de las áreas fundamentales de ciencias de la computación, dedicada al estudio del diseño y análisis de algoritmos. Aún cuando hoy en día es un área muy activa en la industria e investigación, tiene su origen mucho antes de la existencia de los actuales computadores. Los

primeros algoritmos registrados son los algoritmos de multiplicación desarrollados por los egipcios entre los siglos 1700–2000 a.C. Luego, entre los siglos 1600–1800 a.C., los Babilonios desarrollaron varios algoritmos para resolver problemas relevantes, como el de multiplicar números naturales. Estos algoritmos fueron encontrados en tablas de arcilla. Fue Donald Knuth, quien publicó en 1972 la primera traducción al inglés de esos algoritmos ¹, el que permitió entender la relevancia de esas tablas, y comprender que esos eran realmente algoritmos. Luego, hay otros ejemplos como el algoritmo de Euclides para encontrar el máximo común divisor de dos números enteros, que fue definido en el siglo 300 a.C. y es usado hasta el día de hoy, o la Criba de Eratóstenes para números primos, del siglo 200 a.C., por nombrar los más relevantes. Sin duda, los algoritmos han sido muy importantes para el desarrollo de las civilizaciones antiguas, y continúan siendo vitales para el desarrollo de la vida moderna.

La palabra *algoritmo* se origina con el influyente matemático persa Muhammad ibn Mūsā, originario de Khwārizmī, una región que hoy es parte de la república de Uzbekistán. Por su gentilicio es que luego fue conocido como Muhammad ibn Mūsā al-Khwārizmī, que significa *Muhammad ibn Mūsā, el de Khwārizmī*. Luego fue conocido simplemente como Al-Khorezmi, y posteriormente latinizado como *Algorizmi*. Alrededor del año 820. Algorizmi describió métodos para resolver ecuaciones lineales y cuadráticas en su trabajo titulado *Kitāb al-mukhtasar fī hisāb al-ğabr wa-l-muqābalah*, cuya traducción más cercana es *Libro Conciso sobre el Cálculo por Completación y Balance*. Dicho trabajo es más conocido por su forma simple *al-ğabr* o *Al-jabr*, que dio origen a lo que hoy conocemos como *Álgebra*. Su obra fue la primera en introducir conceptos fundamentales del álgebra, como el de sumar (o restar) términos a ambos lados de una ecuación, fundamental para la resolución de ecuaciones lineales. En su obra, Algorizmi además proponía usar el sistema de numeración hindú, que derivó en el sistema posicional decimal que usamos hoy en día, descartando los números romanos utilizados hasta ese momento, que agregan complejidad en su manipulación. Finalmente, la manera en que Algorizmi expuso la solución a problemas matemáticos, usando métodos generales en lugar de estudiar casos particulares como era común en esos tiempos, fue suficiente para nombrar a esos métodos generales para resolver problemas: algoritmos.

Con el paso de los años, los algoritmos fueron ganando cada vez más espacio en el desarrollo de la vida moderna. Los hitos más relevantes de la algoritmia son los siguientes:

- 1671:** se desarrolla el método de Newton-Raphson para encontrar raíces de una función.
- 1706:** John Machin desarrolla un método que permitió computar 100 dígitos decimales de π . Éste fue luego mejorado por Jurij Vega en 1789, para llegar a computar 140 dígitos de π .
- 1802:** Gauss introduce lo que luego se conocería como la transformada rápida de Fourier (FFT, Fast Fourier Transform). Ésta solución es mejorada y expandida por Fourier (año 1822), y luego presentada

¹ D. Knuth. *Ancient Babylonian Algorithms*. Communications of the ACM, Vol. 15 (7), páginas 671–677.

por Cooley y Tukey (año 1965) en su forma más usada hoy en día.

- 1842:** se conoce el algoritmo de Ada Lovelace, el primero desarrollado exclusivamente para un dispositivo de computación. Ada fue la primera en reconocer que la Máquina Analítica de Charles Babbage (desarrollada en 1834) tenía potenciales aplicaciones más allá del puro cálculo de funciones, por lo que se la puede considerar la primera programadora de computadores.
- 1847:** George Bool desarrolla del Álgebra Booleana, fundamental hoy en día para el funcionamiento de los computadores digitales.
- 1887:** Herman Hollerith desarrolla el algoritmo hoy conocido como RadixSort. Era usado para ordenar tarjetas perforadas sobre las Máquinas de Tabulación del mismo Hollerith, introducidas para acelerar la obtención de los resultados de los censos en Estados Unidos.
- 1934:** Delaunay desarrolla su algoritmo de triangulación de conjuntos de puntos, que es clave hoy en día para la representación de terrenos en 3D (entre otras aplicaciones).
- 1936:** Alan Turing desarrolla la Máquina de Turing y el concepto de *computabilidad*. Junto a Kurt Gödel, Alonzo Church, Emil Post, y Stephen Kleene (entre otros), desarrollaron la noción moderna de algoritmo, que mantenemos al día de hoy. Turing también introduce el concepto de computabilidad: la idea de que hay problemas que no pueden resolverse mediante un algoritmo.
- 1945:** John von Neumann desarrolla el algoritmo MergeSort. Éste fue el primer algoritmo en ejecutar en lo que hoy se conoce como el modelo de computación de von Neumann, y que se usa en los computadores actuales. Es dicho modelo, la memoria del computador almacena tanto los datos como los programas.
- 1947:** George Dantzig desarrolla el algoritmo Simplex de programación lineal.
- 1952:** David Huffman desarrolla el algoritmo de compresión de Huffman.
- 1954:** Harold H. Seward desarrolla los algoritmos CountingSort y RadixSort (u ordenamiento digital), el último basado en el antiguo método de ordenamiento de tarjetas perforadas de Herman Hollerith sobre las máquinas de tabulación.
- 1956:** Kruskal desarrolla su algoritmo para calcular árboles recubridores mínimos en grafos.
- 1957:**
- Prim desarrolla su algoritmo para calcular árboles recubridores mínimos en grafos.
 - Bellman y Ford desarrollan su algoritmo de caminos mínimos en grafos.
- 1959:**
- Dijkstra desarrolla su algoritmo de caminos mínimos en grafos.
 - Donald L. Shell desarrolla el algoritmo de ordenamiento ShellSort.
 - John G.F. Francis y Vera Kublanovskaya (de forma independiente) desarrollan el algoritmo de factorización QR, para la descomposición de matrices en álgebra lineal.
- 1960:** Karatsuba presenta su algoritmo de multiplicación de números enteros. Es el primero en mostrar que la multiplicación de números enteros podía hacerse de forma más eficiente que la conocida.

Esto significó un importante avance para multiplicar números de grandes cantidades de dígitos.

- 1962:**
- Adelson-Velsky y Landis presentan los árboles AVL, usados hasta el día de hoy (en alguna de sus variantes) para garantizar la eficiencia en la búsqueda en grandes conjuntos de datos.
 - C. A. R. Hoare desarrolla el algoritmo de ordenamiento QuickSort, uno de los algoritmos de ordenamiento de datos de uso más extendido (hasta el día de hoy).
 - Floyd presenta la versión conocida actualmente de lo que era el algoritmo de Roy (1959) y Warshall (1962), conocido actualmente como el algoritmo Floyd-Warshall. El algoritmo permite calcular caminos mínimos en grafos y calcular la clausura transitiva de un grafo.
 - Ford y Fulkerson desarrollan su algoritmo para calcular flujos máximos en grafos.
- 1964:** J. W. J. Williams desarrolla el algoritmo HeapSort.
- 1965:** Vladimir Levenshtein presenta el algoritmo para calcular la distancia de Levenshtein (o distancia de edición), utilizada para comparar strings, hoy en día con múltiples aplicaciones como la corrección ortográfica automática, y la comparación de secuencias de ADN y proteínas, entre otras.
- 1968:** Peter Hart, Nils Nilsson, y Bertram Raphael desarrollan el algoritmo A^* para recorrido de grafos.
- 1969:** Volker Strassen desarrolla el algoritmo de Strassen para multiplicar matrices, clave para aplicaciones que necesitan multiplicar grandes matrices.
- 1972:**
- Ronald Graham desarrolla el algoritmo scan de Graham, el que permite computar el convex hull (o cierre convexo) de un conjunto de puntos, uno de los algoritmos fundamentales de la geometría computacional.
 - Rudolf Bayer desarrolla los árboles 2-3 (2-3 trees) y B (B-trees), los últimos usados hasta el día de hoy como estructura principal para el almacenamiento de grandes volúmenes de datos en memorias secundarios como los discos duros.
- 1973:**
- R. A. Jarvis desarrolla el algoritmo marcha de Jarvis (Jarvis march), para el cálculo del convex hull de un conjunto de puntos.
 - John Hopcroft y Richard Karp presentan el algoritmo Hopcroft-Karp, para emparejamientos de cardinalidad máxima en grafos bipartitos.
- 1975:** Aho y Corasick presentan el algoritmo Aho-Corasick para búsquedas de múltiples patrones en strings.
- 1976:** Knuth, Morris, y Pratt presentan su algoritmo para búsqueda de patrones en texto.
- 1977:**
- Boyer y Moore presentan su algoritmo para búsqueda de patrones en texto.
 - Rivest, Shamir, y Adleman presentan el algoritmo de encriptación RSA.
 - Lempel y Ziv descubren el algoritmo de compresión de textos conocido como LZ77, base para muchos de los compresores usados hoy en día (e.g., `gzip`).

- 1978:**
 - Lempel y Ziv descubren el algoritmo de compresión de textos conocido como LZ78, como alternativa al algoritmo presentado el año anterior.
 - Guibas y Sedgwick desarrollan los árboles Rojo-Negro (Red-Black trees), como una evolución de los árboles AVL. Estos son usados hoy en día para implementar arreglos asociativos de tipo `map` de lenguajes como C++.
- 1984:** Terry Welch desarrolla el algoritmo de compresión LZW, una variante de LZ78 usada como parte del formato de imágenes GIF.
- 1985:** Sleator y Tarjan descubren los Splay Trees.
- 1994:** Burrows y Wheeler descubren la transformada Burrows-Wheeler, sobre la que se basan diversos algoritmos de compresión como `bzip2`.
- 1998:** Larry Page presenta el algoritmo PageRank, que es la base del buscador google. Éste es, quizás, el algoritmo más ejecutado de la historia: se ejecuta millones de veces por hora, con cada búsqueda en google.
- 2001:** Lempel–Ziv–Markov chain algorithm for compression developed by Igor Pavlov
- 2002:** Girvan y Newman presentan su algoritmo para detectar comunidades en sistemas complejos.
- 2002:** Agrawal, Kayal, y Saxena presentan el algoritmos AKS para chequeo de primalidad.
- 2004:** Bender y Farach Colton desarrollan su estructura de datos para resolver la operación LCA sobre árboles.

La lista no es completa, y puede ser extendida, pero ilustra los hitos más importantes en el desarrollo de la algoritmia.

1.6. ¿Por Qué Estudiar Algoritmos?

Programar es mucho más que aprender a escribir código usando algún lenguaje de programación. Un buen programador debe, además, tener habilidad para resolver problemas complejos, y la capacidad de medir la eficiencia de una solución algorítmica. La algoritmia es una de las bases fundamentales de ciencias de la computación. Aunque otras áreas han ganado terreno últimamente —e.g., ciencia de datos, inteligencia artificial y deep learning—, la algoritmia sigue siendo fundamental para proveer soluciones eficiente a muchos de los problemas que aparecen en esas áreas. Es, de alguna manera, un área transversal a ciencias de la computación. A lo largo del libro daremos soluciones algorítmicas a problemas que aparecen en áreas como estructuras de datos, bases de datos, sistemas operativos, lenguajes de programación, biología computacional (o bioinformática), minería de datos, computación paralela, álgebra lineal, geometría, cálculo, matemáticas discretas, y estadística, entre otras.

1.7. Nuestro Enfoque para el Estudio de Algoritmos

Presentamos a continuación el enfoque que seguiremos para el diseño y análisis de algoritmos. La Figura 1.1 muestra el esquema principal de nuestro enfoque, que es el siguiente:

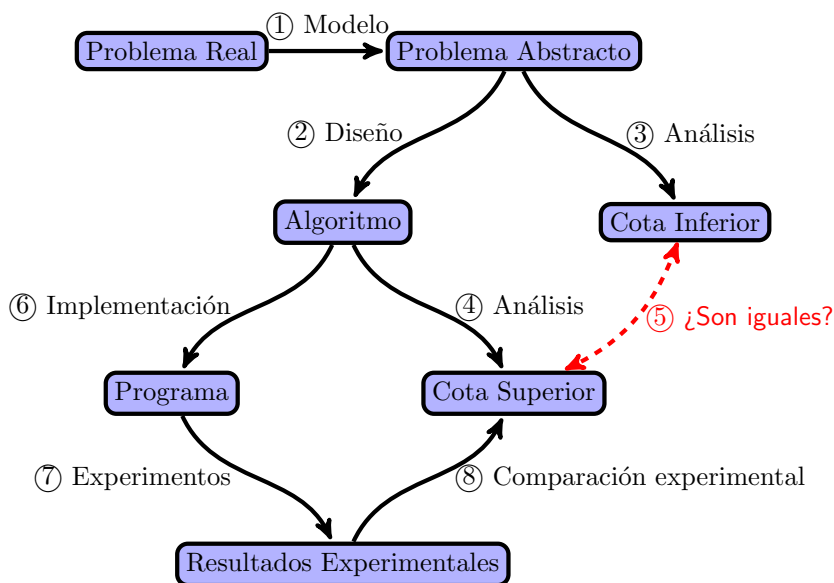


Figura 1.1: Nuestro enfoque para el estudio de algoritmos.

1. Los problemas del mundo real usualmente son modelados para convertirlos en problemas abstractos. De esta manera, simplifican el diseño y análisis de algoritmos que los resuelven.
2. A partir del problema abstracto, y usando técnicas de diseño de algoritmos, obtenemos un algoritmo que resuelve dicho problema. Parte importante de este trabajo está dedicado al diseño de algoritmos, en donde estudiaremos las técnicas de diseño más importantes.
3. A partir del problema abstracto, pero esta vez usando técnicas de análisis sobre el **problema**, obtenemos cotas inferiores para éste. Parte de este trabajo estará dedicado a obtener cotas inferiores de los problemas mas emblemáticos. Este tipo de análisis entrega información valiosa para el diseño de algoritmos, tal como veremos más adelante.
4. Al usar técnicas de análisis sobre un algoritmo se obtiene el tiempo de ejecución del mismo. Esos tiempos de ejecución definirán, a su vez, cotas superiores para la complejidad del problema que resuelve el algoritmo.
5. Si la cota superior obtenida a partir del algoritmo es igual a la cota inferior demostrada para el problema, hemos encontrado la complejidad del problema. El algoritmo obtenido se conoce como *algoritmo óptimo* (es decir, el mejor algoritmo posible para resolver el problema subyacente).
6. Al implementar un algoritmo usando un lenguaje de computación, se obtiene un programa que resuelve el problema. En este trabajo,

la implementación de algoritmos estará cubierta por proyectos al final de cada capítulo.

7. A partir de pruebas experimentales realizadas con la implementación del algoritmo, se obtienen resultados experimentales para el tiempo de ejecución de éste.
8. La evaluación de los resultados experimentales del algoritmo permite cotejarla con el análisis teórico realizado anteriormente. A partir de esto se puede concluir respecto a la practicidad de un algoritmo.

A lo largo de este libro, continuamente seguiremos caminos mostrados en el diagrama de la figura.

1.8. Cómo Usar este Libro

Este libro está centrado en las principales técnicas de diseño y análisis de algoritmos, también con una mirada en la complejidad de problemas. Principalmente, el documento está organizado por técnicas de diseño. Esto significa que un mismo problema puede ser tratado en varios capítulos, desde distintos puntos de vista respecto al diseño. El libro está orientado principalmente a estudiantes de carreras de pregrado en ciencias de la computación, cubriendo los temas principales de un curso obligatorio de diseño y análisis de algoritmos. Sin embargo, también se tratan temas que pueden ser usados para un curso de algoritmos avanzado.

En el libro se revisan mucho de los algoritmos más relevantes y conocidos. Sin embargo, también se tratan otros no tan conocidos, pero que tienen valor práctico y/o pedagógico. El enfoque está puesto no sólo en revisar mucho de los algoritmos más reconocidos, sino que también se busca enseñar el diseño y análisis de algoritmos, y para eso es necesario tratar problemas nuevos y no tan conocidos.

1.9. Algoritmos en la Industria

Los avances en tecnologías de la información no sólo han incrementado la cantidad de servicios disponibles, sino que además han incrementado notablemente el volumen de datos que deben manipular muchas de las empresas prestadoras de esos servicios. Algunos ejemplos típicos son datos de interacción de usuarios en medios sociales, datos de movilidad de personas (obtenidos a través del uso de teléfonos celulares), datos obtenidos a través de la Internet de las Cosas, grandes bases de datos como la web o medios sociales, bases de datos geográficas y de mapas, bibliotecas digitales, repositorios de código fuente, bases de datos biológicas, escenarios tridimensionales de videojuegos y realidad virtual, por nombrar algunos ejemplos. La industria se ha visto no sólo en la necesidad de dar servicios de forma eficiente, sino que también manipular estas grandes cantidades de datos para sacarles algún tipo de provecho. Por lo tanto, necesitan contar con ingenieros con una sólida formación algorítmica. Esto se puede notar en el tipo de entrevistas laborales que han implementado las empresas más importantes, basadas

principalmente en la formación algorítmica del postulante. A lo largo del libro, y siempre que sea posible, se vincularán los temas estudiados con aplicaciones concretas en la industria. Además, muchos de los ejercicios al final de las secciones del libro son (o se parecen) a preguntas de dichas entrevistas.

1.10. Algoritmos en Competencias de Programación

Desde hace unos años, las competencias de programación se han vuelto más comunes y populares. Las mismas consisten en resolver un conjunto de problemas algorítmicos en un tiempo establecido para la competencia (por ejemplo, 1 día, o 5 horas). Unas de las competencias más conocidas es la ACM ICPC, que se realiza a nivel regional por subcontinentes, para luego realizar la final mundial con los mejores equipos clasificados de las competencias regionales. En el libro, resolveremos problemas que frecuentemente aparecen en dichas competencias y, siempre que sea posible, al final de cada capítulo ilustraremos las técnicas estudiadas con problemas que han aparecido en competencias de programación.

Conceptos Preliminares

En este capítulo revisaremos algunos conceptos matemáticos importantes para el desarrollo de este libro.

2.1. Conjuntos

Un *conjunto* A es una agrupación o colección de elementos distintos ¹ tomados de un *universo* U —el *universo de referencia*—, que generalmente se hace para una mejor manipulación desde el punto de vista matemático o computacional. El concepto de conjunto es fundamental en diversas áreas de las matemáticas y ciencias de la computación. De hecho, juega un rol central en matemáticas discretas, pero también en el diseño y análisis de algoritmos.

1: Si existen elementos repetidos, lo llamaremos *multiconjunto*.

Formas de Definir Conjuntos

Definir formalmente un conjunto A significa indicar de forma clara y sin ambigüedad alguna cuáles elementos del universo U pertenecen a A y cuáles no. Existen tres formas principales de definir conjuntos: por *extensión*, por *comprensión*, y por *inducción*. Estudiaremos ahora las dos primeras, dejando la definición por inducción para más tarde.

Definición por Extensión. Un conjunto de elementos es definido por *extensión* si se enumeran todos sus elementos, uno por uno, encerrados dentro de llaves ‘{’ y ‘}’ y separados por comas. Por ejemplo, $\{1, 3, 5, 7, 9\}$ es un conjunto con todos los números impares ≤ 10 . Este tipo de definición es preferible cuando se trata de conjuntos finitos de pocos elementos. Para conjuntos finitos más grandes, a veces se puede usar algún tipo de generalización para definirlos por extensión. Por ejemplo, el conjunto $\{1, 3, 5, \dots, 2i + 1, \dots, 999\}$ contiene a todos los números impares positivos que son menores a 1000. De hecho, también permite definir cierto tipo de conjuntos infinitos, como por ejemplo $\{1, 3, 5, \dots, 2i + 1, \dots\}$, que representa al conjunto infinito de números impares positivos.

Definición por Comprensión. Para ciertos tipos de conjuntos es complicado —y, en muchos casos, imposible— definirlos por extensión o usando generalizaciones como las mostradas. En esos casos, una definición por *comprensión* puede ser preferible, la cual describe los elementos que pertenecen al conjunto a través de propiedades que tienen en común. Por ejemplo, la definición $\{x \in \mathbb{N} \mid 1 \leq x < 1000 \wedge x \text{ es impar}\}$ describe por comprensión al conjunto de los impares positivos menores a 1000. La parte que está a la izquierda de la barra ‘|’ indica la forma que tienen los elementos que forman el conjunto, mientras que la parte derecha

indica las propiedades que deben cumplir dichos elementos. El conjunto en cuestión estará formado por *todos* los elementos que cumplan con las propiedades establecidas. Computacionalmente, almacenar un conjunto por extensión significa almacenar cada uno de sus elementos explícitamente. Almacenar un conjunto por comprensión, por otro lado, no necesita el almacenamiento explícito de sus elementos, sino más bien una manera de determinar los elementos que lo forman (la propiedad que está a la derecha de ‘|’ en la definición).

Pertenencia de un Elemento a un Conjunto

Sea A un conjunto. Diremos que cada uno de los elementos que forman el conjunto *pertenece a* A . La *cardinalidad* (o *tamaño*) de un conjunto finito A será denotada con $|A|$, y representa la cantidad de elementos que pertenecen al mismo. Formalmente, diremos que $x \in A$ si el elemento x pertenece a A . En otro caso, diremos que $x \notin A$. Aunque parezca un concepto muy simple, el chequeo de pertenencia de un elemento a un conjunto es un problema fundamental en algoritmia, con aplicaciones muy diversas. De hecho, éste será uno de los problemas emblemáticos estudiados a lo largo de este libro.

Un conjunto A es *subconjunto* de otro conjunto B si todo elemento de A también pertenece a B . Si existe la posibilidad de que ambos conjuntos sean iguales, lo denotaremos $A \subseteq B$. Si, por otro lado, existe al menos un elemento $x \in B$ tal que $x \notin A$, entonces diremos que la inclusión es estricta —o que A es *subconjunto propio* de B —, y lo denotaremos $A \subset B$.

El Conjunto Vacío y el Conjunto Potencia

El conjunto vacío es uno que no tiene ningún elemento, y se denota con \emptyset . Por lo tanto, se cumple $|\emptyset| = 0$ y $\emptyset \subseteq A$, para todo conjunto A .

Por otro lado, dado un conjunto A , su *conjunto potencia* $\mathcal{P}(A)$ —también conocido como el *conjunto de partes* de A — es el conjunto formado por todos los subconjuntos de A ². Por ejemplo, si $A = \emptyset$, entonces $\mathcal{P}(A) = \{\emptyset\}$ ³. Si $B = \{1\}$, $\mathcal{P}(B) = \{\emptyset, \{1\}\}$. Ahora, si $C = \{1, 2, 3, 4\}$, entonces

$$\mathcal{P}(C) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{4\}, \{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \\ \{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}, \{2, 3, 4\}, \{1, 2, 3, 4\}\}.$$

Note que $|\mathcal{P}(A)| = 2^{|A|}$. Para los tres ejemplos estudiados anteriormente tenemos $|\mathcal{P}(\emptyset)| = 2^0$, $|\mathcal{P}(\{1\})| = 2^1$, y $|\mathcal{P}(\{1, 2, 3, 4\})| = 2^4$.

Operaciones sobre Conjuntos

Definimos a continuación las operaciones más importantes sobre conjuntos, siendo A , B , y C conjuntos con universo U ⁴.

2: Usualmente, llamaremos *familia de conjuntos* a un conjunto de conjuntos.

3: Note que $\emptyset \neq \{\emptyset\}$.

4: En capítulos posteriores mostraremos cómo computar eficientemente muchas de estas operaciones.

Unión de Conjuntos. La unión de dos conjuntos A y B se define como:

$$A \cup B = \{x \mid x \in A \vee x \in B\}.$$

Por ejemplo, si $A = \{1, 3, 7, 9\}$ y $B = \{2, 3, 5, 7\}$, entonces $A \cup B = \{1, 2, 3, 5, 7, 9\}$. Note que el resultado de la unión es, a su vez, un conjunto, por lo que no tiene elementos repetidos.

La unión de conjuntos tiene las siguientes propiedades:

- **Propiedad Conmutativa:** $A \cup B = B \cup A$.
- **Propiedad Asociativa:** $A \cup (B \cup C) = (A \cup B) \cup C$. Como consecuencia, la operación puede definirse sobre múltiples conjuntos, de la forma $A_1 \cup \dots \cup A_k$. Usaremos $\bigcup_{i=1}^k A_i$ para denotar esa misma unión de forma resumida.
- **Propiedad de Identidad** (elemento neutro): $A \cup \emptyset = A$.
- **Propiedad de Dominación:** $A \cup U = U$.
- **Propiedad de Idempotencia:** $A \cup A = A$.

Intersección de Conjuntos. La intersección de dos conjuntos A y B se define como:

$$A \cap B = \{x \mid x \in A \wedge x \in B\}.$$

Por ejemplo, si $A = \{1, 3, 7, 9\}$ y $B = \{2, 3, 5, 7\}$, entonces $A \cap B = \{3, 7\}$.

Diremos que dos conjuntos son *disjuntos* si $A \cap B = \emptyset$. La intersección de conjuntos tiene las siguientes propiedades:

- **Propiedad Conmutativa:** $A \cap B = B \cap A$.
- **Propiedad Asociativa:** $A \cap (B \cap C) = (A \cap B) \cap C$. Al igual que para la unión, esta última propiedad permite definir una intersección de forma general como $A_1 \cap \dots \cap A_k = \bigcap_{i=1}^k A_i$.
- **Propiedad de Identidad** (elemento neutro): $A \cap U = A$.
- **Propiedad de Dominación:** $A \cap \emptyset = \emptyset$.
- **Propiedad de Idempotencia:** $A \cap A = A$.
- **Propiedad Distributiva:**
 1. $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$.
 2. $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$.

Diferencia. La diferencia entre dos conjuntos A y B se define como:

$$A \setminus B = \{x \mid x \in A \wedge x \notin B\}.$$

Por ejemplo, si $A = \{1, 3, 7, 9\}$ y $B = \{2, 3, 5, 7\}$, entonces $A \setminus B = \{1, 9\}$.

Complemento. El complemento de un conjunto A se define como:

$$\overline{A} = \{x \mid x \in U \wedge x \notin A\}.$$

Las siguiente propiedades están relacionadas con el complemento de un conjunto:

- $\overline{(\overline{A})} = A.$
- $A \cup \overline{A} = U.$
- $A \cap \overline{A} = \emptyset.$
- **Leyes de De Morgan:**
 1. $\overline{A \cup B} = \overline{A} \cap \overline{B}.$
 2. $\overline{A \cap B} = \overline{A} \cup \overline{B}.$

5: El nombre “producto cartesiano” deriva de René Descartes, cuyo estudio originó este concepto.

Producto Cartesiano. El producto cartesiano ⁵ de dos conjuntos A y B se define como:

$$A \times B = \{(a, b) \mid a \in A \wedge b \in B\}.$$

Esto es, $A \times B$ es el conjunto de todos los posibles pares ordenados —o *duplas*— (a, b) tal que $a \in A$ y $b \in B$.

En general, para los conjuntos A_1, A_2, \dots, A_k , con $k \geq 2$, definimos

$$A_1 \times A_2 \times \dots \times A_k = \{(a_1, a_2, \dots, a_k) \mid a_1 \in A_1 \wedge a_2 \in A_2 \wedge \dots \wedge a_k \in A_k\}.$$

Llamaremos *k-upla* (o *k-tupla*) a cada elemento (a_1, a_2, \dots, a_k) del producto cartesiano $A_1 \times \dots \times A_k$.

Notaciones para Conjuntos de Números

En adelante usaremos las notaciones estándar para los conjuntos de números más típicos, como se indica a continuación.

Números Naturales. Denotaremos con \mathbb{N} al conjunto infinito de números naturales $\{1, 2, \dots\}$. Cuando necesitemos incluir el 0, lo denotaremos \mathbb{N}^0 . Finalmente, denotaremos $\mathbb{N}_n = \{1, \dots, n\}$ y $\mathbb{N}_n^0 = \{0, 1, \dots, n\}$, para $n \geq 1$. Para $x, y \in \mathbb{N}$, $x \leq y$, usaremos $[x..y]$ para denotar al intervalo entero correspondiente al conjunto $\{x, x+1, \dots, y\}$.

Números Enteros. Denotaremos con \mathbb{Z} al conjunto infinito de números enteros $\{\dots, -2, -1, 0, 1, 2, \dots\}$. Usaremos la misma notación $[x..y]$ para representar un intervalo de números enteros, con $x, y \in \mathbb{Z}$ y $x \leq y$.

Números Reales. Denotaremos con \mathbb{R} al conjunto infinito de números reales. Algunos subconjuntos de interés son los números reales positivos $\mathbb{R}^+ = \{x \in \mathbb{R} \mid x > 0\}$ y el conjunto de números reales negativos $\mathbb{R}^- = \{x \in \mathbb{R} \mid x < 0\}$. Alternativamente, se puede definir \mathbb{R}_0^+ y \mathbb{R}_0^- si necesitamos incluir al 0. Para $x, y \in \mathbb{R}$, $x \leq y$, denotaremos con $[x, y]$ al intervalo de números correspondiente al conjunto $\{z \in \mathbb{R} \mid x \leq z \leq y\}$.

2.2. Funciones y Relaciones Binarias

Definición 2.2.1 Sean A y B dos conjuntos no vacíos. Se denomina *relación binaria de A en B* a cualquier conjunto $R \subseteq A \times B$.

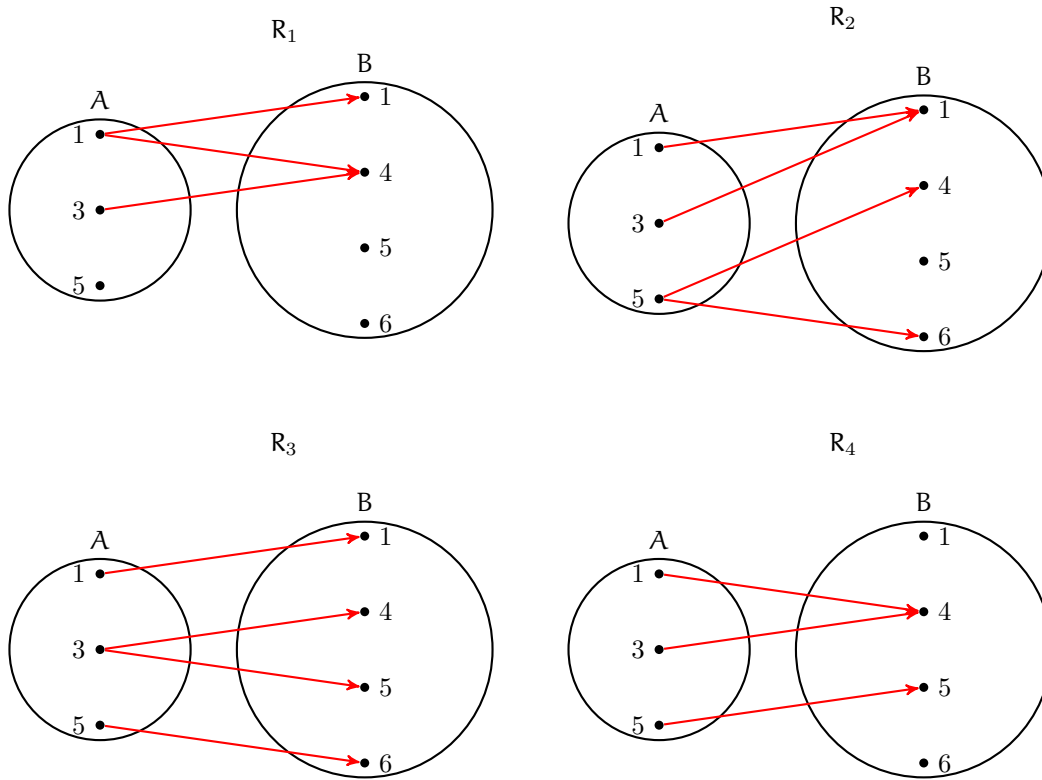


Figura 2.1: Las 4 relaciones del Ejemplo 2.2.1 sobre los conjuntos $A = \{1, 3, 5\}$ y $B = \{1, 4, 5, 6\}$.

Llamaremos al conjunto A el dominio de R , y a B el codominio de R .

Si $(x, y) \in R$, para una relación R , es común escribir xRy .

Ejemplo 2.2.1 Sean $A = \{1, 3, 5\}$ y $B = \{1, 4, 5, 6\}$. El producto cartesiano de esos conjuntos es $A \times B = \{(1, 1), (1, 4), (1, 5), (1, 6), (3, 1), (3, 4), (3, 5), (3, 6), (5, 1), (5, 4), (5, 5), (5, 6)\}$. Algunos ejemplos de relaciones binarias sobre esos conjuntos son las siguientes:

- $R_1 = \{(1, 1), (1, 4), (3, 4)\}$;
- $R_2 = \{(1, 1), (3, 1), (5, 4), (5, 6)\}$;
- $R_3 = \{(1, 1), (3, 4), (3, 5), (5, 6)\}$; y
- $R_4 = \{(1, 4), (3, 4), (5, 5)\}$.

La Figura 2.1 muestra estas relaciones de forma gráfica. Allí, se representan los conjuntos A y B usando diagramas de Venn. Las relaciones, por otro lado, son representadas con flechas: por cada par (x, y) perteneciente a una relación, se dibuja una flecha $x \rightarrow y$.

Definición 2.2.2 Dada una relación $R \subseteq A \times B$, definimos la relación inversa $R^{-1} \subseteq B \times A$ tal que

$$R^{-1} = \{(y, x) \mid (x, y) \in R\}. \quad (2.1)$$

Tipos de Relaciones

Existen diferentes tipos de relaciones, dependiendo de las propiedades que cumplan. Revisamos a continuación algunas de ellas.

Relación total: Una relación $R \subseteq A \times B$ es total si $\forall x \in A, \exists y \in B$ tal que $(x, y) \in R$, y se denota $R : A \dashv\vdash B$.

Función: Una relación $R \subseteq A \times B$ es función si $\forall x \in A, \forall y, y' \in B, (x, y) \in R \wedge (x, y') \in R \Rightarrow y = y'$, y se denota $R : A \longrightarrow B$. Si R es una función, denotaremos $R(x) = y$ si y sólo si $(x, y) \in R$. Alternativamente, también diremos que R mapea x en y , y por lo tanto usaremos *mapeo* como sinónimo de función.

Relación inyectiva: Una relación $R \subseteq A \times B$ es inyectiva si $\forall x, x' \in A, \forall y \in B, (x, y) \in R \wedge (x', y) \in R \Rightarrow x = x'$, y se denota $R : A \longleftarrow B$.

Relación sobreyectiva (o suryectiva): Una relación $R \subseteq A \times B$ es sobreyectiva —o suryectiva— si $\forall y \in B, \exists x \in A$ tal que $(x, y) \in R$, y se denota $R : A \twoheadrightarrow B$.

Bijección (mapeo uno a uno): Una relación $R \subseteq A \times B$ es una biyección —o mapeo uno a uno— si cumple con ser función, total, inyectiva, y sobreyectiva.

Ejemplo 2.2.2 Las relaciones de la Figura 2.1 cumplen con:

- R_1 no es total, no es función, no es inyectiva, y no es sobreyectiva.
- R_2 es total, no es función, no es inyectiva, y no es sobreyectiva.
- R_3 es total, no es función, es inyectiva, y es sobreyectiva.
- R_4 es total, es función, no es inyectiva, no es sobreyectiva.

Ninguna de esas relaciones es una biyección.

Estas propiedades pueden comprobarse de forma gráfica sobre los diagramas de la Figura 2.1:

- Una relación es total si se observa al menos una flecha saliendo desde cada $x \in A$.
- Una relación es función si a lo más una flecha sale desde cada $x \in A$.
- Una relación es inyectiva si a lo más una flecha llega a cada $y \in B$.
- Una relación es sobreyectiva si hay al menos una flecha llegando a cada $y \in B$.

Ejemplo 2.2.3 Sean $A = \{\text{Santiago, Valparaíso, Concepción, Futrono, Temuco}\}$, y $B = \{1, 2, 3, 4, 5\}$. Consideremos la relación $R \subseteq A \times B$:

$$R = \{(\text{Concepción}, 1), (\text{Futrono}, 2), (\text{Santiago}, 3), (\text{Temuco}, 4), (\text{Valparaíso}, 5)\}.$$

Esta relación es una biyección: note cómo mapea cada uno de los elementos de A en B . Una posible interpretación de la relación es que mapea nombres de ciudades en Chile a su correspondiente rango lexicográfico dentro del conjunto A . Por ejemplo, Concepción tiene rango lexicográfico 1, ya que $R(\text{Concepción}) = 1$.

Una propiedad importante de las biyecciones es la siguiente.

Lema 2.2.1 Si $R \subseteq A \times B$ es una biyección, entonces $|A| = |B|$.

Como resultado, una aplicación es la de demostrar que dos conjuntos A y B tienen la misma cardinalidad: sólo necesitamos definir una biyección entre ellos para probarlo. Esto parece una tarea trivial cuando A y B son conjuntos finitos pequeños, ya que la biyección se puede definir por extensión. Sin embargo, para conjuntos de tamaño suficientemente grande o infinitos, una definición por comprensión es preferible ⁶, lo cual en general deja de ser trivial.

6: Si los conjuntos son infinitos, la biyección debe ser, obligatoriamente, definida por comprensión.

Relaciones con Dominio y Codominio Coincidente

Un caso particular de interés son las relaciones $R \subseteq A \times A$, para algún conjunto no vacío A . En este caso, se dice que R es una *relación sobre A* . Por ejemplo, podemos definir la relación $\leq = \{(x, y) \in \mathbb{N} \times \mathbb{N} \mid x \leq y\}$. En este caso, hablamos de *la relación \leq sobre los números naturales*. Para este tipo de relaciones se definen las siguientes propiedades:

Reflexiva: Una relación $R \subseteq A \times A$ es reflexiva si $\forall x \in A, (x, x) \in R$.

Simétrica: Una relación $R \subseteq A \times A$ es simétrica si $\forall x, y \in A, (x, y) \in R \Rightarrow (y, x) \in R$.

Antisimétrica: Una relación $R \subseteq A \times A$ es antisimétrica si $\forall x, y \in A, (x, y) \in R \wedge (y, x) \in R \Rightarrow x = y$.

Transitiva: Una relación $R \subseteq A \times A$ es transitiva si $\forall x, y, z \in A, (x, y) \in R \wedge (y, z) \in R \Rightarrow (x, z) \in R$.

Completa: Una relación $R \subseteq A \times A$ es completa si $\forall x, y \in A, (x, y) \in R \vee (y, x) \in R$.

Órdenes Parciales y Totales

Los *órdenes parciales* y *órdenes totales* son tipos particulares de relaciones binaria que serán de interés a lo largo de este libro —por ejemplo, para ordenar conjuntos, o para obtener estadísticas de orden en un conjunto. En particular, destacamos los siguientes:

Orden Parcial: Una relación $R \subseteq A \times A$ es de orden parcial si es reflexiva, antisimétrica, y transitiva.

Orden Total: Una relación $R \subseteq A \times A$ es de orden total si es un orden parcial y cumple con ser completa.

Ilustramos a continuación algunos ejemplos de órdenes totales que serán útiles más adelante.

La Relación de Inclusión de Conjuntos. Estudiemos la relación de inclusión de conjuntos \subseteq , definida sobre $\mathcal{P}(\mathbb{N})$. Dicha relación contiene todos los pares de conjuntos de números naturales tal que uno está incluido en el otro. Analizamos a continuación sus propiedades:

- Es reflexiva: para todo conjunto $S \in \mathcal{P}(\mathbb{N})$ se cumple $S \subseteq S$.
- Es antisimétrica: para todo par de conjuntos $S_1, S_2 \in \mathcal{P}(\mathbb{N})$, si $S_1 \subseteq S_2$ y $S_2 \subseteq S_1 \Rightarrow S_1 = S_2$.

- Es transitiva: para todo $S_1, S_2, S_3 \in \mathcal{P}(\mathbb{N})$, si $S_1 \subseteq S_2$ y $S_2 \subseteq S_3 \Rightarrow S_1 \subseteq S_3$.
- No es completa: por ejemplo, para $S_1 = \{1, 2\}$ y $S_2 = \{2, 3, 4\}$ se cumple que $S_1 \not\subseteq S_2$ y $S_2 \not\subseteq S_1$.

Podemos concluir que la relación \subseteq sobre conjuntos es un orden parcial, pero no un orden total.

La Relación \leq sobre los Naturales. Estudiemos ahora la relación \leq , definida sobre los números naturales, que contiene todos los pares de números tal que el primero es menor o igual que el segundo. Dicha relación tiene las siguientes propiedades:

- Es reflexiva: para todo número natural $n \in \mathbb{N}$ se cumple $n \leq n$.
- Es antisimétrica: para todo par de números naturales $n_1, n_2 \in \mathbb{N}$ se tiene que si $n_1 \leq n_2$ y $n_2 \leq n_1 \Rightarrow n_1 = n_2$.
- Es transitiva: para todo $n_1, n_2, n_3 \in \mathbb{N}$ se cumple que $n_1 \leq n_2$ y $n_2 \leq n_3 \Rightarrow n_1 \leq n_3$.
- Es completa: para todo $n_1, n_2 \in \mathbb{N}$ se cumple que $n_1 \leq n_2$ o $n_2 \leq n_1$.

Podemos concluir que la relación \leq es un orden total sobre los naturales ⁷.

7: De hecho, también es un orden total sobre los enteros y reales.

Un Orden Total Sobre Subconjuntos de \mathbb{N}_n . Ilustramos a continuación una posible —aunque no necesariamente realista— relación de orden total sobre subconjuntos de \mathbb{N}_n , para $n \in \mathbb{N}$. Dado un conjunto $S \subseteq \mathbb{N}_n$, la idea es representarlo mediante un arreglo ⁸ de bits $B_S[1..n]$, de forma tal que para todo $i \in S$ se tiene $B_S[i] = 1$, y $B_S[i] = 0$ si $i \notin S$. Por ejemplo, para $n = 8$ y $S = \{1, 2, 4, 7\}$, tenemos $B_S[1..8] = (1, 1, 0, 1, 0, 0, 1, 0)$. El arreglo B_S es conocido como el *vector característico* del conjunto S .

8: O secuencia, o vector.

Luego, cada conjunto podría ser interpretado como un número binario de n bits, cuyo valor es

$$v_n(S) = \sum_{i=1}^n B_S[i] \cdot 2^{i-1}. \quad (2.2)$$

Para el conjunto S del ejemplo anterior, se tiene que $v_8(S) = 2^0 + 2^1 + 2^3 + 2^6 = 75$. De esta forma, a cada conjunto se asocia un número natural que lo representa en el intervalo $[0..2^n-1]$. De hecho, la relación entre conjuntos y su correspondiente entero $v_n(S)$ es una biyección. Definimos a continuación la relación

$$\sqsubseteq = \{(S_1, S_2) \in \mathcal{P}(\mathbb{N}_n) \times \mathcal{P}(\mathbb{N}_n) \mid v_n(S_1) \leq v_n(S_2)\}. \quad (2.3)$$

Dado que \leq es una relación de orden total sobre los naturales, \sqsubseteq también es un orden total sobre subconjuntos de números naturales.

Un Orden Total para Conjuntos de Intervalos. Sea \mathcal{I} el conjunto de todos los intervalos válidos sobre \mathbb{N} , y sea \leq una relación sobre \mathcal{I} tal

que

$$\leq = \{(I_i, I_j) \in \mathcal{I} \times \mathcal{I} \mid I_i.s \leq I_j.s\}, \quad (2.4)$$

en donde para un intervalo I_i , la notación $I_i.s$ hace referencia al extremo inferior del intervalo. De esta manera, la relación \leq permite comparar dos intervalos en términos de sus extremos inferiores. Dado esto, note que la relación \leq cumple con las propiedades reflexiva, antisimétrica, transitiva, además de ser completa, por lo que es un orden total sobre \mathcal{I} .

Un orden total similar podría definirse usando los extremos superiores de los intervalos. Alternativamente, uno podría definir un orden que involucre a ambos extremos de los intervalos, de la siguiente manera:

$$\preceq = \{(I_i, I_j) \in \mathcal{I} \times \mathcal{I} \mid I_i.s \leq I_j.s \wedge I_i.e \leq I_j.e\}, \quad (2.5)$$

en donde $I_i.e$ denota el extremo superior del intervalo I_i . Aunque esta relación cumple con las propiedades reflexiva, antisimétrica, y transitiva, no cumple con ser completa: para los intervalos $[1..5]$ y $[2..3]$, por ejemplo, se tiene que $[1..5] \not\preceq [2..3]$ y $[2..3] \not\preceq [1..5]$. Por lo tanto, \preceq es un orden parcial pero no un orden total.

Un Orden Total para Conjuntos de Puntos. Las relaciones definidas anteriormente sobre conjuntos de intervalos se pueden extender para conjuntos de puntos en el plano \mathbb{R}^2 , definiendo órdenes totales usando sólo una de las coordenadas de los puntos. Similarmente, para la relación:

$$\preceq = \{(p_i, p_j) \in \mathbb{R} \times \mathbb{R} \mid p_i.x \leq p_j.x \wedge p_i.y \leq p_j.y\}, \quad (2.6)$$

se cumple que es un orden parcial pero no total. Dados dos puntos p_i y p_j , si $p_i \preceq p_j$ se dice que p_j *domina* al punto p_i . Todas estas relaciones se pueden extender a puntos de dimensiones mayores, con las mismas propiedades.

Órdenes Totales y Ordenamiento de Conjuntos. Las relaciones de orden total serán esenciales para ciertos problemas que estudiaremos en adelante. Por ejemplo, una relación de orden total definida sobre un conjunto A permite que éste pueda ser ordenado, tal como lo ilustra el siguiente ejemplo.

Ejemplo 2.2.4 Sea $A = \{1, 3, 7, 9\}$, y sea $\preceq = \{(3, 3), (3, 9), (3, 1), (3, 7), (9, 9), (9, 1), (9, 7), (1, 1), (1, 7), (7, 7)\}$ una relación arbitraria que cumple con ser un orden total. Este orden total permite ordenar al conjunto A de la siguiente manera: $3 \preceq 9 \preceq 1 \preceq 7$.

Aunque la definición matemática de conjunto no implica ningún orden entre sus elementos ⁹, en ciencia de la computación *ordenar un conjunto* significa organizar sus elementos de manera que se puedan acceder de forma ordenada. En general, ordenar un conjunto permite resolver algunas operaciones sobre él de forma eficiente, y de allí su importancia. Notablemente, la operación de determinar la pertenencia de un elemento

9: Por ejemplo, el conjunto $\{1, 2, 3, 4\}$ es igual al conjunto $\{3, 2, 1, 4\}$.

en un conjunto es una de las más beneficiadas si se ordena el conjunto: piense cuánto más simple es buscar a una persona en un listado ordenado alfabéticamente.

Note que la existencia de un orden total sobre un conjunto finito de n elementos implica que el conjunto tiene un elemento mínimo y un elemento máximo respecto a ese orden. Es más, en general tiene un elemento k -ésimo: el elemento que ocupa la k -ésima posición si se ordena el conjunto, para $1 \leq k \leq n$.

Definición 2.2.3 Sea A un conjunto finito y no vacío sobre el que se ha definido una relación de orden total R . Se define el rango —u ordinal— de un elemento $x \in A$ como

$$\text{rango}(x) = |A| - |\{(x, y) \in R\}| + 1.$$

Por ejemplo, para el conjunto A y la relación \preceq definida en el Ejemplo 2.2.4, tenemos $\text{rango}(3) = 1$, $\text{rango}(9) = 2$, $\text{rango}(1) = 3$, y $\text{rango}(7) = 4$.

Definición 2.2.4 Para un conjunto A sobre el que se ha definido una relación de orden total, definimos el estadístico de orden k de A como el elemento $x \in A$ tal que $\text{rango}(x) = k$.

Obtener estadísticos de orden k de un conjunto es importante en la práctica, en particular para herramientas estadísticas. Un ejemplo típico es obtener la mediana de un conjunto: el elemento con rango $\lfloor \frac{n+1}{2} \rfloor$.

Respecto a los órdenes definidos anteriormente, podemos concluir que la relación \subseteq no permite ordenar conjuntos de conjuntos, y que un conjunto de puntos puede ser ordenado en base a una de sus coordenadas, pero no en base a todas como lo plantea la relación \preceq .

Órdenes Totales Definidos por Extensión y Comprensión. Como todo conjunto, las relaciones de orden total pueden definirse por extensión o comprensión. Computacionalmente, la manera en que se definan tiene consecuencias sobre la eficiencia de los algoritmos que necesiten manipularlos. Una definición por comprensión no necesita almacenar los pares ordenados que definen el orden, sino que debe existir un método para determinar si un par ordenado dado pertenece o no a la relación. En los ejemplos estudiados anteriormente, las relaciones \leq (sobre los naturales) y \subseteq (sobre subconjuntos de los naturales) pueden ser definidas por comprensión:

- \leq : usualmente, ciertos modelos de computación como las RAM¹⁰ tienen instrucciones que permiten comparar dos naturales de forma rápida¹¹ y determinar la relación de orden entre ellos¹².
- \subseteq : las Ecuaciones (2.2) y (2.3) muestran la manera de comparar dos conjuntos sin necesidad de almacenar todos los pares de la relación. Sin embargo, esta vez el cómputo necesita más trabajo que en el caso de \leq , ya que no puede resolverse mediante una única instrucción.

Por otro lado, tenemos las definiciones por extensión, como la relación \preceq del Ejemplo 2.2.4. Éstas permiten, en principio, definir cualquier

10: Por Random Access Machine, en inglés.

11: En menos de 1 nanosegundo en los computadores actuales.

12: De hecho, también existen instrucciones para comparar enteros y números reales.

orden necesario, lo cual es una ventaja. Sin embargo, las consecuencias computacionales son las siguientes:

- Necesita espacio para almacenar la relación, el cual puede ser excesivo si el conjunto A sobre el que se define el orden es muy grande.
- Para poder determinar si se cumple $x \preceq y$ o no, se debe determinar si $(x, y) \in \preceq$ o no. Es decir, debemos determinar la pertenencia del par (x, y) en la relación de orden, lo cual necesita tiempo de cómputo no despreciable, como veremos más adelante.

Notación para Órdenes Totales. En adelante, usaremos \preceq para denotar una relación de orden total genérica. Además, para simplificar usaremos las siguientes notaciones basadas en el orden \preceq :

- $x \preceq y$ denota $(x, y) \in \preceq$.
- $x \triangleleft y$ denota $(x, y) \in \preceq \wedge x \neq y$.
- $x \succeq y$ denota $(y, x) \in \preceq$.
- $x \triangleright y$ denota $(y, x) \in \preceq \wedge x \neq y$.

Aún cuando usemos el símbolo $x \preceq y$ para comparar dos elementos x e y , lo leeremos como “ x menor o igual a y ” o “ x precede a y ”.

Relaciones de Equivalencia

Definición 2.2.5 Una relación $R \subseteq A \times A$ es una relación de equivalencia si es reflexiva, simétrica, y transitiva.

Si R es una relación de equivalencia y se cumple $(x, y) \in R$, entonces diremos que x e y son equivalentes en R . De forma similar a las relaciones de orden, las relaciones de equivalencia serán clave para resolver diferentes problemas, como estudiaremos más adelante.

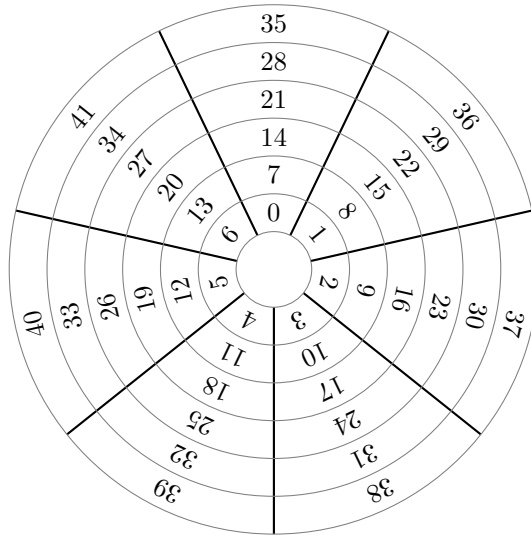
La Relación de Congruencia Módulo entre Números Naturales. Consideremos la relación de congruencia entre números naturales, que definimos en lo que sigue. Dados $a \in \mathbb{N}^0$ y $n \in \mathbb{N}$, definimos la operación módulo como:

$$a \bmod n = a - \left\lfloor \frac{a}{n} \right\rfloor n. \quad (2.7)$$

En otras palabras, $a \bmod n$ es el resto de la división entera a/n . Eso significa que a es divisible por n si y sólo si $a \bmod n = 0$.

Luego, dados $a, b \in \mathbb{N}^0$ y $n \in \mathbb{N}$, diremos que a y b son *congruentes módulo n* , denotado $a \equiv b \pmod{n}$, si existe $k \in \mathbb{Z}$ tal que $a - b = kn$. Esto es, la diferencia entre a y b es múltiplo de n o, alternativamente, $a \bmod n = b \bmod n$. La Figura 2.2 muestra un diagrama que permite ver la congruencia módulo 7 de los naturales \mathbb{N}_{41}^0 . Allí se puede ver, por ejemplo, que $19 \equiv 12 \pmod{7}$, ya que $19 - 12 = 1 \cdot 7$, y $19 \equiv 5 \pmod{7}$ ya que $19 - 5 = 2 \cdot 7$. De hecho, todos los números que son equivalentes módulo 7 se encuentran en la misma partición del círculo.

Consideremos entonces la relación $R_{\equiv n} = \{(a, b) \mid a, b \in \mathbb{N}^0, a \equiv b \pmod{n}\}$. Dicha relación tiene las siguientes propiedades:



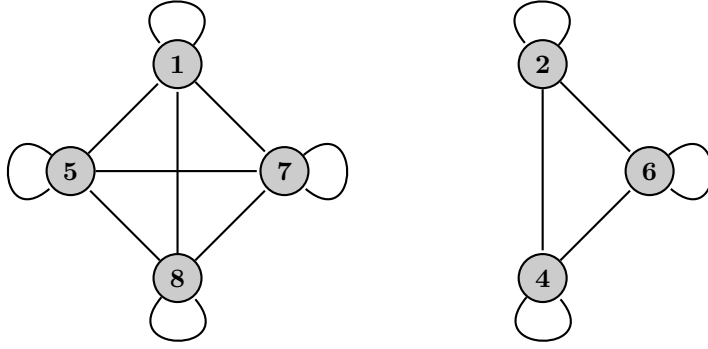


Figura 2.3: Diagrama para la relación de equivalencia R del Ejemplo 2.2.5, definida sobre el conjunto $A = \{1, 2, 4, 5, 6, 7, 8\}$. Note cómo la relación de equivalencia particiona al conjunto A en 2 clases de equivalencia, $[1] = \{1, 5, 7, 8\}$ y $[2] = \{2, 4, 6\}$.

elección de esos elementos como representantes de las clases es arbitraria, en general cualquier otro elemento podría usarse como representante.

Consideremos nuevamente el caso de la congruencia entre números naturales. Al ser la relación $a \equiv b \pmod{n}$ de equivalencia, podemos definir la clase de equivalencia de $a \in \mathbb{N}^0$ de la siguiente manera:

$$[a]_{\equiv_n} = \{b \in \mathbb{N}^0 \mid b \equiv a \pmod{n}\}.$$

Para el ejemplo de la Figura 2.2, tenemos $[5]_{\equiv_7} = \{5, 12, 19, 26, 33, 40, \dots\}$. Nuevamente, las 7 distintas clases de equivalencia del ejemplo se pueden ver claramente en la figura a partir de los arcos que se han agregado al círculo.

Definición 2.2.7 (Partición de un conjunto) *Una partición de un conjunto A es una familia de subconjuntos $P = \{A_1, \dots, A_p\}$, con $p \geq 1$, tal que se cumple:*

- Para $1 \leq i \leq p$, $A_i \neq \emptyset$.
- Para $1 \leq i, j \leq p$, $i \neq j \Rightarrow A_i \cap A_j = \emptyset$.
- $\bigcup_{i=1}^p A_i = A$.

Llamaremos parte a cada una de los conjuntos A_i que conforman una partición.

Toda relación de equivalencia sobre un conjunto A induce una partición de A —i.e., las clases de equivalencia—, y viceversa, tal como lo indica el siguiente teorema.

Teorema 2.2.2 *Toda relación de equivalencia $R \subseteq A \times A$ induce una partición $P_R = \bigcup_{x \in A} [x]$ sobre el conjunto A .*

Demostración. Probamos las 3 propiedades necesarias para que P_R sea una partición. (1) Obviamente, cada una de las clases de equivalencia es no vacía, porque al menos contienen al representante de la clase. (2) Además, las clases de equivalencia son disjuntas de a pares, lo que puede ser probado usando el siguiente argumento: si dos elementos x e y pertenecen a distintas clases de equivalencia, entonces $(x, y) \notin R$, por lo que $[x] \cap [y] = \emptyset$. (3) Finalmente, ya que R es reflexiva, todo elemento $x \in A$ aparece en al menos un par en R , y por lo tanto $\bigcup_{A_i \in P_R} A_i = A$. ■

Teorema 2.2.3 *Toda partición de un conjunto A induce una relación de equivalencia $R \subseteq A \times A$.*

Demostración. Sea $P = \{A_1, \dots, A_p\}$ una partición de un conjunto no vacío A . A partir de P , definimos la siguiente relación:

$$R = \{(x, y) \mid x, y \in A_i, 0 \leq i \leq p\}.$$

Esta relación es de equivalencia, ya que es reflexiva, simétrica, y transitiva. ■

Como resultado, uno puede representar una relación de equivalencia manteniendo la correspondiente partición, ya que son equivalentes. Estudiaremos en capítulos posteriores estructuras de datos que permiten mantener una partición de un conjunto, y por lo tanto la correspondiente relación de equivalencia.

2.3. La Función Factorial

Definición 2.3.1 *Dado un número natural $n \in \mathbb{N}^0$, se define la función $n!$ (el factorial de n) como*

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1,$$

para $n > 0$, y $0! = 1$.

A continuación estudiaremos algunos aspectos relacionados a esta función que serán de utilidad. Note que para n suficientemente grande, $n!$ crece más rápidamente que a^n , para toda constante $a \in \mathbb{N}$. Por otro lado, $n!$ crece más lentamente que n^n . En muchas situaciones necesitaremos acotar alguna función $f(n!)$, como por ejemplo $\lg(n!)$. Usar aproximaciones del factorial en estos casos suele ser útil, ya que se basan en otro tipo de funciones más simples de acotar. Revisamos a continuación las aproximaciones más conocidas para $n!$.

Una manera simple de aproximar funciones como $\lg(n!)$ es usar el hecho de que $n! \leq n^n$ para $n \geq 0$ ¹³, por lo que:

$$\lg(n!) \leq \lg(n^n) = n \lg n,$$

en donde la igualdad de la derecha es obtenida por propiedades de logaritmos. Aunque esta aproximación es poco ajustada, suele ser suficiente en aplicaciones para las que el error introducido sea tolerable.

En este caso, se aproxima $n!$ usando su logaritmo natural, $\ln(n!) = \sum_{i=1}^n \ln i$. Dado que la función \ln es creciente, esta suma puede acotarse usando integrales de la siguiente forma:

$$\int_1^n \ln(x) dx \leq \sum_{i=1}^n \ln i \leq \int_0^n \ln(x+1) dx,$$

13: Asumiremos $0^0 = 1$.

lo cual, resolviendo, produce:

$$n \ln \left(\frac{n}{e} \right) + 1 \leq \ln n! \leq (n+1) \ln \left(\frac{n+1}{e} \right) + 1,$$

lo cual nuevamente nos indica que $\ln n! \sim n \ln n$. Para despejar $n!$, se aplica la función exponencial en los tres lados de la desigualdad y se llega a:

$$\left(\frac{n}{e} \right)^n e \leq n! \leq \left(\frac{n+1}{e} \right)^{n+1} e. \quad (2.8)$$

Una mejor aproximación para $n!$ es la de Stirling, la cual tiene un error más bajo que la anterior cuando n es grande. Esta aproximación establece que, para $n \geq 1$, se cumple:

$$n! = \sqrt{2\pi n} \left(\frac{n}{e} \right)^n \left(1 + \frac{1}{12n} + \varepsilon(n) \right), \quad (2.9)$$

en donde el error es $\varepsilon(n) \leq c/n^2$, para una constante $c \in \mathbb{N}$. Note que esta aproximación también implica que

$$\left(\frac{n}{e} \right)^n \leq n!. \quad (2.10)$$

Usando esta aproximación, se puede aproximar $\lg(n!)$ de la siguiente manera:

$$\begin{aligned} \lg(n!) &\approx \lg \left(\sqrt{2\pi n} \left(\frac{n}{e} \right)^n \right) \\ &= n \lg n - n \lg e + \frac{\lg n}{2} + \frac{\lg 2\pi}{2}. \end{aligned} \quad (2.11)$$

Finalmente, la aproximación de Ramanujan establece que para $n \geq 1$ se cumple:

$$n! = \sqrt{2\pi n} \left(\frac{n}{e} \right)^n \left(1 + \frac{1}{2n} + \frac{1}{8n^2} \right)^{1/6}. \quad (2.12)$$

Ésta es una aproximación levemente más precisa que la de Stirling, aunque la tasa de error de ambas sean del mismo orden. Sin embargo, la de Stirling será suficiente para los propósitos de este libro.

2.4. Sucesiones, Strings, Permutaciones, Combinaciones

Sucesiones

Una *sucesión* o *secuencia* infinita es una función discreta $f: \mathbb{N} \mapsto S$, que mapea números naturales a elementos de otro conjunto S . Llamaremos términos de la sucesión a cada uno de los $f(i)$, para $i \in \mathbb{N}$. Para simplificar, usualmente usaremos la notación f_i para referirnos a $f(i)$. Al escribir los términos de una sucesión usaremos la notación $\langle f_1, f_2, \dots \rangle$. Aquí se puede notar el hecho de que una sucesión le asigna

un valor f_i a cada natural i . A diferencia de un conjunto, al escribir los elementos de una sucesión el orden de los elementos importa. Además, una sucesión puede tener elementos repetidos —no es requisito que f sea inyectiva. De forma similar se puede definir una sucesión finita $f : \mathbb{N}_n \mapsto S$ para un número natural n , denotada $\langle f_1, \dots, f_n \rangle$. Para una sucesión finita, es común denotar $f[i] = f_i$.

Strings

Un *string* o *cadena* es una sucesión finita $s : \mathbb{N}_n \mapsto \Sigma$, en donde $\Sigma = \mathbb{N}_{\sigma-1}^0$ es llamado el *alfabeto* del string y $|s| = n$ denotará la *longitud* —o *largo*— del string. Por ejemplo, si $\Sigma = \{0, 1\}$ es el alfabeto binario, algunos posibles strings sobre ese alfabeto son 0 , 1 , 00 , 01 , 10 , 11 , y así siguiendo¹⁴. Por conveniencia, definimos el string vacío, denotado ε , como un string que no tiene ningún elemento. Así, $|\varepsilon| = 0$.

14: Al mostrar los elementos de un string en general omitiremos los corchetes angulares de la sucesión y las comas que separan los elementos de la misma. De esta manera, $00101 = \langle 0, 0, 1, 0, 1 \rangle$.

Ejemplo 2.4.1 Para $\Sigma = \{s_1, s_2, s_3, s_4\}$, los posibles strings distintos de longitud 2 son:

- $s_1s_1, s_1s_2, s_1s_3, s_1s_4,$
- $s_2s_1, s_2s_2, s_2s_3, s_2s_4,$
- $s_3s_1, s_3s_2, s_3s_3, s_3s_4,$
- $s_4s_1, s_4s_2, s_4s_3, s_4s_4.$

Dado que para cada uno de los n elementos de un string podemos elegir cualquiera de los σ elementos de Σ , permitiendo repeticiones, se tiene que la cantidad de strings distintos de longitud n es:

$$S(\sigma, n) = \sigma^n. \quad (2.13)$$

Finalmente, denotaremos Σ^* al conjunto infinito de todos los strings de longitud ≥ 0 formados a partir de símbolos del alfabeto Σ . Eso implica que $\varepsilon \in \Sigma^*$ y $\Sigma \subset \Sigma^*$. Además, denotaremos $\Sigma^+ = \Sigma^* - \{\varepsilon\}$.

Permutaciones

Definición 2.4.1 Una *permutación* $\langle a_1, \dots, a_n \rangle$ de un conjunto finito S de n elementos de algún tipo es una función biyectiva $\pi : \mathbb{N}_n \mapsto S$ tal que $\pi(i) = a_i$, para $1 \leq i \leq n$.

Por ejemplo, para $S = \{\text{Concepción}, \text{Futrono}, \text{Santiago}, \text{Temuco}\}$, una posible permutación está dada por la función

$$\pi = \{(2, \text{Concepción}), (4, \text{Futrono}), (1, \text{Santiago}), (3, \text{Temuco})\}.$$

Usando notación de sucesión, tenemos

$$\langle \pi(1), \pi(2), \pi(3), \pi(4) \rangle = \langle \text{Santiago}, \text{Concepción}, \text{Temuco}, \text{Futrono} \rangle.$$

Informalmente podemos decir que una permutación “muestra” los elementos de S en un orden dado, sin repeticiones. Al escribir una permutación particular, por simplicidad en general usaremos $\pi(1), \pi(2), \dots, \pi(n)$, omitiendo los corchetes angulares de la sucesión.

Ejemplo 2.4.2 Si $S = \{s_1, s_2, s_3\}$, las distintas permutaciones de S son las siguientes 6:

$s_1, s_2, s_3; s_1, s_3, s_2; s_2, s_1, s_3; s_2, s_3, s_1; s_3, s_1, s_2; s_3, s_2, s_1;$

mientras que si $S = \{s_1, s_2, s_3, s_4\}$, las distintas permutaciones de S son las siguientes 24:

- $s_1, s_2, s_3, s_4; s_1, s_2, s_4, s_3; s_1, s_3, s_2, s_4; s_1, s_3, s_4, s_2; s_1, s_4, s_2, s_3;$
 $s_1, s_4, s_3, s_2;$
- $s_2, s_1, s_3, s_4; s_2, s_1, s_4, s_3; s_2, s_3, s_1, s_4; s_2, s_3, s_4, s_1; s_2, s_4, s_1, s_3;$
 $s_2, s_4, s_3, s_1;$
- $s_3, s_2, s_1, s_4; s_3, s_2, s_4, s_1; s_3, s_1, s_2, s_4; s_3, s_1, s_4, s_2; s_3, s_4, s_2, s_1;$
 $s_3, s_4, s_1, s_2;$
- $s_4, s_2, s_3, s_1; s_4, s_2, s_1, s_3; s_4, s_3, s_2, s_1; s_4, s_3, s_1, s_2; s_4, s_1, s_2, s_3;$
 $s_4, s_1, s_3, s_2.$

Como lo sugiere el ejemplo anterior, la cantidad de permutaciones distintas para un conjunto de n elementos es $n! = n \times (n-1) \times (n-2) \times \cdots \times 1$. Esto es porque para seleccionar el primer elemento de la permutación tenemos n alternativas distintas; para el segundo tenemos $n-1$ alternativas, que son las que nos quedan luego de fijar uno como primer elemento; de la misma manera, el tercer elemento puede ser seleccionado de $n-2$ formas, porque ya fijamos los 2 primeros; y así siguiendo hasta que al n -ésimo elemento podemos seleccionarlo de una única forma. Dada que cada una de las cantidades de formas de elegir un elemento se debe combinar con la cantidad de formas de elegir el resto de elementos, todas esas cantidades deben multiplicarse produciendo $n!$ como resultado.

Una k -permutación de un conjunto S de n elementos es una permutación de $k \leq n$ elementos de S . Así, una permutación de S equivale a una n -permutación.

Ejemplo 2.4.3 Para $S = \{s_1, s_2, s_3, s_4\}$, las 12 2-permutaciones distintas son:

- $s_1, s_2; s_1, s_3; s_1, s_4,$
- $s_2, s_1; s_2, s_3; s_2, s_4,$
- $s_3, s_1; s_3, s_2; s_3, s_4,$
- $s_4, s_1; s_4, s_2; s_4, s_3.$

La cantidad de k -permutaciones distintas de un conjunto de n elementos es:

$$P(n, k) = n \times (n-1) \times \cdots \times (n-k+1) = \frac{n!}{(n-k)!}, \quad (2.14)$$

con un argumento similar al anterior, sólo que seleccionamos k elementos en lugar de n .

Observación 2.4.1 Si elegimos $k \leq n$ elementos arbitrarios de S , estos aparecen en $k!$ de las $P(n, k)$ k -permutaciones distintas.

La justificación es que son permutaciones de k elementos, por lo que en total son $k!$.

Permutaciones de \mathbb{N}_n . Las permutaciones de \mathbb{N}_n son un caso particular de permutaciones que se pueden definir formalmente de la siguiente manera.

Definición 2.4.2 Una permutación $\langle a_1, \dots, a_n \rangle$ de \mathbb{N}_n es una función biyectiva $\pi: \mathbb{N}_n \rightarrow \mathbb{N}_n$, tal que $\pi(i) = a_i$, para $1 \leq i \leq n$.

15: De hecho, también es una biyección y en este caso particular también una permutación de \mathbb{N}_n .

Dado que una permutación π es una biyección, su inversa π^{-1} tal que $\pi^{-1}(a_i) = i$ es también una función ¹⁵. Esto significa que dado que $\pi(i) = a_i$ es el elemento que está en la posición i de la permutación, $\pi^{-1}(a_i)$ es la posición de la permutación π en la que se ubica a_i .

Con esta definición, podemos representar una permutación π usando la siguiente notación de dos filas:

$$\pi = \begin{pmatrix} 1 & 2 & \cdots & n \\ a_1 & a_2 & \cdots & a_n \end{pmatrix}.$$

Ejemplo 2.4.4 Consideremos la siguiente permutación de \mathbb{N}_{10} : 3, 6, 4, 10, 9, 2, 7, 1, 5, 8. Su representación es:

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 3 & 6 & 4 & 10 & 9 & 2 & 7 & 1 & 5 & 8 \end{pmatrix}.$$

Esta notación permite una representación computacional muy simple para las permutaciones de \mathbb{N}_n : un arreglo $A_\pi[1..n]$ de manera que $A_\pi[i] = \pi(i)$, para $i = 1, \dots, n$. Para calcular $\pi^{-1}(a_i)$ usando el arreglo A_π , la idea es buscar la posición i del arreglo tal que $A_\pi[i] = a_i$, lo cual puede requerir revisar todo el arreglo. Otra alternativa es almacenar π^{-1} explícitamente en otro arreglo, lo cual duplica el uso de espacio pero permite calcular la permutación inversa más rápidamente.

Ciclos de una Permutación de \mathbb{N}_n . Estudiamos a continuación la estructura de ciclos de una permutación. Cada elemento de una permutación π de \mathbb{N}_n pertenece a un único ciclo de la permutación. Los elementos que conforman el ciclo del elemento i son: $i, \pi(i), \pi(\pi(i)), \pi(\pi(\pi(i)))$, y así siguiendo hasta obtener nuevamente i .

Ejemplo 2.4.5 La permutación

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 3 & 6 & 4 & 10 & 9 & 2 & 7 & 1 & 5 & 8 \end{pmatrix},$$

consiste de los siguientes 4 ciclos ¹⁶:

- $1 \rightarrow 3 \rightarrow 4 \rightarrow 10 \rightarrow 8 \rightarrow 1$,
- $2 \rightarrow 6 \rightarrow 2$,
- $5 \rightarrow 9 \rightarrow 5$,
- $7 \rightarrow 7$

16: Note cómo la estructura de ciclos de la permutación induce una relación de equivalencia —en este caso de 4 clases de equivalencia— entre los elementos de \mathbb{N}_{10} .

Note que $i \rightarrow j$ en la estructura de ciclos de una permutación no sólo nos dice que $\pi(i) = j$, sino que también $\pi^{-1}(j) = i$. Entonces, para calcular $\pi^{-1}(j)$ sólo necesitamos recorrer el ciclo de j hasta encontrar

$i \rightarrow j$. En algunos casos, eso permite calcular π^{-1} sin tener que revisar todo el arreglo, lo que es más eficiente. Como la estructura de ciclos es implícita a la permutación, sólo necesitamos almacenar el arreglo A_π . Para calcular $\pi^{-1}(i)$, se comienza con $j \leftarrow i$, y mientras $A_\pi[j] \neq i$ se itera haciendo $j \leftarrow A_\pi[j]$. Cuando esta condición se haga falsa, el valor actual de j es la respuesta. En el Ejemplo 2.4.5, para calcular $\pi^{-1}(6)$, se comienza con $j \leftarrow 6$. Luego, como $A_\pi[6] = 2 \neq 6$ se hace $j \leftarrow 2$. Finalmente, $A_\pi[2] = 6$ y el proceso se detiene, por lo que $\pi^{-1}(6) = 2$.

Combinaciones

Dado un conjunto finito S de $|S| = n$ elementos, una k -combinación de S es un subconjunto $S' \subseteq S$ tal que $|S'| = k$, para $k \leq n$.

Ejemplo 2.4.6 Para el conjunto $S = \{s_1, s_2, s_3, s_4\}$, las diferentes 2-combinaciones son:

- $\{s_1, s_2\}, \{s_1, s_3\}, \{s_1, s_4\},$
- $\{s_2, s_3\}, \{s_2, s_4\},$
- $\{s_3, s_4\}.$

Note la diferencia entre una k -permutación y una k -combinación. En las k -permutaciones el orden de los elementos sí importa, porque hablamos de secuencias —o sucesiones— de elementos. Por lo tanto las 2-permutaciones s_1s_2 y s_2s_1 son distintas. Por otro lado, en las k -combinaciones el orden de los elementos no importa, ya que hablamos de conjuntos. Por lo tanto, las 2-combinaciones $\{s_1, s_2\}$ y $\{s_2, s_1\}$ son iguales.

La cantidad de k -combinaciones distintas para un conjunto de n elementos es:

$$C(n, k) = \frac{n!}{k!(n-k)!} = \frac{P(n, k)}{k!}. \quad (2.15)$$

La justificación es que a cada una de las $C(n, k)$ k -combinaciones le corresponden $k!$ distintas k -permutaciones —recuerde la Observación 2.4.1. Por ejemplo, la 2-combinación $\{s_1, s_2\}$ del Ejemplo 2.4.6 corresponde a las $2! = 2$ siguientes 2-permutaciones que aparecen en el Ejemplo 2.4.3: s_1s_2 y s_2s_1 . Como en total hay $\frac{n!}{(n-k)!}$ k -permutaciones, se cumple que $C(n, k) \times k! = P(n, k)$. Despejando $C(n, k)$ obtenemos el resultado.

$C(n, k)$ es más conocido como *coeficiente binomial* y denotado como:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \quad (2.16)$$

lo cual se lee como *combinaciones de n sobre k* , o simplemente *n sobre k* , y corresponde al número de subconjuntos distintos de tamaño k que pueden formarse a partir de elementos tomados de un conjunto de tamaño n .

Propiedades y Cotas. Demostramos a continuación algunas propiedades y cotas importantes relacionados a las combinaciones.

Teorema 2.4.2 (Simetría del combinatorio) Para $n, k \in \mathbb{N}^0$ tal que $k \leq n$, se cumple

$$\binom{n}{k} = \binom{n}{n-k}.$$

Demostración. Simplemente hay que notar que

$$\binom{n}{n-k} = \frac{n!}{(n-k)!(n-(n-k))!} = \frac{n!}{(n-k)!k!} = \binom{n}{k}.$$

■

Teorema 2.4.3 (Teorema del binomio) Para todo par de números reales a y b , y $n \in \mathbb{N}$, se cumple

$$(a+b)^n = \sum_{k=0}^n \binom{n}{k} a^{n-k} b^k.$$

Antes de demostrar este teorema, estudiemos el siguiente caso particular para ganar intuición:

$$\begin{aligned} (a+b)^3 &= (a+b)(a+b)(a+b) \\ &= \underbrace{aaa}_{\binom{3}{0}a^3b^0} + \underbrace{aab + aba + baa}_{\binom{3}{1}a^2b} + \underbrace{abb + bab + bba}_{\binom{3}{2}ab^2} + \underbrace{bbb}_{\binom{3}{3}a^0b^3} \\ &= a^3 + 3a^2b + 3ab^2 + b^3 \\ &= \sum_{k=0}^3 \binom{3}{k} a^{3-k} b^k. \end{aligned}$$

Demostración (Teorema del binomio). Para demostrar el teorema, es suficiente entender que en $(a+b)^n$, el término $a^{n-k}b^k$ aparece $\binom{n}{k}$ veces, con $k = 0, \dots, n$. ■

Un caso particular de este teorema es para $a = b = 1$, produciendo el siguiente resultado.

Corolario 2.4.4 $2^n = \sum_{k=0}^n \binom{n}{k}$.

Notar que estamos sumando

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \dots + \binom{n}{n},$$

esto es, la cantidad de subconjuntos de tamaño 0, la cantidad de subconjuntos de tamaño 1, y así hasta la cantidad de subconjuntos de tamaño n . En otras palabras, estamos sumando la cantidad de subconjuntos distintos de un conjunto de tamaño n , lo cual es 2^n , la cardinalidad del conjunto de partes $\mathcal{P}(S)$.

Demostramos a continuación una cota importante para $\binom{n}{k}$.

Teorema 2.4.5 Para $1 \leq k \leq n$ se cumple

$$\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \left(\frac{en}{k}\right)^k.$$

Demostración. Para demostrar la cota inferior, tenemos:

$$\begin{aligned} \binom{n}{k} &= \frac{n(n-1) \cdots (n-k+1)(n-k)!}{k!(n-k)!} \\ &= \frac{n(n-1) \cdots (n-k+1)}{k(k-1) \cdots 1} \\ &= \left(\frac{n}{k}\right) \left(\frac{n-1}{k-1}\right) \cdots \left(\frac{n-k+1}{1}\right) \\ &\geq \left(\frac{n}{k}\right)^k. \end{aligned}$$

Para la cota superior, recuerde que la aproximación de Stirling también implica que $k! \geq \left(\frac{k}{e}\right)^k$ —recuerde la Ecuación (2.10). Entonces podemos acotar

$$\begin{aligned} \binom{n}{k} &= \frac{n(n-1) \cdots (n-k+1)}{k(k-1) \cdots 1} \\ &\leq \frac{n^k}{k!} \\ &\leq \left(\frac{en}{k}\right)^k. \end{aligned}$$

■

A partir de esto podemos obtener el siguiente resultado, que será de importancia más adelante para demostrar algunas cotas inferiores. La idea es simplemente aplicar \lg a cada lado de la desigualdad del Teorema 2.4.5, y luego usar propiedades de los logaritmos.

Corolario 2.4.6 Para $1 \leq k \leq n$ se cumple

$$k \lg \frac{n}{k} \leq \lg \binom{n}{k} \leq k \lg e + k \lg \frac{n}{k}.$$

Para concluir esta sección, estudiemos una definición alternativa de $\binom{n}{k}$ a través del conocido Teorema de Pascal, y el concepto del triángulo de Pascal que definimos a continuación. Sea C una matriz de dos dimensiones infinita hacia la derecha y abajo, tal que para $0 \leq k \leq n$ se define $C[n, k] = \binom{n}{k}$. Dado que $k \leq n$, la matriz es triangular inferior,

y tiene la siguiente forma:

$$\begin{array}{ccccccc}
 & & & & & & \binom{0}{0} \\
 & & & & & & \binom{1}{0} & \binom{1}{1} \\
 & & & & & \binom{2}{0} & \binom{2}{1} & \binom{2}{2} \\
 & & & \binom{3}{0} & \binom{3}{1} & \binom{3}{2} & \binom{3}{3} \\
 & \binom{4}{0} & \binom{4}{1} & \binom{4}{2} & \binom{4}{3} & \binom{4}{4} \\
 \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots
 \end{array}$$

Si acomodamos un poco las entradas de la matriz obtenemos la versión más conocida del triángulo:

$$\begin{array}{cccccccc}
 & & & & & & & \binom{0}{0} \\
 & & & & & & \binom{1}{0} & \binom{1}{1} \\
 & & & & \binom{2}{0} & \binom{2}{1} & \binom{2}{2} \\
 & & \binom{3}{0} & \binom{3}{1} & \binom{3}{2} & \binom{3}{3} \\
 \binom{4}{0} & \binom{4}{1} & \binom{4}{2} & \binom{4}{3} & \binom{4}{4}
 \end{array}$$

La Figura 2.4 muestra las primeras 17 líneas del triángulo de Pascal. Usaremos esta figura para ilustrar las siguientes propiedades:

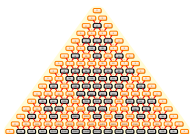
1. Para $1 \leq k \leq n$ se tiene:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

En la Figura 2.4, esta relación se indica con flechas, que marcan los valores que se suman para obtener cada entrada del triángulo.

Demostraremos esta propiedad más abajo, en el Teorema 2.4.7.

2. Si se suman los valores de cualquier fila $n \geq 0$, se obtiene $\binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{n} = 2^n$, como lo indica el Corolario 2.4.4.
3. La simetría del número combinatorio (Teorema 2.4.2) puede observarse en cada fila del triángulo: en la fila n , la entrada $\binom{n}{k}$ es igual a la entrada $\binom{n}{n-k}$, para $1 \leq k \leq n$.
4. El valor máximo de $\binom{n}{k}$ se da para $k = \lfloor n/2 \rfloor$. Si observa la Figura 2.4, notará cómo los valores de cada fila crecen hasta su punto medio, en que se alcanza el máximo, para luego descender de forma simétrica a la primera mitad de la fila —tal como lo indica el item anterior.
5. Si se usa un color diferente para los valores pares e impares del triángulo, se obtiene un Triángulo de Sierpiński. La Figura 2.4 también ilustra este hecho —véase también la miniatura al margen.



Demostramos a continuación el Teorema de Pascal, relacionado a la propiedad del item 1 en la lista anterior.

Teorema 2.4.7 (Teorema de Pascal) *Para $1 \leq k \leq n$, se cumple*

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

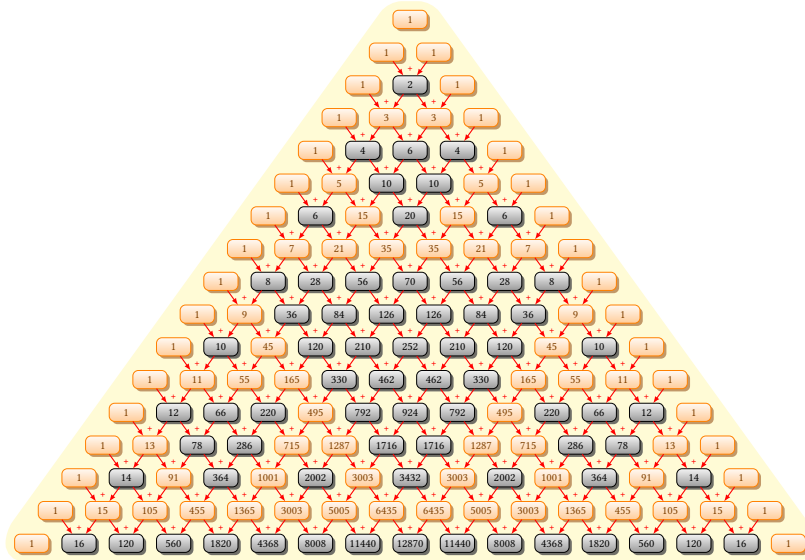


Figura 2.4: Las primeras 17 filas del triángulo de Pascal. Los valores pares e impares han sido coloreados de distinta manera para hacer notar el correspondiente Triángulo de Sierpiński.

Demostración. Sea $S = \{s_1, \dots, s_n\}$ un conjunto de n elementos. Sea además S' cualquier subconjunto de S con $|S'| = n-1$ elementos. Sea s_i el elemento que no está en S' , para $1 \leq i \leq n$. Cualquier subconjunto de S de tamaño k puede ser formado como a continuación:

- Como un subconjunto de tamaño k de S' , de los cuales hay $\binom{n-1}{k}$ distintos.
- Como un subconjunto de tamaño $k-1$ de S' , más el elemento s_i . Tenemos $\binom{n-1}{k-1}$ subconjuntos de esta forma.

Dado que estos dos casos son excluyentes, debemos sumar esas dos cantidades para obtener $\binom{n}{k}$, lo que demuestra el teorema. ■

2.5. Glosario de Conceptos Matemáticos Básicos

Repasamos a continuación algunos conceptos matemáticos preliminares, que necesitaremos más adelante.

Logaritmos y Potencias

En este trabajo haremos uso intensivo de logaritmos y potencias, por lo que es conveniente repasar sus propiedades más importantes.

Definición 2.5.1 La función logaritmo es definida como

$$\log_b a = c \Leftrightarrow a = b^c.$$

Las siguientes identidades son verdaderas para los logaritmos y las potencias:

1. $\log_b (ac) = \log_b a + \log_b c$.
2. $\log_b \frac{a}{c} = \log_b a - \log_b c$.
3. $\log_b (a^c) = c \log_b a$.

4. $\log_b a = \frac{\log_c a}{\log_c b}$.
5. $b^{\log_c a} = a^{\log_c b}$.
6. $(b^a)^c = b^{ac}$.
7. $b^a b^c = b^{a+c}$.
8. $\frac{b^a}{b^c} = b^{a-c}$.

Siguiendo la notación estándar usada en la literatura, en adelante denotaremos $\lg n = \log_2 n$, y $\ln n = \log_e n$.

Funciones Piso y Techo

Para todo número real x , las funciones *piso* y *techo* se definen como a continuación:

1. $\lfloor x \rfloor$ = el mayor entero que es menor o igual a x .
2. $\lceil x \rceil$ = el menor entero que es mayor o igual a x .

Series

Repasamos a continuación algunas propiedades y los tipos de series (o sumas) más comunes y que necesitaremos más adelante.

Factorización de Series. Asumiendo que a no depende de i , podemos factorizar una serie de la siguiente manera:

$$\sum_{i=1}^n af(i) = a \sum_{i=1}^n f(i).$$

Revirtiendo el Orden. Usando la propiedad conmutativa de la suma, podemos revertir el orden en que se hacen éstas como a continuación:

$$\sum_{i=1}^n \sum_{j=1}^m f(i, j) = \sum_{j=1}^m \sum_{i=1}^n f(i, j).$$

Acotando Series con Integrales. Si f es una función no decreciente, se puede acotar una suma con una integral, como a continuación:

$$\int_{a-1}^b f(x) dx \leq \sum_{i=a}^b f(i) \leq \int_a^{b+1} f(x) dx.$$

A menudo, es útil particionar una suma como sigue para poder acotarla de forma más simple:

$$\sum_{i=1}^n f(i) = \sum_{i=1}^j f(i) + \sum_{i=j+1}^n f(i).$$

Serie Telescópica. Para toda función f , una serie telescópica tiene la forma $\sum_{i=1}^n (f(i) - f(i-1))$. Estas series tienen la particularidad de que cada término cancela elementos del término anterior, excepto el primer término (en el que $-f(0)$ no se cancela) y el último término (en el que $f(n)$ no se cancela). Por lo tanto, se tiene:

$$\sum_{i=1}^n (f(i) - f(i-1)) = f(n) - f(0).$$

Serie Geométrica. Para una constante real $0 < a \neq 1$, una serie geométrica tiene la forma:

$$\sum_{i=0}^n a^i = \frac{a^{n+1} - 1}{a - 1}.$$

En el caso en que a sea una constante real tal que $0 < a < 1$, tenemos:

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1 - a}.$$

Serie Lineal Exponencial. La siguiente combinación de dos sumas conocidas se llama suma lineal exponencial, y tiene la siguiente expansión, para una constante real $0 < a \neq 1$ y $n \geq 2$:

$$\sum_{i=1}^n i a^i = \frac{a - (n+1)a^{n+1} + na^{n+2}}{(1-a)^2}.$$

Números Armónicos. El n -ésimo número Armónico \mathcal{H}_n es definido como:

$$\mathcal{H}_n = \sum_{i=1}^n \frac{1}{i}.$$

Las propiedades de estos números han sido estudiadas por diversos matemáticos desde varios siglos atrás, siendo Leonhard Euler uno de los primeros de los que se tenga registro.

Una propiedad importante de los números Armónicos que nos será de utilidad es la siguiente:

$$\ln(n+1) \leq \mathcal{H}_n \leq 1 + \ln(n). \quad (2.17)$$

La demostración queda como ejercicio para el lector.

De hecho, la similitud entre \mathcal{H}_n y $\ln n$ es tal que para $n \rightarrow \infty$, es posible acotar su diferencia con una constante:

$$\lim_{n \rightarrow \infty} (\mathcal{H}_n - \ln n) = \gamma,$$

siendo $\gamma = 0,577215664901532 \dots$ la constante de Euler-Mascheroni. Además, \mathcal{H}_n puede aproximarse con la siguiente expansión:

$$\mathcal{H}_n = \gamma + \ln n + \frac{1}{2}n^{-1} - \frac{1}{12}n^{-2} + \frac{1}{120}n^{-4} - \frac{1}{252}n^{-6} + \dots$$

Regla de L'Hôpital

La regla de L'Hôpital será de utilidad para comparar la velocidad de crecimiento de dos funciones. Si se tiene

$$\lim_{n \rightarrow \infty} f(n) = +\infty,$$

y

$$\lim_{n \rightarrow \infty} g(n) = +\infty,$$

entonces

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{f'(n)}{g'(n)},$$

donde $f'(n)$ y $g'(n)$ denotan las derivadas de f y g , respectivamente.

En este capítulo revisaremos los conceptos algorítmicos básicos necesarios para el correcto diseño y análisis de algoritmos en capítulos posteriores.

3.1. Modelos de Computación

Antes de diseñar y analizar algoritmos, es importante definir el *modelo de computación* subyacente. Esto significa definir el tipo de computador hipotético sobre el cual se ejecutarán nuestros algoritmos. Cada modelo de computación especifica las operaciones elementales que el computador hipotético puede ejecutar. Esto afecta la manera en que uno diseña y analiza algoritmos sobre esos modelos, y de ahí la importancia de aclararlo desde un principio. Dos modelos de computación típicos son las Máquinas de Turing (MT) y las Máquinas de Acceso Aleatorio (RAM, por sus siglas en inglés). Por ejemplo, para sumar dos números enteros positivos n y m en una MT, es necesario realizar una cantidad de operaciones proporcional a $n + m$ ¹. En una RAM, por otro lado, la suma de dos números enteros puede computarse mucho más eficientemente con una única operación primitiva de suma.

1: Los detalles del algoritmo quedan como ejercicio para el lector.

El Modelo de Computación Word RAM

Debido a su similitud con los computadores actuales, el modelo de computación word RAM es uno de los más usados en la literatura, y será el más usado a lo largo de este trabajo. En ciertos casos, sin embargo, necesitaremos usar otros modelos de computación, los que serán debidamente definidos. Una word RAM (o simplemente RAM) es un computador hipotético que consiste de una Unidad Central de Procesamiento (o CPU, en inglés) y una memoria principal M , tal como describimos a continuación.

CPU. La CPU de una RAM consiste principalmente de:

1. **Registros:** Un conjunto de k registros del procesador, que denotaremos R_1, \dots, R_k . Cada registro permite almacenar una secuencia de w bits, siendo éste el parámetro más importante del modelo. Típicamente, cada registro puede almacenar un número entero o real de w bits —usando una representación adecuada de punto fijo o flotante en el caso de los números reales. Usaremos el término *palabra de máquina* para referirnos a una secuencia de w bits que puede ser almacenada en uno de los registros del procesador y, por lo tanto, w es el tamaño de la palabra de máquina².
2. **Instrucciones:** Un conjunto de instrucciones —u *operaciones elementales*— que la CPU es capaz de ejecutar. Las instrucciones más típicamente soportadas son: asignaciones (\leftarrow), operaciones

2: Los valores más típicos en la práctica son $w = 32$ o $w = 64$.

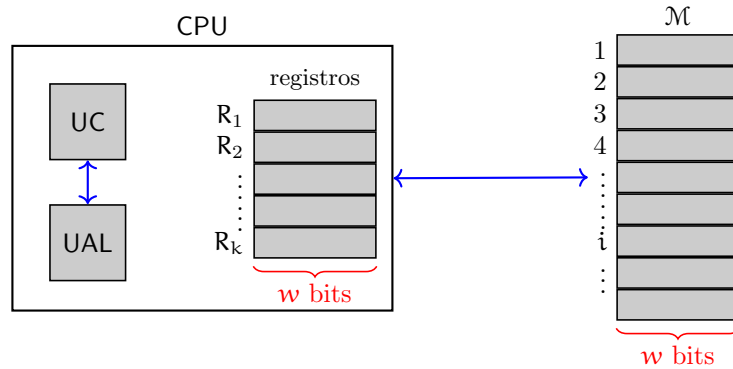


Figura 3.1: Un esquema del modelo de computación RAM.

aritméticas y lógicas (+, ×, −, /, **and**, **or**, **not**), operaciones relacionales (=, ≤, ≥, <, >, ≠), saltos implícitos (**if**, **while**) o explícitos (**return**, **break**, etc.), operaciones a nivel de bits al estilo del lenguaje C/C++, como \ll y \gg (shifts a izquierda y derecha), **&** (**and** a nivel de bits), **|** (**or** a nivel de bits), \oplus (**xor** a nivel de bits), \sim (negación a nivel de bits), entre otras operaciones. Las operaciones aritméticas y lógicas usualmente están implementadas en lo que se conoce como la *Unidad Aritmética y Lógica* (UAL) del procesador.

3. **Unidad de Control (UC):** Es la encargada de la correcta ejecución de cada una de las instrucciones que conforman un programa. En una RAM, la ejecución de un programa procede por *pasos discretos* o *ciclos del procesador*. Asumiremos que la CPU es capaz de ejecutar una única instrucción por ciclo del procesador ³.

3: En el caso de las operaciones a nivel de bits, asumiremos que operan sobre los w bits de una palabra de máquina por ciclo del procesador. Ésta es una característica distintiva del modelo word RAM, en comparación con el modelo RAM original.

4: Eso significa en el orden que un algoritmo necesite hacerlo.

5: La suposición de memoria infinita permite evitar detalles como la disponibilidad de memoria para el correcto funcionamiento de un algoritmo.

Memoria. Por su parte, la memoria \mathcal{M} de una RAM está formada por una cantidad infinita de celdas de w bits cada una, $\mathcal{M}[1], \mathcal{M}[2], \dots$. Las celdas pueden ser accedidas de manera aleatoria ⁴, y cada acceso toma un único ciclo del procesador. Es esta característica la que da el nombre a este modelo —*random access*. Asumiremos que el acceso a una celda de memoria se hace para lectura o escritura de la misma. La memoria del computador almacena los datos que serán procesados por los algoritmos. Aún cuando asumimos que la cantidad de memoria disponible es infinita, obviamente un algoritmo sólo puede usar una cantidad finita de la misma ⁵.

La Figura 3.1 ilustra una RAM. Note la diferencia de este modelo con, por ejemplo, una Máquina de Turing, que sólo permite el acceso secuencial a las celdas de la memoria: para acceder a la celda $\mathcal{M}[i]$ de la memoria (o cinta), debemos mover el cabezal secuencialmente por sobre las celdas que separan la celda actual de la celda de destino $\mathcal{M}[i]$.

Programando una RAM

En general, en este libro describiremos los algoritmos usando pseudocódigo similar a algún lenguaje de alto nivel ⁶. Asumiremos que existe un compilador que traduce las construcciones de alto nivel como if-then-else, while, for, loop-until, llamadas a funciones, operadores

6: Algunos autores, como Donald Knuth, prefieren definir un lenguaje tipo assembly para una máquina hipotética (la máquina MIX en el caso de Knuth), y de esa manera tener absoluto control sobre la cantidad de instrucciones ejecutadas por un algoritmo.

aritméticos y lógicos, y asignaciones, entre otras, a las correspondientes instrucciones a nivel de máquina como mover datos, operaciones aritméticas, saltos condicionales e incondicionales, y demás.

Bloques de Código. Para mejorar la legibilidad, el alcance de los bloques de código correspondientes a las construcciones if-then-else, while, for, y loop-until, entre otras, estarán delimitados por una línea vertical a la izquierda del bloque. Los bloques serán finalizados con la palabra clave **end**.

Comentarios en el Código. En general usaremos comentarios en el código para ayudar en la lectura de los algoritmos. Los mismos aparecerán a la derecha en la línea correspondiente, indicados por el símbolo `//`.

Variables, Arreglos, y Punteros. Una *variable* es simplemente una celda de memoria a la que se le ha dado un nombre representativo para identificarla. El valor almacenado en una celda de memoria puede variar a lo largo del tiempo —y de acuerdo a lo que necesite un algoritmo que la manipule. Siempre que se entienda por el contexto, para simplificar el código no haremos declaración de variables. Para cambiar el valor de una variable, usaremos la operación de asignación, denotada con \leftarrow . Así, por ejemplo, $A \leftarrow 1$ asigna el valor 1 a la celda de memoria correspondiente a la variable A . Asumiremos que las variables no tienen ningún valor inicial específico, por lo que deben ser adecuadamente inicializadas cuando sea necesario. Asumiremos también que la asignación de un valor almacenado en una celda de memoria de w bits a otra variable del mismo tipo toma un único ciclo del procesador. Si una variable ha sido almacenada en la celda $M[i]$ de la memoria, diremos que *su dirección es i* .

Un *arreglo* $A[1..n]$ ⁷, por otro lado, permite agrupar bajo un mismo nombre una colección de n valores del mismo tipo. Cada uno de los elementos del arreglo es identificado con un índice en el intervalo $[1..n]$. Usaremos la notación habitual de corchetes para acceder a cada uno de los elementos: el primer elemento del arreglo es $A[1]$, el segundo $A[2]$, y así siguiendo. Cuando sea necesario, usaremos $A[i..j]$, con $1 \leq i \leq j \leq n$, para denotar el subarreglo de los elementos con índices en $[i..j]$. En la memoria de una RAM, un arreglo se almacena en una zona contigua como lo ilustra la Figura 3.2, permitiendo el acceso eficiente a cada uno de los elementos que lo conforman. Llamaremos *dirección base* del arreglo A a la dirección de memoria que almacena el elemento $A[1]$.

Un *puntero*⁸ p es una variable que almacena la dirección de memoria de otra variable q . De esta manera, diremos que p apunta a q . Supongamos que la variable p ha sido almacenada en la celda $M[i]$ de la memoria, y q en la celda $M[j]$. Entonces, la celda $M[i]$ —correspondiente al puntero p — en realidad almacena el valor j —la dirección de q . Vea la Figura 3.3 para una ilustración. Si un puntero p apunta a una variable q , lo dibujaremos con una flecha desde p a q . Un puntero permite manipular a la variable apuntada de la forma en que sea necesaria: se puede acceder a su valor, y modificarlo cuando sea necesario.

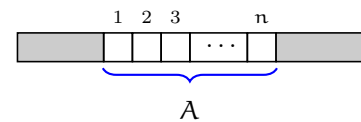


Figura 3.2: Disposición de un arreglo $A[1..n]$ en la memoria de una RAM.

7: Por *array* en inglés, alternativamente llamados *vectores*.

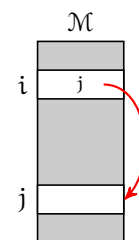


Figura 3.3: Un puntero almacenado en la celda $M[i]$, apuntando a la variable almacenada en la celda $M[j]$.

8: O *apuntador*.

9: Sería muy ineficiente tener que copiar un arreglo al pasarlo como parámetro.

Descripción de los Algoritmos. Para describir algoritmos usaremos una notación similar a las funciones de los lenguajes de alto nivel, indicando el nombre del algoritmo y sus parámetros formales. Siempre que corresponda, describiremos en el código la entrada y la salida del algoritmo. Asumiremos que los parámetros del algoritmo (i.e., los datos de entrada) se pasan tal como en el lenguaje C: por valor (o copia) en el caso de las variables simples, o como la dirección base en el caso de un arreglo ⁹. En este último caso, también indicaremos los índices que limitan el arreglo. Finalmente, usaremos la palabra clave **return** para finalizar la ejecución de un algoritmo y retornar el control a quien invocó al algoritmo. Además de retornar el control, se indicará el valor de retorno del algoritmo.

3.2. Problemas Abstractos

Un *problema abstracto* corresponde a una pregunta que se quiere responder o solucionar, y que es descrito mediante:

1. El conjunto de *parámetros* del problema, que corresponden a los datos sobre los que se plantea el problema, y
2. Las propiedades que deben cumplir las *soluciones* del problema.

Por ejemplo, el problema de determinar el máximo común divisor entre dos números enteros x e y . Aquí, los parámetros del problema son los enteros x e y , mientras que la solución es un entero z que es el máximo común divisor entre x e y . Otro ejemplo es el de determinar la pertenencia de un elemento x en un conjunto S , en donde tanto x como S son los parámetros del problema, mientras que la solución es SÍ o NO, dependiendo de si $x \in S$ o no.

En la descripción de un problema, los parámetros sólo son definidos formalmente, pero su valor no es especificado. Esto es análogo a cuando uno define una función —que implementa un algoritmo— usando un lenguaje de programación. Allí se deben definir los parámetros formales, pero su valor no se especifica sino hasta que la función es invocada.

Instancias y Soluciones de un Problema

10: O *caso particular*, aunque preferimos *instancia* por su correspondiente en inglés, *instance*.

Una *instancia* ¹⁰ de un problema abstracto \mathcal{P} corresponde a la asignación de valores particulares a los parámetros del problema. Para el ejemplo del máximo común divisor, una posible instancia es $x = 24$, $y = 44$. En general, un problema abstracto está asociado a un conjunto de instancias, las cuales corresponden a todos los posibles casos particulares del problema.

Definición 3.2.1 Para un problema abstracto \mathcal{P} , denotaremos $\mathcal{I}_{\mathcal{P}}$ al conjunto —potencialmente infinito— de sus instancias.

Por ejemplo, para el problema de determinar si x es divisible por y , con $x, y \in \mathbb{N}$ y $x \geq y$, las infinitas instancias del problema son todos los pares de números naturales (x, y) tal que $x \geq y$. Volviendo a la analogía con la definición de una función usando un lenguaje de programación,

una instancia corresponde a los parámetros actuales —o parámetros de entrada— en la invocación de una función.

A cada instancia de un problema le corresponde *una o más soluciones*, que son los resultados que deben obtenerse al resolver la instancia dada del problema. Definimos formalmente dicho conjunto de soluciones.

Definición 3.2.2 *Para un problema abstracto \mathcal{P} , denotaremos $\mathcal{S}_{\mathcal{P}}$ al conjunto —potencialmente infinito— de soluciones correspondientes a todas las instancias en $\mathcal{I}_{\mathcal{P}}$.*

Codificación de Instancias y Soluciones

Una instancia debe ser codificada adecuadamente para ser procesada por un algoritmo. Formalmente, una codificación es una biyección

$$e : \mathcal{I}_{\mathcal{P}} \xleftrightarrow{\sim} \{0, 1\}^*,$$

que mapea cada instancia del problema \mathcal{P} a su correspondiente string binario —o secuencia binaria— que la identifica ¹¹. Por ejemplo, en una RAM un entero se representa usualmente como un número binario de w bits. Un conjunto de n números enteros, por otro lado, podría representarse como una secuencia de n números binarios de w bits cada uno. En general, en adelante asumiremos que las instancias a resolver han sido codificadas adecuadamente, según corresponda. Esto significa que cuando digamos que un algoritmo trabaja sobre una instancia, en realidad significa que trabaja sobre una codificación de la instancia.

11: En términos formales, el alfabeto elegido para la codificación no necesariamente tiene que ser binario. Sin embargo, ésta es la alternativa más práctica en general.

Tamaño de una Instancia

Al analizar algoritmos es común modelar su eficiencia ¹² como una función del *tamaño de la instancia* que se está procesando. Es importante, entonces, comprender qué dicho concepto, que se define de manera diferente dependiendo del modelo de computación empleado. Para el modelo de Máquinas de Turing, por ejemplo, muchos autores coinciden en definir el tamaño de una instancia como la cantidad de bits usados en su codificación. Esto es principalmente porque una Máquina de Turing debe recorrer la codificación de la instancia de forma secuencial para procesarla. Así, una instancia más grande en general implica una codificación más grande, lo cual a su vez implica mayor tiempo de ejecución.

En el modelo RAM, por otro lado, se asume que cada celda de la memoria puede almacenar un valor entero (o un puntero), y que éstas pueden ser accedidas en un único ciclo del procesador. Así, definir el tamaño de la instancia como el tamaño de su codificación deja de tener sentido. En cambio, preferiremos usar los parámetros del problema —y no su codificación— como tamaño de una instancia. Esto es, llamaremos tamaño de una instancia $I \in \mathcal{I}_{\mathcal{P}}$ a uno —o más— de los parámetros del problema que mida el número de componentes de una instancia. Usaremos $|I|$ para denotar el tamaño de la instancia I .

Los siguientes ejemplos permiten ilustrar la idea:

12: Por ejemplo, su tiempo de ejecución o el espacio de memoria utilizado.

1. Para el problema de ordenar un conjunto $S \subseteq U$ sobre el que se ha definido una relación de orden total, el tamaño de una instancia corresponde a $|S|$, el tamaño del conjunto a ordenar.
2. Para el problema de determinar si un número natural n es primo o no, el tamaño de una instancia para este problema es el valor particular n .
3. Para el problema de multiplicar dos matrices de $n \times n$, podría usarse a n como tamaño de la instancia ya que es el parámetro que determina el tamaño de las matrices a multiplicar.
4. Para el problema de multiplicar dos números enteros de n dígitos, el tamaño de una instancia es n . Esto es porque el trabajo de multiplicación se hará sobre los dígitos de los números involucrados, por lo que es de esperar que el tiempo de ejecución de un algoritmo que resuelva este problema sea una función del número de dígitos.

En ciertos casos, el tamaño de una instancia puede estar dado por más de un parámetro. Por ejemplo, el último problema mencionado en el ítem 4 arriba podría ser reformulado para multiplicar dos números de n y m dígitos. En ese caso, una instancia será de tamaño (n, m) . En general, si el tamaño de una instancia depende de $k > 1$ parámetros del problema, lo representaremos usando una k -tupla ordenada.

Definición 3.2.3 Para un problema abstracto \mathcal{P} , sea $\mathcal{I}_{\mathcal{P},n} \subseteq \mathcal{I}_{\mathcal{P}}$ el conjunto de todas las instancias de \mathcal{P} que tienen tamaño n , y $\mathcal{S}_{\mathcal{P},n} \subseteq \mathcal{S}_{\mathcal{P}}$ el conjunto de soluciones correspondientes a las instancias de tamaño n .

Definición Formal

Al definir un problema abstracto \mathcal{P} , con sus parámetros y las propiedades que debe cumplir la solución al problema, implícitamente estamos definiendo una relación binaria total $R_{\mathcal{P}} \subseteq \mathcal{I}_{\mathcal{P}} \times \mathcal{S}_{\mathcal{P}}$, tal que

$$R_{\mathcal{P}} = \{(I, S) \mid S \text{ es solución de } I\}, \quad (3.1)$$

que relaciona a cada instancia de \mathcal{P} con sus correspondientes soluciones¹³. Obviamente, esta relación es puramente conceptual, con el fin de formalizar la definición de un problema abstracto.

13: Para ciertos problemas, puede existir más de una solución posible para una instancia.

3.3. Tipos de Problemas Abstractos

En general, los problemas abstractos se pueden dividir en cuatro tipos principales que estudiamos a continuación.

Problemas de Decisión

Un *problema de decisión* es uno al que a cada instancia le corresponde una solución del tipo “SÍ” o “NO”. La Figura 3.4 muestra conceptualmente una relación $R_{\mathcal{P}}$ correspondiente a un problema de decisión.

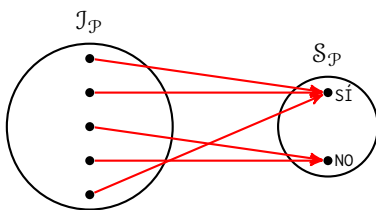


Figura 3.4: Esquema conceptual de una relación $R_{\mathcal{P}}$ para un problema de decisión.

Algunos ejemplos son los siguientes: (i) Dado un conjunto S con universo U y un elemento $x \in U$, ¿ $x \in S$?; (ii) Dados dos números enteros x e y , tal que $x \geq y$, ¿Es x divisible por y ?; (iii) Dado un número natural n , ¿Es primo?; (iv) Dado un tablero de ajedrez de $n \times n$ casilleros ($n \geq 1$) y n reinas, ¿Es posible colocar las reinas en el tablero tal que no puedan atacarse entre ellas?; (v) Dadas tres matrices A , B , y C de $n \times n$, ¿Se cumple que $A \times B = C$?; (vi) Dado un mapa con ciudades y carreteras que las unen, y dadas dos ciudades c_o y c_d en el mapa, ¿Existe una manera de ir desde c_o a c_d pasando por menos de c ciudades intermedias?; (vii) Dado un conjunto P de puntos en el plano, tal que cada punto p_i tiene un peso $w(p_i)$, y un rectángulo r con lados paralelos a los ejes de coordenadas, ¿Hay algún punto de P dentro de r cuyo peso sea mayor a un valor W ?

Para un problema de decisión \mathcal{P} , sea $\mathcal{J}'_{\mathcal{P}} \subseteq \mathcal{J}_{\mathcal{P}}$ el subconjunto de sus instancias que tienen respuesta “Sí”. Por ejemplo, para el problema de determinar si un entero x es divisible por y , tenemos $\mathcal{J}'_{\mathcal{P}} = \{(x, y) \in \mathbb{Z} \times \mathbb{Z} \mid x \text{ es divisible por } y\}$. Así, el problema de decisión original se transforma en una instancia del problema de pertenencia en el conjunto $\mathcal{J}'_{\mathcal{P}}$: para resolver una instancia I de \mathcal{P} hay que determinar si se cumple $I \in \mathcal{J}'_{\mathcal{P}}$ o no.

Problemas de Búsqueda

Un *problema de búsqueda* es uno en el que es suficiente encontrar una solución —cualquiera de las posibles— a una instancia del problema. Sea $R_{\mathcal{P}}(I) = \{s \in \mathcal{S}_{\mathcal{P}} \mid (I, s) \in R_{\mathcal{P}}\}$ el conjunto de soluciones de una instancia $I \in \mathcal{J}_{\mathcal{P}}$. Conceptualmente, un problema de búsqueda se puede resolver recorriendo $\mathcal{S}_{\mathcal{P}}$ exhaustivamente, chequeando cada $s \in \mathcal{S}_{\mathcal{P}}$ hasta encontrar la primera para la que se cumple $s \in R_{\mathcal{P}}(I)$. Esto implica resolver el problema de pertenencia de una solución s en el conjunto $R_{\mathcal{P}}(I)$. Vea la Figura 3.5 para una ilustración. De hecho, note que estrictamente hablando sólo es necesario recorrer el conjunto $\mathcal{S}_{\mathcal{P},|I|}$ para buscar la solución, y no todo $\mathcal{S}_{\mathcal{P}}$. Algunos ejemplos son los siguientes: (i) Dados $x, y \in \mathbb{Z}$, calcular el máximo común divisor; (ii) Dado un tablero de ajedrez de $n \times n$ casilleros ($n \geq 1$) y n reinas, encontrar (de ser posible) una forma de colocar las reinas en el tablero tal que no puedan atacarse entre ellas; (iii) Dadas dos matrices A y B de $n \times n$, encontrar una matriz C de $n \times n$ tal que $A \times B = C$; (iv) Dado un mapa con ciudades y carreteras que las unen, y dadas dos ciudades c_o y c_d en el mapa, encontrar (de ser posible) un camino que permita viajar desde c_o a c_d pasando por menos de c ciudades intermedias; (v) Dado un conjunto P de puntos en el plano, tal que cada punto p_i tiene un peso $w(p_i)$, y un rectángulo r con lados paralelos a los ejes de coordenadas, encontrar (de ser posible) un punto de P dentro de r cuyo peso sea mayor a un valor W .

Si para cada instancia de un problema existe una única solución ¹⁴, entonces el problema se conoce como *problema función*. De los ejemplos anteriores, sólo dos cumplen con esta propiedad: el problema del máximo común divisor y el de multiplicación de matrices.

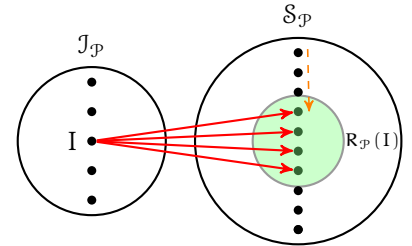


Figura 3.5: Esquema conceptual de la relación $R_{\mathcal{P}}$ para un problema de búsqueda.

14: Esto es, $\forall I \in \mathcal{J}_{\mathcal{P}}, |R_{\mathcal{P}}(I)| = 1$.

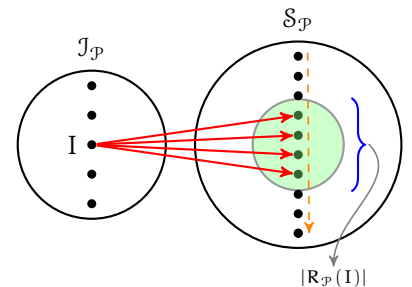


Figura 3.6: Esquema conceptual de la relación $R_{\mathcal{P}}$ para un problema de conteo.

Problemas de Reporte y Conteo

Un *problema de reporte* busca obtener $R_{\mathcal{P}}(I)$ —es decir, computar el conjunto de todas las soluciones de la instancia I —, mientras que un *problema de conteo* busca calcular $|R_{\mathcal{P}}(I)|$ —esto es, la cantidad de soluciones de una instancia dada. Conceptualmente, estos tipos de problemas se pueden resolver “recorriendo” $S_{\mathcal{P}}$ exhaustivamente, y para cada $s \in S_{\mathcal{P}}$, si $s \in R_{\mathcal{P}}(I)$ se lo reporta o se lo cuenta —dependiendo si es un problema de búsqueda o conteo, respectivamente. Algunos ejemplos son los siguientes: (i) Dado un tablero de ajedrez de $n \times n$ casilleros ($n \geq 1$), y n reinas, ¿De cuántas maneras se pueden colocar las reinas en el tablero tal que no puedan atacarse entre ellas? (ii) Dado un mapa con ciudades y carreteras que las unen, y dadas dos ciudades c_o y c_d en el mapa, ¿Cuántos caminos permiten viajar desde c_o a c_d pasando por menos de c ciudades intermedias? (iii) Dado un conjunto P de puntos en el plano, tal que cada punto p_i tiene un peso $w(p_i)$, y un rectángulo r con lados paralelos a los ejes de coordenadas, contar cuántos puntos de P hay dentro de r , tal que su peso sea mayor a un valor W .

Problemas de Optimización

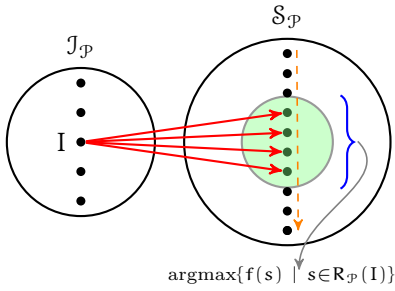


Figura 3.7: Esquema conceptual de la relación $R_{\mathcal{P}}$ para un problema de optimización.

A diferencia de un problema de búsqueda, reporte, o conteo, un *problema de optimización* busca la mejor solución para $I \in \mathcal{I}_{\mathcal{P}}$. Sea $f : S_{\mathcal{P}} \mapsto \mathbb{R}$ una *función objetivo* que mide la calidad de una solución. Entonces, el problema consiste en encontrar $\operatorname{argmax}\{f(s) \mid s \in R_{\mathcal{P}}(I)\}$, i.e., la solución s de I que maximiza $f(s)$. Para eso, conceptualmente, también se debe recorrer $S_{\mathcal{P}}$, chequeando el valor de f para cada $s \in R_{\mathcal{P}}(I)$, para determinar aquella que maximice su valor. Algunos ejemplos son los siguientes: (i) Encontrar el polígono convexo de área mínima que contiene a un conjunto de puntos en el plano. (ii) Dados dos strings s y t , ¿Cuál es la mínima cantidad de inserciones, borrados, y sustituciones de símbolos necesarios para transformar un string en otro? (iii) Dado un mapa con ciudades y carreteras que las unen, y dadas dos ciudades c_o y c_d en el mapa, encontrar el camino que permita viajar desde c_o a c_d pasando por la menor cantidad posible de ciudades intermedias. (iv) Dado un conjunto P de puntos en el plano, tal que cada punto p_i tiene un peso $w(p_i)$, y un rectángulo r con lados paralelos a los ejes de coordenadas, encontrar el punto de P que está dentro de r , tal que su peso sea el mayor de todos.

El Chequeo de Pertenencia y la Resolución de Problemas

Remarcar que todos los tipos de problemas necesitan, al menos conceptualmente, hacer chequeos de pertenencia en conjuntos.

3.4. Algoritmos: Definición Formal

Luego de haber intentado definir informalmente lo que es un algoritmo en la Sección 1.2 (en la página 2), estamos ahora en condiciones de dar una definición formal.

Definición 3.4.1 Para un problema abstracto \mathcal{P} , sea:

- $\mathcal{I}_{\mathcal{P}}$ el conjunto, posiblemente infinito, de todas las instancias de \mathcal{P} .
- $\mathcal{S}_{\mathcal{P}}$ el conjunto de soluciones a todas las instancias de \mathcal{P} .
- $\mathcal{R}_{\mathcal{P}} \subseteq \mathcal{I}_{\mathcal{P}} \times \mathcal{S}_{\mathcal{P}}$ la relación binaria entre instancias de \mathcal{P} y sus soluciones.
- $\mathcal{R}_{\mathcal{P}}(I) = \{s \mid (I, s) \in \mathcal{R}_{\mathcal{P}}\}$ es el conjunto de soluciones de una instancia $I \in \mathcal{I}_{\mathcal{P}}$.

Formalmente, todo algoritmo $\mathcal{A}_{\mathcal{P}}$ que resuelve el problema \mathcal{P} es una representación por comprensión del conjunto $\mathcal{R}_{\mathcal{P}}(I)$, para toda $I \in \mathcal{I}_{\mathcal{P}}$.

Esta definición tiene varias consecuencias, que discutimos a continuación: (1) El hecho de que $\mathcal{A}_{\mathcal{P}}$ represente a $\mathcal{R}_{\mathcal{P}}(I)$ por comprensión evita tener que representarlo por extensión, lo cual no sería práctico para la mayoría de los problemas de interés. (2) Para computar $\mathcal{R}_{\mathcal{P}}(I)$, el algoritmo $\mathcal{A}_{\mathcal{P}}$ debe recibir la instancia I como *entrada*. Así, un algoritmo recibe una instancia del problema como entrada y produce un solución —o conjunto de soluciones— como *salida*. (3) Exige que un algoritmo resuelva **toda** instancia I del problema. (4) Exige que un algoritmo produzca exactamente la solución —o soluciones— correspondiente a la instancia. Un algoritmo es *correcto* cuando todas las instancias son resueltas adecuadamente. La demostración de corrección de un algoritmo es un paso importante en el diseño del mismo.

3.5. El Rol de la Algoritmia en la Resolución de Problemas

La *algoritmia* es un área de ciencias de la computación que se enfoca en el estudio del diseño y análisis de algoritmos, y los aspectos relacionados a esto. Su principal objetivo es, por lo tanto, la resolución de problemas de forma eficiente en términos de los recursos computacionales utilizados. En la Sección 3.2, cuando definimos los distintos tipos de problemas, mostramos cómo se podrían resolver conceptualmente. Por ejemplo, los problemas de decisión podrían resolverse mediante el problema de pertenencia de una instancia en el conjunto de instancias “SÍ”. Por otro lado, para los problemas de búsqueda, conteo, y optimización, la idea principal era recorrer el conjunto de soluciones del problema, $\mathcal{S}_{\mathcal{P}}$, buscando aquellas que correspondan a la instancia que se quiere resolver. Sin embargo, aunque son conceptualmente correctos, el lector comprenderá que en general esos enfoques no son eficientes: el conjunto de soluciones de un problema puede ser muy grande, e.g., de tamaño exponencial o factorial respecto al tamaño de la instancia. Un recorrido exhaustivo sobre un conjunto de ese tamaño es, obviamente, no práctico.

Como resultado, podemos decir que la algoritmia se encarga de estudiar maneras de evitar las búsquedas exhaustivas en el espacio de soluciones, proponiendo técnicas para desarrollar algoritmos más eficientes, y herramientas de análisis para mostrar qué tan eficientes son esos algoritmos. Sin embargo, para aquellos problemas en que no se nos ocurra

una manera más eficiente de resolverlo que revisando exhaustivamente el conjunto de soluciones, la algoritmia también provee herramientas para mostrar que ciertos problemas no pueden —probablemente— ser resueltos más eficientemente que de esa manera.

3.6. Tiempo de Ejecución de un Algoritmo

15: Definir el modelo de computación subyacente es importante para poder definir lo que significa una operación elemental.

El *tiempo de ejecución* de un algoritmo es la cantidad de operaciones elementales¹⁵ que el algoritmo ejecuta para resolver una instancia dada del problema. Definimos este concepto formalmente como a continuación.

Definición 3.6.1 (Tiempo de Ejecución) *Sea A un algoritmo que resuelve un problema abstracto \mathcal{P} , y sea $I \in \mathcal{I}_{\mathcal{P}}$ una instancia particular de \mathcal{P} . El tiempo de ejecución del algoritmo A para la instancia I , denotado $T_A(I)$, es la cantidad de operaciones elementales que ejecuta A para resolver I .*

El tiempo de ejecución es un factor clave para comparar algoritmos que resuelven un mismo problema: un menor tiempo de ejecución significa un algoritmo más eficiente. El hecho de medir el tiempo de ejecución en base a la cantidad de operaciones elementales ejecutadas para procesar una instancia, permite independizarnos de las implementaciones particulares que pueda tener un algoritmo. Recuerde que asumimos que cada operación elemental se ejecuta en un ciclo del procesador.

Ejemplo 3.6.1 Consideremos un conjunto S de n elementos sobre el que se ha definido una relación de orden total \preceq . Asumiendo que el conjunto ha sido codificado —o almacenado— en un arreglo $S[1..n]$, el Algoritmo 1 permite encontrar el mayor elemento en el conjunto. Recuerde que al ser \preceq un orden total, cada uno de los elementos del conjunto tiene un rango. En este caso, estamos buscando el elemento con rango n . Al inspeccionar el algoritmo, podemos ver que ejecuta $n - 1$ ciclos **for**. Por cada uno de esos ciclos, se comparan dos posiciones del arreglo en el **if** de la línea 3: la que contiene el máximo visto hasta el momento, y la que contiene el i -ésimo elemento del arreglo, siendo i el índice del ciclo. Si denotamos con c la cantidad de operaciones ejecutadas en cada ciclo, y teniendo en cuenta que además el algoritmo ejecuta dos asignaciones antes del ciclo **for**, y una operación **return** al finalizar el ciclo, podemos decir que el tiempo de ejecución del algoritmo es $n - 1$ comparaciones usando el orden total \preceq , y $c \cdot (n - 1) + 3$ operaciones. Esto significa que su tiempo de ejecución crece linealmente con respecto al tamaño del arreglo de entrada.

A menudo, en lugar de contar todas las operaciones que ejecuta un algoritmo —como hicimos en el Ejemplo 3.6.1— contaremos sólo cierto tipo de operaciones que, por alguna razón, sean más importantes que otras. La importancia de una operación está dada por el costo computacional de ejecutarla. Al medir el tiempo de ejecución en términos de

Algoritmo 1: MÁXIMO($S[1..n]$)

Entrada: Un conjunto no necesariamente ordenado representado por un arreglo $S[1..n]$ sobre el que se ha definido una relación de orden total \preceq .

Salida: La posición m tal que $S[m]$ es el máximo elemento de S respecto a \preceq .

```

1  $m \leftarrow 1$ 
2 for  $i \leftarrow 2$  to  $n$  do
3   if  $S[m] \preceq S[i]$  then
4      $m \leftarrow i$ 
5   end
6 end
7 return  $m$ 

```

esas operaciones costosas, estaremos penalizando más a los algoritmos que ejecuten una mayor cantidad de esas operaciones. Algunas de las operaciones típicas suelen ser las comparaciones \preceq entre elementos de un conjunto, las multiplicaciones y divisiones, y los accesos a celdas de memoria, entre otras.

Para el caso particular del Algoritmo 1, es común medir sólo la cantidad de comparaciones \preceq ejecutadas. Esto es porque, dependiendo de la relación de orden \preceq y el tipo de los elementos del conjunto, cada comparación de elementos puede ser costosa de calcular. Algunos ejemplos son los siguientes:

- \preceq podría ser una relación de orden definida por extensión, como el ejemplo de la relación \preceq del Ejemplo 2.2.4 (página 21). En estos casos, y como ya hemos dicho, la relación debe ser almacenada explícitamente en la memoria del computador RAM. Luego, para determinar si $S[m] \preceq S[i]$ se cumple o no, debemos determinar si $(S[m], S[i]) \in \preceq$, lo cual puede ser costoso. Como veremos más adelante, para determinar la pertenencia de un elemento a un conjunto es necesario buscar el elemento dentro de la representación del conjunto. Esta operación puede tomar varios ciclos del procesador cada vez que es ejecutada, y por lo tanto dominar el tiempo de ejecución del algoritmo.
- \preceq podría estar definida por comprensión, y por lo tanto existir un algoritmo que permita determinar la pertenencia $(S[m], S[i]) \in \preceq$ sin necesidad de almacenar la relación explícitamente, ni tener que buscar el par en la relación —como en el caso anterior. Sin embargo, el algoritmo que permita decidir la pertenencia podría ser costoso, tal como es el caso de la relación \sqsubseteq definida en las Ecuaciones (2.3) y (2.2) —en la página 20.

En adelante, cuando estudiemos algoritmos que funcionan en base a relaciones de orden total, mediremos su tiempo de ejecución en cantidad de comparaciones, \preceq . Los ejemplos más típicos de esta clase de algoritmos corresponden a los problemas de ordenar un conjunto, determinar la pertenencia de un elemento en un conjunto ordenado, y encontrar el elemento con rango k en un conjunto (para $1 \leq k \leq n$), entre otros.

3.7. Análisis de Mejor y Peor Caso

16: El problema de determinar la pertenencia de un elemento en un conjunto es fundamental en computación, con aplicaciones de todo tipo. De hecho, al finalizar la sección anterior relacionamos el problema de implementar un orden total con el problema de determinar la pertenencia de un elemento a un conjunto.

Es común que un algoritmo no ejecute la misma cantidad de operaciones para resolver cada una de las instancias del problema, sino que su tiempo de ejecución dependa de cuál caso se esté procesando. Por ejemplo, consideremos el Algoritmo 2, que recibe un elemento $x \in \mathcal{U}$ y un conjunto $S \subseteq \mathcal{U}$, y determina si $x \in S$ o no ¹⁶. El algoritmo asume que el conjunto ha sido representado usando un arreglo no necesariamente ordenado de n elementos. El algoritmo retorna la posición i tal que

Algoritmo 2: PERTENENCIA(x , $S[1..n]$)

Entrada: Un conjunto no necesariamente ordenado representado por un arreglo $S[1..n] \subseteq \mathcal{U}$, y un elemento $x \in \mathcal{U}$.

Salida: la posición i tal que $S[i] = x$, o $n + 1$ si $x \notin S$.

```

1 for i ← 1 to n do
2   if x = S[i] then
3     return i
4   end
5 end
6 return n + 1                                     // Indica que x ∉ S

```

$S[i] = x$. Si $x \notin S$, devuelve el valor $n + 1$. Como puede verse, el algoritmo determina la pertenencia de x buscándolo secuencialmente en el arreglo. Generalmente, resolveremos la pertenencia mediante una búsqueda del elemento x en la codificación del conjunto S . La *búsqueda secuencial* —como suele conocerse a este algoritmo— es uno de los métodos más primitivos para resolver este problema.

Este algoritmo ilustra lo que hemos mencionado: su tiempo de ejecución depende del elemento a buscar, ya que la posición en que se ubica el elemento buscado determina la cantidad de ciclos que ejecuta. A continuación, formalizamos el concepto de comportamiento de mejor y peor caso de un algoritmo.

El *análisis de mejor caso* de un algoritmo consiste en calcular su tiempo de ejecución considerando la(s) instancia(s) que produce(n) la ejecución de la menor cantidad posible de operaciones. Formalmente:

Definición 3.7.1 (Tiempo de Ejecución de Mejor Caso) *Sea \mathcal{A} un algoritmo que resuelve un problema abstracto \mathcal{P} . El tiempo de ejecución de mejor caso para \mathcal{A} sobre todas las instancias de tamaño n se define como:*

$$t_{\mathcal{A}}(n) \equiv \min\{T_{\mathcal{A}}(I) \mid I \in \mathcal{I}_{\mathcal{P},n}\}.$$

Por ejemplo, para el Algoritmo 2, el mejor caso consiste en buscar el elemento $x = S[1]$. Esto produce la ejecución de un único ciclo. Sin embargo, este tipo de análisis es optimista y no aporta demasiada información respecto al tiempo de ejecución de una instancia menos optimista. Al analizar algoritmos, siempre es conveniente ser relativamente pesimista.

El *análisis de peor caso* de un algoritmo consiste en calcular su tiempo de ejecución considerando la(s) instancia(s) que produce(n) la ejecución

de la mayor cantidad posible de operaciones elementales. Formalmente:

Definición 3.7.2 (Tiempo de Ejecución de Peor Caso) *Sea \mathcal{P} un problema abstracto cuyo conjunto de instancias es $\mathcal{I}_{\mathcal{P}}$. Sea A un algoritmo que resuelve el problema \mathcal{P} . El tiempo de ejecución de peor caso para A sobre todas las instancias de tamaño n se define como:*

$$T_A(n) \equiv \max\{T_A(I) \mid I \in \mathcal{I}_{\mathcal{P},n}\}.$$

Para el ejemplo del Algoritmo 2, el peor caso consiste en buscar el elemento $x = S[n]$ —o, alternativamente, un elemento que no está en el conjunto. Eso produce la ejecución de n ciclos, que es el máximo en este caso. Al ser pesimista, este tipo de análisis previene al usuario del algoritmo sobre el peor escenario en términos de tiempo de ejecución y le permitiría tomar recaudos. Sin embargo, hay que ser cuidadosos: un análisis de peor caso puede ser demasiado pesimista, y penalizar demasiado a algoritmos que tienen malos peores casos cuya probabilidad de ocurrir es muy baja. Es probable que se descarten algoritmos por casos que raramente ocurrirán en la práctica. Para subsanar esto, estudiaremos otras alternativas de análisis más adelante.

3.8. Notaciones Asintóticas

Tal como vimos, el tiempo de ejecución del Algoritmo 2 depende de la instancia del problema que se resuelva. Esto es muy común al analizar algoritmos, y se debe distinguir el caso —e.g., mejor o peor caso— que se está analizando. Sin embargo, a veces es preferible —o necesario— denotar el tiempo de ejecución de un algoritmo de tal manera que se incluya a **todas** las instancias del problema. Las *notaciones asintóticas* permiten que en lugar de expresar el tiempo de ejecución de cada instancia de forma exacta, podamos acotar el tiempo de ejecución de todas las instancias. Estudiaremos a continuación las principales notaciones asintóticas usadas en la literatura.

Notación O. Para expresar formalmente que una función $f : \mathbb{N} \mapsto [0, +\infty)$ crece más lentamente o igual que otra función $g : \mathbb{N} \mapsto [0, +\infty)$ ¹⁷, usaremos la notación $f(n) \in O(g(n))$, que se lee “ $f(n)$ es O grande de $g(n)$ ”, en donde definimos:

$$\begin{aligned} O(g(n)) = \{f : \mathbb{N} \mapsto [0, +\infty) \mid \\ \exists c \in \mathbb{R}^+, \\ \exists n_0 \in \mathbb{N}, \\ f(n) \leq c \cdot g(n), \forall n \geq n_0\}. \end{aligned} \quad (3.2)$$

Esto es, $O(g(n))$ es el conjunto de todas las funciones f que, a partir de un valor n_0 , pueden ser acotadas superiormente por un múltiplo de la función g . Vea la Figura 3.8 para una ilustración. Note que la única manera de acotar superiormente a una función f de la manera indicada en la definición, es que f crezca más lentamente (o igual) que la función g . Por ejemplo, supongamos que $f(n) = n^2$ y $g(n) = n$. Notar que no

17: En otras palabras, g es una cota superior para f .

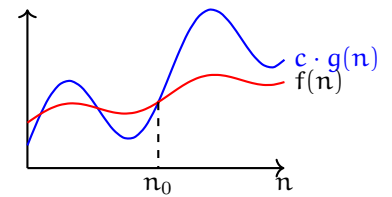


Figura 3.8: Ejemplo de una función $f(n) \in O(g(n))$, asumiendo que para todo $n \geq n_0$ se cumple que $f(n) \leq c \cdot g(n)$.

existen constantes c y n_0 tal que $n^2 \leq c \cdot n, \forall n \geq n_0$. Por lo tanto, $n^2 \notin O(n)$. Entonces, reformulando, $O(g(n))$ es el conjunto de todas las funciones f que crecen más lentamente (o igual) que g .

Ejemplo 3.8.1 Considere el conjunto

$$O(n) = \{1, 2, 3, \dots, \lg \lg n, \dots, \lg n, \dots, \sqrt{n}, \dots, n, 2n, 3n, n+1, \dots\},$$

en donde se han listado sólo las funciones más típicas. Note que el conjunto es infinito.

La notación $O(\cdot)$ permite acotar superiormente el tiempo de ejecución de un algoritmo, prescindiendo de los detalles de la función que modela el tiempo. Esto es, si un algoritmo tiene tiempo de ejecución de peor caso $T(n)$, podemos asegurar que el tiempo de ejecución del algoritmo es $O(T(n))$, sin importar la instancia que se esté resolviendo.

Note el rol de la constante c en la definición de $O(\cdot)$, permitiendo simplificar una función $T(n)$. Por ejemplo, si $T(n) = 3n^2 + 4n + 2$, note que $3n^2 + 4n + 2 \leq 9n^2, \forall n \geq 1$. Por lo tanto, $T(n) \in O(n^2)$, que es más conciso y elegante que su forma original, y aún mantiene suficiente información para conocer la velocidad de crecimiento de la función original. En general, vamos a simplificar lo más posible los tiempos de ejecución al usar notaciones asintóticas, eliminando constantes y términos de orden inferior ¹⁸.

Las clases de funciones más típicas que encontraremos al analizar algoritmos son las siguientes:

- $O(1)$: se usa para denotar a las funciones constantes, cualquiera sea la constante. Por ejemplo, $15 \in O(1)$, dado que $15 \leq 15 \cdot 1, \forall n \geq 1$. Por ende, se usa $O(1)$ en lugar de $O(15)$, al ser una alternativa más concisa y elegante.
- $O(\lg \lg n)$: las funciones log-logarítmicas, de crecimiento muy lento.
- $O(\lg n)$: se usa para denotar las funciones logarítmicas. En general, asumiremos que los logaritmos son en base 2. Sin embargo, se cumple que $\log_a(n) \in O(\log_b n)$, para cualquier par de bases a y b . Ver la sección de ejercicios al final de esta sección.
- $O(\sqrt{n})$: se usa para denotar las raíces cuadradas.
- $O(n)$: se usa para denotar las funciones lineales.
- $O(n \lg n)$: se usa para denotar las funciones superlineales del tipo $n \lg n$.
- $O(n^2)$: se usa para denotar los polinomios de grado 2. En general, $O(n^k)$ se usa para denotar los polinomios de grado $k \geq 1$. Se puede demostrar que todo polinomio de grado k es $O(n^k)$. Ver la sección de ejercicios al final de esta sección.
- $O(2^n)$: se usa para denotar las funciones exponenciales.
- $O(n!)$: se usa para denotar las funciones factoriales.

Existen, obviamente, más clases de funciones que las mencionadas —de hecho, hay una cantidad infinita de ellas. Sin embargo, esas son las más típicas.

18: En la práctica, sin embargo, los factores constantes son importantes. Suponga dos algoritmos A_1 y A_2 que resuelven el mismo problema con tiempos de ejecución de peor caso $T_1(n) = n$ y $T_2(n) = 2n$, respectivamente. Entonces, ante una misma entrada, A_1 ejecutará la mitad de las operaciones que ejecuta A_2 .

Cotas ajustadas...

Al usar la notación $O(\cdot)$ para acotar superiormente el tiempo de ejecución de un algoritmo, se debe emplear la cota superior más ajustada posible. Por ejemplo, el Algoritmo 1 tiene tiempo de ejecución $T(n) = c \cdot (n - 1) + 3$. Por lo tanto, podríamos decir que $T(n) \in O(n^2)$, lo cual es matemáticamente correcto. Sin embargo, $O(n^2)$ es una cota superior no ajustada para $c \cdot (n - 1) + 3$. Una cota de este tipo no es apropiada ya que subestima al algoritmo. De hecho, podríamos pensar en un caso extremo, y decir que casi cualquier algoritmo interesante en la práctica tiene tiempo de ejecución acotado —burdamente— por $O(n^n)$. Sin embargo, dicha cota —obviamente no ajustada— no entregará información útil respecto al tiempo de ejecución del algoritmo. Una cota ajustada en este caso sería $T(n) \in O(n)$.

Aunque la notación $O(\cdot)$ es la más conocida y enseñada en cursos básicos de estructuras de datos y algoritmos y, probablemente, la más comúnmente usada en la literatura académica y científica, no es la única notación asintótica existente, como veremos a continuación.

Notación Ω . Para expresar formalmente que una función $f : \mathbb{N} \mapsto [0, +\infty)$ crece más rápidamente o igual que otra función $g : \mathbb{N} \mapsto [0, +\infty)$, usaremos la notación $f(n) \in \Omega(g(n))$, en donde definimos:

$$\begin{aligned} \Omega(g(n)) = \{f : \mathbb{N} \mapsto [0, +\infty) \mid \\ \exists c \in \mathbb{R}^+, \\ \exists n_0 \in \mathbb{N}, \\ f(n) \geq c \cdot g(n), \forall n \geq n_0\}. \end{aligned} \quad (3.3)$$

De forma similar a la notación $O(\cdot)$, las únicas funciones $f(n)$ que cumplen la condición para pertenecer a $\Omega(g(n))$ son aquellas que crecen más rápidamente (o igual) que $g(n)$. Esto es, $\Omega(g(n))$ es el conjunto de todas las funciones que pueden ser acotadas inferiormente por un múltiplo de la función $g(n)$.

Ejemplo 3.8.2 Considere el conjunto

$$\Omega(n) = \{n, 2n, n+1, \dots, n \lg n, \dots, n^2, n^3, \dots, n^k, \dots, 2^n, \dots, n!, \dots\},$$

en donde se han listado sólo las funciones más típicas. Note que el conjunto es infinito.

La notación Ω , en general, no es de mucha utilidad para acotar inferiormente el tiempo de ejecución de un algoritmo. Esto es, si el tiempo de ejecución de mejor caso del algoritmo es $t(n)$, podemos decir que el tiempo de ejecución del algoritmo es $\Omega(t(n))$, sin importar la instancia que se esté resolviendo. Sin embargo, esto por sí sólo no entrega demasiada información respecto al comportamiento del algoritmo: en general, el mejor caso $t(n)$ es muy optimista y puede alejarse del peor caso.

A diferencia de esto último, algo que aporta más información es usar una *cota inferior del problema* para acotar inferiormente el tiempo de ejecución de un algoritmo. Estudiaremos el concepto de cotas inferiores y la complejidad de los problemas más adelante, en el Capítulo 7.

Notación Θ . Para expresar formalmente que una función $f : \mathbb{N} \mapsto [0, +\infty)$ crece asintóticamente a la misma velocidad que otra función $g : \mathbb{N} \mapsto [0, +\infty)$, usaremos la notación $f(n) \in \Theta(g(n))$, en donde definimos:

$$\begin{aligned} \Theta(g(n)) = \{f : \mathbb{N} \mapsto [0, +\infty) \mid \\ \exists c_1, c_2 \in \mathbb{R}^+, \\ \exists n_0 \in \mathbb{N}, \\ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0\}. \end{aligned} \quad (3.4)$$

Las únicas funciones $f(n)$ que pueden ser acotadas tanto inferior como superiormente por múltiplos de una misma función $g(n)$ son aquellas que tienen su misma velocidad de crecimiento.

Ejemplo 3.8.3 Considere el conjunto

$$\Theta(n) = \{n, 2n, 3n, n+1, \dots\},$$

en donde se han listado sólo las funciones más típicas. Note que el conjunto es infinito.

A partir de los Ejemplos 3.8.1, 3.8.2, y 3.8.3, podemos notar lo siguiente:

Observación 3.8.1 Para cualquier función $g : \mathbb{N} \mapsto [0, +\infty)$, se cumple que

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n)).$$

Al analizar algoritmos, usaremos la notación $\Theta(T(n))$ para indicar que el tiempo de ejecución de un algoritmo es asintóticamente igual a una función $T(n)$ para todas las instancias del problema. Esto quiere decir que cualquier ejecución del algoritmo toma tiempo proporcional a $T(n)$. En general, un algoritmo tiene tiempo de ejecución $\Theta(T(n))$ si y sólo si los tiempos de ejecución de mejor y peor caso son asintóticamente iguales, es decir, $t(n) \in \Theta(T(n))$.

¿Cuándo usar Θ y cuándo O ?

Diversos autores —quizás el caso más emblemático sea D. Knuth— advierten sobre el uso inadecuado de las notaciones asintóticas. Es importante usar la notación adecuada en cada caso, de manera que exprese exactamente lo que se quiere decir. Al acotar el tiempo de ejecución de un algoritmo, Θ es en general preferible por sobre O , ya que entrega más información. Sin embargo, no siempre es posible usar Θ .

Ilustramos los anterior con el siguiente ejemplo.

Ejemplo 3.8.4 Analizando nuevamente el Algoritmo 2, sean $t(n)$ y $T(n)$ los tiempos de mejor y peor caso, respectivamente, de una búsqueda exitosa —es decir, el tiempo de ejecución resultante de buscar un elemento presente en el conjunto. De forma similar, sean $t'(n)$ y $T'(n)$ los tiempos de mejor y peor caso de una búsqueda infructuosa —es decir, el tiempo de ejecución resultante de buscar un elemento que no está en el conjunto. Acotemos a continuación los tiempos generales —es decir, sin importar la instancia— de búsqueda exitosa e infructuosa:

- Respecto a las búsquedas exitosas, el mejor caso corresponde a buscar el elemento $x = S[1]$, ya que la ejecución termina en la primera iteración del algoritmo. De esta forma, $t(n) = 1$ iteración. Por otro lado, el peor caso corresponde a buscar el elemento $x = S[n]$, que requiere de $T(n) = n$ iteraciones. Dado que $t(n) \notin \Theta(T(n))$, no podemos decir que el tiempo general de búsqueda exitosa es $\Theta(n)$, sino más bien que es $O(n)$.
- Respecto a las búsquedas infructuosas, note que cualquiera de ellas necesita recorrer el arreglo completo. Por lo tanto, $t'(n) \in \Theta(T'(n))$, lo que significa que el tiempo general de búsqueda infructuosa es $\Theta(n)$.

Estudiemos a continuación una versión del Algoritmo 2, ahora para resolver la pertenencia en un conjunto ordenado S sobre el que se ha definido un orden total \preceq . El Algoritmo 3 implementa la búsqueda de x asumiendo que el conjunto es representado por el arreglo ordenado $S[1..n]$. Note la diferencia con el Algoritmo 2: al estar ordenado el

Algoritmo 3: PERTENENCIA($x, S[1..n]$)

Entrada: Un conjunto ordenado (asumiendo la relación de orden total \preceq) representado por un arreglo $S[1..n] \subseteq U$, y un elemento $x \in U$.

Salida: La posición i tal que $S[i] = x$, o $n + 1$ si $x \notin S$.

```

1  $i \leftarrow 1$ 
2 while  $i \leq n$  do
3   if  $x \preceq S[i]$  then
4     break
5   end
6    $i \leftarrow i + 1$ 
7 end
8 if  $i \leq n$  and  $x = S[i]$  then
9   return  $i$                                 // Búsqueda exitosa
10 else
11   return  $n + 1$                             // Indica que  $x \notin S$ 
12 end

```

arreglo, la búsqueda puede detenerse apenas se cumpla $x \preceq S[i]$. Respecto al tiempo de ejecución del algoritmo, el tiempo de búsqueda exitosa se mantiene como el Algoritmo 2, siendo $O(n)$ en general. Para las búsquedas infructuosas, en cambio, ahora el tiempo pueden acotarse con $O(n)$. Esto es porque no necesariamente deben recorrer el arreglo completo para finalizar.

Notación o. Para expresar formalmente que una función $f : \mathbb{N} \mapsto [0, +\infty)$ crece estrictamente más lento (en términos asintóticos) que otra función $g : \mathbb{N} \mapsto [0, +\infty)$, usaremos la notación $f(n) \in o(g(n))$, en donde definimos:

$$\begin{aligned} o(g(n)) = \{f : \mathbb{N} \mapsto [0, +\infty) \mid \\ \forall c \in \mathbb{R}^+, \\ \exists n_0 \in \mathbb{N}, \\ f(n) < c \cdot g(n), \forall n \geq n_0\}. \end{aligned} \quad (3.5)$$

Como es de esperar, esta definición es similar a la de la notación $O(\cdot)$, en donde ahora se usa ‘<’ en lugar de ‘ \leq ’. Sin embargo, hay otra diferencia sutil, aunque importante: para que $f(n) \in o(g(n))$, la desigualdad se debe cumplir para toda constante c , no basta con exhibir una única constante como antes. Por ejemplo, consideremos $f(n) = 2n$ y $g(n) = n$. Dado que ambas funciones son lineales, y por lo tanto crecen a la misma velocidad, es claro que debería ocurrir $2n \notin o(n)$. Sin embargo, supongamos que se usa el cuantificador “ $\exists c$ ” en la definición. En ese caso, si tomamos $c = 3$, tenemos que $2n < 3n$, $\forall n \geq 1$ y por lo tanto tendríamos, incorrectamente, que $2n \in o(n)$. El cuantificador adecuado aquí es “ $\forall c$ ”.

Ejemplo 3.8.5 Considere el conjunto

$$o(n) = \{1, 2, 3, \dots, \lg \lg n, \dots, \lg n, \dots, \sqrt{n}, \dots\},$$

en donde se han listado sólo las funciones más típicas. Note que el conjunto es infinito.

Observación 3.8.2 Para cualquier función $g : \mathbb{N} \mapsto [0, +\infty)$, se tiene que

$$o(g(n)) = O(g(n)) - \Theta(g(n)).$$

Reservaremos esta notación para especificar cotas superiores estrictas.

Notación ω . Para expresar formalmente que una función $f : \mathbb{N} \mapsto [0, +\infty)$ crece estrictamente más rápido (en términos asintóticos) que otra función $g : \mathbb{N} \mapsto [0, +\infty)$, usaremos la notación $f(n) \in \omega(g(n))$, en donde definimos:

$$\begin{aligned} \omega(g(n)) = \{f : \mathbb{N} \mapsto [0, +\infty) \mid \\ \forall c \in \mathbb{R}^+, \\ \exists n_0 \in \mathbb{N}, \\ f(n) > c \cdot g(n), \forall n \geq n_0\}. \end{aligned} \quad (3.6)$$

Ejemplo 3.8.6 Considere el conjunto

$$\omega(n) = \{\dots, n \lg n, \dots, n^2, 2n^2, \dots, n^k, \dots, 2^n, \dots, n!, \dots\},$$

en donde se han listado sólo las funciones más típicas. Note que el conjunto es infinito.

Observación 3.8.3 Para cualquier función $g : \mathbb{N} \mapsto [0, +\infty)$, se tiene que

$$\omega(g(n)) = \Omega(g(n)) - \Theta(g(n)).$$

Reservaremos esta notación para especificar cotas inferiores estrictas.

Propiedades de las Notaciones Asintóticas. Las siguientes propiedades de las notaciones asintóticas son importantes y útiles en la práctica.

Teorema 3.8.4 (Regla de la Suma) Sean $f_1, f_2, g_1, g_2 : \mathbb{N} \mapsto [0, +\infty)$ funciones. Si $f_1(n) \in O(g_1(n))$ y $f_2(n) \in O(g_2(n))$, entonces

$$f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

Demostración. Por hipótesis tenemos que $\exists c_1 \in \mathbb{R}, c_1 > 0, \exists c_2 \in \mathbb{R}, c_2 > 0, \exists n_1 \in \mathbb{N}, \exists n_2 \in \mathbb{N}$, tal que:

$$f_1(n) \leq c_1 \cdot g_1(n), \quad \forall n \geq n_1,$$

y

$$f_2(n) \leq c_2 \cdot g_2(n), \quad \forall n \geq n_2.$$

Si sumamos a ambos lados de esas desigualdades, tenemos que $\forall n \geq \max\{n_1, n_2\}$:

$$f_1(n) + f_2(n) \leq c_1 \cdot g_1(n) + c_2 \cdot g_2(n).$$

Por lo tanto, $\forall n \geq \max\{n_1, n_2\}$ también se cumple:

$$\begin{aligned} f_1(n) + f_2(n) &\leq c_1 \cdot \max\{g_1(n), g_2(n)\} + c_2 \cdot \max\{g_1(n), g_2(n)\} \\ &= (c_1 + c_2) \cdot \max\{g_1(n), g_2(n)\}. \end{aligned}$$

Si definimos $c = c_1 + c_2$, concluimos que

$$f_1(n) + f_2(n) \leq c \cdot \max\{g_1(n), g_2(n)\},$$

lo cual, por definición de O , significa que

$$f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

■

Intuitivamente, lo que la Regla de la Suma establece es que si tenemos dos algoritmos cuyos tiempos de ejecución son $f_1(n) \in O(g_1(n))$ y $f_2(n) \in O(g_2(n))$, el tiempo total resultante de ejecutar ambos algoritmos en secuencia está acotado superiormente (en términos asintóticos) por el más lento de los dos algoritmos. Por ejemplo, si los tiempos de ejecución son $5n + 2 \in O(n)$ y $n^2 + 3n - 1 \in O(n^2)$, entonces la

suma de esos tiempos es $n^2 + 8n + 1$, lo cual es $O(n^2)$. Existen las versiones análogas de la Regla de la Suma para las notaciones Ω y Θ . Su demostración queda como ejercicio para el lector.

Teorema 3.8.5 (Regla del Producto) Sean $f_1, f_2, g_1, g_2 : \mathbb{N} \mapsto [0, +\infty)$ funciones. Si $f_1(n) \in O(g_1(n))$ y $f_2(n) \in O(g_2(n))$, entonces

$$f_1(n) \cdot f_2(n) \in O(g_1(n) \cdot g_2(n)).$$

La demostración es similar a la de la Regla de la Suma, y se deja como ejercicio para el lector.

Además de estas dos reglas, muchas de las propiedades de los números reales son válidas también para las notaciones asintóticas —asumiendo que $f(n)$ y $g(n)$ son asintóticamente positivas. Se listan a continuación dichas propiedades. La demostración de las mismas se deja como ejercicio para el lector.

Teorema 3.8.6 (Transitividad) Para las funciones $f, g, h : \mathbb{N} \mapsto [0, +\infty)$ se cumplen las siguientes propiedades:

1. $f(n) \in \Theta(g(n))$ y $g(n) \in \Theta(h(n))$ implica $f(n) \in \Theta(h(n))$.
2. $f(n) \in O(g(n))$ y $g(n) \in O(h(n))$ implica $f(n) \in O(h(n))$.
3. $f(n) \in \Omega(g(n))$ y $g(n) \in \Omega(h(n))$ implica $f(n) \in \Omega(h(n))$.
4. $f(n) \in o(g(n))$ y $g(n) \in o(h(n))$ implica $f(n) \in o(h(n))$.
5. $f(n) \in \omega(g(n))$ y $g(n) \in \omega(h(n))$ implica $f(n) \in \omega(h(n))$.

Teorema 3.8.7 (Reflexividad) Para toda función $f : \mathbb{N} \mapsto [0, +\infty)$ se cumple:

1. $f(n) \in \Theta(f(n))$.
2. $f(n) \in O(f(n))$.
3. $f(n) \in \Omega(f(n))$.

Teorema 3.8.8 (Simetría) Para todas las funciones $f, g : \mathbb{N} \mapsto [0, +\infty)$ se cumple: $f(n) \in \Theta(g(n))$ si y sólo si $g(n) \in \Theta(f(n))$.

Teorema 3.8.9 (Simetría Transpuesta) Para las funciones $f, g : \mathbb{N} \mapsto [0, +\infty)$ se cumplen las siguientes propiedades:

1. $f(n) \in O(g(n))$ si y sólo si $g(n) \in \Omega(f(n))$.
2. $f(n) \in o(g(n))$ si y sólo si $g(n) \in \omega(f(n))$.

Al cumplirse las propiedades de transitividad, reflexividad, simetría, y simetría transpuesta, podemos hacer una analogía entre la comparación asintótica de funciones f y g y números reales a y b . La siguiente tabla muestra la analogía entre las diferentes notaciones asintóticas (línea de arriba) y los correspondientes operadores relaciones (línea de abajo):

O	Ω	Θ	o	ω
\leq	\geq	$=$	$<$	$>$

Sin embargo, hay una propiedad que no se cumple para las notaciones asintóticas: la *tricotomía*. Ésta establece que para dos números reales a y b , exactamente una de las siguientes comparaciones es verdadera: $a < b$, $a = b$, o $a > b$. Esto es, cualquier par de números reales pueden ser comparados. Sin embargo, no todas las funciones son asintóticamente comparables. Puede pasar que $f(n) \notin O(g(n))$ y $f(n) \notin \Omega(g(n))$, como es el caso de las funciones n y $n^{1+\sin n}$.

La Regla del Límite para Comparar Funciones. Dadas dos funciones $f(n)$ y $g(n)$, usualmente uno quiere saber si una crece más rápido que la otra, con el objetivo de compararlas. Una manera de resolver esto es mediante la Regla del Límite. Si existe el límite:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k, \quad (3.7)$$

entonces tenemos:

1. Si $k = \infty$, entonces $f(n) \in \omega(g(n))$.
2. Si $k = 0$, entonces $f(n) \in O(g(n)) \wedge g(n) \notin O(f(n))$. Es decir, $f(n) \in o(g(n))$.
3. Si $k \neq 0$ y $k \neq \infty$, entonces $f(n) \in \Theta(g(n))$.

Ejercicios

1. Graficar las siguientes funciones de n . Para cada función, establecer el rango de valores de n para los cuales la función es la más eficiente (si se pensara como el tiempo de ejecución de un algoritmo).

$$4n^2, \log_3 n, 3^n, 20n, 2, \log_2 n, n^{2/3}$$

2. Usando la definición de O , mostrar que $a \in O(1)$ para cualquier constante natural a . Además, mostrar que $a \in O(n)$.
3. Calcule las siguientes sumas:

- a) $1 + 3 + 5 + 7 + \dots + 999$.
- b) $2 + 4 + 8 + 16 + \dots + 1024$.
- c) $\sum_{i=3}^{n+1} 1$.
- d) $\sum_{i=3}^{n+1} i$.
- e) $\sum_{i=0}^{n-1} i(i+1)$.

4. Encontrar el orden de crecimiento de las siguientes sumas. Usar la notación $\Theta(g(n))$ con la función $g(n)$ más ajustada posible

- a) $\sum_{i=0}^{n-1} (i^2 + 1)^2$.
- b) $\sum_{i=2}^{n-1} \log_2(i^2)$.
- c) $\sum_{i=1}^n (i+1)2^{i-1}$.
- d) $\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} i+j$.

5. Indicar cuáles de las siguientes afirmaciones son verdaderas y cuáles son falsas:

- a) $n^2 \in O(n^3)$
- b) $n^3 \in O(n^2)$
- c) $n^2 \in \Omega(n^3)$
- d) $2^{n+1} \in O(2^n)$

- e) $(n+1)! \in O(n!)$
- f) $2^{2n} \in O(2^n)$
- g) $f(n) \in O(n) \Rightarrow 2^{f(n)} \in O(2^n)$
- h) $3^n \in O(2^n)$
- i) $\log n \in O(n^{1/2})$
- j) $n^{1/2} \in O(\log n)$
- k) $n^i \in O(n^j)$ si $i < j$
- l) $n! \in \Theta((n+1)!)$
- m) $(n+a)^2 \in \Theta(n^2)$, para a constante.
- n) $n! \in \Omega(2^n)$.

6. Demostrar las siguientes inclusiones estrictas:

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^k) \subset O(2^n) \subset O(n!)$$

7. Ordenar las siguientes funciones por velocidad de crecimiento, desde la que crece más lentamente a la que crece más rápidamente.

$$4n^2, \log_3 n, n!, 3^n, 20n, 2, \log_2 n, n^{2/3}.$$

Ayuda: usar la aproximación de Stirling para clasificar a $n!$, la cual indica que $\log n! \approx n \log n$.

8. Ordenar las siguientes funciones por velocidad de crecimiento, desde la que crece más lentamente a la que crece más rápidamente:

$$n, \sqrt{n}, \log n, \log \log n, \log^2 n, n/\log n, \sqrt{n} \log^2 n, (1/3)^n, (3/2)^n, 33, n^2.$$

9. Demostrar las siguientes propiedades de la notación O :

- a) Si $f_1(n) \in O(g_1(n))$ y $f_2(n) \in O(g_2(n))$, entonces $f_1(n) + f_2(n) \in O(\max(g_1(n), g_2(n)))$.
- b) $\forall a, b > 1$, se tiene que $\log_a n \in O(\log_b n)$
- c) Si $f(n) \in O(g(n))$ y $g(n) \in O(h(n)) \Rightarrow f(n) \in O(h(n))$.

10. Asumiendo que $T_1(n) \in \Omega(f(n))$, $T_1(n) \in O(h(n))$, $T_2(n) \in \Omega(g(n))$, y $T_2(n) \in O(t(n))$, demostrar o refutar las siguientes afirmaciones:

- a) $T_1(n) + T_2(n) \in \Omega(\max(f, g))$.
- b) $T_1(n) \cdot T_2(n) \in \Omega(f \cdot g)$.
- c) $T_1(n) + T_2(n) \in O(\max(h, t) + \min(h, t))$.

11. a) Demostrar que para un polinomio de grado k ,

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_0,$$

(con $a_k > 0$), se cumple $p(n) \in \Theta(n^k)$.

- b) Demostrar que las funciones exponenciales a^n tienen diferentes ordenes de crecimiento, para diferentes valores de $a > 0$. En otras palabras $a^n \notin \Theta(b^n)$ si y solo si $a \neq b$.

12. a) ¿Cuál es el menor entero k tal que $\sqrt{n} \in O(n^k)$?

b) ¿Cuál es el menor entero k tal que $n \log n \in O(n^k)$?

13. Responda si cada una de las siguientes afirmaciones es verdadera o falsa. Justifique por qué, o por qué no.

- a) $2n \in \Theta(3n)$.
- b) $2^n \in \Theta(3^n)$.

14. Para cada uno de los siguientes pares de funciones, determinar si $f(n) \in O(g(n))$, $f(n) \in \Omega(g(n))$ o $f(n) \in \Theta(g(n))$. Justificar su

respuesta mostrando el desarrollo que lo llevó a tomar la decisión.
Asuma que los logaritmos son en base 2.

- a) $f(n) = \log(n^2)$; $g(n) = \log n + 5$.
- b) $f(n) = \sqrt{n}$; $g(n) = \log n^2$.
- c) $f(n) = \log n^2$; $g(n) = \log n$.
- d) $f(n) = n$; $g(n) = \log n^2$.
- e) $f(n) = n \log n$; $g(n) = \log n^2$.
- f) $f(n) = \log n^2$; $g(n) = \log^2 n$.
- g) $f(n) = \log 10$; $g(n) = 10$.
- h) $f(n) = 2^n$; $g(n) = 10n^2$.
- i) $f(n) = 2^n$; $g(n) = n \log n$.
- j) $f(n) = 3^n$; $g(n) = 2^n$.
- k) $f(n) = 2^n$; $g(n) = n^n$.

15. Demostrar que $\sum_{i=1}^n \lceil \log i \rceil \in O(n \log n)$.

16. La varianza muestral de n observaciones x_1, \dots, x_n con media muestral

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n},$$

puede ser computada como

$$\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n - 1}$$

o alternativamente

$$\frac{\sum_{i=1}^n x_i^2 - (\sum_{i=1}^n x_i)^2 / n}{n - 1}.$$

Encuentre y compare el número de divisiones, multiplicaciones, y sumas/restas que se requieren para computar la varianza de acuerdo a cada una de esas fórmulas.

17. Dado un polinomio de grado n , $p(x) = a_0 + a_1x + \dots + a_nx^n$.

- a) Describir un algoritmo de tiempo $\Theta(n^2)$ para computar $p(x)$, para un valor de x dado.
- b) Considere reescribir $p(x)$ como a continuación:

$$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \dots + x(a_{n-1} + xa_n) \dots))),$$

lo que se conoce como el método de Horner. Use notación asintótica para caracterizar el número de operaciones aritméticas que lleva a cabo el método.

18. Determinar (usando notación asintótica) el tiempo de ejecución en el peor caso de los siguientes fragmentos de código. Asuma que todas las variables son de tipo `int`. Expresé el tiempo en función de la variable n .

- a)

```
sum = 0;
for (i=0; i<3; i++)
    for (j=0; j<n; j++)
        sum++;
```
- b)

```
sum = 0;
for (i=0; i<n*n; i++)
    sum++;
```
- c)

```
for (i=0; i<n-1; i++)
    for (j=i+1; j<n; j++) {
```

```

        tmp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = tmp;
    }
d) sum = 0;
   for (i=1; i<=n; i++)
       for (j=1; j<=n; j=j*2)
           sum++

```

19. Obtener en notación O el tiempo de ejecución de peor caso del siguiente algoritmo:

```

int buscar(int *v, int n, int e) {
    int i = 1;
    while (i < n && e != v[i])
        i++;
    if (i < n)
        return i;
    else
        return -1; // elemento no encontrado
}

```

20. En cada uno de los siguientes casos, encuentre velocidades de crecimiento X tal que:

- Si $T(n) = X$, entonces $T(2n) = X^2$. Es decir, una velocidad de crecimiento X tal que al duplicar el tamaño de la entrada, el tiempo obtenido es el cuadrado del tiempo original.
- Si $T(n) = X$, entonces $T(2n) = X+1$. Es decir, una velocidad de crecimiento X tal que al duplicar el tamaño de la entrada se incrementa el tiempo de ejecución solamente en 1.
- Si $T(n) = X$, entonces $T(n^2) = X+1$. Es decir, una velocidad de crecimiento X tal que al elevar al cuadrado el tamaño de la entrada se incrementa el tiempo de ejecución solamente en 1.
- Si $T(n) = X$, entonces $T(n+1) = 2X$. Es decir, una velocidad de crecimiento X tal que al incrementar en 1 el tamaño de la entrada se duplica el tiempo de ejecución.

21. Demostrar que si f y g son dos funciones crecientes, entonces $f \in O(g) \Leftrightarrow g \in \Omega(f)$.

22. Demostrar que si $f \in O(g) \wedge g \notin O(f) \Rightarrow f \in o(g)$.

23. Demostrar que $o(f(n)) \cap \omega(f(n)) = \emptyset$.

24. Suponiendo que $T_1 \in O(f)$ y que $T_2 \in O(f)$, indicar cuáles de las siguientes afirmaciones son ciertas:

- $T_1 + T_2 \in O(f)$.
- $T_1 - T_2 \in O(f)$.
- $T_1/T_2 \in O(1)$.
- $T_1 \in O(T_2)$.

25. La proposición $O(f(n)) - O(f(n)) = 0$, ¿Es verdadera o falsa? Argumente.

26. Demostrar que para cualquier constante $k > 0$ se verifica que $\log^k n \in O(n)$.

27. En los siguientes items, acotar b asintóticamente de manera que $f(n) \in o(n)$. Su cota debe ser lo más general posible, en el sentido

de incluir la mayor cantidad de funciones que cumplan con la condición. Justifique brevemente su respuesta.

- a) $f(n) = \frac{n}{b}$.
- b) $f(n) = \frac{n \lg n}{b}$.
- c) $f(n) = b \cdot \sqrt{n}$.
- d) $f(n) = \frac{n \cdot b}{\lg(\sqrt{n})}$.
- e) $f(n) = b \cdot \frac{\sqrt{n}}{\lg n}$.

28. ¿Cuál es el error en el siguiente argumento? Dado que $n \in O(n)$ y $kn \in O(n)$ para k constante, entonces tenemos que

$$\sum_{k=1}^n kn = \sum_{k=1}^n O(n) = O(n^2).$$

29. Suponga que se cuenta con $n > 2$ monedas idénticas (indistinguibles entre ellas). Suponga además que exactamente una de las monedas es falsa, y su peso es distinto al de las monedas legítimas. Se tiene acceso a una balanza de dos platos, la cual permite colocar objetos en cada uno de sus platos, indicando de qué lado está el mayor peso. Cada uso de la balanza se cobra, por lo que debe usarse la menor cantidad de veces posible.

- a) Diseñe un algoritmo que permita determinar si la moneda falsa tiene peso mayor o menor que las monedas legítimas en tiempo $\Theta(1)$ usos de la balanza.
- b) Diseñe un algoritmo eficiente para encontrar la moneda falsa. Analice su algoritmo y determine su costo usando la notación asintótica más adecuada.

La inducción matemática es una poderosa técnica de demostración que, además, será de utilidad como:

- Técnica de diseño de algoritmos, cuando la emparentemos con su análoga en computación: la recursión. Así, estudiaremos la capacidad de resolver problemas grandes a partir de las soluciones a problemas del mismo tipo, pero más pequeños.
- Técnica alternativa para definir conjuntos por extensión, conocida como *inducción estructural*.
- Técnica para definir funciones matemáticas, conocidas como *ecuaciones de recurrencia*.

Ilustramos en este capítulo los 4 usos principales de la inducción.

4.1. Uso 1: La Inducción como Técnica de Demostración

La idea fundamental detrás de la inducción matemática es ilustrada en el siguiente ejemplo.

Ejemplo 4.1.1 Los Axiomas de Peano establecen que:

1. 1 es un número natural.
2. Todo número natural x tiene un sucesor $\text{suc}(x)$ que también es un número natural.
3. 1 no es el sucesor de ningún número natural.
4. Dados dos números naturales x e y , $\text{suc}(x) = \text{suc}(y) \Leftrightarrow x = y$.
5. Si una propiedad es verdadera para el 1, y si la verdad de esa propiedad para un número x implica que también es verdad para $\text{suc}(x)$, entonces la propiedad es verdadera para todos los números naturales.

Note que el quinto axioma puede verificarse fácilmente. Si una propiedad se cumple para el 1 y además se cumple para el sucesor, entonces inmediatamente se cumple para el 2. Pero eso implica que también se cumple para el 3, por lo cual también se cumple para el 4, y así siguiendo sobre todos los naturales.

El quinto axioma en este ejemplo es conocido como *el axioma de inducción*, y refleja el concepto que queremos definir. Para probar que cierta propiedad —o proposición— se cumple para todos los naturales, debemos ser capaces de probar primero que se cumple para $n = 1$ ¹. Luego, se debe demostrar que el hecho de que la proposición se cumpla para n implica que también se cumple para $\text{suc}(n)$, con lo cual estaremos probando inductivamente que se cumple para todos los naturales. Note la diferencia con las demostraciones tradicionales,

1: O algún otro valor inicial adecuado.

que prueban directamente que una propiedad se cumple para todos los naturales. En muchos casos, demostrar por inducción es más simple.

Sea $\mathbb{P}(n)$ una proposición que se quiere demostrar, la cual está asociada a una variable n . Por ejemplo, podemos querer demostrar que $\sum_{i=1}^n i = \frac{n(n+1)}{2}$ se cumple para todo $n \in \mathbb{N}$. Una demostración por inducción para $\mathbb{P}(n)$ procede de la siguiente manera:

Caso base: Se debe chequear que $\mathbb{P}(n)$ se cumple para un valor inicial $n = n_0$.

Hipótesis inductiva: Se asume que $\mathbb{P}(n-1)$ se cumple.

Paso inductivo: Para todo $n > n_0$, se debe demostrar que el hecho de que se cumpla $\mathbb{P}(n-1)$ implica que $\mathbb{P}(n)$ se cumple.

Al igual que en el ejemplo de los Axiomas de Peano, esto es suficiente para demostrar que $\mathbb{P}(n)$ se cumple para todo $n \geq n_0$.

A continuación, ilustramos el concepto con algunos ejemplos.

La Serie Aritmética. El siguiente resultado será de interés en numerosas ocasiones más adelante.

Lema 4.1.1 Para todo $n \in \mathbb{N}$ se cumple $\sum_{i=1}^n i = \frac{n(n+1)}{2}$.

Demostración. Por inducción sobre la variable n :

- *Caso base:* notar que la propiedad se cumple para $n = 1$, ya que ambos lados de la ecuación son iguales.
- *Hipótesis inductiva:* asumimos que se cumple $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$.
- *Paso inductivo:* para todo $n > 1$, probaremos que si la propiedad se cumple para $n-1$, entonces también se cumple para n . Note que:

$$\begin{aligned} \sum_{i=1}^n i &= \left(\sum_{i=1}^{n-1} i \right) + n \\ &\stackrel{\text{h.i.}}{=} \frac{n(n-1)}{2} + n \\ &= n \left(1 + \frac{n-1}{2} \right) = \frac{n(n+1)}{2}. \end{aligned}$$

■

Suma de Números Impares. Demostremos a continuación que la suma de los primeros n números impares es n^2 .

Lema 4.1.2 Para todo $n \in \mathbb{N}$ se cumple $\sum_{i=1}^n (2i-1) = n^2$.

Demostración. Por inducción sobre n :

- *Caso base:* para $n = 1$, la propiedad se cumple ya que la suma es 1, lo que es igual a $1^2 = 1$.
- *Hipótesis inductiva:* asumimos que $\sum_{i=1}^{n-1} (2i-1) = (n-1)^2$.

- *Paso inductivo:* para $n > 1$ tenemos que:

$$\begin{aligned}\sum_{i=1}^n (2i-1) &= \sum_{i=1}^{n-1} (2i-1) + 2n-1 \\ &\stackrel{\text{h.i.}}{=} (n-1)^2 + 2n-1 \\ &= n^2.\end{aligned}$$

■

Un Resultado Relacionado a la Serie Geométrica. Otro resultado de interés es el siguiente.

Lema 4.1.3 Para todo $n \in \mathbb{N}^0$ se cumple $\sum_{i=0}^n \frac{1}{2^i} < 2$.²

2: Recuerde que $\sum_{i=0}^{\infty} \frac{1}{2^i}$ es una serie geométrica con razón $a = 1/2$, por lo que la suma es igual a $\frac{1}{1-a} = 2$.

Demostración. Por inducción sobre n :

- *Caso base:* para $n = 0$ la suma produce 1, lo cual obviamente es < 2 , por lo que el Lema se cumple para este caso.
- *Hipótesis inductiva:* asumimos que se cumple $\sum_{i=0}^{n-1} \frac{1}{2^i} < 2$.
- *Paso inductivo:* para todo $n > 0$, demostramos ahora que si la propiedad se cumple para $n-1$, entonces también se cumple para n . Aquí es importante notar lo siguiente:

$$\begin{aligned}\sum_{i=0}^n \frac{1}{2^i} &= 1 + \sum_{i=1}^n \frac{1}{2^i} \\ &= 1 + \frac{1}{2} \underbrace{\sum_{i=0}^{n-1} \frac{1}{2^i}}_{< 2, \text{ por h.i.}} \\ &\quad \underbrace{\hspace{1.5cm}}_{< 1},\end{aligned}$$

lo cual es < 2 , ya que el segundo término es < 1 . Esto completa la demostración.

■

Cubriendo un Tablero con Poliomínos. En 1954, Solomon Golomb —siendo en ese momento un estudiante de Harvard— introdujo el concepto de *poliomínos* y planteó problemas relacionados a ellos. Un poliomínó de grado n es una figura formada por n cuadrados unitarios pegados entre sí por sus lados. Por ejemplo, un poliomínó de grado 1 —o *monominó*— es un simple cuadrado unitario \square . Un poliomínó de grado 2 —o *dominó*— es de la forma $\square\square$. Un poliomínó de grado 3 —o *triominó*— puede tener forma de I, $\square\square\square$, o forma de L, \square
 \square . Usaremos inducción para demostrar la siguiente propiedad.

Lema 4.1.4 Todo tablero bidimensional de $2^n \times 2^n$ casillas unitarias (para $n \geq 1$) al que se le ha removido una casilla arbitraria puede ser cubierto completamente con triominós L.

Demostración. Por inducción sobre n :



Figura 4.1

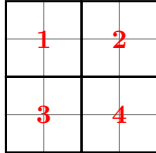


Figura 4.2

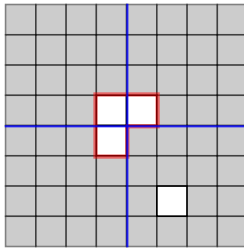


Figura 4.3

- *Caso base*: para $n = 1$, tenemos un tablero de 2×2 casillas. Cualquiera sea la casilla que removamos, las restantes 3 casillas pueden ser cubiertas usando un único triominó L, tal como lo muestran los diagramas de la Figura 4.1 —la casilla blanca es la que ha sido removida.
- *Hipótesis inductiva*: todo tablero de $2^{n-1} \times 2^{n-1}$ casillas al que se le ha removido una única casilla puede ser cubierto completamente por triominós L.
- *Paso inductivo*: Consideremos un tablero de $2^n \times 2^n$, para $n > 1$. Note que este tablero se puede descomponer en 4 tableros de $2^{n-1} \times 2^{n-1}$, enumerados como en la Figura 4.2. Sin pérdida de generalidad, asumamos que la casilla removida está en el cuadrante 4. Luego, removemos las siguientes 3 casillas: la inferior derecha del cuadrante 1, la inferior izquierda del cuadrante 2, y la superior derecha del cuadrante 3. El resultado se muestra en la Figura 4.3, con las casillas removidas en blanco, y una casilla cualquiera removida en el cuadrante 4 —esa es la casilla faltante original. Finalmente, cubrimos con triominós cada uno de los cuadrantes, con su casilla removida. Dado que son de tamaño $2^{n-1} \times 2^{n-1}$, es posible cubrirlos por hipótesis inductiva. Finalmente, colocamos un triominó en el centro del tablero para cubrir las 3 casillas removidas, dejando sin cubrir únicamente la casilla removida originalmente.

■

4.2. Uso 2: Definición Inductiva de Conjuntos

Además de las dos formas de definir conjuntos estudiadas anteriormente —extensión y comprensión—, es posible definir los elementos que conforman un conjunto usando inducción. Dichas definiciones consisten de:

- Los *casos base*, que definen los elementos básicos que pertenecen al conjunto que se está definiendo.
- La *regla inductiva*, que define cómo construir nuevos elementos del conjunto a partir de elementos que ya pertenecen al mismo.

Los únicos elementos que pertenecen al conjunto que se está definiendo son aquellos que pueden ser formados mediante la aplicación de las reglas anteriores.

Ejemplo 4.2.1 El conjunto \mathbb{N} de números naturales puede definirse por inducción de la siguiente manera ³:

- *Caso base*: $1 \in \mathbb{N}$.
- *Regla inductiva*: si $n \in \mathbb{N}$, entonces $n + 1 \in \mathbb{N}$.

Otro ejemplo —esta vez relacionado a strings sobre un alfabeto Σ — es el siguiente.

3: Esta definición de los naturales corresponde a Giuseppe Peano.

Ejemplo 4.2.2 El conjunto infinito Σ^* de todos los posibles strings para un alfabeto Σ se puede definir inductivamente de la siguiente manera:

- *Caso base:* $\varepsilon \in \Sigma^*$.
- *Regla inductiva:* si $x \in \Sigma^*$, entonces $\forall s \in \Sigma, xs \in \Sigma^*$.

Estudiaremos más adelante definiciones inductivas de conjuntos más complejos.

4.3. Uso 3: Ecuaciones de Recurrencia

Una *ecuación de recurrencia* permite definir una función matemática en término de sí misma. Esto es, se define $f(n)$ en términos de $f(m)$, para $m < n$. El siguiente ejemplo ilustra su uso.

Ejemplo 4.3.1 Para $n \in \mathbb{N}^0$, se define su factorial de la siguiente manera:

$$n! = \begin{cases} n(n-1)!, & n \geq 1; \\ 1, & n = 0. \end{cases}$$

Las ecuaciones de recurrencia son una manera matemáticamente válida de definir funciones, que constan de dos partes principales:

- El *caso base*, o *caso conocido*, que en el Ejemplo 4.3.1 es $0! = 1$; y
- La *definición recurrente*, que construye la solución para n en base a soluciones para valores $m < n$. En el caso particular del Ejemplo 4.3.1, construye $n!$ en base a $(n-1)!$.

Note la similitud con la inducción matemática, que ilustraremos con el ejemplo del factorial: una vez definido el caso base $0!$, podemos asumir que sabemos computar correctamente $(n-1)!$. Luego, para $n > 0$, mostramos cómo calcular $n!$ en base a $(n-1)!$ que, por hipótesis inductiva, ya sabemos calcular. Eso es suficiente para calcular el factorial de cualquier número en \mathbb{N}^0 . Esto es, $1! = 1 \cdot 0! = 1$, por lo tanto ya sabemos cómo calcularlo. Entonces, tenemos $2! = 2 \cdot 1! = 2$, lo que nos lleva a poder calcular $3! = 3 \cdot 2! = 6$, y así siguiendo para todos los naturales.

4.4. Uso 4: Algoritmos Recursivos

Un algoritmo recursivo permite resolver una instancia de un problema a partir de resolver subinstancias más pequeñas del mismo. Si este proceso se aplica repetidamente, eventualmente alcanzaremos instancias tan pequeñas —y, por lo tanto, fáciles de resolver— que puedan ser resueltas directamente. De esta manera, se construye la solución a la instancia original del problema a partir de soluciones a subinstancias más pequeñas. Éste es un concepto que usaremos frecuentemente en adelante —en diferentes variantes— para diseñar algoritmos, y que está fuertemente relacionado con la inducción matemática.

Denotemos con $\mathcal{A}_{\mathcal{P}}(I)$ la ejecución de un algoritmo $\mathcal{A}_{\mathcal{P}}$ sobre una instancia I del problema \mathcal{P} , la cual tiene tamaño $|I| = n$. Un algoritmo recursivo $\mathcal{A}_{\mathcal{P}}$ consiste básicamente de dos partes:

Base de la Recursión: Si el tamaño de la instancia del problema es suficientemente pequeña, $n \leq \tau$, para algún valor τ especificado, el problema se resuelve directamente.

Recursión: En otro caso, dividimos la instancia I del problema en k subinstancias I_1, \dots, I_k de tamaño n_1, \dots, n_k , respectivamente, tal que $n_1, \dots, n_k < n$. Luego, se construye la solución para I a partir de las soluciones $\mathcal{A}_{\mathcal{P}}(I_1), \dots, \mathcal{A}_{\mathcal{P}}(I_k)$ de las subinstancias ⁴.

4: En un lenguaje de programación, una función recursiva es una que se invoca a sí misma, lo cual es equivalente a nuestra definición.

En un algoritmo recursivo, el caso base es obligatorio: debe haber al menos uno para evitar que la recursión sea infinita. Por otro lado, la parte recursiva construye la solución a partir de la solución a los subinstancias del problema. Note que, de alguna manera, estamos asumiendo que sabemos cómo resolver las subinstancias mediante $\mathcal{A}_{\mathcal{P}}(I_i)$, lo cual es similar a plantear una hipótesis inductiva al usar inducción. En resumen, los casos base muestran cómo resolver las instancias más simples, mientras que la recursión muestra cómo resolver el problema original a partir de resolver subproblemas más pequeños —que asumimos ya sabemos resolver.

Ejemplo 4.4.1 Consideremos el problema de imprimir, en orden, cada uno de los números naturales del conjunto \mathbb{N}_n , para un valor $n \geq 1$ dado. El Algoritmo 4 muestra el proceso recursivo. El caso base que define el algoritmo es $n = 1$, que simplemente imprime el 1. Luego, planteamos la hipótesis inductiva ⁵: sabemos cómo imprimir los primeros $n-1$ naturales. Esto implica que `GENERARNATURALES($n-1$)` realiza correctamente ese proceso. A continuación, resolvemos para $n > 1$. Note que el proceso de imprimir los primeros n naturales puede descomponerse en: (1) imprimir los primeros $n-1$ naturales —algo que ya sabemos hacer, por hipótesis inductiva—; y (2) imprimir n . Note cómo el algoritmo primero chequea el caso base para, en caso de no cumplirse la condición, ejecutar el caso recursivo. Esto es típico de cualquier algoritmo recursivo.

5: ¿O deberíamos llamarla *hipótesis recursiva*?

Algoritmo 4: `GENERARNATURALES(n)`

```

1 if  $n = 1$  then
2   | Imprimir 1
3 else
4   | GENERARNATURALES( $n - 1$ )
5   | Imprimir  $n$ 
6 end
```

Al comienzo puede resultar complicado el hecho de escribir la solución a un problema en base a resolver subproblemas más pequeños usando el mismo algoritmo que estamos diseñando. Un consejo es concentrarse en 3 aspectos principales:

1. Definir los **casos base** correctamente.
2. Definir adecuadamente las **subinstancias que deben resolverse recursivamente**. Es importante que esas subinstancias

alcancen, en algún momento, alguno de los casos base definidos.

3. **Combinar, de ser necesario, las soluciones a las subinstancias** de forma adecuada para obtener la respuesta al problema original.

Si hacemos esto correctamente para el problema original, entonces las subinstancias —que serán resueltas recursivamente con el mismo algoritmo— serán resueltas correctamente ⁶, por lo que su solución puede ser usada para resolver el problema original de forma segura ⁷. Esto es similar a la inducción matemática, en donde uno: (1) se demuestra el caso base, que equivale al caso base de la recursión; luego (2) se asume que la propiedad se cumple para $< n$, estableciendo la hipótesis inductiva. Esto equivale a asumir que las soluciones de las k subinstancias son correctas; finalmente (3) se demuestra la propiedad para n , construyendo en base a la hipótesis inductiva —es decir, combinando las soluciones a las subinstancias que, por hipótesis inductiva, tienen solución correcta. Diversos autores notan esto mismo, como Erickson ⁸ (quien llama “*hada de la recursión*” al proceso que hemos explicado), o Graham, Knuth, y Patashnik ⁹ (que llaman “*nombrar y conquistar*” a un proceso similar al explicado); incluso, hay autores como Manber ¹⁰ que propone directamente el diseño de algoritmos por inducción, y muestra que muchas de las técnicas de diseño clásicas de algoritmos no son más que variantes de inducción matemática.

6: *boom!*

7: **boooooom!**

8: Jeff Erickson. *Algorithms*, 2019.

9: Ronald L. Graham, Donald E. Knuth, Oren Patashnik. *Concrete Mathematics*, 1989.

10: Udi Manber. *Introduction to Algorithms: A Creative Approach*, 1989.

Ejemplo 4.4.2 Estudiemos a continuación un algoritmo recursivo para el cómputo del factorial de un número, función definida en la ecuación de recurrencia del Ejemplo 4.3.1 (página 69). En esa definición podemos ver el caso base, que define $0! = 1$ y la recursión que define $n!$ en términos de multiplicar la solución a $(n - 1)!$ por n . El Algoritmo 5 muestra una implementación de esta definición.

Algoritmo 5: FACTORIAL(n)

```

1 if  $n = 0$  then
2   | return 0
3 else
4   | return  $n \times \text{FACTORIAL}(n - 1)$ 
5 end
```

Intuitivamente...

Pruebe que el tiempo de ejecución del Algoritmo 5 es $\Theta(n)$.

Respecto a la recursión en la práctica...

Aunque la recursión es una herramienta poderosa que permite definir algoritmos que en general son elegantes y fáciles de entender, es importante notar que no siempre las soluciones recursivas son prácticas, por diversas razones técnicas. La primera razón es que las invocaciones recursivas agregan costos adicionales constantes, tanto por la invocación misma, el paso de parámetros, y el retorno. Una implementación iterativa del mismo proceso podría producir

una ejecución más eficiente en la práctica —es importante, sin embargo, notar que el tiempo de ejecución asintótico del algoritmo no se modifica. Existen técnicas que permiten eliminar la recursión para obtener código iterativo, como por ejemplo la *eliminación de recursión de cola*. La segunda razón para evitar la recursión en la práctica es la *profundidad de la recursión*, que es la máxima cantidad de invocaciones pendientes de una función recursiva, en espera del resultado de invocaciones posteriores. Por ejemplo, la recursión del Algoritmo 5 tiene profundidad n . En el sistema de ejecución cada invocación es almacenada en una pila, que tiene un tamaño máximo fijo y limitado y que puede ser alcanzado relativamente fácil si la profundidad es grande como la del Algoritmo 5.

Los Números de Fibonacci y la Granja de Conejos

Leonardo de Pisa, conocido también como *Fibonacci*, fue un reconocido matemático italiano del siglo XIII, que definió (entre otras cosas) la sucesión infinita de números naturales que lleva su nombre:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...

Su motivación era modelar la cría de conejos, que describimos a continuación. Supongamos que: (1) una pareja de conejos debe tener al menos un mes de edad para poder cruzarse, (2) que el período de gestación es de exactamente 1 mes, (3) siempre dan a luz una única pareja de conejos, y (4) una pareja de conejos se cruza apenas es posible hacerlo. Comenzamos el mes 0 con $F_0 = 0$ conejos. Al comenzar el mes 1, adquirimos una pareja de conejos recién nacidos (llamaremos c_1 a esta pareja), por lo que tenemos $F_1 = 1$. Para simplificar el estudio, asumimos que las parejas de conejos no mueren, y están indefinidamente en edad de reproducción. El crecimiento de nuestra granja de conejos es el siguiente:

Comienzo del mes 0: tenemos $F_0 = 0$ parejas de conejos.

Comienzo del mes 1: adquirimos la pareja de conejos c_1 , recién nacidos. Tenemos $F_1 = 1$ parejas de conejos.

Comienzo del mes 2: la pareja c_1 cumple 1 mes de edad y se reproduce. Todavía tenemos $F_2 = 1 + 0 = 1$ pareja de conejos.

Comienzo del mes 3: la pareja c_1 da a luz a la pareja c_2 , y vuelve a cruzarse. Tenemos $F_3 = 1 + 1 = 2$ parejas.

Comienzo del mes 4: la pareja c_1 da a luz a la pareja c_3 y vuelven a cruzarse, y la pareja c_2 se cruza. Tenemos $F_4 = 2 + 1 = 3$ parejas.

Comienzo del mes 5: la pareja c_1 da a luz a la pareja c_4 , la pareja c_2 hace lo propio con la pareja c_5 . Las parejas c_1 , c_2 , y c_3 se cruzan. Tenemos $F_5 = 3 + 2 = 5$ parejas.

⋮

Comienzo del mes i : las F_{i-2} parejas que se cruzaron al comienzo del mes anterior dan a luz F_{i-2} nuevas parejas, que se suman a las F_{i-1} parejas que ya teníamos, por lo que ahora tenemos $F_i = F_{i-1} + F_{i-2}$ parejas. Las F_{i-1} parejas en edad de reproducción se cruzan, y darán a luz al comienzo del mes $i + 1$.

Entonces, el i -ésimo número de la secuencia de Fibonacci $F_i = F_{i-1} + F_{i-2}$, es la cantidad de parejas de conejos que tenemos al comienzo del

i -ésimo mes de cría, para $i \geq 2$, suponiendo que comenzamos el mes 0 sin conejos ($F_0 = 0$), y al comienzo del mes 1 adquirimos una pareja de conejos ($F_1 = 1$).

La ecuación de recurrencia que define la sucesión de Fibonacci es:

$$F(n) = \begin{cases} F(n-1) + F(n-2), & n \geq 2; \\ 0, & n = 0; \\ 1, & n = 1. \end{cases} \quad (4.1)$$

Esta definición recurrente es implementada en el Algoritmo 6, que permite calcular el n -ésimo número de la sucesión. Note cómo el **if** de

Algoritmo 6: FIB(n)

```

1 if  $n \leq 1$  then
2   | return  $n$ 
3 else
4   | return FIB( $n-1$ ) + FIB( $n-2$ )
5 end

```

la línea 1 chequea ambos casos base usando una única condición.

Un concepto relacionado a los algoritmos recursivos es el *árbol de recursión*, que muestra gráficamente cómo se efectúa el cómputo para una entrada dada. Por ejemplo, la Figura 4.4 muestra el árbol de recursión para el cómputo de FIB(5) usando el Algoritmo 6. La raíz del árbol

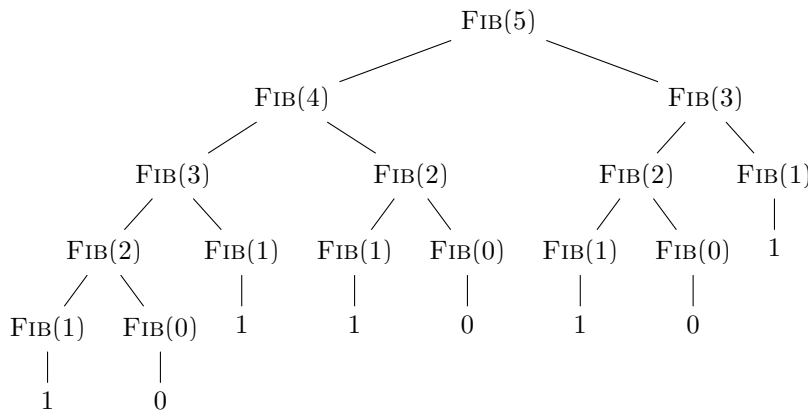


Figura 4.4: Árbol de recursión para el cálculo de FIB(5), usando el Algoritmo 6.

corresponde a FIB(5), es decir, al algoritmo FIB aplicado sobre el problema original. Para computar FIB(5), antes se deben resolver FIB(4) y FIB(3), por lo tanto la raíz tiene dos hijos correspondientes a las respectivas invocaciones recursivas del algoritmo. Así, cada nodo expande las ramas necesarias para resolver la subinstancia correspondiente del problema y por lo tanto el árbol completo muestra el cómputo necesario para resolver la instancia original del problema. Los nodos externos del árbol corresponden a los casos base. En este caso, note que 5 nodos externos corresponden al caso base de valor 1. Por la forma en que esos valores son usado para calcular los números de Fibonacci, esto significa que $FIB(5) = 5$. Note también que la cantidad de nodos del árbol de recursión —para una entrada dada— está en estrechamente relacionada con el tiempo de ejecución del algoritmo: más nodos signi-

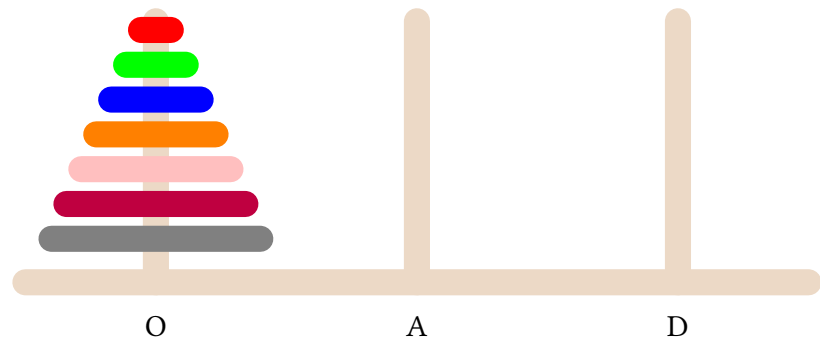


Figura 4.5: Configuración inicial para las Torres de Hanoi de 7 discos.

fica más tiempo. Esto es usado a menudo como ayuda en el análisis de algoritmos recursivos, como veremos más adelante.

Intuitivamente

¿Cuál es el tiempo de ejecución de $\text{FIB}(n)$ si se usa el Algoritmo 6?

Las Torres de Hanoi y el Fin del Mundo

Las Torres de Hanoi son un puzzle creado por el matemático francés Édouard Lucas d'Amiens (1842–1891) en el año 1883. Según la leyenda, en el instante en que el mundo fue creado, se les entregó a los monjes del gran templo de Benarés, India, un tablero con 3 agujas verticales (etiquetadas O, A y D, de izquierda a derecha) y 64 discos de oro. Los discos son todos de distinto tamaño y con un orificio en el centro, inicialmente colocados en la aguja O —es decir, la aguja izquierda. En dicha aguja inicial, los discos están ordenados por tamaño: el mayor en la base, el menor en la cima. La tarea de los monjes es mover, sin descanso, los 64 discos desde la aguja O a la aguja D, usando la aguja central A como auxiliar para el proceso y siguiendo estas reglas:

- Sólo se puede mover un disco a la vez, desde una aguja a otra, colocándolo en el tope de la pila de la aguja destino.
- Sólo se permite apilar un disco sobre otro de mayor tamaño. Así, las pilas de discos siempre estarán ordenadas por tamaño decreciente desde la base a la cima.
- Sólo se puede mover el disco que está en el tope de una pila de discos.

Según la leyenda, una vez que todos los discos estén en la aguja D, ocurrirá el fin del mundo. ¿Cuánto tiempo nos quedará? Después de todo, 64 discos no parece una gran cantidad...

En general, se puede jugar con la cantidad de discos que uno desee. La Figura 4.5 muestra la configuración inicial para las Torres de Hanoi con 7 discos. El pensamiento recursivo será clave para resolver este puzzle.

Spoiler alert...

A continuación se mostrará la solución al juego. Antes, trate de entenderlo usted mismo.

Para resolver el puzzle con $n \geq 1$ discos de forma recursiva, estudiemos primero el caso base: un tablero con 1 disco en la aguja O. Dicho caso se resuelve moviendo $O \rightarrow D$. Luego, asumimos que ya sabemos resolver el puzzle para $n - 1$ discos, lo que será nuestra hipótesis inductiva. Mostramos ahora cómo resolver para $n > 1$ discos. Note que para mover los n discos desde O hasta D, el primer obstáculo son los $n - 1$ discos que están sobre el mayor disco de la aguja O. En el resultado final, ese disco debe estar en la base de la aguja D. Sin embargo, ya sabemos (por hipótesis inductiva) cómo resolver el problema de trasladar $n - 1$ discos desde una aguja a otra. En este caso, queremos mover $n - 1$ discos desde O a A, para dejar libre el camino al disco más grande. Eso significa resolver una instancia de Torres de Hanoi con $n - 1$ discos desde O a A, usando la aguja D como auxiliar. Luego de esto, podemos mover el disco más grande desde O a D. Finalmente, debemos mover (siguiendo las reglas) los $n - 1$ discos que están en A a D, usando a O como auxiliar, algo que ya sabemos hacer por hipótesis inductiva.

Llamemos $H_n(O, A, D)$ al algoritmo que resuelve las Torres de Hanoi para n discos, desde la aguja O a la D, usando A como auxiliar¹¹. Asumiendo que el caso base $H_0(O, A, D)$ se resuelve trivialmente, $H_n(O, A, D)$, para $n \geq 1$, puede resolverse recursivamente de la siguiente manera:

1. Resolver $H_{n-1}(O, D, A)$.
2. Mover el disco desde O a D.
3. Resolver $H_{n-1}(A, O, D)$.

Eso es todo lo que necesitamos para resolver el problema. La Figura 4.6 muestra el árbol de recursión para una instancia de 3 discos. Observe cómo las agujas cambian de rol a medida que se avanza en la recursión. Los movimientos de discos entre agujas corresponden a las hojas del árbol. Por ejemplo, la hoja de más a la izquierda corresponde a mover el disco más pequeño desde O a D. Eso significa que para resolver el juego con 3 discos, se necesitan 7 movimientos usando este algoritmo. El Algoritmo 7 muestra el proceso descrito. Se asume que las agujas

11: Graham, Knuth, y Patashnik llaman a esto “name and conquer”, traducido como “ponle nombre y conquista”, ya que facilita pensar en los subproblemas recursivos.

Algoritmo 7: HANOI(O, A, D, n)

```

1 if n = 0 then
2   | return
3 else
4   | HANOI(O, D, A, n - 1)
5   | Imprimir O → D
6   | HANOI(A, O, D, n - 1)
7 end

```

han sido enumeradas de alguna manera, usando enteros. Por ejemplo, $D = 1$, $T = 2$, y $D = 3$. El algoritmo imprime los movimiento que deber ejecutarse para resolver el juego.

Para pensar...

¿Cuántos movimientos necesitará el problema original de 64 discos que el creador le dio a los monjes? ¿Si cada movimiento se hace en un segundo, cuánto tiempo toma resolver eso?

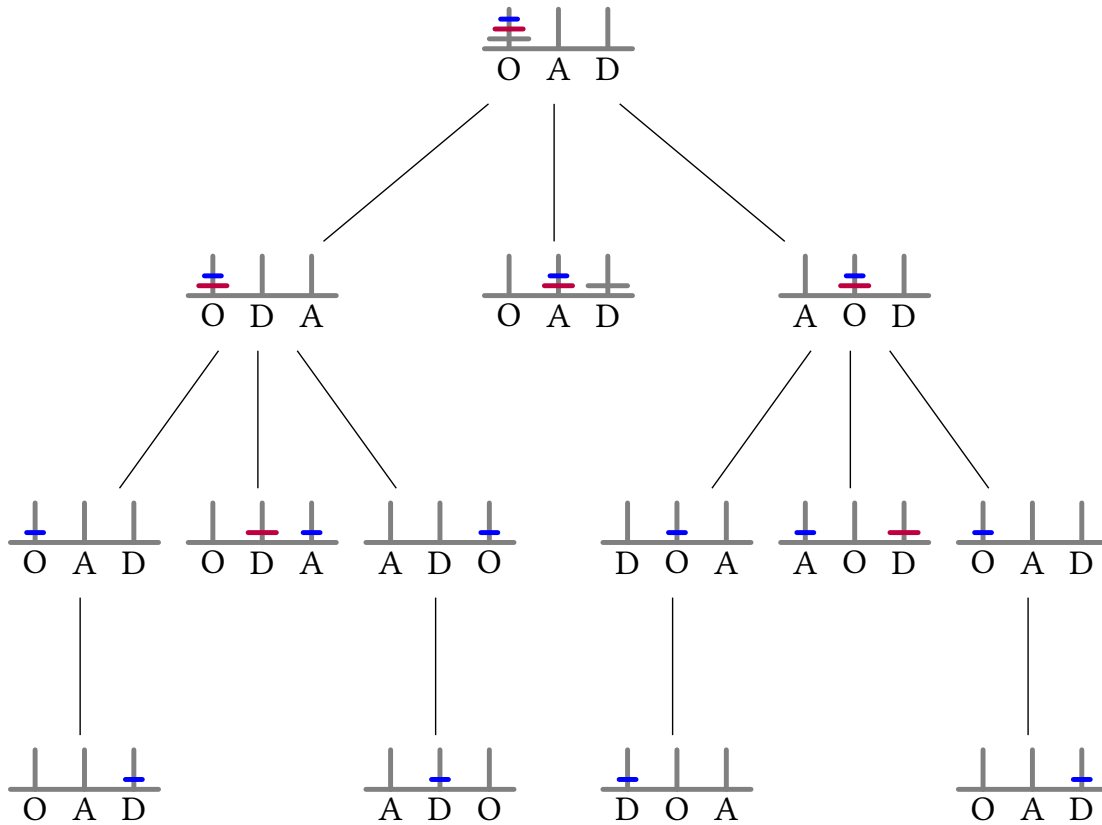


Figura 4.6: Árbol de recursión para resolver torres de hanoi con 3 discos. Sólo las hojas del árbol corresponden al movimiento de discos.

4.5. Resolución de Ecuaciones de Recurrencia

Una ecuación de recurrencia permite definir una función en términos de sí misma, tal como lo vimos anteriormente en las Ecuaciones (??) y (4.1). La misma debe incluir un caso base ¹², y la definición recurrente. Este tipo de ecuaciones aparecen naturalmente al analizar algoritmos recursivos. Por ejemplo, el tiempo de ejecución del algoritmo expresado en la Ecuación (??) e implementado en el Algoritmo 5 puede modelarse con la ecuación de recurrencia:

$$T_{\text{fact}}(n) = \begin{cases} T_{\text{fact}}(n-1) + 1, & n > 0; \\ 1, & n = 0. \end{cases} \quad (4.2)$$

Esto es, el tiempo $T_{\text{fact}}(n)$ que toma resolver `FACTORIAL(n)` es el tiempo que toma resolver `FACTORIAL(n-1)` (es decir, $T_{\text{fact}}(n-1)$) más el tiempo que toma multiplicar por n (asumiendo que esa multiplicación toma 1 unidad de tiempo). Para el caso base, sólo debemos retornar 1, lo que toma 1 unidad de tiempo. Aunque en general es bastante sencillo obtener estas ecuaciones de recurrencia para el tiempo de ejecución de un algoritmo recursivo, y la función está bien definida en términos matemáticos, la definición recurrente de la función no nos da demasiadas pistas acerca de qué función matemática está siendo modelada por esa ecuación. Eso impide, entre otras cosas, estudiar

12: Alternativamente, casos de borde, condiciones iniciales, o casos conocidos, los llamaremos indistintamente en adelante

su velocidad de crecimiento. En esta sección estudiaremos formas de resolver ecuaciones de recurrencia, para obtener su forma cerrada (o no recurrente).

De manera similar, el tiempo de ejecución $T_{\text{fib}}(n)$ asociado a la Ecuación (4.1) para calcular el n -ésimo número de Fibonacci es:

$$T_{\text{fib}}(n) = \begin{cases} T_{\text{fib}}(n-1) + T_{\text{fib}}(n-2) + 1, & n \geq 2; \\ 1, & n = 0; \\ 1, & n = 1. \end{cases}$$

Note la sutil diferencia entre ésta y la Ecuación (4.1).

En general, una ecuación de recurrencia T define una secuencia infinita de números enteros

$$\langle a_0, a_1, \dots, a_i, \dots \rangle$$

tal que

$$a_0 = T(0), a_1 = T(1), \dots, a_i = T(i), \dots$$

La ecuación del ejemplo anterior corresponde a la secuencia

$$\langle 1, 1, 3, 5, 9, 15, 25, 41, 67, \dots \rangle.$$

Ecuaciones Lineales Homogéneas

Tienen la forma

$$a_0 T(n) + a_1 T(n-1) + \dots + a_k T(n-k) = 0,$$

en donde $a_0, \dots, a_k \in \mathbb{R}$ son constantes, $a_k \neq 0$, y $1 \leq k \leq n$.

Para resolverlas se hace el cambio:

$$T(n-i) = x^{k-i},$$

para $i = 0, \dots, k$, obteniendo la *ecuación característica* asociada:

$$a_0 x^k + a_1 x^{k-1} + \dots + a_k = 0.$$

Sean r_1, \dots, r_k las raíces del polinomio. Dependiendo de su multiplicidad, tenemos dos casos.

Caso 1: Raíces Distintas. Si todas las raíces de la ecuación característica son distintas, la solución de la ecuación de recurrencia está dada por:

$$T(n) = c_1 r_1^n + c_2 r_2^n + \dots + c_k r_k^n, \quad (4.3)$$

en donde los coeficientes c_i se determinan planteando un sistema de ecuaciones a partir de las condiciones iniciales de la definición.

13: Note que φ_+ es en realidad la constante $\varphi = 1,618033989\dots$, conocida como razón áurea, o golden ratio, para la que se han probado una gran cantidad de propiedades.

Ejemplo 4.5.1 Encontremos la forma cerrada de la Ecuación (4.1), que nos dará la función no recurrente para calcular los números de Fibonacci. Note que la ecuación puede escribirse como:

$$F(n) - F(n-1) - F(n-2) = 0,$$

por lo que corresponde a una ecuación lineal homogénea con ecuación característica: $x^2 - x - 1 = 0$. Este polinomio cuadrático tiene dos raíces distintas, $r_1 = \frac{1+\sqrt{5}}{2}$ y $r_2 = \frac{1-\sqrt{5}}{2}$. Por simplicidad, llamaremos $\varphi_+ = \frac{1+\sqrt{5}}{2}$ y $\varphi_- = \frac{1-\sqrt{5}}{2}$ a esas raíces¹³. De acuerdo a la Ecuación (4.3), la solución cerrada para $F(n)$ es:

$$F(n) = c_1 \varphi_+^n + c_2 \varphi_-^n.$$

Para completar la definición de $F(n)$, encontramos los valores de c_1 y c_2 , usando las condiciones iniciales para plantear el siguiente sistema de ecuaciones:

$$\begin{aligned} F(0) &= c_1 \varphi_+^0 + c_2 \varphi_-^0 = c_1 + c_2 = 0. \\ F(1) &= c_1 \varphi_+ + c_2 \varphi_- = 1. \end{aligned}$$

Resolviendo obtenemos $c_1 = \frac{1}{\sqrt{5}}$ y $c_2 = -\frac{1}{\sqrt{5}}$. Esto significa que la forma cerrada que buscamos es:

$$F(n) = \frac{1}{\sqrt{5}} \varphi_+^n - \frac{1}{\sqrt{5}} \varphi_-^n.$$

Note que esto nos da una forma cerrada de calcular los números de Fibonacci, sin necesidad de recursión. Sólo hay que tener en cuenta que las raíces cuadradas pueden tener una cantidad infinita de dígitos decimales, lo que necesitaría una cantidad infinita de bits para ser representada de forma exacta en un computador. Entonces, en general esta fórmula nos da una aproximación a los números de Fibonacci.

Caso 2: Raíces con Multiplicidad Mayor a 1. Si r_1, r_2, \dots, r_s ($s \leq k$) son las raíces distintas de la ecuación característica de una ecuación de recurrencia homogénea lineal, cada una con multiplicidad m_1, m_2, \dots, m_s , entonces la ecuación característica puede expresarse como

$$(x - r_1)^{m_1} \times (x - r_2)^{m_2} \times \dots \times (x - r_s)^{m_s} = 0. \quad (4.4)$$

La solución a la ecuación de recurrencia es

$$T(n) = r_1^n \sum_{i=1}^{m_1} c_{1i} n^{i-1} + \dots + r_s^n \sum_{i=1}^{m_s} c_{si} n^{i-1}.$$

Nuevamente los coeficientes c_{ij} se obtienen mediante las condiciones iniciales.

Ejemplo 4.5.2 Resolvamos la ecuación de recurrencia definida como:

$$T(n) = \begin{cases} 5T(n-1) - 8T(n-2) + 4T(n-3), & n \geq 3 \\ 0, & n = 0 \\ 1, & n = 1 \\ 2, & n = 2 \end{cases}$$

Note que la secuencia infinita de números correspondiente es:

$$\langle 0, 1, 2, 2, -2, -18, \dots \rangle.$$

La parte recurrente de la misma puede escribirse como:

$$T(n) - 5T(n-1) + 8T(n-2) - 4T(n-3) = 0,$$

por lo que es una ecuación de recurrencia lineal homogénea, con ecuación característica:

$$x^3 - 5x^2 + 8x - 4 = 0,$$

la cual tiene raíces $r_1 = 2$ (con multiplicidad $m_1 = 2$) y $r_2 = 1$ (con multiplicidad $m_2 = 1$). Por lo tanto, la ecuación característica puede escribirse como:

$$(x - 2)^2 \times (x - 1) = 0,$$

y la ecuación de recurrencia tiene la siguiente solución:

$$T(n) = r_1^n(c_{11} + c_{12}n) + r_2^n c_{21} = 2^n(c_{11} + c_{12}n) + c_{21}.$$

Usando las 3 condiciones iniciales sobre esta ecuación, planteamos el siguiente sistema de ecuaciones:

$$\begin{aligned} T(0) &= c_{11} + c_{21} = 0, \\ T(1) &= 2c_{11} + 2c_{12} + c_{21} = 1, \\ T(2) &= 4c_{11} + 8c_{12} + c_{21} = 2. \end{aligned}$$

Resolviendo tenemos que $c_{11} = 2$, $c_{12} = -1/2$, y $c_{21} = -2$, por lo que el resultado final es

$$T(n) = 2^n \left(2 - \frac{1}{2}n \right) - 2 = 2^{n+1} - n2^{n-1} - 2.$$

Se recomienda chequear que, para los primeros valores de n , esta función genera la misma secuencia de números que la ecuación de recurrencia.

Ecuaciones de Recurrencia Lineales No Homogéneas

Tienen la forma

$$a_0T(n) + a_1T(n-1) + \dots + a_kT(n-k) = b^n p(n). \quad (4.5)$$

En donde $a_0, \dots, a_k, b \in \mathbb{R}$ son constantes, $a_i \neq 0$, $1 \leq k \leq n$, $p(n)$ es un polinomio en n de grado d . Para esta ecuación de recurrencia se

define la ecuación característica:

$$(a_0x^k + a_1x^{k-1} + \dots + a_k)(x - b)^{d+1} = 0.$$

Se procede de forma similar a los casos anteriores, encontrando las raíces r_1, \dots, r_s del polinomio $a_0x^k + a_1x^{k-1} + \dots + a_k$.

Ejemplo 4.5.3 Determinemos la cantidad de movimientos realizador por el algoritmo que resuelve las Torres de Hanoi (Algoritmo 7). Notar que el caso base realiza $T(0) = 0$ movimientos. Para $n > 0$, el algoritmo resuelve 2 problemas de $n - 1$ discos, y luego realiza 1 movimiento. Esto se puede escribir formalmente de la siguiente manera:

$$T(n) = \begin{cases} 2T(n-1) + 1, & n \geq 1, \\ 0, & n = 0. \end{cases}$$

Note que la secuencia de enteros asociada a esta ecuación es:

$$\langle 0, 1, 3, 7, 15, 31, 63, 127, 255, 511, \dots \rangle.$$

Esto sugiere una cantidad exponencial de movimientos para resolver el problema. La ecuación de recurrencia puede ser escrita de la siguiente manera:

$$T(n) - 2T(n-1) = 1,$$

en donde $p(n) = 1$, de grado $d = 0$, y $b^n = 1$, por lo tanto $b = 1$ (de acuerdo a la Ecuación (4.5)). Entonces, la ecuación característica puede escribirse como

$$(x - 2)(x - 1) = 0,$$

la cual tiene raíces $r_1 = 2$ (de multiplicidad $m_1 = 1$) y $r_2 = 1$ (de multiplicidad $m_2 = 1$). Por lo tanto,

$$T(n) = c_1 2^n + c_2.$$

Usando las condiciones iniciales, planteamos el sistema de ecuaciones:

$$\begin{aligned} T(0) &= c_1 + c_2 = 0, \\ T(1) &= 2c_1 + c_2 = 1. \end{aligned}$$

Note que para la segunda ecuación, no tenemos un caso base en la definición que nos diga cuánto vale $T(1)$. Por lo tanto, hay que usar la definición recurrente para obtener el/los valor/es necesario/s. Resolviendo ese sistema, tenemos que $c_1 = 1$ y $c_2 = -1$, por lo que

$$T(n) = 2^n - 1.$$

Tal como habíamos sugerido, se necesitan una cantidad exponencial de movimientos para resolver las Torres de Hanoi.

Para el problema del fin del mundo, con $n = 64$, hacen falta

$$2^{64} - 1 = 1,8446744073709551615 \times 10^{19} \text{ movimientos.}$$

Asumiendo que los monjes realizan un movimiento por segundo, sin detenerse, van a necesitar ~ 584.942 millones de años para terminar el juego. Dado que la edad estimada del universo es ~ 14.000 millones de años, aún queda bastante tiempo para llegar al fin del mundo.

Cambio de Variable. Considere una ecuación de recurrencia como la siguiente:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + n, & n > 1; \\ 1, & n = 1. \end{cases} \quad (4.6)$$

Como veremos más adelante, esta ecuación está relacionada a un algoritmo de ordenamiento muy conocido. Examinando la ecuación, a primera vista no parece tener la forma adecuada para clasificarla como lineal homogénea, ni como lineal no homogénea. Sin embargo, si asumimos $n = 2^k$ (y, por lo tanto, $k = \lg n$), para $k \geq 1$ (es decir, asumimos que n es una potencia de 2), tenemos que

$$T(2^k) = 2T\left(\frac{2^k}{2}\right) + 2^k = 2T(2^{k-1}) + 2^k.$$

Si, además hacemos el cambio de variable $t(k) = T(2^k)$, la ecuación nos queda:

$$t(k) = 2t(k-1) + 2^k,$$

la cual sí es una ecuación de recurrencia lineal no homogénea:

$$t(k) - 2t(k-1) = 2^k,$$

con polinomio $p(k) = 1$, de grado $d = 0$, y $b^k = 2^k$, por lo que $b = 2$. Por lo tanto tenemos ecuación característica:

$$(x-2)(x-2) = (x-2)^2 = 0,$$

la que tiene una única raíz $r_1 = 2$ de multiplicidad $m_1 = 2$. Por lo tanto, la solución es:

$$t(k) = r_1^k(c_{11} + c_{12}k) = c_{11}2^k + c_{12}k2^k.$$

Deshaciendo el cambio, tenemos que:

$$T(2^k) = c_{11}2^k + c_{12}k2^k,$$

y como $n = 2^k$ y por ende $k = \lg n$, tenemos

$$T(n) = c_{11}n + c_{12}n \lg n.$$

Resolviendo, se llega a $c_{11} = 1$ y $c_{12} = 1$, por lo que

$$T(n) = n \lg n + n.$$

El desarrollo se deja como ejercicio.

Ejercicios

1. Obtener el tiempo de ejecución de los siguientes algoritmos recursivos:

```
a) int recursiva(int n) {
    int i, y;

    if (n <= 1) return n+1;
    else {
        for (i = 0, y = 0; i < n; i++)
            y = 2 * y;
        return y + recursiva(n/2) + recursiva(n/2);
    }
}

b) int log2(int n) {
    if (n == 1) return 0;
    else
        return 1 + log2(n/2);
}
```

2. Considere el siguiente algoritmo recursivo:

```
int Misterio(int *A, int n) {
    if (n == 1) return A[0];
    else {
        int temp = Misterio(A, n-1);
        if (temp <= A[n-1]) return temp;
        else return A[n-1];
    }
}
```

- a) ¿Qué es lo que computa este algoritmo?
 - b) Establecer una ecuación de recurrencia para modelar el tiempo de ejecución del algoritmo, y resolverla.
3. Resolver las siguientes ecuaciones de recurrencia y acotarlas asintóticamente.
 - a) $T(n) = 3T(n-1) + 4T(n-2)$ si $n > 1$; $T(0) = 0$; $T(1) = 1$.
 - b) $T(n) = 2T(n-1) - (n+5)3^n$ si $n > 0$; $T(0) = 0$.
 - c) $T(n) = 4T(n/2) + n^2$ si $n > 4$, n potencia de 2; $T(1) = 1$; $T(2) = 8$.
 - d) $T(n) = 2T(n/2) + n \lg n$ si $n > 1$, n potencia de 2.
 - e) $T(n) = 3T(n/2) + 5n + 3$ si $n > 1$, n potencia de 2.
 - f) $T(n) = 2T(n/2) + \lg n$ si $n > 1$, n potencia de 2.
 - g) $T(n) = 2T(n^{1/2}) + \lg n$ con $n = 2^{2^k}$, $k \geq 1$; $T(2) = 1$.
 - h) $T(n) = 5T(n/2) + (n \lg n)^2$ si $n > 1$, n potencia de 2; $T(1) = 1$.
 - i) $T(n) = 2T(n/2) + n \lg n$ si $n > 1$, n potencia de 2.
 - j) $T(n) = T(n-1) + 2T(n-2) - 2T(n-3)$ si $n > 2$; $T(n) = 9n^2 - 15n + 106$ si $n = 0, 1, 2$.
 - k) $T(n) = \frac{3}{2}T(n/2) - \frac{1}{2}T(n/4) - \frac{1}{n}$ si $n > 2$; $T(1) = 1$; $T(2) = \frac{3}{2}$.
 - l) $T(n) = 2T(n/4) + n^{1/2}$ si $n > 4$, n potencia de 4.

- m) $T(n) = 4T(n/3) + n^2$ si $n > 3$, n potencia de 3.
- n) $T(n) = T(n-1) + 5$ si $n > 1$; $T(1) = 0$.
- \hat{n}) $T(n) = 3T(n-1)$ si $n > 1$; $T(1) = 4$.
- o) $T(n) = T(n-1) + n$ si $n > 0$; $T(0) = 0$.
- p) $T(n) = T(n/2) + n$ si $n > 1$; $T(1) = 1$ (resolver asumiendo $n = 2^k$).
- q) $T(n) = T(n/3) + 1$ si $n > 1$; $T(1) = 1$ (resolver asumiendo $n = 3^k$).

4. Resolver la ecuación $T(n) = \frac{1}{n} \sum_{i=0}^{n-1} T(i) + cn$, con $T(0) = 0$.
5. Dado el siguiente algoritmo recursivo:

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

Establezca una ecuación de recurrencia para determinar qué función de n computa el algoritmo. Resuelva la ecuación usando el método de la ecuación característica (visto en clases) para encontrar dicha función.

6. Encuentre la forma cerrada de la ecuación de recurrencia:

$$T(n) = T\left(\frac{n}{2}\right) + n \log_2 n,$$

para n una potencia de 2, y con condiciones iniciales $T(1) = 1$, $T(2) = 3$.

7. Encuentre la forma cerrada de la ecuación de recurrencia:

$$T(n) = 2T(n-1) + 3^n(n+1),$$

con condición inicial $T(0) = 0$.

8. Encuentre la forma cerrada de la ecuación de recurrencia $T(n) = T(n^{1/2}) + 1$, para $n > 1$, y $T(1) = 1$.

Estructuras de Datos Fundamentales

5

Una *estructura de datos* es una manera particular de organizar datos en la memoria de un computador, de manera que puedan ser usados y procesados eficientemente mediante un algoritmo. En el desarrollo de este trabajo, las estructuras de datos serán fundamentales para que los algoritmos manipulen los datos de forma eficiente ¹. Revisamos en este capítulo las estructuras de datos fundamentales, que serán necesarias para el correcto entendimiento de varios algoritmos más adelante.

1: No sólo los datos de entrada, sino que también lo de salida.

5.1. Pilas

Aunque es muy simple, el concepto de pila ² es fundamental para el diseño de algoritmos —y la computación en general. Una pila en la vida real representa una colección de objetos que han sido colocados uno encima del otro sobre alguna base. Por ejemplo, una pila de libros, de hojas de papel, de sillas, o de platos. La única forma de interactuar con una pila es a través de su cima (o tope): (1) sólo podemos observar el elemento que está en la cima, ya que el resto tiene elementos encima; (2) para agregar un nuevo elemento a la pila, lo debemos apilar sobre la cima actual; y (3) sólo se puede eliminar (o desapilar) el elemento que está actualmente en la cima. Cualquier intento de interacción con un elemento que no sea el que está en la cima, llevará al derrumbe de la pila.

2: *Stack*, en inglés.

Operaciones

Una pila P implementa las siguientes operaciones:

- P.PUSH(x):** agrega el elemento x a la pila P , colocándolo en la cima de la misma;
- P.POP():** extrae el elemento que está en la cima de la pila P , asumiendo que la pila tiene elementos;
- P.CIMA():** permite consultar el elemento que se encuentra en la cima de la pila P ;
- P.TAM():** devuelve la cantidad de elementos almacenados en la pila P ; y
- P.INICIAR():** inicializa la pila como vacía.

Ejemplo 5.1.1 La Figura 5.1 (izquierda) muestra una pila inicialmente vacía, y sobre la que anteriormente se realizaron las operaciones:

- P.PUSH(A),
- P.PUSH(B),
- P.PUSH(C),

y sobre la que se está realizando la operación P.PUSH(D) para obtener la pila que está a la derecha de la figura.

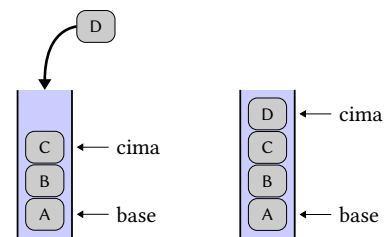


Figura 5.1: Ejemplo de operaciones sobre una pila.

Las pilas memorizan el orden inverso en que sucedió una serie de eventos. Por ejemplo, al ver la secuencia D, C, B, A en la pila que está a la derecha en la Figura 5.1 (leyendo desde la cima a la base), sabemos que los elementos llegaron en orden inverso: A, B, C, D. Eso corresponde a una política conocida como *LIFO* (*Last In First Out*, o *último en llegar, primero en salir*), la cual es fundamental para aplicaciones como la operación “deshacer” en editores de texto, “volver atrás” en browsers de la web, y la pila de ejecución de su lenguaje de programación favorito, por nombrar algunas.

Aplicación: Evaluación de Expresiones en Notación Posfija

La notación natural para expresiones aritméticas es la *infija*, en donde los operadores se colocan entre los operandos sobre los que se aplican. Por ejemplo, $3 \times 5 + x$ es una expresión en notación infija. A pesar de su claridad para la lectura humana, esta notación necesita definir la precedencia y asociatividad de los distintos operadores para funcionar correctamente. Además, se deben usar paréntesis para forzar el cambio en esas reglas. En el ejemplo anterior, $3 \times 5 + x$ usualmente es equivalente a $(3 \times 5) + x$, ya que usualmente la multiplicación tiene más precedencia que la suma. Si se quiere realizar primero la suma, se debe escribir explícitamente $3 \times (5 + x)$. Todo esto complica la evaluación automática de expresiones en esa notación.

Un ejemplo típico de aplicación de pilas es la evaluación de expresiones en *notación posfija* —o *notación polaca inversa*. En esta notación, los operadores aparecen inmediatamente después de sus operandos. Por ejemplo, la expresión infija $3 \times (5 + x)$ se escribe $3\ 5\ x\ +\ \times$ en notación posfija. Esta notación fue creada para simplificar la evaluación de expresiones: no se necesita definir precedencia ni asociatividad de operadores, ni el uso de paréntesis. Además, la evaluación se puede hacer de forma simple usando una pila como mostramos a continuación.

La idea es que la secuencia posfija se procesa de izquierda a derecha, considerando los siguientes casos:

- Si el elemento actual es un operando, se coloca en una pila.
- Si el elemento actual es un operador, los dos elementos (asumiendo operaciones binarias) que están en la cima de la pila se reemplazan por el correspondiente resultado de aplicarles el operador.

En el segundo caso, si hay menos de dos elementos en la pila, la expresión está mal formada y se debe informar el error. Luego de procesar toda la expresión —y si no se han detectado errores anteriormente— la pila debe contener un único elemento: el resultado de la evaluación. Si la pila tiene más de un elemento, la expresión está mal formada y se debe informar el error.

Ejemplo 5.1.2 La expresión $3\ 5\ 2\ +\ \times\ 8\ +$ (que equivale a $3 \times (5 + 2) + 8$) se evalúa tal como lo indica la Figura 5.2. Al finalizar el proceso, la pila tiene un único elemento, por lo que la expresión está bien formada y el valor en la cima de la pila es el resultado de la evaluación —en este caso, 29.

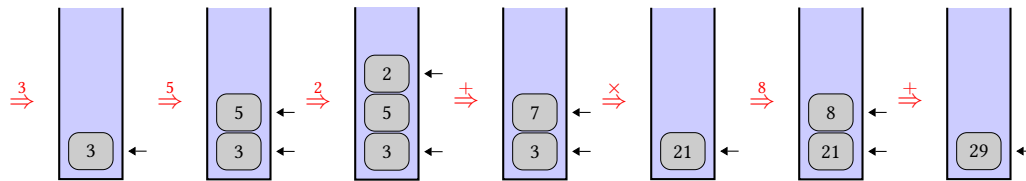


Figura 5.2: Evaluación de la expresión posfija $3\ 5\ 2\ +\ \times\ 8\ +$ usando una pila. El resultado de la evaluación es 29, como puede verse en la cima de la pila al finalizar la ejecución.

En algunas calculadoras científicas comerciales antiguas, las expresiones debían introducirse directamente en notación posfija. Eso permitía una lógica interna mucho más simple y eficiente, aunque cargaba al usuario la manipulación de la expresión posfija. Actualmente las expresiones se introducen en notación infija, la que es luego traducida a notación posfija por la calculadora para una evaluación más eficiente. Otro ejemplo son los compiladores de los lenguajes de programación, que traducen las expresiones a notación posfija para generar código de forma más simple. Por ejemplo, una expresión como $a + b$ genera código del tipo: (1) mover el valor de a al registro R_1 del procesador; (2) mover el valor de b al registro R_2 del procesador; (3) sumar R_1 y R_2 . Eso es equivalente a la expresión posfija $a\ b\ +$.

Aplicación: Chequeo de Anidamiento de Paréntesis

Otro ejemplo clásico de uso de pilas es en el chequeo de anidamiento de paréntesis: dada una secuencia que contiene diversos tipos de paréntesis (por ejemplo ' $()$ ', ' $\{\}$ ', ' $[]$ ', ' $\langle\rangle$ ', '**begin-end**', etc.), hay que determinar si la misma está bien formada o no. Esto es, si cada paréntesis que abre tiene su correspondiente paréntesis que cierra del mismo tipo, anidados correctamente. Esta es una tarea típica de los analizadores sintácticos de compiladores de lenguajes de programación. La idea es muy simple: se lee la secuencia de izquierda a derecha, y los paréntesis de apertura se van colocando en una pila. Cuando se encuentra un paréntesis de cierre, se debe chequear que el que está actualmente en la cima de la pila coincida en tipo de paréntesis. Si es así, se elimina el paréntesis de la cima de la pila, y se sigue procesando la entrada. Si, por el contrario, los tipos no coinciden, significa que la expresión de entrada no está correctamente anidada. Al finalizar el procesamiento de la expresión, la pila debe quedar vacía, ya que de otra manera la expresión estaba mal formada.

Implementación de Pilas

Una pila puede implementarse de forma muy simple y eficiente utilizando un arreglo —o *buffer*— $A[1..m]$ como estructura base para almacenar los elementos, para algún valor predefinido $m \geq 1$. El elemento que está en la base de la pila es $A[1]$, mientras que el elemento de la cima se encuentra en $A[n]$, siendo $n \leq m$ una variable que indica la cantidad actual de elementos que tiene la pila. De esta forma, la operación INICIAR() sólo hace $n \leftarrow 0$, mientras que PUSH(x) simplemente hace $n \leftarrow n + 1$ y luego $A[n] \leftarrow x$ ³. Por otro lado, POP() sólo necesita hacer $n \leftarrow n - 1$. Finalmente, CIMA() sólo necesita devolver $A[n]$. De esta forma, todas las operaciones toman tiempo $O(1)$.

3: Cuando $n = m$, la pila ha consumido todo el espacio disponible en el buffer. Si en esa condición se intenta hacer PUSH(), ocurre un *rebalse de la pila*, o *stack overflow* en inglés.

Más allá de la simpleza y eficiencia de esta implementación, hay que tener especial cuidado con la elección del valor m , el tamaño máximo permitido para la pila. En muchas situaciones no es posible determinar este valor *a priori*, y elegir un valor demasiado grande puede desperdiciar mucha memoria. Una alternativa en esos casos es permitir que la pila tenga más de m elementos, haciendo crecer el buffer A antes que ocurra un rebalse. Típicamente, eso implica pedir memoria para un nuevo buffer $A'[1..m']$ tal que $m' \geq m$, luego copiar los elementos de A a A' , y finalmente liberar la memoria usada por A . Al ser éste un proceso muy costoso (en particular por el proceso de copia de los elementos entre arreglos), m' debe elegirse de forma tal que el llenado de la pila no ocurra muy frecuentemente.

Éste es el conocido problema del arreglo extensible, que es fundamental para mantener un buffer dinámico como en este caso —estudiaremos este problema en detalle en la Sección 6.2. Una estrategia bastante usada en la práctica ⁴ consiste en definir $m' = 2m$. Es decir, se duplica el tamaño del buffer cada vez que rebalsa. Eso garantiza que desde el último rebalse se necesiten m inserciones en la pila para volver a llenar el buffer y tener que copiar $2m$ elementos a un nuevo buffer. Respecto al uso de espacio de esta estrategia, en el peor caso se desperdicia la mitad de las entradas del arreglo, lo que puede ser un problema.

4: E.g., por el container `std::vector<>` de C++, para implementar la operación `push_back()`.

5.2. Colas

Las estructuras de datos de tipo cola también son fundamentales en computación. La idea es similar al de las colas (o filas) de espera de atención por algún servicio —como, por ejemplo, un banco o supermercado. Una cola “memoriza” el orden de llegada de las personas al centro de atención. El primero que está en la fila es el primero que llegó, y va a ser el primero en ser atendido. Cada vez que llega una nueva persona, por su parte, se anexa al final de la fila en espera de atención. Este tipo de política es conocida como *FIFO* (por *First-In First-Out*, o *primero en llegar, primero en salir*).

A diferencia de las pilas —en donde uno interactúa con un único extremo, la cima— en las colas debemos interactuar con los dos extremos de la estructura: (1) sólo podemos observar el elemento que está en el frente de la cola, ya que el resto está cubierto por otros elementos; (2) un nuevo elemento se anexa al final de la cola, detrás de todos los restantes; y (3) sólo se puede eliminar el elemento que está actualmente en el frente de la cola —es lo mismo que ocurre con el primero de la fila en un banco luego de ser atendido y abandonar el banco.

Operaciones

Una cola Q implementa las siguientes operaciones:

Q.ENQUEUE(x): agrega el elemento x a la cola Q , detrás de los restantes elementos;

Q.DEQUEUE(): extrae el elemento que está al frente de la cola Q ;

Q.FRENTE(): devuelve el elemento que está al frente en la cola Q ;

Q.TAM(): devuelve la cantidad de elementos almacenados en la cola Q;
y
Q.INICIAR(): inicializa la cola como vacía.

Ejemplo 5.2.1 La Figura 5.3 muestra una cola Q inicialmente vacía sobre la que se realizaron las siguientes operaciones:

- Q.ENQUEUE(A),
- Q.ENQUEUE(B),
- Q.ENQUEUE(C),
- Q.ENQUEUE(D).
- Q.DEQUEUE().

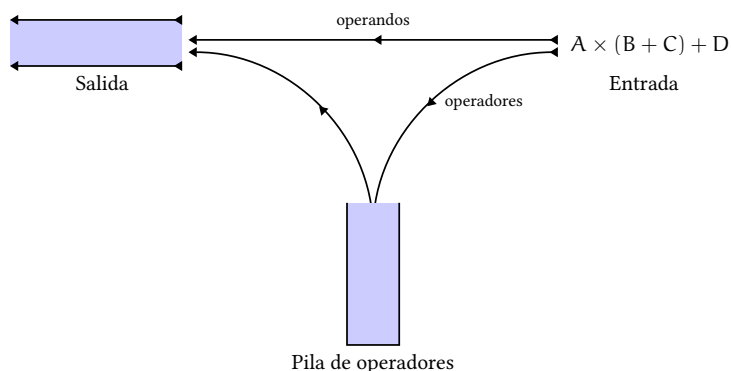
Como en la Figura 5.3, al dibujar una cola usaremos flechas en los bordes para indicar el sentido del movimiento de los elementos dentro de la misma.

Las colas son usadas frecuentemente en aplicaciones como sistemas operativos, por ejemplo para el administrador de la CPU, o para administrar dispositivos de entrada/salida —e.g., para administrar accesos al disco, o a una impresora, por nombrar sólo algunas.

Aplicación: El Algoritmo del Patio de Maniobras

El *algoritmo del patio de maniobras* —*shunting yard* en inglés— fue presentado por Edger Disjkstra en 1961, con la finalidad de traducir expresiones aritméticas en notación infija a la correspondiente notación posfija. Como lo mencionamos en un ejemplo anterior, la mayoría de las calculadoras necesitan realizar esta transformación para evaluar expresiones infijas eficientemente. El algoritmo maneja una pila de operadores P y una cola de salida Q —en donde será escrita la expresión posfija. El nombre del algoritmo viene de que su funcionamiento se parece al de un patio de maniobras de una estación de trenes.

La disposición de las estructuras de datos empleadas, así como los sentidos en que pueden moverse los elementos dentro del patio de maniobras es la siguiente:



Imagine que la expresión que queremos transformar es un tren, y cada uno de los componentes de la expresión —e.g., operadores, operandos, paréntesis— es un vagón del tren. Usaremos el patio de maniobras

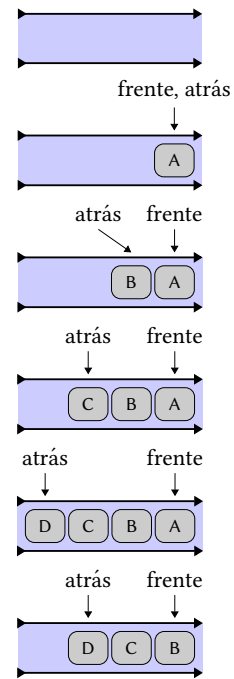


Figura 5.3: Operaciones sobre una cola.

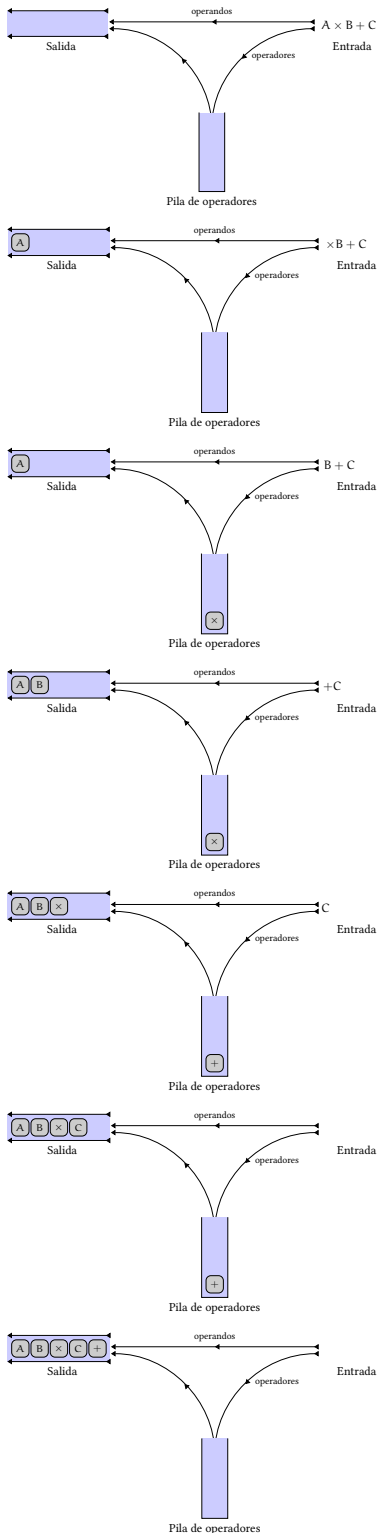


Figura 5.4: Aplicación del algoritmo de maniobras sobre la expresión infija $A \times B + C$.

para reorganizar los vagones y producir una salida que corresponda a la notación posfija de la expresión de entrada.

Sea $E[1..n]$ la expresión aritmética de entrada en notación infija, compuesta de n elementos. Asumimos que se ha definido la precedencia de operadores para la expresión de entrada y, para simplificar, todos los operadores tienen asociatividad de izquierda a derecha. El algoritmo procede como a continuación:

- Para cada elemento $E[i]$, $i = 1, \dots, n$, hacer:
 1. Si $E[i]$ es un operando, hacer $Q.ENQUEUE(E[i])$.
 2. Sino, si $E[i] = '('$, hacer $P.PUSH(E[i])$.
 3. Sino, si $E[i]$ es un operador, hacer:
 - a) Si la precedencia de $E[i]$ es mayor que la precedencia del elemento $P.CIMA()$, hacer $P.PUSH(E[i])$.
 - b) Sino, mientras la precedencia del elemento que está en la cima de P es mayor que la de $E[i]$, hacer $Q.ENQUEUE(P.CIMA())$ y luego $P.POP()$. Finalmente, hacer $P.PUSH(E[i])$.
 4. Sino, si $E[i]$ es $)$, mientras $P.CIMA() \neq '('$, hacer $Q.ENQUEUE(P.CIMA())$ y luego $P.POP()$. Finalmente, hacer $P.POP()$ para eliminar $'('$.
- Una vez consumida toda la expresión como explica el paso anterior, se deben traspasar a Q todos los operadores que aún están en P . Para esto, mientras P tenga elementos se hace $Q.ENQUEUE(P.CIMA())$ y luego $P.POP()$.

La Figura 5.4 muestra la ejecución de este algoritmo sobre la expresión $A \times B + C$, obteniendo como resultado la expresión $A B \times C +$.

Implementación de Colas

La implementación eficiente de colas es levemente más complicada que la implementación de pilas. Si se usa un arreglo (o buffer) $Q[1..m]$ para soportar la estructura de datos, debemos mantener variables para indicar la posición de ambos extremos de la cola dentro del arreglo —a diferencia de las pilas, en donde sólo uno de sus extremos, la cima, es variable. En particular, mantenemos la variable p que indica la posición del primer elemento de la cola, y la variable u que indica la posición del último elemento. La operación $INICIAR()$ inicializa dichas variables como $p \leftarrow 1$, $u \leftarrow 0$.

Para implementar la operación $DEQUEUE()$, simplemente se hace $p \leftarrow p + 1$. La operación $ENQUEUE(x)$, por otro lado, se implementa haciendo $u \leftarrow u + 1$ y luego $Q[u] \leftarrow x$. Sin embargo, se debe tener en cuenta que los extremos de la cola pueden alcanzar el límite superior del arreglo, sin que esté totalmente ocupado. Una solución eficiente en este caso es el concepto de *arreglo circular*, esto es, un buffer que es conceptualmente visto como circular. Eso significa que al elemento $Q[m]$ le sigue el elemento $Q[1]$. Cuando cualquiera de las dos variables, p o u , alcanzan el extremo superior de Q , circulan para retornar a la posición 1. De esta manera, tanto $ENQUEUE(x)$ como $DEQUEUE()$ tienen tiempo de ejecución $O(1)$.

La Figura 5.5 ilustra la idea de arreglo circular. La zona sombreada es la que contiene los elementos de la cola. La figura también muestra el arreglo lineal Q , que es el que almacena realmente los elementos.

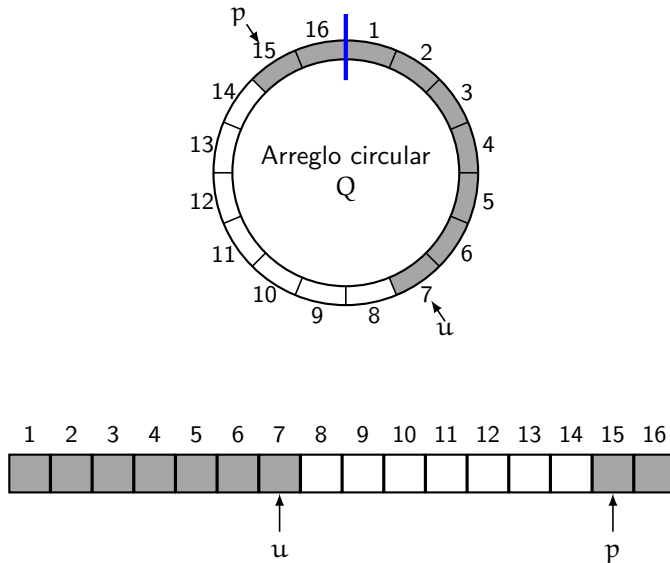


Figura 5.5: Implementación de una cola usando un arreglo circular $Q[1..16]$. Arriba se muestra la representación conceptual de la idea, mientras que abajo se muestra la representación real de la cola dentro del arreglo soporte Q .

5.3. Listas

Una lista L es una secuencia de n elementos $L = \langle a_1, a_2, \dots, a_n \rangle$ tal que a_1 es el primer elemento de la lista, a_2 es el segundo elemento de la lista, y en general a_i es el i -ésimo elemento de la lista. Una propiedad importante es que cada elemento en la lista tiene un único predecesor —excepto a_1 , que no tiene— y un único sucesor —excepto a_n , que no tiene. Por esta propiedad, a las listas se las conoce como estructuras de datos lineales. Una versión alternativa son las *listas circulares*, en donde el predecesor de a_1 es a_n , y por lo tanto el sucesor de a_n es a_1 .

Existen numerosas aplicaciones que necesitan mantener secuencias de elementos, por lo que las estructuras de tipo lista son de importancia. A diferencia de las pilas y colas, una lista es más general, en el sentido de que uno puede interactuar con cualquiera de los elementos que la conforman.

Operaciones

Típicamente, uno opera e interactúa con los elementos de la lista a medida que la recorre. Por lo tanto, vamos a suponer que se mantiene registro de cuál es el *elemento actual* de la lista ⁵. Definimos a continuación las operaciones permitidas sobre una lista L , muchas de las cuales operan sobre el elemento actual a_i , para $1 \leq i \leq n$:

L.INICIO(): hace que el elemento actual de la lista sea a_1 .

L.FIN(): hace que el elemento actual de la lista sea a_n .

L.IRA(j): hace que el elemento actual sea a_j , para $1 \leq j \leq n$.

L.SIG(): si $i < n$, hace que el elemento actual sea a_{i+1} . Si, por otro lado, el elemento actual es a_n : (1) si la lista es circular, hace que el elemento actual sea a_1 ; o (2) si la lista no es circular, el elemento actual pasa a ser el elemento ficticio a_{n+1} . Posteriores invocaciones a SIG() estando en a_{n+1} no tienen efecto, manteniéndose en a_{n+1} .

5: Si es necesario, se podría mantener registro de varios elementos sobre los que se está actuando, a modo de iteradores.

L.ANT(): si $i > 1$, hace que el elemento actual sea a_{i-1} . Si, por otro lado, el elemento actual es a_1 : (1) si la lista es circular, entonces hace que el elemento actual sea a_n ; o (2) si la lista no es circular, el elemento actual pasa a ser el elemento ficticio a_0 . Posteriores invocaciones a **ANT()** estando en a_0 no tienen efecto, manteniéndose en a_0 .

L.POS(): devuelve i , la posición actual dentro de la lista.

L.ELEM(): si $1 \leq i \leq n$, devuelve el elemento actual a_i . En otro caso, la operación no tiene efecto.

L.INSERTAR(elem): si $1 \leq i \leq n$, inserta un nuevo elemento **elem** justo antes del elemento actual a_i (el cual, por ende, se vuelve a_{i+1}).

L.BORRAR(): si $1 \leq i \leq n$, elimina el elemento actual a_i .

L.ANEXAR(elem): agrega un nuevo elemento $a_{n+1} = \text{elem}$ al final de la lista.

L.TAM(): devuelve n , el tamaño actual de la lista.

L.INICIAR(): inicializa la lista como vacía.

A modo de ejemplo, el Algoritmo 8 muestra el recorrido de los elementos de una lista, asumiendo que la función **PROCESAR** efectivamente permite procesar un elemento a_i de alguna manera.

Algoritmo 8: **RECORRERLISTA**($L = \langle a_1, \dots, a_n \rangle$)

Entrada: una lista $L = \langle a_1, \dots, a_n \rangle$.

```

1 L.INICIO()
2 while L.POS() ≤ n do
3   | PROCESAR(L.ELEM())
4   | L.SIG()
5 end
```

Aplicación: Representación de Polinomios y el Algoritmo de Horner

Sea $p(x) = c_0 + c_1x + c_2x^2 + \dots + c_nx^n$ un polinomio de grado n , de una única variable x , en donde los valores c_i son coeficientes constantes, y $c_n \neq 0$. Una manera típica de representar polinomios es mediante el vector de coeficientes $\langle c_0, c_1, \dots, c_n \rangle$. Dado que la posición de cada c_i dentro del vector es importante —ya que eso indica el x^i al que multiplica—, se puede representar el vector de coeficientes usando una lista de $n + 1$ elementos $\langle a_1, a_2, \dots, a_{n+1} \rangle$ tal que $a_i = c_{i-1}$, para $i = 1, \dots, n + 1$.

La evaluación de un polinomio para un valor particular de x es una tarea importante en muchas aplicaciones. El Algoritmo 9 implementa dicha evaluación, recorriendo la lista para obtener cada uno de los coeficientes del polinomio, los cuales son multiplicados por el correspondiente valor x^i . Es importante estudiar cómo resolver x^i para alcanzar un algoritmo eficiente. Si se resuelve simplemente aplicando la definición:

$$x^i = \underbrace{x \times x \cdots \times x}_{i-1 \text{ multiplicaciones}},$$

implica realizar $i - 1$ multiplicaciones, para $i = 1, \dots, n + 1$. Es fácil ver que la cantidad total de multiplicaciones es la serie aritmética

Algoritmo 9: EVALUAR($L = \langle a_1, \dots, a_{n+1} \rangle, x$)

Entrada: un polinomio p representado mediante su lista de coeficientes $L = \langle a_1, \dots, a_{n+1} \rangle$, y un valor $x \in \mathbb{R}$.

Salida: el valor $p(x)$.

```

1 L.INICIO()
2  $r \leftarrow 0$ 
3  $i \leftarrow 0$ 
4 while L.POS()  $\leq n + 1$  do
5    $r \leftarrow r + L.ELEM() \times x^i$ 
6   L.SIG()
7    $i \leftarrow i + 1$ 
8 end
9 return  $r$ 
```

$\sum_{i=1}^{n+1} i - 1 = \sum_{i=1}^n i = \frac{n(n+1)}{2}$. Note que el algoritmo realiza siempre esta cantidad de multiplicaciones, por lo que el tiempo de ejecución medido en cantidad de multiplicaciones es $\Theta(n^2)$ ⁶. Lamentablemente, este algoritmo no es eficiente para n suficientemente grande. En capítulos posteriores estudiaremos una manera de calcular x^i usando $\lg i$ multiplicaciones, lo cual reduciría el tiempo total de este algoritmo a $O(n \lg n)$ —bastante mejor.

Afortunadamente, hay una manera aún más eficiente de evaluar un polinomio, que estudiamos a continuación. Horner notó que un polinomio puede ser escrito como:

$$p(x) = c_0 + x(c_1 + x(c_2 + x(c_3 + \dots + x(c_{n-1} + c_n x))))). \quad (5.1)$$

Con este simple cambio, el proceso de evaluación mostrado en el Algoritmo 10 realiza $\Theta(n)$ multiplicaciones, aún mejor que antes. Más adelante en esta sección mostraremos cómo implementar L.ANT() en tiempo $O(1)$ de manera que el Algoritmo 10 sea de tiempo total $\Theta(n)$, y no sólo la cantidad de multiplicaciones esté acotada de esa forma.

Algoritmo 10: HORNER($L = \langle a_1, \dots, a_{n+1} \rangle, x$)

Entrada: un polinomio p representado mediante su lista de coeficientes $L = \langle a_1, \dots, a_{n+1} \rangle$, y un valor $x \in \mathbb{R}$.

Salida: el valor $p(x)$.

```

1 L.FIN()
2  $r \leftarrow 0$ 
3 while L.POS()  $\geq 1$  do
4    $r \leftarrow (r + L.ELEM()) \times x$ 
5   L.ANT()
6 end
7 return  $r$ 
```

6: El tiempo puede hacerse $O(n^2)$ agregando un chequeo para que no se realicen multiplicaciones cuando el correspondiente coeficiente c_i es 0.

Aplicación: Representación de Polígonos

Un polígono \mathcal{P} de n puntos en el plano 2 dimensional puede ser definido como una secuencia de puntos $\langle p_1, p_2, \dots, p_n \rangle$, tal que:

- para $i = 1, \dots, n-1$ se cumple que $\overline{p_i p_{i+1}}$ es un lado del polígono,
y

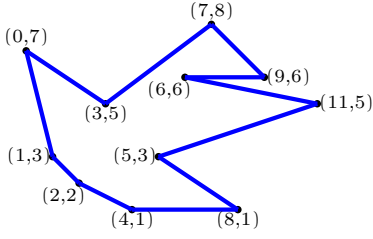


Figura 5.6: Un polígono representado por la lista de puntos $\langle (2,2), (4,1), (8,1), (5,3), (11,5), (6,6), (9,6), (7,8), (3,5), (0,7), (1,3), (5,3) \rangle$.

- $\overline{p_n p_1}$ es un lado del polígono.

Por lo tanto, es natural representar dicha secuencia usando una lista circular $L = \langle p_1, \dots, p_n \rangle$. La Figura 5.6 muestra un polígono representado por la lista circular de puntos $\langle (2,2), (4,1), (8,1), (5,3), (11,5), (6,6), (9,6), (7,8), (3,5), (0,7), (1,3) \rangle$.

Cualquiera de los puntos del polígono puede ser elegido como el primer elemento de la lista —por la circularidad de la misma. Además, puede usarse cualquiera de los dos sentidos posibles para los puntos: horario o antihorario (este último es el caso de la Figura 5.6). En general, en la literatura se prefiere asumir el sentido antihorario, por lo que asumiremos lo mismo.

A modo de ilustración, estudiamos a continuación el problema de determinar si un polígono \mathcal{P} de $n \geq 3$ lados es convexo o no. Tal como hemos asumido, supongamos que \mathcal{P} ha sido representado con una lista circular en sentido antihorario. Recordemos que para que un polígono sea convexo, cada uno de sus ángulos internos debe ser convexo —es decir, menor o igual a 180 grados. Visto de otro modo, un ángulo cualquiera de un polígono, conformado por los puntos p_{i-1} , p_i , y p_{i+1} , es convexo si al movernos en el sentido $p_{i-1} \rightarrow p_i \rightarrow p_{i+1}$ —antihorario— se realiza un giro a la izquierda —o no se realiza ningún giro si son 3 puntos colineales. Por ejemplo, considere los puntos $(4,1)$, $(8,1)$, y $(5,3)$ del polígono de la Figura 5.6. Note que al movernos $(4,1) \rightarrow (8,1) \rightarrow (5,3)$ se hace un giro a la izquierda, por lo que el ángulo es convexo. Por otro lado, si hacemos $(8,1) \rightarrow (5,3) \rightarrow (11,5)$ en la misma figura, se hace un giro a la derecha, ya que el ángulo correspondiente es cóncavo.

A continuación, mostramos una operación que permite determinar si un ángulo definido a partir de tres puntos es convexo o no. Usaremos dicha operación para chequear cada uno de los n ángulos internos del polígono. Sean $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, y $p_3 = (x_3, y_3)$. El área (con signo) del triángulo formado por esos tres puntos puede calcularse con el determinante:

$$AS(\Delta(p_1, p_2, p_3)) = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1 y_2 + x_2 y_3 + x_3 y_1 - x_1 y_3 - x_2 y_1 - x_3 y_2. \quad (5.2)$$

Lo que nos interesa de esta expresión es su signo, ya que::

- $AS(\Delta(p_1, p_2, p_3)) > 0$: significa que en el movimiento $p_1 \rightarrow p_2 \rightarrow p_3$, en p_2 se hace un giro a la izquierda. Esto es, p_3 está a la izquierda de $\overrightarrow{p_1 p_2}$, por lo que el ángulo interno en p_2 es convexo.
- $AS(\Delta(p_1, p_2, p_3)) < 0$: significa que en el movimiento $p_1 \rightarrow p_2 \rightarrow p_3$, en p_2 hacemos un giro a la derecha. Esto es, p_3 está a la derecha de $\overrightarrow{p_1 p_2}$, por lo que el ángulo interno en p_2 es cóncavo.
- $AS(\Delta(p_1, p_2, p_3)) = 0$: significa que p_1 , p_2 , y p_3 son puntos colineales, esto es, el ángulo interno en p_2 es llano.

Como dijimos, el algoritmo que determina la convexidad simplemente recorre el polígono, chequeando cada uno de sus ángulos. Como cada invocación a AS toma tiempo $O(1)$, el tiempo total para los chequeos

es $O(n)$ —el peor caso necesita una cantidad lineal de chequeos, pero pueden ser menos.

Aplicación: Administración Dinámica de Memoria

La administración dinámica de memoria es fundamental para el sistema de ejecución de la mayoría de los lenguajes de programación. De esta manera, los lenguajes permiten al programador pedir memoria cuando el programa está en ejecución. Esto corresponde a las funciones estándar `malloc` y `free` del lenguaje C, y a `new` y `delete` de C++. Estudiamos a continuación una manera simple de implementar un administrador de memoria, para ilustrar el uso de listas ⁷.

Supongamos que disponemos de un bloque de memoria contigua $B[1..M]$ de M celdas de memoria. Esta es la memoria disponible para administrar dinámicamente. A lo largo de la ejecución de un programa, éste solicitará bloques de memoria contigua de un tamaño variable (digamos que la i -ésima solicitud requiere m_i celdas) usando la operación `new`, y además liberará bloques de memoria asignados anteriormente usando la operación `delete`. Administraremos los bloques de memoria usando las siguientes dos listas:

Lista de bloques disponibles, L_1 : cada nodo esta lista almacenará un bloque de memoria disponible, representado por dos números enteros. Estos indican la celda de comienzo y de final que corresponde al bloque disponible. Inicialmente, esta lista contiene un único nodo que representa a $B[1..M]$ —i.e., el bloque completo está disponible. Los nodos de la lista se mantienen ordenados por la celda de comienzo de los bloques libres.

Lista de bloques asignados, L_2 : esta lista se mantiene de manera similar a la anterior, salvo que no es necesario mantener el orden de los nodos. Inicialmente, la lista está vacía, ya que no hay bloques asignados.

Dado un requerimiento de memoria de m_i celdas, debemos recorrer la lista L_1 hasta encontrar el primer bloque de tamaño $m' \geq m_i$ capaz de satisfacer el requerimiento —estrategia conocida como *first-fit* en la literatura. Si $m' > m_i$, se debe agregar el nuevo bloque de tamaño m_i a la lista L_2 , y se debe modificar el tamaño del bloque disponible en la lista L_1 , que ahora es de $m' - m_i$ celdas.

La liberación de memoria, por otro lado, se hará indicando la celda de comienzo del bloque a liberar. Dicho bloque debe ser buscado en la lista L_2 —recorriéndola—, para ser eliminado de la misma y asignado a la lista L_1 , en la posición adecuada ⁸. Se debe tener en cuenta que si existen bloques libres contiguos en L_1 , los mismos deben ser unificados en un único bloque más grande. Esto es para evitar el particionamiento del espacio disponible. Por ejemplo, supongamos que la lista L_1 contiene los bloques $B[i..j]$ y $B[k..l]$, y luego se libera el bloque $B[j + 1..k - 1]$ de L_2 , por lo que ahora está disponible. Entonces, los tres nodos contiguos de L_1 , $B[i..j]$, $B[j + 1..k - 1]$, y $B[k..l]$, deben unificarse en un único bloque disponible $B[i..l]$.

7: El ejemplo es sólo a modo de ilustración, por lo que ciertos aspectos pueden ser implementados más eficientemente con pequeñas modificaciones.

8: Esta operación puede hacerse más eficientemente, sin necesidad de recorrer L_2 , introduciendo pequeños cambios a este esquema.

Implementación de Listas

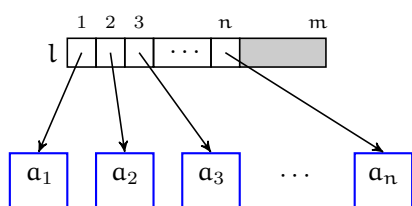
Estudiamos a continuación tres maneras típicas de implementar una lista.

Listas Basadas en Arreglos. La lista se representa usando un arreglo $l[1..m]$, con $m \geq n$, tal que para $1 \leq i \leq n$ se cumple $l[i] = a_i$. Además, se mantiene una variable entera c que indica la posición actual dentro de la lista, y una variable entera $n \leq m$ que indica la cantidad de elementos que tiene la lista —los cuales están almacenados en $l[1..n]$. De esta manera, la operación **INICIO()** simplemente hace $c \leftarrow 1$, **FIN()** es $c \leftarrow n$, **IRA(j)** es $c \leftarrow j$, **SIG()** es $c \leftarrow c + 1$, **ANT()** es $c \leftarrow c - 1$, **ELEM()** devuelve $l[c]$, y **TAM()** simplemente devuelve n ⁹. Así, todas esas operaciones pueden implementarse en tiempo $O(1)$. La operación **ANEXAR($elem$)** hace $n \leftarrow n + 1$, y luego $l[n] \leftarrow elem$, por lo que también toma tiempo constante.

9: Obviamente, se debe hacer la verificación de errores respecto a la posición actual i .

Por otro lado, las operaciones **INSERTAR()** y **BORRAR()** —que permiten dinamismo en la lista—, son más costosas en esta implementación. El problema es que se asocia de forma estricta cada elemento a_i con la celda $l[i]$ del arreglo. Al insertar o eliminar un elemento, cambia el índice de ciertos elementos en la lista, lo que lleva a cambiar de posición a esos elementos dentro del arreglo. De forma más clara, supongamos que se quiere insertar un nuevo elemento en la posición c de la lista. Eso significa que el nuevo elemento será a_c luego de la inserción, mientras que los anteriores elementos a_c, \dots, a_n ahora serán a_{c+1}, \dots, a_{n+1} . Por lo tanto, los elementos a_c, \dots, a_n deben ser desplazados una posición a la derecha dentro del arreglo l , para luego almacenar el nuevo elemento en la celda $l[c]$. Algo similar ocurre para los borrados, en donde los elementos involucrados deben ser movidos una posición a la izquierda. Eso toma tiempo $O(n)$, lo cual es ineficiente para aplicaciones que realicen una gran cantidad de inserciones y borrados.

Además de que el dinamismo tiene un costo asintótico alto en este tipo de listas, cuando los elementos que almacena la lista son muy grandes —por ejemplo, una lista de imágenes, o de puntos 1000-dimensionales—, la constante asociada a $O(n)$ puede ser grande. Esto afectaría aún más el tiempo de ejecución en la práctica. La razón es el desplazamiento de elementos se hacen mediante asignaciones. Recuerde que en un ciclo de CPU de un computador RAM, se puede asignar una variable de w bits —e.g., un entero— a otra. Si cada elemento de la lista usa más de w bits, se requieren varios ciclos de la CPU para desplazar un único elemento, lo que incrementa considerablemente el tiempo total. Por ejemplo, pensemos en una lista de puntos 1000-dimensionales. Cada elemento de la lista podría requerir $1000w$ bits, por lo tanto cada uno requiere de 1000 ciclos del procesador para ser desplazado. Si necesitásemos desplazar 1000 millones de esos puntos en una lista, y suponiendo que cada ciclo toma 1 nanosegundo, se necesitarían 1000 segundos —unos 16.6 minutos— para completar la tarea.



Dicha constante puede reducirse almacenando los elementos en un espacio separado, pedido dinámicamente, y manteniendo punteros a ellos en $l[1..m]$. De esta forma, los desplazamientos al insertar y borrar se hacen sobre los punteros, que en general son de w bits y pueden ser

asignados en un único ciclo por una RAM, sin importar el tamaño de los objetos almacenados en la lista. De hecho, el lenguaje Python usa esta implementación de listas. El ejemplo anterior, que tomaba 1000 segundos, ahora tomaría 1 segundo usando esta técnica. El espacio total usado es de $\Theta(n + m)$.

Para finalizar, es importante discutir qué hacer cuando el arreglo l se llena por completo, luego de sucesivas inserciones. Note que estamos ante el mismo problema de mantener un buffer que crece dinámicamente, mencionado en la Sección 5.1 al implementar una pila. Como ya se dijo, estudiaremos este tema en detalle en el Capítulo 6.

Listas Enlazadas. A diferencia de las listas basadas en arreglos discutidas anteriormente, las listas enlazadas son más flexibles respecto a la celda de memoria que almacena un elemento a_i de la lista. Esa poca flexibilidad hace que las inserciones y borrados en una lista basada en arreglo requieran desplazar elementos dentro del espacio de almacenamiento, para ajustarlos a sus nuevos índices dentro de la lista.

En una lista enlazada, los elementos no son almacenados de forma contigua en la memoria, sino que cada elemento ocupa una celda de memoria determinada por el administrador de memoria dinámica del sistema. Sea $\text{dir}(a_i)$ la celda de la memoria \mathcal{M} que almacena al elemento a_i de la lista. En dicha celda almacenaremos —junto con el elemento a_i — el valor $\text{dir}(a_{i+1})$, el enlace —o puntero— al siguiente elemento de la lista. En otras palabras, $\mathcal{M}[\text{dir}(a_i)] = (a_i, \text{dir}(a_{i+1}))$. Para marcar el final de la lista, junto con a_n se almacena una dirección de memoria que pueda ser reconocida como inválida, y que denotaremos nil . En nuestro computador RAM, podría ser $\text{nil} = 0$, ya que la memoria \mathcal{M} no tiene celda 0¹⁰. Siguiendo a la literatura, llamaremos *nodo* a cada par $(a_i, \text{dir}(a_{i+1}))$. Si c es un puntero a un nodo de la lista enlazada, usaremos $c.e$ para denotar al elemento a_i del par ordenado, mientras que $c.s$ será el puntero al siguiente elemento.

10: De hecho, algunos lenguajes de programación como C o C++ la constante NULL equivale a 0

En resumen, en esta implementación necesitamos almacenar lo siguiente:

- Un puntero $p = \text{dir}(a_0)$ al primer nodo de la lista. El resto de los elementos de la lista se pueden acceder desde allí, siguiendo el enlace almacenado en cada nodo.
- Un puntero $u = \text{dir}(a_n)$, que permite acceder rápidamente al último nodo de la lista.
- Un puntero c , que apunta al nodo que contiene el elemento actual de la lista.
- Una variable n , que indica la cantidad de elementos de la lista.

La Figura 5.7 muestra una lista enlazada de 6 elementos. Cada nodo se grafica con un rectángulo que almacena el elemento a_i y el puntero al siguiente nodo, representado como una flecha en el diagrama. El último nodo de la lista apunta a nil , que se indica mediante dos líneas verticales.

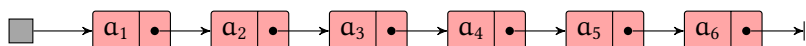


Figura 5.7: Una lista enlazada $L = \langle a_1, a_2, a_3, a_4, a_5, a_6 \rangle$.

Respecto a implementación de las operaciones sobre esta representación, tenemos lo siguiente:

INICIO() y FIN(): note que pueden implementarse en tiempo constante, simplemente haciendo $c \leftarrow p$ o $c \leftarrow u$, respectivamente, por lo tanto toman tiempo $O(1)$.

IR_A(j): note que no sabemos de antemano en qué celda de la memoria está almacenado a_j . El único nodo de la lista que conoce dónde está almacenado a_j es el que almacena a a_{j-1} . Pero, nuevamente, para saber dónde está almacenado a_{j-1} tenemos que saber dónde está almacenado a_{j-2} , y así siguiendo hasta el primer elemento de la lista. Eso implica que para poder llegar al nodo de a_j , debemos seguir una secuencia de j punteros desde el inicio de la lista. En el peor caso, hay que visitar los n nodos de la lista, por lo que el tiempo de ejecución es $O(n)$.

SIG(): simplemente hay que hacer $c \leftarrow c.s$, lo cual toma tiempo $O(1)$.

ANT(): en este caso, si el elemento actual de la lista es a_i , necesitamos acceder a a_{i-1} . Sin embargo, esto requeriría recorrer la lista desde el inicio en tiempo $O(n)$, lo cual es poco eficiente. Para poder implementar esta operación en tiempo constante, debemos representar cada nodo de la lista de esta forma:

$$(a_i, \text{dir}(a_{i-1}), \text{dir}(a_{i+1})).$$

Esto es, con cada elemento a_i almacenamos un puntero no sólo al nodo siguiente en la lista, sino que también un puntero al nodo anterior. Este tipo de implementación se conoce como *lista doblemente enlazada*. Esto agrega espacio adicional $\Theta(n)$ a la implementación, pero permite implementar los retrocesos en la lista de forma eficiente. Esto es importante para aplicaciones que necesitan recorrer una lista de ambas direcciones, o en dirección reversa, como era el caso del Algoritmo 10.

ELEM(): devuelve $c.e$, en tiempo $O(1)$.

TAM(): devuelve n , en tiempo $O(1)$.

ANEXAR(elem): se debe crear un nuevo nodo que almacena $(elem, nil)$, almacenado en la dirección de memoria $\text{dir}(elem)$. Luego, se hace $u.s \leftarrow \text{dir}(elem)$, con lo que el antiguo último nodo ahora apunta al nuevo nodo creado. Finalmente, se hace $u \leftarrow \text{dir}(elem)$. Todo esto toma tiempo $O(1)$, asumiendo que el manejo de memoria dinámica se hace en tiempo constante ¹¹.

INSERTAR(elem): suponga que a_i es el elemento actual de la lista. Para agregar el nuevo elemento, se debe crear un nuevo nodo en la lista que almacena $(elem, \text{dir}(a_i))$. De esta forma, el nuevo nodo estará justo antes del nodo de a_i . Luego, hay que modificar el nodo de a_{i-1} , para que ahora almacene $(a_{i-1}, \text{dir}(elem))$. Así, el nuevo nodo queda justamente entre los nodos que contienen a a_{i-1} y a_i . Sin embargo, para resolver la operación de forma eficiente, es necesario mantener en todo momento —durante la operación de la lista— un puntero a al nodo anterior al nodo actual —apuntado por c . Es importante notar que hay casos particulares de inserción, como por ejemplo inserción al comienzo o al final de la lista, o inserción en una lista vacía. Dichos detalles se omiten aquí por simplicidad. El tiempo total es $O(1)$.

BORRAR(): asumiendo que el elemento actual es a_i , para implementar

11: Esto es cierto si se tiene un administrador de memoria especializado para la lista, que sólo permita manejar pedidos de memoria dinámica para un tamaño constante establecido, que en este caso correspondería al tamaño necesario para implementar cada nodo. Recuerde el administrador de memoria dinámica estudiado anteriormente en esta sección.

esta operación se debe “desenganchar” el nodo correspondiente de la lista enlazada. Esto se logra haciendo $a.s \leftarrow \text{dir}(a_{i+1})$, y $c \leftarrow \text{dir}(a_{i+1})$ (aquí, a es el puntero al nodo anterior usado para la operación `INSERTAR()`). Luego, se debe liberar la memoria que contenía al nodo de a_i . El tiempo total es $O(1)$.

Las listas enlazadas soportan eficientemente las operaciones de inserción y borrado. Sin embargo, no son capaces de implementar eficientemente accesos aleatorios a elementos de la lista. En resumen, las listas enlazadas son adecuadas para aplicaciones que necesiten recorrer una lista, a la vez que se realizan inserciones/borrados en el camino. Un ejemplo puede ser el manejo de polígonos, en el que se necesiten agregar o borrar puntos a medida que lo se recorre.

Respecto a su adopción en la práctica, es importante notar que hay aplicaciones importantes que utilizan listas enlazadas. Uno de los usos más emblemáticos sea, quizás, en la implementación de la clase `list` del lenguaje C++, que utiliza listas doblemente enlazadas ya que permiten iteradores bidireccionales.

Listas Basadas en Xor. La idea es implementar una lista doblemente enlazada, pero condensando en un único puntero información que permita avanzar al nodo siguiente y retroceder al nodo anterior —como si tuviésemos dos punteros por nodo. La ventaja es el ahorro de espacio: se tiene la funcionalidad y eficiencia de una lista doblemente enlazada, pero usando espacio similar al de una lista enlazada simple. Basaremos la implementación en la operación *o exclusivo* (xor) a nivel de bits, que denotamos con el operador \oplus , y que definimos a continuación ¹². Sean $p[1..w]$, $q[1..w]$, y $r[1..w]$ números enteros sin signo, representados en binario como una secuencia de w bits. Sea $p \oplus q = r$, entonces para $i = 1, \dots, w$ se tiene $r[i] = 1$ si y sólo si $p[i] \neq q[i]$ —i.e., $r[i] = 0$ en otro caso. Note que para la operación \oplus se cumplen las siguientes propiedades:

1. $(p \oplus q) \oplus p = q$.
2. $(p \oplus q) \oplus q = p$.
3. $p \oplus 0 = p$.
4. $p \oplus p = 0$.
5. $p \oplus q = q \oplus p$.
6. $(p \oplus q) \oplus r = p \oplus (q \oplus r)$.

Al igual que antes, sea $\text{dir}(a_i)$ a la dirección de memoria correspondiente al nodo de a_i . Entonces, para $i = 2, \dots, n-1$, en esta representación el nodo de a_i se representa de la forma:

$$(a_i, \text{dir}(a_{i-1}) \oplus \text{dir}(a_{i+1})),$$

El nodo de a_1 , por otro lado, almacena $(a_1, 0 \oplus \text{dir}(a_2)) = (a_1, \text{dir}(a_2))$, mientras que el nodo de a_n almacena $(a_n, \text{dir}(a_{n-1}) \oplus 0) = (a_n, a_{n-1})$.

Para poder recorrer la lista, se mantienen tres punteros: un puntero $c = \text{dir}(a_i)$ al nodo actual, un puntero $a = \text{dir}(a_{i-1})$ al nodo anterior, y un puntero $s = \text{dir}(a_{i+1})$ al nodo siguiente. El puntero a y la información almacenada en el nodo actual son suficientes para avanzar

12: Asumimos que dicha operación es soportada por una instrucción de la máquina RAM, y que usa un único ciclo para obtener el resultado.

al nodo de a_{i+1} , ya que

$$\text{dir}(a_{i-1}) \oplus (\text{dir}(a_{i-1}) \oplus \text{dir}(a_{i+1})) = \text{dir}(a_{i+1}).$$

Note que esta misma fórmula aplicada sobre el nodo correspondiente a a_n produce $\text{dir}(a_{n-1}) \oplus (\text{dir}(a_{i-1}) \oplus \emptyset) = \emptyset$, lo cual puede usarse para detener el recorrido de la lista. Algo similar ocurre con a_1 cuando se retrocede en la lista. Para retroceder en la lista, es suficiente usar el puntero s y la información almacenada en el puntero del nodo actual.

Ejercicios

1. Dado un polígono \mathcal{P} y un punto x en el plano, escribir un algoritmo que determine si x está contenido dentro del polígono o no.
2. Conjunto de puntos en el plano, encontrar el cuadrado con los lados paralelos a los ejes de coordenadas que tenga área mínima.
3. Polígono convexo de área mínima que contiene a otro polígono? Es como el convex hull de un polígono, quizás sirva para entender luego el Graham scan.
4. La representación de polinomios que hemos definido usa espacio $\Theta(n)$ para un polinomio de grado n . Modifique dicha representación para que use espacio $O(n)$.

5.4. Árboles

Los *árboles enraizados* —o árboles con raíz— son una estructura de datos fundamental en ciencias de la computación, utilizados en general para representar datos de forma jerárquica. Un árbol se construye sobre un conjunto N de *elementos* o *nodos* —usaremos indistintamente ambos nombres. Dado un árbol A y un elemento $x \in N$, diremos que $x \in A$ si alguno de los elementos de A es x . Además, si A_1 y A_2 son árboles, entonces

$$A_1 \cap A_2 = \{x \in N \mid x \in A_1 \wedge x \in A_2\}, \text{ y } A_1 \cup A_2 = \{x \in N \mid x \in A_1 \vee x \in A_2\}.$$

Definimos ahora el conjunto \mathcal{A} de todos los posibles árboles que podemos construir con elementos de N . La definición se hace de forma inductiva.

Definición 5.4.1 *El conjunto \mathcal{A} de todos los posibles árboles enraizados formados a partir de un conjunto de elementos N se construye con las siguientes reglas:*

Caso Base: $\forall x \in N, \langle x \rangle \in \mathcal{A}$.

Paso Inductivo: Si $A_1, \dots, A_k \in \mathcal{A}$ son árboles tal que $\bigcap_{i=1}^k A_i = \emptyset$, entonces

$$A = \langle x, A_1, \dots, A_k \rangle \in \mathcal{A},$$

$$\text{para } x \in N \setminus \bigcup_{i=1}^k A_i.$$

En ambos puntos de la definición, el elemento x es la *raíz* del árbol resultante. El caso base de la definición corresponde a árboles de un único elemento. Esos son usados posteriormente por la definición inductiva para construir árboles más grandes. En el paso inductivo, diremos que x es *padre* de cada una de las raíces r_1, \dots, r_k de los árboles A_1, \dots, A_k , respectivamente. De forma simétrica, los *hijos* de r son r_1, \dots, r_k . Cada uno de los árboles A_1, \dots, A_k es un subárbol del árbol A . Si un elemento x no tiene hijos en el árbol, diremos que x es una *hoja* del árbol.

Ejemplo 5.4.1 Por ejemplo, si $S = \{r, s, t, u, v, w, x, y, z\}$, entonces

$$\langle r, \langle s, \langle v \rangle, \langle w \rangle \rangle, \langle t \rangle, \langle u, \langle x \rangle, \langle y \rangle, \langle z \rangle \rangle \rangle$$

es un posible árbol sobre S . La Figura 5.8 muestra la representación jerárquica típica para este árbol. La raíz se dibuja en la parte superior del árbol. En general, un elemento se dibuja por sobre sus hijos. Luego, se une con una línea —o arco— a cada elemento con sus hijos. De esta forma, las hojas del árbol quedan en la parte inferior del mismo.

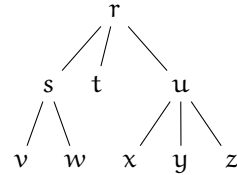


Figura 5.8: Representación jerárquica del árbol $\langle r, \langle s, \langle v \rangle, \langle w \rangle \rangle, \langle t \rangle, \langle u, \langle x \rangle, \langle y \rangle, \langle z \rangle \rangle \rangle$. La raíz del árbol es r , mientras que v, w, t, x, y, z son hojas.

Note que la condición $A_1 \cap \dots \cap A_k = \emptyset$ en la Definición 5.4.1 exige que los subárboles que forman al árbol A sean disjuntos respecto a los elementos que contienen. Además, por la construcción todo elemento x del árbol tiene un único padre —excepto la raíz, que no tiene. Por otro lado, todo elemento x tiene $g_x \geq 0$ hijos. Llamaremos a g_x el *grado* del elemento x . El hecho de poder tener más de un hijo por elemento diferencia a los árboles de las listas ¹³. El *grado del árbol* A es el número $\max\{g_x \mid x \in A\}$.

Diremos que una secuencia de elementos $\langle v_1, v_2, \dots, v_\ell \rangle$ es un *camino* de largo $\ell - 1$ en un árbol si para cada $1 \leq i \leq \ell - 1$ se cumple que v_i es padre de v_{i+1} . Por ejemplo, $\langle r, s, w \rangle$ es un camino de largo 2 en el árbol de la Figura 5.8. Otros caminos son $\langle r, t \rangle$, $\langle r, u, x \rangle$, y $\langle u, y \rangle$. Finalmente, $\langle r, s, z \rangle$ no es un camino, ya que s no es padre de z . Un elemento x de un árbol es *ancestro* de un elemento y si existe un camino desde x a y . En el árbol de la Figura 5.8, r y s son ancestros de w , mientras que t no lo es. Si x es ancestro de y , entonces podemos decir que y es un *descendiente* de x . Note que todos los elementos de un árbol son descendientes de la raíz del mismo.

Ejemplo 5.4.2 Consideremos el siguiente ejemplo de uso de árboles. Sean $A, B, C, D, E, F, G, H, I, J, K$ conjuntos del algún tipo y con el mismo universo U , para los que se cumple lo siguiente:

- $B \subseteq A, C \subseteq A, D \subseteq A$, y $B \cap C \cap D = \emptyset$.
- $E \subseteq B$.
- $I \subseteq C, J \subseteq C$, y $I \cap J = \emptyset$.
- $F \subseteq D, G \subseteq D, H \subseteq D$, y $F \cap G \cap H = \emptyset$.
- $K \subseteq H$.

Como los conjuntos involucrados en cada uno de los puntos anteriores son disjuntos, podemos definir un árbol de la siguiente manera: cada relación de inclusión $S' \subseteq S$ indica que S' es hijo de S en el árbol. De

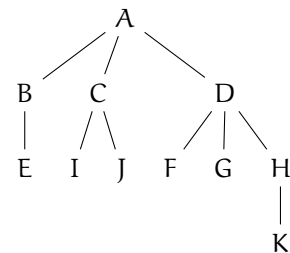


Figura 5.9: Esquema de árbol para el Ejemplo 5.4.2.

13: De hecho, una lista puede ser vista como un caso particular de árbol, en el que cada elemento tiene un único hijo.

esta manera, el árbol resultante para las anteriores condiciones es el de la Figura 5.9. De esta forma, dados dos conjuntos S y S' , uno puede determinar si $S' \subseteq S$: sólo hay que chequear si S es ancestro de S' . Aquí es de ayuda que la inclusión de conjuntos es una relación transitiva.

Una *rama* en un árbol es un camino desde la raíz hasta una hoja. De hecho, cada hoja de un árbol define una rama del mismo. Por ejemplo, el árbol de la Figura 5.8 tiene 6 ramas: $\langle r, s, v \rangle$, $\langle r, s, w \rangle$, $\langle r, t \rangle$, y así siguiendo.

Diremos que un elemento x tiene *profundidad* d si el único camino desde la raíz a x tiene largo d . En el árbol de la Figura 5.8, r tiene profundidad 0, t tiene profundidad 1, y z tiene profundidad 2. La *altura* de un árbol A , denotada $h(A)$, corresponde al largo de la rama más larga del mismo, y puede definirse por inducción estructural como a continuación:

- *Caso base*: si $A = \langle x \rangle$, para $x \in N$, la altura es $h(\langle x \rangle) = 0$.
- *Paso inductivo*: Si $A = \langle x, A_1, \dots, A_k \rangle$, entonces la altura es $h(\langle x, A_1, \dots, A_k \rangle) = 1 + \max\{h(A_1), \dots, h(A_k)\}$.

Por ejemplo, la altura del árbol de la Figura 5.8 es 2, mientras que la del árbol de la Figura 5.9 es 3 —que corresponde a la rama $\langle A, D, H, K \rangle$. Un *nivel* en un árbol es el conjunto de todos los nodos que tienen la misma profundidad. Para el árbol de la Figura 5.9, el nivel 0 es el conjunto $\{A\}$, el nivel 1 es $\{B, C, D\}$, el nivel 2 es $\{E, I, J, F, G, H\}$, y el nivel 3 es $\{K\}$.

Finalmente, llamaremos *bosque* a un conjunto de $k \geq 0$ árboles A_1, \dots, A_k .

Estudiamos a continuación los principales tipos particulares de árboles.

Árboles Ordinales

En un *árbol ordinal*, los hijos de un elemento están ordenados de manera que existe un primer hijo, un segundo hijo, y así siguiendo. Diremos que el i -ésimo hijo de un nodo tiene rango i entre sus hermanos. Aunque ciertas aplicaciones limitan el grado máximo del árbol, en general podemos asumir que el grado es arbitrario y no está acotado superiormente: cada elemento puede tener tantos hijos como sea necesario.

En este tipo de árboles, normalmente uno necesita manipular los hijos de un elemento x secuencialmente, desde el primero al último. Por lo tanto, las operaciones más importantes a implementar son:

PRIMER_HIJO(x): devuelve el primer hijo de x .

SIG_HIJO(x): obtiene el siguiente hijo del elemento x .

HIJO(x, i): obtiene el hijo con rango i del elemento x . Si x tiene menos de i hijos, devuelve el valor inválido `nil`.

Además, debería ser posible agregar y eliminar elementos de forma dinámica en un árbol ordinal.

La Figura 5.10 muestra un ejemplo de árbol ordinal, con los elementos representados por círculos. Junto a cada elemento, se muestra su rango.

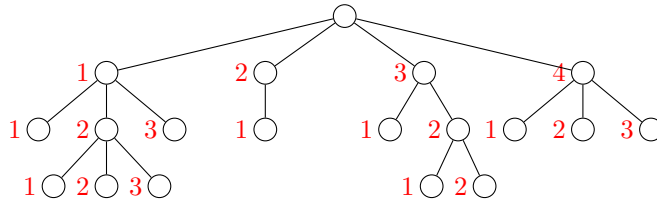


Figura 5.10: Un árbol ordinal. A la izquierda de cada elemento se muestra su rango.

Implementación. Para implementar un árbol ordinal permitiendo que cada elemento del árbol tenga tantos hijos como sea necesario, se representa a cada elemento del árbol usando nodos similares a los de las listas enlazadas. Cada nodo tiene dos punteros:

- Puntero al primer hijo; y
- Puntero al siguiente hermano.

Con esta técnica simple es posible mantener una lista enlazada de hijos de un elemento, la cual puede crecer para agregar los hijos que se necesiten. Así, las operaciones `PRIMER_HIJO()` y `SIG_HIJO()` toman tiempo $O(1)$, mientras que `HIJO(x, i)` toma tiempo $O(i)$ ya que hay que recorrer la lista enlazada de los hijos de x .

Árboles Cardinales

Un *árbol cardinal de grado k* es uno cuyo grado está acotado a un valor máximo k , y cada uno de sus elementos tiene, conceptualmente, k ranuras —una para cada uno de sus, a lo más, k hijos. Cada uno de esos k hijos puede estar presente o no, por lo que cada nodo debe indicar esa información —e.g., si un hijo no está presente, su ranura estará vacía. Diremos que el hijo correspondiente a la ranura i de un elemento x es el *hijo de x rotulado i* . A diferencia de los árboles ordinales, es el rótulo el que distingue a los hijos de un elemento y no la posición entre sus hermanos.

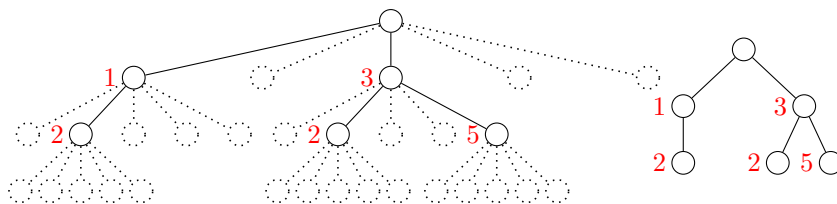
La operación más importante que implementa este tipo de árboles es:

HIJO(x, i): obtiene el hijo con rótulo i del elemento x . Si dicho hijo no existe para x , devuelve el valor inválido `nil`.

Note la diferencia con la operación homónima de los árboles ordinales, para los cuales el argumento i de la operación es interpretado como un número ordinal —i.e., se quiere descender el i -ésimo hijo. Por otro lado, el mismo argumento para la operación de los árboles cardinales es un número cardinal —i.e., se quiere descender al hijo i . Tal como en los árboles ordinales, también se puede agregar y borrar elementos del árbol.

La Figura 5.11 muestra un ejemplo de árbol cardinal de grado máximo $k = 5$. En el árbol de la izquierda de la figura, para cada elemento se muestran sus 5 ranuras conceptuales. Aquellas que corresponden a hijos que no están presentes en el árbol se muestran con líneas punteadas. El árbol de la izquierda es el mismo, pero muestra sólo los hijos que están presentes. En ambos casos, junto a cada elemento se muestra su rótulo.

Figura 5.11: Un árbol cardinal de grado máximo $k = 5$. En el árbol de la izquierda, las líneas punteadas corresponden a elementos que no están presentes en el árbol, por lo tanto la ranura correspondiente en el elemento padre es vacía. A la derecha se muestra el mismo árbol, pero incluyendo sólo los elementos presentes. A la izquierda de cada elemento se muestra su rótulo.



Implementación y el Problema del Diccionario. Para implementar la operación $\text{HIJO}(x, i)$ eficientemente en un árbol cardinal, a cada elemento x del árbol se debe asociar la función

$$D_x = \{(i, h_i) \mid i \in \mathbb{N}_k \wedge h_i \text{ es el hijo de } x \text{ con rótulo } i\}.$$

Luego, se construye una estructura de datos sobre D_x tal que permita resolver la siguiente consulta: dado un valor i , determinar no sólo la pertenencia de un par en D_x cuya primera componente sea i , sino que también obtener el valor h_i asociado. Este es un caso particular de un problema fundamental en estructuras de datos ¹⁴, conocido como el *problema del diccionario* o, alternativamente, *arreglos asociativos*, *mapas*, o *tablas de símbolos*.

De forma general, el problema del diccionario consiste en construir una estructura de datos para almacenar una relación R que contiene pares del tipo (llave, valor asociado), en donde llave es un valor único para cada par —i.e., la relación R es una función. Luego, la estructura resuelve consultas en las que uno aporta una llave como dato de búsqueda, y obtiene como respuesta el valor asociado a dicha llave. Dicho problema es un caso particular del problema de búsqueda y chequeo de pertenencia en un conjunto —que hemos estudiado de forma preliminar, y muy primitivamente, e.g., en los Algoritmos 2 y 3. Existen soluciones más eficientes que dichos algoritmos, que estudiaremos más adelante. De hecho, muchos lenguajes de programación proveen soluciones para este problema, como los *diccionarios* de Python y los *map* de C++.

En el caso particular de los árboles cardinales, el dominio de la relación D_x , para todo elemento x del árbol, es \mathbb{N}_k . Una estructura simple para este caso es un arreglo $d_x[1..k]$ que almacene información respecto a cada uno de los hijos del nodo x . Esto es, que dé acceso a los hijos de x que están presentes, y que marque a aquellos que no lo están, ambos casos en $\Theta(1)$. Desafortunadamente, el uso de espacio de una solución así sería de $\Theta(kn)$ celdas de memoria. Es posible, sin embargo, resolver este problema con espacio $\Theta(n)$, aunque en ciertos casos sacrificando levemente el tiempo de acceso a los hijos. Estudiaremos este tema más adelante.

Árboles Binarios

Un caso particular de árboles cardinales son los *árboles binarios*, uno de las clases de árboles más típicas, y que merecen ser estudiados en profundidad.

14: Probablemente el más importante de todos los problemas de estructuras de datos.

Definición 5.4.2 El conjunto \mathcal{A}_B de todos los posibles árboles binarios enraizados formados a partir de un conjunto de elementos S se construye con las siguientes reglas:

Caso Base: $\square \in \mathcal{A}_B$, es el árbol binario vacío.

Paso Inductivo: Si $B_1, B_2 \in \mathcal{A}_B$ tienen $n_1 \geq 0$ y $n_2 \geq 0$ elementos, respectivamente, tal que $B_1 \cap B_2 = \emptyset$, y $x \in S \setminus (A_1 \cup A_2)$, entonces

$$\langle B_1, x, B_2 \rangle \in \mathcal{A}_B,$$

y tiene $n = n_1 + n_2 + 1$ elementos.

En el árbol $\langle B_1, x, B_2 \rangle$ definido arriba, el elemento x es conocido como la *raíz del árbol*, el árbol B_1 es el *subárbol izquierdo* de x , mientras que B_2 es el *subárbol derecho*. Si x_1 es la raíz de B_1 y x_2 es la raíz de B_2 , llamamos a x_1 el *hijo izquierdo* de x , mientras que x_2 es el *hijo derecho*. Si alguno de los hijos de x es un árbol vacío, se dice que el hijo correspondiente es vacío. A los elementos \square del árbol se los conoce como elementos externos —o, indistintamente, nodos externos. El resto de los elementos del árbol se llaman elementos internos —o nodos internos.

Definimos a continuación la altura h de un árbol binario B , usando inducción sobre la estructura del árbol:

- *Caso base:* si $B = \square$, la altura es $h(\square) = -1$.
- *Paso inductivo:* si $B = \langle B_1, x, B_2 \rangle$, la altura es $h(\langle B_1, x, B_2 \rangle) = 1 + \max\{h(B_1), h(B_2)\}$.

Las principales operaciones que debe implementar un árbol binario son:

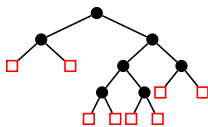
HIJO_IZQ(x): devuelve el hijo izquierdo del elemento x ; y

HIJO_DER(x): devuelve el hijo derecho del elemento x .

Además, se deben permitir inserciones de nuevos elementos y borrado de elementos existentes.

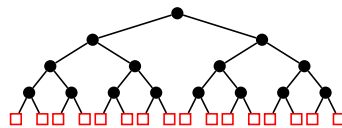
La implementación clásica de árboles binarios almacena los elementos internos en nodos similares a las listas enlazadas —asignados dinámicamente. El nodo correspondiente a un elemento x almacena el valor de x y dos punteros, uno para cada hijo. Si alguno de los hijos de un nodo interno es un nodo externo, el puntero correspondiente es *nil*.

Caracterización de Árboles Binarios. Un árbol binario es *completo* si cada uno de sus elementos tiene 0 o 2 hijos, tal como ocurren en el siguiente ejemplo:

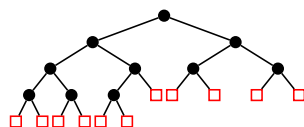


Por otro lado, un árbol binario de altura h es *perfecto* si todos sus niveles están llenos. Esto es, todos los nodos hasta el nivel $h - 1$ tienen

grado 2, mientras que los nodos en el nivel h tienen grado 0. El siguiente es un ejemplo de árbol perfecto de altura $h = 3$:



Finalmente, un árbol binario de altura h es *casi perfecto* si es un árbol perfecto hasta el nivel $d - 1$, y los elementos del último nivel (h) están lo más a la izquierda posible. Note que un árbol perfecto es un caso particular de árbol casi perfecto. El siguiente es un ejemplo de árbol casi perfecto para $h = 3$:



Propiedades de los Árboles Binarios. Demostramos a continuación propiedades importantes de los árboles binarios. Dado un árbol binario con n elementos internos, e elementos externos, y altura h , se cumplen las siguientes propiedades:

1. $e = n + 1$. Se puede demostrar por inducción sobre n :
 - *Caso base*: para $n = 0$, se tiene el árbol \square , con $n + 1 = 1$ elemento externo.
 - *Hipótesis inductiva*: el teorema es verdadero para todo árbol con hasta $n - 1$ elementos internos.
 - *Paso inductivo*: Sea $B = \langle B_1, x, B_2 \rangle$ un árbol de $n \geq 1$ elementos internos. Sean n_1 y n_2 el número de elementos internos en los árboles B_1 y B_2 , respectivamente. Por lo tanto, $n = n_1 + n_2 + 1$. Note además que $n_1 < n$ y $n_2 < n$. Eso significa que la hipótesis inductiva se cumple para B_1 y B_2 , por lo que tienen $n_1 + 1$ y $n_2 + 1$ elementos externos, respectivamente. Eso implica que B tiene $n_1 + 1 + n_2 + 1 = n + 1$ elementos externos, demostrando el teorema.
2. Si el árbol es perfecto $\Rightarrow n = 2^{h+1} - 1$. Se puede demostrar por inducción sobre h :
 - *Caso base*: para $h = 0$, el árbol tiene $n = 1$ elemento interno. Dado que $2^1 - 1 = 1$, la propiedad se cumple para $h = 0$.
 - *Hipótesis inductiva*: un árbol perfecto de altura $h - 1$ tiene $2^h - 1$ elementos.
 - *Paso inductivo*: note que cualquier árbol perfecto de altura $h > 0$ está formado por dos árboles perfectos de altura $h - 1$, que son los hijos de la raíz del árbol. Por lo tanto, para esos dos subárboles perfectos se cumple la hipótesis inductiva, por lo tanto la cantidad total de elementos en ellos es $2 \cdot (2^h - 1) = 2^{h+1} - 2$. A eso hay que sumarle 1, por el elemento raíz del árbol, así que la cantidad de elementos que tiene es $2^{h+1} - 1$, demostrando la propiedad.
3. $h + 1 \leq n \leq 2^{h+1} - 1$. Se puede verificar fácilmente considerando los dos árboles binarios extremos con n elementos. La idea es

acotar la cantidad de elementos internos, dada la altura del árbol. Para la cota inferior, considere un árbol binario lineal, en el que cada elemento tiene un único hijo. Ese es el árbol binario de altura h que tiene la menor cantidad de elementos: $h + 1$. Para la cota superior, note que un árbol perfecto de altura h es el árbol binario de altura h con la mayor cantidad posible de elementos internos. La propiedad anterior prueba que $n = 2^{h+1} - 1$ en ese caso.

4. $\lg(n + 1) - 1 \leq h \leq n - 1$. Se puede probar de manera similar a la propiedad anterior, pero ahora acotando la altura del árbol binario de n elementos internos.

Árboles Binarios de Búsqueda

Los *árboles binarios de búsqueda* —ABB, para abreviar— son un tipo particular de árbol binario utilizado principalmente para resolver el problema de pertenencia en un conjunto S sobre el que se ha definido una relación de orden total \leq . Tal como en los Algoritmos 2 y 3, resolveremos la pertenencia mediante una búsqueda en el conjunto. Para permitir una búsqueda eficiente, los elementos del árbol deben respetar el orden indicado en la siguiente definición.

Definición 5.4.3 Sea S un conjunto sobre el que se ha definido un orden total \leq . El conjunto \mathcal{A}_{BB} de todos los posibles árboles binarios de búsqueda sobre S se define a partir de las siguientes reglas inductivas:

Caso Base: $\square \in \mathcal{A}_{BB}$, es el árbol binario de búsqueda vacío.

Paso Inductivo: Si $B_1, B_2 \in \mathcal{A}_{BB}$ tal que $B_1 \cap B_2 = \emptyset$, y sea $x \in S \setminus (B_1 \cup B_2)$, entonces

$$\langle B_1, x, B_2 \rangle \in \mathcal{A}_{BB} \iff \max\{B_1\} \leq x \leq \min\{B_2\}.$$

En otras palabras, $\langle B_1, x, B_2 \rangle$ es un ABB si B_1 y B_2 son ABBs, y además se cumple que todos los elementos de B_1 son \leq que x , y x es \leq que todos los elementos de B_2 .

La Figura 5.12 muestra un posible ABB para el conjunto $S = \{5, 10, 15, 18, 20, 33, 38, 39, 47, 51\}$, usando la relación de orden total \leq sobre los naturales. En la figura, los nodos externos del árbol se muestran como cuadrados rojos.

Estudiamos a continuación cómo resolver el problema de pertenencia en un conjunto dinámico S ¹⁵ usando un ABB. Mostraremos también cómo resolver otras consultas de interés sobre un conjunto. En lo que sigue, sea $B = \langle B_1, x, B_2 \rangle$ un ABB que representa al conjunto S .

Pertenencia. Para determinar si $y \in S$ o no, debemos buscarlo en B . Si B es el árbol vacío \square , la búsqueda se detiene sin éxito: obviamente, $y \notin S$. En otro caso, se compara a y con la raíz x , teniendo en cuenta los casos:

- $y = x$: La búsqueda finaliza, ya que $y \in S$.

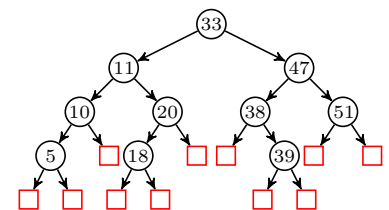


Figura 5.12: Un posible ABB para el conjunto $\{5, 10, 15, 18, 20, 33, 38, 39, 47, 51\}$ de $n = 10$ elementos y usando el orden total \leq sobre los naturales. Los nodos internos están representados con círculos, mientras que los nodos externos están representados con cuadrados rojos.

15: En un conjunto dinámico uno puede agregar nuevos elementos al conjunto, y eliminar elementos existentes.

- $y < x$: Esto significa que, de pertenecer al conjunto, y está almacenado en el subárbol izquierdo B_1 . La búsqueda continúa en dicho subárbol, siguiendo las mismas reglas aquí explicadas.
- $y > x$: De forma similar al caso anterior, la búsqueda debe continuar dentro del subárbol derecho B_2 .

Note que si $y \notin S$, la búsqueda alcanzará un nodo externo \square del árbol, tal como lo ilustra el siguiente ejemplo.

Ejemplo 5.4.3 Si buscamos el elemento $y = 25$ en el ABB de la Figura 5.12, debemos bajar primero por la izquierda del 33, luego por la derecha del 11, y finalmente por la derecha del 20, para finalizar en ese nodo externo. Eso indica que el 25 no está en el árbol, ya que agotamos todas las posibilidades en la búsqueda sin encontrarlo.

En general, y debido al orden de los elementos del ABB, para buscar un elemento y se sigue un único camino dentro del árbol, desde la raíz hasta el nodo que almacena a y —o, eventualmente, hasta un nodo externo cuando $y \notin S$. Ése es el único camino que puede contener al elemento buscado, por lo que si llegamos a un nodo externo sin encontrarlo estaremos seguros de que no está en el conjunto. En cada nodo del árbol, es posible descartar ramas completas del árbol de forma segura, haciendo una única comparación entre elementos. Esto es importante para la eficiencia del algoritmo, ya que trata de evitar comparar con todos los elementos del conjunto durante la búsqueda —recuerde los Algoritmos 2 y 3.

Para un ABB de altura h , el tiempo de búsqueda en el peor caso es de $h + 1$ comparaciones ¹⁶. En el ABB de la Figura 5.12, el peor caso corresponde a buscar los elementos 5, 18, o 39, realizando 4 comparaciones. En general, el tiempo de búsqueda es de $O(h)$ comparaciones, por lo que ABBs con forma balanceada son preferibles por sobre los árboles más altos ¹⁷. Más adelante, acotaremos h para un ABB.

Es importante notar que este mismo proceso de búsqueda se puede aplicar para el problema del diccionario —que definimos cuando implementamos árboles cardinales. En ese caso, cada elemento del árbol es un par (x, i) , siendo x la clave e i la información satélite asociada a x . El orden de los nodos del ABB debe hacerse con respecto a x ¹⁸, ya que es lo que se usa para buscar. De hecho, los arreglos asociativos `map` de C++ están implementados de esta forma, utilizando una variante de ABB que garantizan eficiencia de búsqueda.

Máximo y Mínimo. Para encontrar el mínimo y máximo de un conjunto usando un ABB, usaremos también una búsqueda sobre el árbol —aunque diferente a la usada para la pertenencia. De hecho, en este caso estamos buscando un elemento que no conocemos. Sin embargo, para el caso del máximo hay una propiedad que se cumple para la raíz x del ABB: $x \leq \max\{S\}$, por lo que la búsqueda debe continuar en el subárbol derecho B_2 . Dentro del ABB B_2 procedemos de la misma manera, ya que el elemento buscado $\max\{S\}$ será obviamente \geq a la raíz del subárbol B_2 . La búsqueda continúa de la misma manera, descendiendo siempre por la derecha, hasta que el hijo derecho del nodo actual sea vacío. Por ejemplo, para buscar el máximo del ABB de

16: Es decir, el peor caso es buscar el elemento que está más distante de la raíz.

17: En este contexto, *árbol balanceado* significa un árbol que tiende a parecerse a un árbol perfecto y, por lo tanto, tiene altura cercana a $\lg n$.

18: Asumiendo que existe una relación de orden total sobre esos valores.

la Figura 5.12, descendemos por la derecha hasta llegar al nodo que contiene al 51, que es el máximo. Para buscar el mínimo se procede de forma similar, pero siguiendo la rama de más a la izquierda de B. El tiempo de ejecución es, nuevamente, de $O(h)$ comparaciones.

Inserción. Para agregar un nuevo elemento y a S , debemos insertarlo en el ABB B —que, luego de la operación, representará al conjunto $S \cup \{y\}$. De hecho, comúnmente los ABBs se construyen por sucesivas inserciones, comenzando desde un árbol vacío, que representa al conjunto vacío. La inserción de cada elemento debe hacerse en el lugar adecuado del ABB, manteniendo el orden. La idea es que una posterior consulta de pertenencia —u otro tipo de búsqueda— lo encuentre. Todas las inserciones en un ABB se realizarán en nodos externos del mismo, los que serán reemplazados por un nuevo nodo interno. Esto es, los \square del árbol son potenciales puntos de inserción¹⁹. Por ejemplo, para insertar el 25 en el ABB de la Figura 5.12 manteniendo el orden del ABB, el único posible punto de inserción es el \square que es hijo derecho del 20.

Para determinar el punto de inserción del nuevo elemento y y asegurar que no se insertan elementos que ya existen en el conjunto —un conjunto no admite elementos repetidos—, la inserción debe ser precedida por una búsqueda del elemento y a insertar. Hay dos posibles resultados para dicha búsqueda:

- *Es exitosa:* significa que $y \in S$, por lo que la inserción fracasa.
- *Es infructuosa:* significa que $y \notin S$, lo que nos asegura que no agregaremos elementos repetidos. Pero, además, nos da información respecto al punto de inserción: el \square en donde la búsqueda fracasa es el único punto de inserción posible para y . Dicho nodo externo debe reemplazarse por $\langle \square, y, \square \rangle$.

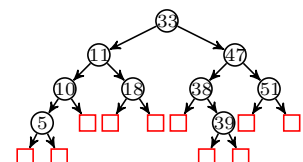
Respecto al tiempo de ejecución de una inserción, note que el mismo depende de la búsqueda previa a la inserción, que ya dijimos que ejecuta $O(h)$ comparaciones, siendo h la altura del ABB.

Borrado. Borrar un elemento y de un ABB es algo más complicado que la inserción. La razón es que, a diferencia de las inserciones —que ocurren en los nodos externos del árbol—, los borrados ocurren en cualquier nodo. Por ejemplo, uno podría querer borrar el 33 del ABB de la Figura 5.12. El problema es cómo reestructurar el árbol al eliminar uno de sus nodos, y seguir manteniendo el orden entre sus elementos. Tal como para las inserciones, el borrado debe ser precedido de una búsqueda. Si ésta es infructuosa, significa que el elemento a borrar no pertenece al conjunto, y el borrado fracasa. Por otro lado, si la búsqueda es exitosa, nos permite encontrar el nodo y a borrar. En particular, hay 3 casos de borrado, dependiendo de la ubicación de y en el árbol:

Caso 1: Ambos hijos de y son nodos externos, por lo que simplemente se puede eliminar el nodo, transformándolo en un nodo externo.

Caso 2: Uno de los hijos de y es externo, mientras que el otro es interno. En este caso, simplemente hay que reemplazar a y en su padre por el hijo de y que es interno. Note que con este cambio en el ABB no se modifica el orden de los elementos

19: Use el ABB de la Figura 5.12 para notar que los $n + 1$ nodos externos de un ABB corresponden a los $n + 1$ posibles puntos de inserción en un conjunto ordenado: el \square de la izquierda corresponde a todos los elementos menores que 5; el siguiente corresponde a los elementos mayores que 5 y menores que 10; y así siguiendo.



del mismo. La figura al margen muestra el ABB resultante de borrar el 20 del árbol de la Figura 5.12. Note cómo su nodo es simplemente reemplazado por su hijo izquierdo, 18. Dado que todos los elementos del subárbol del 18 son mayores que 11, no se modifica el orden de los elementos del ABB.

Caso 3: Ambos hijos de y son internos. En este caso, reemplazaremos y con otro elemento y' del ABB, el cual esté en un nodo que sea más simple de borrar —e.g., que sea alguno de los dos casos anteriores. Para respetar el orden del ABB —y así evitar reorganizar el árbol completamente—, y' debe cumplir con:

$$\text{máx}\{B_1\} \leq y' \leq \text{mín}\{B_2\}.$$

Hay sólo dos elementos en el ABB que cumplen con esta condición:

- $\text{máx}\{B_1\}$, y
- $\text{mín}\{B_2\}$.

Afortunadamente, para ambos elementos se cumple lo siguiente:

- El nodo correspondiente a $\text{máx}\{B_1\}$ tiene, al menos, su hijo derecho vacío. Supongamos, por contradicción, que el hijo derecho es un nodo interno. Eso significa que almacena un valor mayor a $\text{máx}\{B_1\}$, lo cual es una contradicción.
- El nodo correspondiente a $\text{mín}\{B_2\}$ tiene, al menos, su hijo izquierdo vacío, lo cual puede demostrarse de forma similar.

Por ejemplo, supongamos que se quiere eliminar $y = 33$ del ABB de la Figura 5.12. Entonces, el elemento $\text{máx}\{B_1\} = 20$ tiene el hijo derecho vacío, mientras que $\text{mín}\{B_2\} = 38$ tiene el hijo izquierdo vacío. Supongamos entonces que $y' \leftarrow \text{máx}\{B_1\}$. La idea es trasladar el valor y' al nodo que almacena el elemento y a borrar. Luego, hay que borrar el nodo original que almacenaba a y' . Por lo expuesto anteriormente, ese borrado corresponde ya sea al Caso 1 o Caso 2, explicados anteriormente. La Figura 5.13 muestra el ABB resultante de borrar el 33 del ABB de la Figura 5.12.

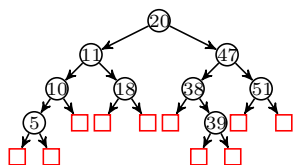


Figura 5.13: Borrado del elemento 33 del ABB de la Figura 5.12, el cual es reemplazado por el 20, que es el máximo elemento de su subárbol izquierdo.

Análisis de la Altura Máxima de un ABB. A causa del proceso de inserción, la forma de un ABB depende del orden en que sus elementos fueron insertados. Por ejemplo, en el ABB de la Figura 5.12, el 33 fue el primer elemento insertado, ya que está en la raíz. Luego, el siguiente elemento insertado fue el 11 o el 47, y así siguiendo. Este es un aspecto muy importante de los ABB, y que influye en la eficiencia de búsqueda —de la cual dependen todas las operaciones que hemos mencionado.

Ya hemos dicho que el tiempo de búsqueda en un ABB es de $O(h)$, siendo h la altura del árbol. Desafortunadamente, la altura de un ABB de n elementos puede ser hasta $n - 1$: piense en insertar los n elementos del conjunto en orden, desde el menor al mayor. El árbol resultante será lineal, en donde el hijo izquierdo de cada nodo es externo, estando presente sólo el hijo derecho. El mismo efecto se logra si se insertan los elementos en orden inverso, o de forma alternada mayor-menor —que produciría un árbol en zigzag—, entre otros muchos órdenes que producen un ABB de altura $n - 1$. Como consecuencia, el tiempo de búsqueda en un ABB es de $O(n)$ comparaciones, no diferenciándose en este sentido de los Algoritmos 2 y 3. Sin embargo, ésta cota de

peor caso es muy pesimista, tal como mostraremos en la Sección 6.1: si los elementos son insertados de forma aleatoria en un ABB, el árbol resultante tiende a ser más balanceado y por lo tanto el tiempo de búsqueda es mucho más eficiente que n comparaciones.

La Cantidad de Árboles de cada Tipo

El conteo de objetos o estructuras combinatorias distintas de cierto tipo es una de las tareas más importantes en combinatoria. Contaremos a continuación cuántos árboles distintos se pueden formar con n nodos, para los tipos de árboles estudiados anteriormente. Al contar, vamos a considerar solamente la *topología* de los árboles —esto es, sólo distinguiremos su forma, sin considerar los valores de cada nodo. De hecho, podemos pensar los n nodos como círculos negros indistinguibles, y queremos saber cuántos árboles distintos podemos formar a partir de ellos.

Árboles Binarios. Sea S un conjunto sobre el que se define el conjunto \mathcal{A}_B de todos los árboles binarios cuyos elementos pertenecen a S . Definimos el conjunto $\mathcal{A}_B(n) \subseteq \mathcal{A}_B$ que contiene a todos los árboles binarios con exactamente $n \geq 0$ nodos internos. La cardinalidad de $\mathcal{A}_B(n)$ corresponde a la cantidad de árboles binarios distintos con n nodos internos, y se define como:

$$\mathcal{C}(n) = |\mathcal{A}_B(n)| = \frac{1}{n+1} \binom{2n}{n}, \quad (5.3)$$

conocido como el *número de Catalan*. La sucesión de dichos números para $n \geq 0$ es $\langle 1, 1, 2, 5, 14, 42, 132, \dots \rangle$. La Figura 5.14 muestra los $\mathcal{C}(3) = 5$ árboles binarios distintos con 3 elementos internos. Para $n \rightarrow \infty$, se cumple

$$\mathcal{C}(n) \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}, \quad (5.4)$$

lo cual se puede demostrar usando la aproximación de Stirling para $n!$ sobre la Ecuación (5.3).

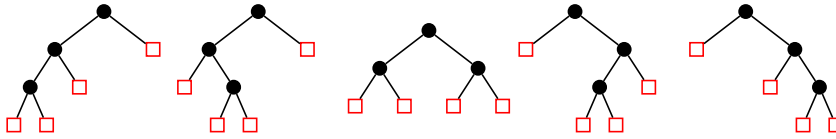


Figura 5.14: Los 5 árboles binarios distintos con $n = 3$ elementos internos.

Árboles Cardinales. La fórmula para árboles binarios puede generalizarse para árboles cardinales de grado k , de la siguiente manera:

$$\mathcal{C}(n, k) = \frac{1}{kn+1} \binom{kn+1}{n}. \quad (5.5)$$

Árboles Ordinales. Mostramos a continuación una biyección entre las topologías de los árboles binarios y árboles ordinales, que permitirá concluir que la cantidad de árboles binarios distintos con n elementos

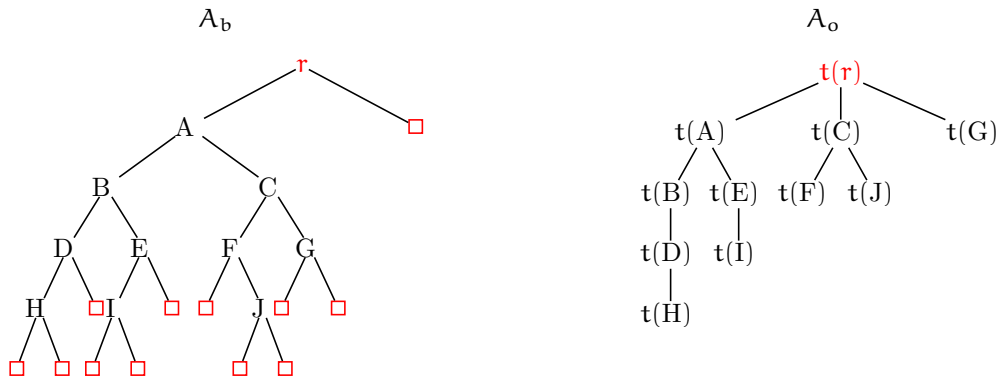


Figura 5.15: Un árbol binario A_b con raíz falsa r (izquierda) y el correspondiente árbol ordinal A_o (derecha). Cada nodo x en A_b tiene su correspondiente nodo $t(x)$ en A_o .

internos es igual a la cantidad de árboles ordinales distintos con $n + 1$ elementos —recuerde el resultado del Lema 2.2.1. La idea es mostrar cómo cualquier árbol binario A_b de n elementos internos puede ser transformado en un árbol ordinal A_o de $n + 1$ elementos, y viceversa.

Mapeo $f : A_b \mapsto A_o$: cada nodo x en A_b corresponderá a un nodo $t(x)$ en A_o . Para simplificar la exposición, agregamos una raíz ficticia r a A_b , tal que la raíz original de A_b es el hijo izquierdo de r —no es necesario representar este nodo en A_b , ya que puede simularse, manteniendo así sus n nodos originales. El nodo r no tiene hijo derecho. Sea $t(r)$ la raíz de A_o . Luego, para cualquier elemento interno x de A_b —incluyendo la raíz ficticia r — se cumple:

- el hijo izquierdo de x corresponde al primer hijo de $t(x)$ en A_o , y
- el hijo derecho de x corresponde al siguiente hermano de $t(x)$ en A_o .

Luego, las operaciones sobre A_b pueden simularse sobre A_o de la siguiente manera:

- $A_b.HIJO_IZQ(x) = A_o.PRIMER_HIJO(t(x))$, y
- $A_b.HIJO_DER(x) = A_o.SIG_HIJO(t(x))$.

Con este mapeo, a todo árbol binario le corresponde un único árbol ordinal, por lo que es una función total. La Figura 5.15 muestra un ejemplo de árbol binario y su correspondiente árbol ordinal.

Mapeo $f^{-1} : A_o \mapsto A_b$: Para concluir la biyección, definimos el mapeo inverso f^{-1} . Todo nodo $t(x)$ en A_o tendrá su nodo correspondiente x en A_b . Obviamente, la raíz $t(r)$ de A_o corresponde a la raíz ficticia r de A_b . Luego, para un nodo $t(x)$ de A_o , sea $t(x_i)$ el i -ésimo hijo de x . Entonces:

- si $i = 1$, x_1 es el hijo izquierdo de x en A_b , y
- si $i > 1$, x_i es el hijo derecho de x_{i-1} en A_b .

Este es el mapeo inverso al explicado anteriormente. Luego, las operaciones sobre A_o pueden simularse de la siguiente manera:

- $A_o.PRIMER_HIJO(t(x)) = A_b.HIJO_IZQ(x)$, y
- $A_o.SIG_HIJO(t(x)) = A_b.HIJO_DER(x)$.

Con este mapeo, a todo árbol ordinal le corresponde un único árbol binario, por lo que f^{-1} es una función total.

De esta forma, f es no sólo una función total, sino que también es inyectiva y sobreyectiva —porque f^{-1} es una función total. Eso significa que el mapeo f es una biyección, y que entonces hay

$$\mathcal{O}(n+1) = \frac{1}{n+1} \binom{2n}{n} \quad (5.6)$$

árboles ordinales distintos de $n+1$ nodos. La Figura 5.16 muestra los 5 árboles ordinales posibles con 4 nodos, que son equivalentes a los 5 árboles binarios distintos con 3 nodos de la Figura 5.14.

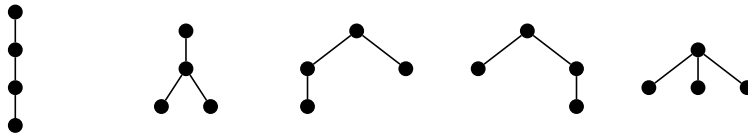


Figura 5.16: Los 5 árboles ordinales distintos con $n = 4$ nodos. Cada árbol corresponde a un árbol binario de 3 nodos de la Figura 5.14.

5.5. Grafos

Los *grafos* son un concepto matemático conocido desde aproximadamente 1736, cuando Leonhard Euler publicó su conocido artículo *Seven Bridges of Königsberg*, siendo actualmente fundamentales en computación y matemática, además de haber sido aplicados con éxito en diversas áreas de la ciencia e ingeniería, como biología, física, e investigación de operaciones, por nombrar algunas.

Definición 5.5.1 *Un grafo es un par ordenado $G = (V, E)$ tal que:*

- *El conjunto $V = \{v_1, \dots, v_n\}$ es el conjunto de vértices del grafo.*
- *La relación binaria $E \subseteq V \times V$ es el conjunto de arcos (o aristas) del grafo.*

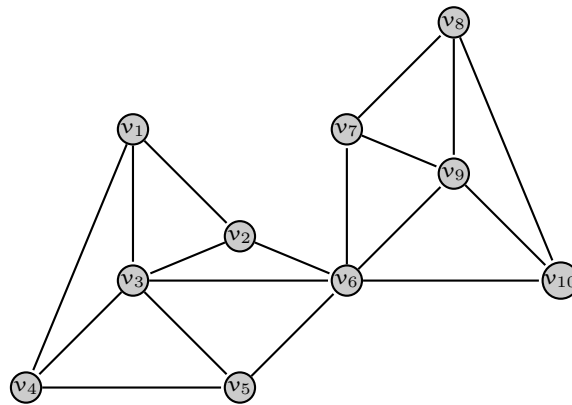
Cada arista $(v_i, v_j) \in E$ relaciona al par de vértices v_i y v_j del grafo. Diremos que v_i y v_j son vértices *adyacentes* (o *vecinos*). Alternativamente, diremos que dicho arco incide en los vértices v_i y v_j .

Generalmente, los grafos se usan para modelar situaciones o realidades que involucran datos y sus relaciones. En general, los vértices se usan para representar los datos de la realidad, mientras que los arcos se usan para representar las relaciones entre esos datos. Por ejemplo, los vértices podrían representar a los usuarios de un medio social, mientras que los arcos representan las relaciones de amistad o seguimiento entre ellos.

Al dibujar un grafo, los vértices se representan como puntos o círculos en el plano, mientras que las aristas son líneas que unen los puntos correspondientes, tal como lo ilustra el siguiente ejemplo.

Ejemplo 5.5.1 La Figura 5.17 muestra un grafo con conjunto de

Figura 5.17: Un grafo con 10 vértices y 18 arcos.



vértices

$$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\},$$

y aristas

$$E = \{(v_1, v_4), (v_1, v_3), (v_1, v_2), (v_2, v_3), (v_2, v_6), (v_6, v_3), (v_6, v_5), (v_5, v_3), (v_5, v_4), (v_4, v_3), (v_6, v_7), (v_6, v_9), (v_6, v_{10}), (v_7, v_9), (v_9, v_{10}), (v_7, v_8), (v_9, v_8), (v_8, v_{10})\}.$$

Tipos de Grafos

Existen principalmente 2 tipos de grafos, que definiremos a continuación.

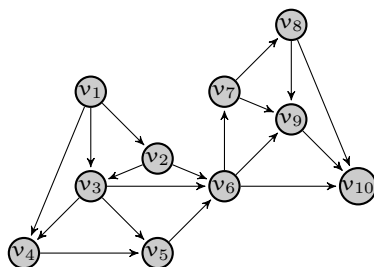


Figura 5.18: Un digrafo.

Grafos Dirigidos. En ciertas aplicaciones es importante saber en qué dirección se da la relación entre dos vértices del grafo. Por ejemplo, si se modela una ciudad y sus calles, es importante indicar el sentido de las mismas. Similarmente, la relación de seguimiento entre usuarios de un medio social como twitter o instagram tiene una dirección: es importante saber qué usuario sigue quién. Para modelar esos casos adecuadamente, se agrega dirección a los arcos del grafo, de forma tal que $(v_i, v_j) \in E$ implica que la relación tiene sentido $v_i \rightarrow v_j$. Llamaremos *grafos dirigidos*, o simplemente *digrafos*, a este tipo de grafos. Al dibujar un digrafo, usaremos flechas en los arcos indicando la dirección de la relación.

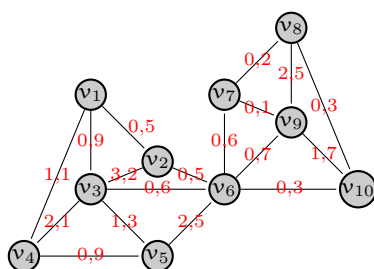


Figura 5.19: Un grafo no dirigido con pesos.

Grafos No Dirigidos. En otras aplicaciones, la relación entre los vértices de un grafo se da en ambos sentidos. Por ejemplo, las relaciones de amistad en medios sociales como facebook, en donde dos usuarios se ponen de acuerdo para establecer una relación de amistad. En ese caso, hablamos de *grafos no dirigidos* (o simplemente *grafos*). En un grafo no dirigido, los arcos se dibujan como líneas sin dirección, como en el grafo de la Figura 5.17.

Grafos con Peso

En algunos escenarios es necesario asociar un *peso* numérico a los arcos del grafo. Por ejemplo, piense en un grafo que modela ciudades y vuelos entre ellos. Se podría asociar un peso a cada arco, indicando el costo monetario del boleto aéreo, o la duración del viaje en horas. Llamaremos *grafos con pesos* o *grafos pesados* a este tipo de grafos. Un grafo con peso puede ser dirigido o no dirigido. Formalmente, el conjunto de aristas de un grafo con pesos se define como una relación ternaria $E \subseteq V \times V \times \mathbb{R}$, tal que $(v_i, v_j, w) \in E$ significa que existe una arista entre v_i y v_j , la cual tiene peso w . Al dibujar este tipo de grafos, agregaremos el peso correspondiente sobre cada arco.

Terminología

Definimos a continuación terminología importante relacionada con los grafos, y que necesitaremos más adelante.

Grado de un Vértice. El *grado* o *valencia* de un vértice es la cantidad de arcos que inciden en él. Formalmente, puede definirse como:

$$\text{grado}(v_i) \equiv |\{(x, y) \in E \mid x = v_i \vee y = v_i\}|.$$

Por ejemplo, en el grafo de la Figura 5.17, el vértice v_6 tiene grado 6, mientras que v_8 tiene grado 3. En el caso de digrafos, se debe distinguir entre grado de entrada y de salida de un vértice, tal como a continuación:

$$\text{grado}^-(v_i) \equiv |\{(v_j, v_i) \in E\}|,$$

y

$$\text{grado}^+(v_i) \equiv |\{(v_i, v_j) \in E\}|.$$

Caminos. Una secuencia de vértices $\langle v_{i_1}, v_{i_2}, \dots, v_{i_\ell} \rangle$ en un grafo es un camino de largo $\ell - 1$ si $(v_{i_j}, v_{i_{j+1}}) \in E$ para todo $1 \leq j < \ell$. Por ejemplo, la secuencia $\langle v_4, v_3, v_1, v_2, v_6 \rangle$ es un camino de largo 4 en el grafo de la Figura 5.17. Sin embargo, esa misma secuencia no es un camino en el digrafo de la Figura 5.18, dado que no respeta las direcciones de los arcos. Para dicho digrafo, la secuencia $\langle v_1, v_2, v_3, v_4, v_5, v_6, v_{10} \rangle$ es un camino de largo 7 desde v_1 a v_{10} .

Algunos tipos particulares de caminos son los siguientes:

- Un camino es *cerrado* si el primer y el último vértice son el mismo. Por ejemplo, ninguno de los caminos mencionados anteriormente son cerrados. Por otro lado, la secuencia $\langle v_1, v_4, v_5, v_3, v_2, v_1 \rangle$ es un camino cerrado en el grafo de la Figura 5.17.
- Un camino es *simple* si todos sus vértices son distintos, excepto quizás el primero y el último si el camino es cerrado. Todos los caminos mencionados en los ejemplos anteriores son simples, mientras que el camino $\langle v_1, v_3, v_5, v_6, v_2, v_3, v_4 \rangle$ no es simple en el grafo de la Figura 5.17.

- Un camino es *euleriano* si pasa por todos los arcos del grafo una única vez.
- Un camino es *hamiltoniano* si visita todos los vértices del grafo una única vez.

Para grafos con peso, el *peso de un camino* es la suma de los pesos de los arcos que lo componen. Por ejemplo, el camino $\langle v_4, v_3, v_1, v_2, v_6 \rangle$ en el grafo de la Figura 5.19 tiene peso 4.

Ciclos. Un *ciclo* es un camino cerrado simple. Por ejemplo, $\langle v_4, v_5, v_6, v_2, v_3, v_4 \rangle$ y $\langle v_6, v_9, v_8, v_7 \rangle$ son dos posibles ciclos en el grafo de la Figura 5.17 —hay varios ciclos más en el grafo. Un grafo *acíclico* es uno que no tiene ciclos. Por ejemplo, el digrafo de la Figura 5.18 es acíclico. Un *ciclo euleriano* es un ciclo que visita todos los arcos del grafo una única vez. Por otro lado, un *ciclo hamiltoniano* es uno que visita todos los vértices del grafo una única vez.

Orden Topológico. El *orden topológico* u *ordenamiento topológico* de un grafo dirigido es un ordenamiento de sus vértices tal que para cada arco $v_i \rightarrow v_j$ del digrafo, v_i aparece antes que v_j en el orden. Por ejemplo, un posible orden topológico para el digrafo de la Figura 5.18 es $v_1, v_3, v_2, v_5, v_4, v_6, v_7, v_{10}, v_9, v_8$. Es importante notar que todo grafo dirigido acíclico —como es el caso del mostrado en la Figura 5.18— tiene al menos un ordenamiento topológico.

Subgrafos. Dado un grafo $G = (V, E)$, un grafo $S = (V', E')$ es subgrafo de G si cumple con:

- $V' \subseteq V$, y
- $E' = \{(v_i, v_j) \in E \mid v_i, v_j \in V'\}$.

Componentes Conexas (Grafos no Dirigidos). Un grafo no dirigido es *conexo* si para cualquier par de vértices v_i y v_j hay al menos un camino entre ellos en el grafo. Por ejemplo, el grafo de la Figura 5.17 es conexo. Una componente conexa de un grafo es un subgrafo conexo maximal. Si un grafo no es conexo, diremos que está formado por $c > 1$ componentes conexas.

Componentes Conexas (Grafos Dirigidos). Un grafo dirigido es *fuertemente conexo* si para cualquier par de vértices v_i y v_j hay al menos un camino entre ellos en el grafo ²⁰. Un grafo dirigido es *débilmente conexo* si es conexo al considerar sus arcos sin dirección —como si fuera un grafo no dirigido.

20: Recuerde que los caminos en un grafo dirigido toman en cuenta la dirección de los arcos.

Árboles. Un grafo $G = (V, E)$ es un *árbol* si cumple con ser conexo y sin ciclos. Aunque se use el mismo nombre, los árboles de la teoría de grafos no deben confundirse con los árboles enraizados estudiados anteriormente. La diferencia principal es que los árboles de la teoría de grafos no tienen un vértice distinguido que actúa como raíz.

Operaciones

Para poder definir algoritmos que operen sobre un grafo $G = (V, E)$, necesitamos implementar las siguientes operaciones.

- G.ADYACENTES(v_i):** permite acceder al conjunto de vecinos $\{v_j \mid (v_i, v_j) \in E \vee (v_j, v_i) \in E\}$ del vértice v_i . El acceso al conjunto es elemento a elemento, en algún orden arbitrario —aunque es un orden reproducible, en general—, similar a un iterador de lenguajes como C++. El uso típico es: **for** $v \in G.ADYACENTES(v_i)$ **do**.
- G.ES_ARCO(v_i, v_j):** devuelve verdadero si $(v_i, v_j) \in E$, falso en otro caso.
- G.NUM_ARCOS():** devuelve $|E|$, el número de arcos del grafo.
- G.NUM_VERT():** devuelve $|V|$, el número de vértices del grafo.
- G.AGREGAR_ARCO(v_i, v_j):** agrega el arco (v_i, v_j) , produciendo $E \leftarrow E \cup \{(v_i, v_j)\}$.
- G.BORRAR_ARCO(v_i, v_j):** elimina el arco (v_i, v_j) , produciendo $E \leftarrow E \setminus \{(v_i, v_j)\}$.
- G.PESO_ARCO(v_i, v_j):** si $(v_i, v_j) \in E$, devuelve el peso del arco. En otro caso, devuelve ∞ .
- G.MARCAR(v_i, c):** asigna la marca c al vértice v_i . Esta operación será importante al recorrer grafos, en particular para evitar caer indefinidamente en ciclos. Durante un recorrido, cada vértice visitado es marcado como tal, evitando de esta forma revisitarlo posteriormente. Asumiremos que $c \in \mathbb{N}$, y que cada vértice puede tener una única marca a la vez.
- G.OBTENER_MARCA(v_i):** devuelve la marca actual del vértice v_i .

Implementación de Grafos

Estudiamos a continuación las dos maneras principales utilizadas para representar la relación binaria $E \subseteq V \times V$ de un grafo $G = (V, E)$. Asumiremos en adelante que $v_i = i$, para $i = 1, \dots, |V|$ ²¹.

21: Eso significa $V = \mathbb{N}_{|V|}$.

Matriz de Adyacencia. La idea es representar la relación E usando una matriz booleana M de $|V| \times |V|$, definida como a continuación. Si G es un grafo no dirigido, definimos

$$M[i, j] = 1 \wedge M[j, i] = 1 \iff (v_i, v_j) \in E.$$

Así, la matriz M es simétrica para grafos no dirigidos. Si, por otro lado, el grafo es dirigido, tenemos

$$M[i, j] = 1 \iff (v_i, v_j) \in E.$$

Note que la matriz reserva espacio para cada uno de los posibles arcos que el grafo pudiese tener. Por lo tanto, el uso de espacio es de $|V|^2$ bits —i.e., $\lceil |V|^2/w \rceil$ celdas de memoria. Almacenamos, además, dos variables enteras, v —el número de vértices del grafo— y e —el número de arcos del grafo. Esto último permite NUM_VERT() y NUM_ARCOS() en tiempo $O(1)$. La Figura 5.20 muestra la matriz de adyacencia correspondiente al grafo de la Figura 5.17.

Respecto a las operaciones, tenemos que ES_ARCO(v_i, v_j) simplemente chequea si $M[i, j] = 1$ o no, AGREGAR_ARCO(v_i, v_j) hace $M[i, j] \leftarrow 1$,

$$M = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Figura 5.20: Matriz de adyacencia para el grafo de la Figura 5.17.

mientras que $\text{BORRAR_ARCO}(v_i, v_j)$ hace $M[i, j] \leftarrow 0$, todas ellas en tiempo $O(1)$. Para la operación $\text{PESO_ARCO}(v_i, v_j)$ en grafos con peso, una alternativa es definir la matriz M de tal forma que $M[i, j]$ almacene el peso del arco (v_i, v_j) —el uso de espacio es $|V|^2$ celdas de memoria, w veces más que la matriz booleana original. Para que las operaciones anteriores funcionen correctamente, se debe almacenar un peso inválido —e.g., ∞ — en las entradas que no corresponden a un arco del grafo. Si por las características de la aplicación no se dispone de un peso inválido —e.g., un caso en que los pesos sean cualquier valor entero de w bits—, se debe agregar una matriz booleana adicional para indicar las entradas válidas —con espacio adicional $\lceil |V|^2/w \rceil$ celdas.

Respecto a las operaciones $\text{MARCAR}(v_i, c)$ y $\text{OBTENER_MARCA}(v_i)$, es necesario mantener un arreglo adicional $m[1..n]$, tal que $m[i]$ almacena la marca correspondiente al vértice v_i . Entonces, $\text{MARCAR}(v_i, c)$ simplemente hace $m[i] \leftarrow c$, mientras que $\text{OBTENER_MARCA}(v_i)$ devuelve $m[i]$. Ambas operaciones toman tiempo $O(1)$.

Para la operación $\text{ADYACENTES}(v_i)$, se mantendrá un arreglo $N[1..n]$ de valores enteros, tal que $N[i]$ indica el vecino actual del vértice v_i . Inicialmente todos los valores del arreglo N estarán en 0. Para devolver a cada uno de los vecinos de v_i , la operación usa una variable auxiliar j que es inicializada como $j \leftarrow N[i]$. Luego, mientras $M[i, j] = 0$ hace $j \leftarrow j + 1$. Cuando la iteración se detiene —porque $M[i, j] = 1$ —, se hace $N[i] \leftarrow j$ y se devuelve el valor j como resultado de la operación. La anterior iteración también debe controlar que $j \leq |V|$. En caso de que esta condición deje de cumplirse, significa que hemos recorrido toda la fila i de M , y ya no hay más vecinos de v_i por reportar. En ese caso, se hace $N[i] \leftarrow 0$ nuevamente, para que futuros usos de la operación sobre v_i sean correctos. En el peor caso, hay que revisar una cantidad de entradas de la matriz que es proporcional a $|V|$ —i.e., toda la fila i de M —, por lo que el tiempo total de $\text{ADYACENTES}()$ es $\Theta(|V|)$ —lo cual es independiente de la cantidad de vecinos de v_i —, y el tiempo por vecino es $O(|V|)$.

Como conclusión, aunque las matrices de adyacencia permiten implementar en tiempo constante algunas de las operaciones de un grafo, éstas no son usualmente las más importantes en la mayoría de las aplicaciones reales, que usualmente necesitan recorrer el grafo de manera eficiente. Para eso se necesitan las operaciones que obtienen los vecinos de un vértice dado, las cuales no tienen garantía de ejecución eficiente

con esta representación. Otro aspecto importante es el elevado uso de espacio de la matriz. Usualmente, los grafos de interés son suficientemente poco densos, en el sentido de que la cantidad total de arcos es bastante menor que $|V|^2$. En esos casos, la cantidad cuadrática de espacio y el elevado tiempo de ejecución para manejar los vecinos de un vértice vuelven prohibitiva a este tipo de implementación. Las matrices de adyacencia sólo se usan en casos en que el grafo sea denso —i.e., $\sim |V|^2$ vértices— o tenga sólo unos pocos vértices —e.g., hasta unos pocos miles.

Listas de Adyacencia. Para cada vértice del grafo almacenaremos una lista de todos sus vértices adyacentes. Sea $\ell[1..|V|]$ el arreglo de dichas listas, de forma tal que

$$\ell[i] = \langle v_j \mid (v_i, v_j) \in E \vee (v_j, v_i) \in E \rangle$$

es la lista de vértices adyacentes al vértice v_i . La Figura 5.21 muestra el conjunto de listas de adyacencia para el digrafo de la Figura 5.18. En la figura se han usado listas enlazadas para representar las $\ell[i]$. Note que en esta definición de las listas de adyacencia, se cumple que $v_i \in \ell[j] \Leftrightarrow v_j \in \ell[i]$. Por ejemplo, $\ell[1] = \langle 2, 3, 4 \rangle$ significa que los arcos (v_1, v_2) , (v_1, v_3) , y (v_1, v_4) pertenecen a E . Además del arreglo de listas, mantenemos las variables v y e —que registran el valor actual de $|V|$ y $|E|$, respectivamente—, y el arreglo de marcas $m[1..|V|]$, tal como para las matrices de adyacencia.

El espacio utilizado por esta representación es de $\Theta(|V| + |E|)$ celdas de memoria, en donde $\Theta(|V|)$ corresponden al arreglo ℓ , mientras que $\Theta(|E|)$ corresponden a las listas²². Esto es bastante más eficiente, en general, que una matriz de adyacencia cuando el grafo es poco denso —que suele ser el caso más típico en la práctica.

Para implementar la operación $ES_ARCO(v_i, v_j)$, debemos chequear la pertenencia de v_j en la lista $\ell[i]$, lo cual toma tiempo $O(|V|)$ —por el recorrido de la lista, que puede tener tamaño $|V|$ en el peor caso²³. La operación $AGREGAR_ARCO(v_i, v_j)$ consiste en agregar v_j en la lista $\ell[i]$ —y v_i en la lista $\ell[j]$ si el grafo es no dirigido—, mientras que $BORRAR_ARCO(v_i, v_j)$ debe eliminar v_j de la lista $\ell[i]$ —y v_i de la lista $\ell[j]$ si el grafo es no dirigido. El tiempo es $O(|V|)$, por las mismas razones anteriores. Si el grafo es con pesos, junto con cada nodo v_j en la lista $\ell[i]$ se debe almacenar el correspondiente peso del arco. Luego, $PESO_ARCO(v_i, v_j)$ se implementa buscando v_j en la lista $\ell[i]$, y devolviendo el peso correspondiente. Nuevamente, el tiempo es $O(|V|)$.

Finalmente, para implementar $ADYACENTES(v_i)$ y permitir el acceso a los vecinos de v_i , simplemente hay que recorrer la lista $\ell[i]$, en tiempo $O(1)$ por vecino reportado. Esto hace que los recorridos en un grafo sea mucho más eficiente al usar este tipo de representación.

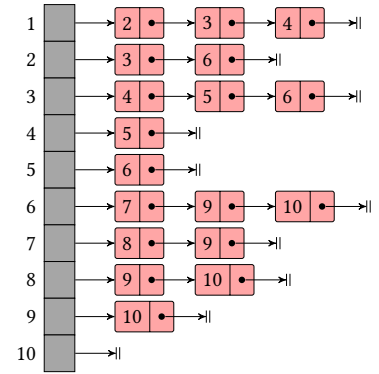


Figura 5.21: Listas de adyacencia (representadas como listas enlazadas) para el digrafo de la Figura 5.18.

22: Note que, en total, hay $|E|$ elementos en las listas.

23: Las listas podrían implementarse con alguna estructura que permita chequear la pertenencia más eficientemente, por ejemplo un ABB o algún tipo de árbol de búsqueda como los que estudiaremos más adelante.

Técnicas de Análisis de Algoritmos

6.1. Análisis de Caso Promedio

En general, el análisis de peor caso de un algoritmo es adecuado y entrega suficiente información respecto al comportamiento del algoritmo. Es el tipo de análisis a elegir cuando se necesita conocer el peor escenario —en términos de tiempo de ejecución— que puede ocurrir al ejecutar un algoritmo. Hay aplicaciones para las que esto es imprescindible, como los sistemas de tiempo real, en donde se debe garantizar que un algoritmo finalizará su ejecución siempre dentro de un rango de tiempo limitado.

Sin embargo, existen algoritmos que tienen un mal comportamiento de peor caso, aún cuando funcionan bien en la práctica. Esto puede indicar que el peor caso ocurre con muy poca probabilidad. En esos casos, un análisis de caso promedio es más adecuado, y da una mejor perspectiva de cómo se va a comportar un algoritmo en la práctica.

Para hacer un análisis de caso promedio, se deben considerar *todas* las posibles instancias del problema sobre las que tuviese que ejecutar el algoritmo, para luego promediar todos los tiempos de ejecución correspondientes. De manera formal, definimos:

Definición 6.1.1 (Tiempo de Ejecución de Caso Promedio) *Sea:*

- \mathcal{P} un problema abstracto cuyo conjunto de instancias es $\mathcal{I}_{\mathcal{P}}$;
- $n \in \mathbb{N}^0$;
- $\mathcal{I}_{\mathcal{P},n} = \{\mathcal{X} \in \mathcal{I}_{\mathcal{P}} \mid |\mathcal{X}| = n\}$ el conjunto de instancias de \mathcal{P} que tienen tamaño n ;
- A un algoritmo que resuelve el problema \mathcal{P} ;
- para una instancia $\mathcal{X} \in \mathcal{I}_{\mathcal{P}}$, sea $T_A(\mathcal{X})$ el tiempo de ejecución del algoritmo A para la instancia \mathcal{X} .

Si todas las instancias de tamaño n tienen la misma probabilidad de ocurrir, $\frac{1}{|\mathcal{I}_{\mathcal{P},n}|}$, el tiempo de ejecución de caso promedio para A sobre instancias de tamaño n se define como:

$$\bar{T}_A(n) \equiv \frac{1}{|\mathcal{I}_{\mathcal{P},n}|} \sum_{\mathcal{X} \in \mathcal{I}_{\mathcal{P},n}} T_A(\mathcal{X}).$$

Un análisis de caso promedio suele, en general, ser más complicado de realizar que un análisis de peor caso. La razón es que puede no ser trivial el hecho de considerar todas las posibles instancias del problema (las cuales, además, son infinitas en muchos casos). A continuación estudiamos diversos ejemplos de análisis de caso promedio, usando distintas variantes de problemas de búsqueda para ilustrar los conceptos.

Tiempo Promedio de Búsqueda Secuencial en Conjunto Desordenado

Estudiamos el problema de búsqueda en un conjunto desordenado, para el cual ya definimos un posible algoritmo asumiendo el conjunto almacenado en un arreglo (ver el Algoritmo 2). Si medimos el tiempo de ejecución en cantidad de comparaciones entre el elemento buscado y elementos del conjunto, ya sabemos que el peor caso de este algoritmo, para búsquedas exitosas e infructuosas, es n comparaciones. Para el análisis de caso promedio, consideramos todos los casos posibles de búsqueda. Para búsquedas infructuosas, sin importar el elemento buscado (que, obviamente, no está en el arreglo), siempre se recorre el arreglo completo, realizando n comparaciones. Entonces, el costo promedio de búsqueda infructuosa es de n comparaciones. La búsqueda exitosa, por otro lado, es más interesante de analizar. Note que buscar el elemento que está en la posición i del arreglo cuesta i comparaciones. Tenemos n casos posibles de búsqueda exitosa. Entonces, el costo promedio de búsqueda exitosa puede calcularse mediante:

$$\bar{T}(n) = \frac{1}{n} \sum_{i=1}^n i = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}. \quad (6.1)$$

Esto significa que en promedio debemos comparar la mitad del arreglo para una búsqueda exitosa. Es importante destacar algo en este punto: note que hemos multiplicado el costo de cada uno de los n casos por $\frac{1}{n}$. Esto significa que cada uno de los costos posibles tiene el mismo peso que el resto. Por lo tanto, nuestro resultado (i.e., $\frac{n+1}{2}$ comparaciones en promedio) es válido sólo si se cumple que cada uno de los elementos del arreglo tiene la misma probabilidad ($\frac{1}{n}$) de ser buscado. Por ejemplo, si en alguna situación práctica ocurre que buscar el primer elemento del arreglo tiene probabilidad, digamos, $\frac{1}{2}$ (es decir, es esperable que la mitad de las búsquedas tengan costo 1), entonces el resultado ya no es válido. En general, el análisis de caso promedio necesita de este tipo de suposiciones para ser válido, las cuales muchas veces son razonables en la práctica, aunque en ocasiones pueden no serlo. Sin embargo, ante la falta de datos respecto a la distribución de los casos, en general es preferible hacer ese tipo de suposiciones para simplificar el análisis. En breve, estudiaremos ejemplos en donde este tipo de suposiciones pueden no ser del todo realistas.

Tiempo Promedio de Búsqueda Secuencial en Conjunto Ordenado

Analizamos a continuación una variante del problema anterior, el de búsqueda en un conjunto ordenado S . Asumimos aquí un orden total \leq sobre los elementos de S , lo cual permite ordenarlo. En particular, vamos a analizar el Algoritmo 3, en el que se asume que el conjunto está almacenado en un arreglo ordenado. El costo promedio de búsquedas exitosas es exactamente igual al de conjuntos desordenados, $\frac{n+1}{2}$ comparaciones. El costo promedio para búsquedas infructuosas, por otro lado, es más interesante en este caso. Suponga el conjunto de 10 elementos mostrado en la Figura 6.1. Los posibles casos de búsqueda infructuosa en este ejemplo son 11, marcados con flechas rojas. En

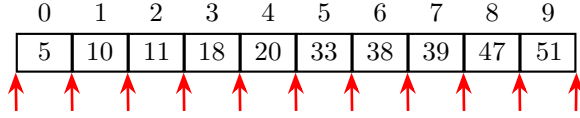


Figura 6.1: Conjunto ordenado de 10 elementos, almacenado en un arreglo ordenado. Las flechas marcan los posibles puntos de fracaso de una búsqueda.

particular, una búsqueda infructuosa en este conjunto ocurre al buscar un número menor que 5 (flecha roja de más a la izquierda), o un número mayor que 5 y menor que 10, o un número mayor que 10 y menor que 11¹, y así siguiendo hasta buscar un número que es mayor que 51. En general, un arreglo ordenado de n elementos tiene $n + 1$ puntos de fracaso. Asumiendo que todos esos puntos de fracaso tienen la misma probabilidad $\frac{1}{n+1}$ de ocurrir, y que para $1 \leq i \leq n$, el i -ésimo punto de fracaso tiene un costo de i comparaciones, mientras que el $(n + 1)$ -ésimo punto de fracaso cuesta n comparaciones, el costo promedio es:

$$\bar{T}(n) = \frac{1}{n+1} \left(\sum_{i=1}^n i + n \right) = \frac{1}{n+1} \cdot \frac{n(n+1)}{2} + \frac{n}{n+1} \approx \frac{n}{2} + 1, \quad (6.2)$$

lo cual es levemente superior al promedio de búsqueda exitosa. Para finalizar el análisis, es importante discutir si la suposición de que los puntos de fracaso son igualmente probables es realista o no. Note que en este problema pueden existir puntos de fracaso cuya probabilidad de ocurrir es nula, como el que está entre $A[1] = 10$ y $A[2] = 11$ para el arreglo de la Figura 6.1 (suponiendo que A almacena valores enteros). Además, hay puntos de fracaso que contienen más elementos que otros, como el que está entre los elementos $A[2] = 11$ y $A[3] = 18$, que contiene a los elementos $\{12, 13, 14, 15, 16, 17\}$; el que está entre los elementos $A[6] = 38$ y $A[7] = 39$, por otro lado, no contiene ningún elemento, y el que está a la derecha de $A[9] = 51$ contiene todos los elementos mayores que 51 y menores o iguales que el máximo valor posible. Sin embargo, aunque probablemente en la práctica no ocurra que los puntos de fracaso son igualmente probables, hacer esa suposición simplifica el análisis y entrega un resultado que probablemente sea cercano a la realidad en muchas situaciones. Esto es una clara desventaja del análisis de caso promedio.

Tiempo de Búsqueda Promedio en Árboles Binarios

Consideremos a continuación el mismo problema de búsqueda en un conjunto ordenado S , el cual ahora ha sido representado mediante un árbol binario de búsqueda. Un árbol binario cuyos nodos almacenan valores pertenecientes a un conjunto S , se define por inducción de la siguiente manera:

Caso Base: El árbol vacío \square es un árbol binario de 0 elementos.

Inducción: Si B_1 y B_2 son dos árboles binarios de $n_1 \geq 0$ y $n_2 \geq 0$ elementos, respectivamente, y $x \in S$ es un elemento, entonces $\langle B_1, x, B_2 \rangle$ es un árbol binario de $n_1 + n_2 + 1$ elementos.

En el árbol $\langle B_1, x, B_2 \rangle$ definido arriba, el elemento x es conocido como la *raíz del árbol*, el árbol B_1 es el *subárbol izquierdo* de x , mientras

¹ Note que si el arreglo almacena números enteros, este punto de fracaso tiene probabilidad 0 de ocurrir.

que B_2 es el *subárbol derecho*. Si x_1 es la raíz de B_1 y x_2 es la raíz de B_2 , llamamos a x_1 el *hijo izquierdo* de x , mientras que x_2 es el *hijo derecho*. Si alguno de los hijos de x es un árbol vacío, se dice que el hijo correspondiente es vacío. Los elementos de un árbol son almacenados en nodos, llamados nodos internos. El resto de los nodos, que no contienen elementos, son llamados nodos externos.

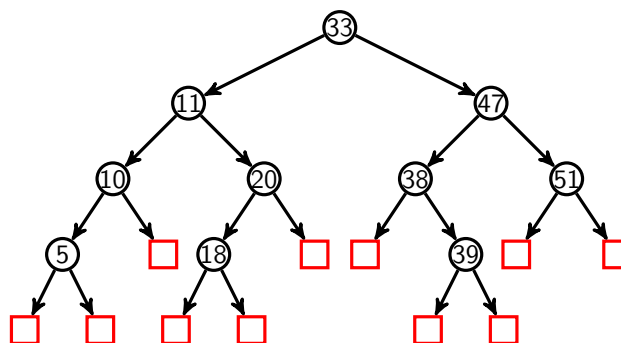
Ejercicio 6.1 Demostrar que un árbol binario de n nodos internos tiene $n + 1$ nodos externos.

Pista: use inducción sobre el número de nodos internos.

Definición 6.1.2 Dado un árbol binario B , llamaremos x_1, x_2, \dots, x_n a sus nodos internos, y $\ell_1, \ell_2, \dots, \ell_{n+1}$ a sus nodos externos.

Asumiendo que existe un orden total \preceq sobre los elementos de S , un *árbol binario de búsqueda* es un árbol binario para el que cada nodo interno almacena un valor mayor que todos los elementos de su subárbol izquierdo, y menor que todos los elementos almacenados en su subárbol derecho. La Figura 6.2 muestra un posible ABB para el conjunto ordenado $\{5, 10, 15, 18, 20, 33, 38, 39, 47, 51\}$, que es el mismo almacenado en el arreglo de la Figura 6.1. En la figura, los nodos externos del árbol

Figura 6.2: Árbol binario de búsqueda del conjunto $\{5, 10, 15, 18, 20, 33, 38, 39, 47, 51\}$ de $n = 10$ elementos. Los nodos internos están representados con círculos, mientras que los nodos externos están representados con cuadrados rojos, los cuales corresponden a las flechas rojas de la Figura 6.1.



se muestran como cuadrados rojos. Note la equivalencia entre los $n + 1$ puntos de fracaso de una búsqueda del arreglo de la Figura 6.1 y los $n + 1$ nodos externos de la Figura 6.2: estos también corresponden a las búsquedas que fracasan en el árbol.

Ejemplo 6.1.1 Por ejemplo, si buscamos el elemento 25, debemos bajar primero por la izquierda del 33, luego por la derecha del 11, y finalmente por la derecha del 20, para finalizar en ese nodo externo. Eso indica que el 25 no está en el árbol, ya que agotamos todas las posibilidades en la búsqueda, sin encontrarlo. Si uno quiere insertar el 25 en el árbol, debería hacerse en este punto de fracaso (nodo externo). Note que la inserción del 25 costaría 3 comparaciones (para encontrar el nodo externo donde insertar). Esto implica que las posteriores búsquedas del 25 costarán 4 comparaciones, una más que al insertarlo.

Usualmente, un ABB se construye por sucesivas inserciones, comenzando desde un árbol vacío. Cada elemento insertado es agregado en el nodo externo adecuado, como se explicó anteriormente. Esto implica que la forma final del árbol depende del orden de inserción de los elementos. La forma del árbol, a su vez, impacta considerablemente en

el costo de búsqueda de un elemento: note que el peor caso de búsqueda tiene costo $1 + h$ comparaciones, siendo h la altura del árbol. En el caso del árbol de la Figura 6.2, el peor caso corresponde a buscar los elementos 5, 18, o 39, que cuestan 4 comparaciones. Dado esto, los árboles con forma balanceada son preferibles por sobre los árboles más altos. Note que en el peor caso, un árbol puede tener altura $h = n - 1$ (un posible ejemplo corresponde a insertar los elementos en orden), por lo que el peor caso de búsqueda en un ABB es de n comparaciones, lo cual coincide con el peor caso de búsqueda secuencial en un arreglo ordenado.

Estudiamos a continuación el costo promedio de búsqueda en un ABB. Una manera simple de hacer este análisis es, a partir de un ABB ya construido, calcular la profundidad promedio de un nodo del ABB. Sin embargo, esto nos daría el costo promedio para ese árbol en particular. Esto es menos interesante que calcular el costo promedio de búsqueda para *cualquier* ABB de n nodos. Para lograr esto, uno podría considerar todos los posibles árboles binarios de n nodos, calcular la profundidad promedio para cada árbol, sumar todos esos promedios, y finalmente promediar entre todos los árboles. Sin embargo, recuerde la Ecuación 5.3 (en la página 111), que indica que la cantidad de árboles binarios diferentes con n nodos internos (para $n \geq 0$) son:

$$C(n) = \frac{1}{n+1} \binom{2n}{n} \approx 4^n.$$

Estos son los conocidos números de Catalan, cuya secuencia para los valores más pequeños de n es: $\langle 1, 1, 2, 5, 14, 42, 132, \dots \rangle$.

Ejercicio 6.2 Si hay $n!$ posibles formas de insertar n elementos en un ABB, ¿Por qué hay menos ($\approx 4^n$) ABBs resultantes?

Esto hace que el método antes propuesto sea muy costoso, ya que implica generar $\sim 4^n$ árboles distintos. Estudiamos en lo que sigue un método menos costoso. Comenzamos definiendo dos conceptos importantes para realizar el análisis.

Definición 6.1.3 (Suma de caminos internos) *Para un ABB de n nodos internos x_1, \dots, x_n , definimos la suma de las longitudes de sus caminos internos como*

$$I_n = \sum_{i=1}^n \text{profundidad}(x_i).$$

Definición 6.1.4 (Suma de caminos externos) *Para un ABB de n nodos internos y $n+1$ nodos externos $\ell_1, \dots, \ell_{n+1}$, definimos la suma de las longitudes de sus caminos externos como*

$$E_n = \sum_{i=1}^{n+1} \text{profundidad}(\ell_i).$$

Ejemplo 6.1.2 Para el ABB de la Figura 6.2, tenemos:

$$I_{10} = 0 + 1 + 2 + 3 + 2 + 3 + 1 + 2 + 3 + 2 = 19,$$

y

$$E_{10} = 4 + 4 + 3 + 4 + 4 + 3 + 3 + 4 + 4 + 3 + 3 = 39.$$

En este ejemplo, note que $E_{10} = I_{10} + 2 \cdot 10 = 19 + 20 = 39$. A continuación probamos esta propiedad.

Teorema 6.1.1 Dado un árbol binario B de $n \geq 0$ nodos, se cumple que

$$E_n = I_n + 2n.$$

Demostración. Procedemos por inducción.

Caso base: Para un árbol binario vacío, $n = 0$, tenemos 0 nodos internos y 1 nodo externo. En ese caso, $E_0 = 0$, $I_0 = 0$, y por lo tanto se cumple $E_0 = I_0 + 2 \cdot 0 = 0$.

Inducción: Asumimos que el teorema es cierto para árboles de $n \geq 0$ nodos. Sean B_1 y B_2 dos árboles binarios de $n_1 \leq n$ y $n_2 \leq n$ nodos internos, respectivamente. Esto significa que para ellos se cumple la hipótesis inductiva (h.i.), por lo que:

$$E_{n_1} = I_{n_1} + 2n_1$$

y

$$E_{n_2} = I_{n_2} + 2n_2.$$

A partir de estos dos árboles (para los que se cumple la h.i.), construiremos un árbol binario B' agregando un nodo raíz como padre de B_1 y B_2 , el cual tendrá $n' = n_1 + n_2 + 1$ nodos internos. Mostraremos que B' también cumple con la h.i.

Primero, determinemos $E_{n'}$ para el ABB que acabamos de construir. Note que B_1 tiene $n_1 + 1$ nodos externos, cuya suma de profundidades es E_{n_1} . En B' , a cada uno de esos nodos externos se les incrementa en 1 su profundidad, por lo tanto el aporte de B_1 a $E_{n'}$ es $E_{n_1} + n_1 + 1$. Lo mismo ocurre con B_2 , por lo tanto su aporte a $E_{n'}$ es $E_{n_2} + n_2 + 1$. En total, tenemos $E_{n'} = E_{n_1} + n_1 + 1 + E_{n_2} + n_2 + 1$.

Algo similar ocurre para $I_{n'}$. En B_1 , a cada uno de los n_1 nodos internos se le incrementa en 1 su profundidad, aportando con $I_{n_1} + n_1$ a $I_{n'}$. Similarmente, para B_2 , el cual aporta $I_{n_2} + n_2$ a $I_{n'}$. Entonces tenemos $I_{n'} = I_{n_1} + n_1 + I_{n_2} + n_2$.

Entonces tenemos:

$$\begin{aligned} E_{n'} - I_{n'} &= (E_{n_1} + n_1 + 1 + E_{n_2} + n_2 + 1) - (I_{n_1} + n_1 + I_{n_2} + n_2) \\ &= E_{n_1} + 1 + E_{n_2} + 1 - I_{n_1} - I_{n_2} \\ &\stackrel{\text{h.i.}}{=} I_{n_1} + 2n_1 + 1 + I_{n_2} + 2n_2 + 1 - I_{n_1} - I_{n_2} \\ &= 2n_1 + 2n_2 + 2 \\ &= 2(n_1 + n_2 + 1) \\ &= 2n'. \end{aligned}$$

Por lo tanto, $E_{n'} = I_{n'} + 2n'$ se cumple para B' . Dado que cualquier árbol binario puede ser construido por inducción tal como se construyó B' (recuerde la definición al comienzo de esta sección), la propiedad se cumple para cualquier árbol binario. ■

Los conceptos I_n y E_n serán clave para calcular el costo promedio de búsqueda en un ABB. Dado un ABB B de n nodos, definimos:

Costo promedio de búsqueda exitosa:

$$C_n = 1 + \frac{I_n}{n}, \quad (6.3)$$

y

Costo promedio de búsqueda infructuosa:

$$C'_n = \frac{E_n}{n+1}. \quad (6.4)$$

Esto significa que el costo promedio de búsqueda exitosa en un ABB corresponde a la profundidad promedio de un nodo interno I_n/n , más 1. Esto es porque la cantidad de comparaciones necesarias para alcanzar cualquier nodo interno es igual a su profundidad más 1. Similarmente, el costo promedio de búsqueda infructuosa en un ABB es igual a la profundidad promedio de un nodo externo (recuerde que una búsqueda infructuosa en un ABB necesita llegar a un nodo externo).

Dado que demostramos que $E_n = I_n + 2n$, tenemos que $I_n = E_n - 2n$, y entonces:

$$\begin{aligned} C_n &= 1 + \frac{I_n}{n} \\ &= 1 + \frac{E_n - 2n}{n} \\ &= \frac{E_n}{n} - 1 \\ &= \frac{n+1}{n+1} \cdot \frac{E_n}{n} - 1 \\ &= \frac{n+1}{n} C'_n - 1. \end{aligned}$$

Por lo tanto, la ecuación que relaciona el costo de las búsquedas exitosas con el de las búsquedas infructuosas es:

$$C_n = \left(1 + \frac{1}{n}\right) C'_n - 1. \quad (6.5)$$

Note que a medida que se insertan más elementos en el árbol (i.e., n es más grande), los costos de búsqueda exitosa e infructuosa son cada vez más cercanos.

Asumimos a continuación que cada una de las $n!$ formas distintas de insertar los elementos en el ABB son igualmente probables. Recuerde que el número de comparaciones necesarias para encontrar un elemento es exactamente uno más que el número de comparaciones que fueron

1: Aquí estamos asumiendo que no hay borrados en el conjunto, los cuales pueden mover elementos en el árbol, y por lo tanto el costo de buscarlos no estará necesariamente relacionado con su costo de inserción.

necesarias para insertar ese elemento en el árbol (revise el Ejemplo 6.1.1)¹. En otras palabras, el costo de búsqueda de un elemento en un ABB es igual al costo de búsqueda infructuosa justo antes de insertarlo, más 1. Esto significa que si el ABB tiene k elementos y se inserta un elemento más, el costo esperado de búsqueda para ese elemento es $1 + C'_k$. En base a esto, si se busca un elemento que está presente en un árbol de n nodos (i.e., búsqueda exitosa), tenemos que considerar que el elemento buscado:

- podría haberse insertado con costo promedio C'_0 en un árbol vacío, por lo que buscarlo cuesta en promedio $1 + C'_0$, o
- podría haberse insertado con costo promedio C'_1 en un árbol con 1 nodo, por lo que buscarlo cuesta en promedio $1 + C'_1$, o
- \vdots
- podría haberse insertado con costo promedio C'_{n-1} en un árbol con $n-1$ nodos, por lo que buscarlo cuesta en promedio $1 + C'_{n-1}$.

Para calcular el costo promedio de búsqueda exitosa, debemos considerar esos n posibles casos en que el elemento podría haberse insertado, y promediarlos:

$$\begin{aligned} C_n &= \frac{(1 + C'_0) + (1 + C'_1) + \cdots + (1 + C'_{n-1})}{n} \\ &= 1 + \frac{C'_0 + C'_1 + \cdots + C'_{n-1}}{n}. \end{aligned} \quad (6.6)$$

Las Ecuaciones (6.5) y (6.6) son dos definiciones distintas para C_n , por lo que las igualamos y obtenemos:

$$(n+1)C'_n = 2n + C'_0 + C'_1 + \cdots + C'_{n-1}. \quad (6.7)$$

Esta recurrencia es fácil de resolver, restando el término anterior:

$$nC'_{n-1} = 2(n-1) + C'_0 + C'_1 + \cdots + C'_{n-2},$$

obteniendo:

$$(n+1)C'_n - nC'_{n-1} = 2 + C'_{n-1}.$$

Por lo tanto

$$C'_n = C'_{n-1} + \frac{2}{n+1}. \quad (6.8)$$

Notar cómo el costo de búsqueda infructuosa crece cada vez más lentamente a medida que crece n . Dado que $C'_0 = 0$, tenemos la siguiente recurrencia:

$$C'_n = \begin{cases} C'_{n-1} + \frac{2}{n+1}, & \text{si } n > 0; \\ 0, & \text{si } n = 0. \end{cases}$$

Esta recurrencia es fácil de resolver, ya que:

$$\begin{aligned}
 C'_n &= C'_{n-1} + \frac{2}{n+1} \\
 &= C'_{n-2} + \frac{2}{n} + \frac{2}{n+1} \\
 &\vdots \\
 &= C'_0 + \frac{2}{2} + \cdots + \frac{2}{n+1} \\
 &= 2 \sum_{k=2}^{n+1} \frac{1}{k} = 2(H_{n+1} - 1),
 \end{aligned} \tag{6.9}$$

siendo $H_n = \sum_{k=1}^n \frac{1}{k}$ el n -ésimo número armónico. Ya que se cumple $\ln(n+1) \leq H_n \leq \ln(n) + 1$, el tiempo promedio de una búsqueda infructuosa es

$$C'_n = 1,386 \lg(n+1). \tag{6.10}$$

Para las búsquedas exitosas, reemplazamos el resultado de (6.9) en (6.5) y obtenemos:

$$C_n = 2 \left(1 + \frac{1}{n} \right) H_n - 3,$$

y entonces

$$C_n = 1,386 \lg n. \tag{6.11}$$

En conclusión, aún cuando al usar un ABB para representar el conjunto ordenado tiene costo de búsqueda $O(n)$, si todos los ordenes de inserción de los elementos en el conjunto son igualmente probables, el costo promedio de búsqueda (tanto exitosa como infructuosa) es logarítmico. En otras palabras, aún cuando el costo de peor caso al representar un conjunto ordenado usando un ABB es el mismo que usar un arreglo ordenado con búsqueda secuencial (i.e., n comparaciones), el costo de caso promedio sí mejora asintóticamente al usar ABBs.

Ejercicio 6.3 Obtener el valor promedio de I_n y E_n , usando resultados anteriores.

Ejercicios

1. Suponga el algoritmo de búsqueda secuencial sobre un arreglo ordenado. Realizar el análisis de caso promedio, tanto para búsqueda exitosa como para búsqueda infructuosa. Haga las suposiciones que sean necesarias para que su análisis sea válido.
2. Suponga una lista enlazada ordenada en la que cada nodo de la misma no sólo tiene un puntero al siguiente nodo de la lista, sino que también tiene un puntero al nodo que está dos posiciones hacia adelante. Defina una estrategia de búsqueda que haga uso eficiente de los dos punteros que hay en cada nodo. Realice el

análisis de caso promedio de dicha estrategia, haciendo todas las suposiciones que sean necesarias para el mismo.

3. Generalizar el ejercicio anterior, en donde ahora cada nodo tiene un puntero al siguiente nodo, y al nodo que está k posiciones hacia adelante en la lista. Diseñe una estrategia de búsqueda que haga uso eficiente de dichos punteros. ¿Cuál es el valor óptimo de k ?
4. ¿Cuál es el valor promedio de I_n (suma de caminos internos) y E_n (suma de caminos externos) para un árbol binario de búsqueda de n nodos? Demuéstrelo formalmente, encontrando los valores exactos en cada caso (como función de n).
5. Considere el siguiente algoritmo de búsqueda (conocido como búsqueda binaria) sobre un arreglo ordenado A de n enteros:

```
int Binaria(int *A, int n, int x)
{
    int L = 0, R = n - 1, m;
    while (L <= R) {
        m = (L + R) / 2;
        if (A[m] < x)
            L = m + 1;
        else if (A[m] > x)
            R = m - 1;
        else
            return m; /* Búsqueda exitosa */
    }
    return -1; /* Búsqueda infructuosa */
}
```

Calcule (de manera formal) el costo promedio de búsqueda exitosa. Para simplificar su análisis, puede asumir que $n = 2^k - 1$, para algún $k \geq 1$. Haga las suposiciones que sean necesarias para que su análisis sea válido.

6. Considere el siguiente algoritmo:

```
int *merge(int *A, int *B, int n)
{
    int i = 0, j = 0, k = 0;
    int *C = malloc(sizeof(int)*n);
    while (i < n && j < n)
        if (A[i] < B[j])
            C[k++] = A[i++];
        else
            C[k++] = B[j++];

    while (i < n) C[k++] = A[i++];
    while (j < n) C[k++] = B[j++];
    return C;
}
```

que recibe dos arreglos ordenados de enteros como entrada, $A[1..n]$ y $B[1..n]$, y produce como resultado un arreglo ordenado $C[1..2n]$, que contiene todos los elementos de A y B . Asuma que el tiempo de ejecución está dado por la cantidad de comparaciones entre elementos de los arreglos A y B (es decir, la cantidad de veces que se ejecuta la comparación $A[i] < B[j]$).

- a) Asumiendo $n = 4$, muestre un ejemplo que produzca el tiempo de ejecución de mejor caso del algoritmo. En base a eso, ¿Cuál es el tiempo de mejor caso, para n general?
- b) Asumiendo $n = 4$, muestre un ejemplo que produzca el tiempo de ejecución de peor caso del algoritmo. En base a eso, ¿Cuál es el tiempo de peor caso, para n general?
- c) Asumiendo $n = 3$, calcule el tiempo de ejecución promedio del algoritmo.

6.2. Análisis Amortizado

En las secciones anteriores hemos estudiado, principalmente, dos tipos de análisis: el de peor caso, y el de caso promedio. El primero tiene la característica de ser pesimista, lo que puede llegar a penalizar a ciertos algoritmos cuyo peor caso es raro u ocurre con probabilidad baja. En esas situaciones, un análisis de caso promedio suele ser más adecuado. Sin embargo, se deben hacer suposiciones sobre el algoritmo (por ejemplo, para que sea válido promediar todos los casos). Dichas suposiciones puede ser poco realistas, tal como hemos visto.

Estudiamos a continuación otro tipo de análisis de algoritmos: análisis amortizado. Suponga una secuencia de n operaciones op_1, op_2, \dots, op_n llevadas a cabo por un algoritmo. En un *análisis amortizado*, básicamente vamos a promediar el tiempo total de esas operaciones sobre las n operaciones. Esto es útil para destacar aquellos algoritmos en los que al ejecutar una secuencia de operaciones, el tiempo promedio por operación es bajo, aún cuando alguna de las operaciones individuales pueda ser costosa. Una ventaja del análisis amortizado respecto al análisis de caso promedio es que no hará falta hacer ninguna suposición respecto al algoritmo y las operaciones: el promedio se hace sobre el tiempo total que toma ejecutar una secuencia de operaciones de peor caso, es decir, una que produce el mayor tiempo de ejecución total. De esta manera, se garantiza el comportamiento promedio por cada operación en el peor caso.

Traslados en Bicicleta

Un ejemplo doméstico de este tipo de análisis es el siguiente. Suponga que necesita trasladarse todos los días a su lugar de estudio, y tiene las siguientes alternativas:

- Usar transporte público, que tiene una tarifa diaria de \$1.
- Usar bicicleta, que requiere un desembolso inicial de \$100 para comprarla, pero le garantiza traslado sin costo por *al menos* d días (es decir, sin sufrir averías). Supongamos que la reparación de averías tiene un costo de \$ a , y nuevamente tiene garantías de *al menos* d días sin nuevas averías.

¿Cuál es el costo total para n días? El caso del transporte público es simple, por lo que conviene concentrarnos en la bicicleta. Debido al desembolso inicial y al costo de reparación, tenemos que el costo total es $C(n) = 100 + a \lfloor \frac{n}{d} \rfloor$, en donde $\lfloor \frac{n}{d} \rfloor$ es la cantidad de averías sufridas en el peor caso (es decir, averías exactamente cada d días) a lo largo

de los n días. El costo de traslado amortizado es $(100 + a\lfloor \frac{n}{d} \rfloor) / n$ por traslado. Por ejemplo, si $a = 5$, $d = 20$, y $n = 365$, el costo amortizado es $\sim 0,52$ por traslado, algo menor al costo diario del transporte público. Esto significa que tanto el alto desembolso inicial como los costos de reparación de averías se amortizan con el uso de la bicicleta.

Implementación de Colas usando Pilas

Consideremos el problema de representar una cola Q usando dos pilas, S_1 y S_2 . Sobre las pilas se han definido las operaciones típicas de PUSH y POP, ambas con tiempo de ejecución $O(1)$, como es habitual. Las operaciones sobre la cola Q se han implementado de la siguiente manera:

Q.ENQUEUE(x): Se implementa haciendo $S_1.PUSH(x)$.

Q.DEQUEUE(): Si S_2 está vacía, primero se traspasan todos los elementos de S_1 a S_2 (uno por uno, usando $S_2.PUSH(S_1.TOP())$ y $S_1.POP()$). Luego, se hace $S_2.POP()$.

Comenzando con Q vacía, ¿Cuál es el tiempo amortizado por operación si se ejecuta una secuencia de n operaciones ENQUEUE y DEQUEUE sobre Q ? Para responder esta pregunta, analicemos primero el costo de cada operación. Claramente, la operación ENQUEUE toma tiempo $O(1)$, mientras que DEQUEUE toma tiempo $O(n)$, dependiendo de la cantidad de elementos que tiene la pila S_1 (que pueden ser hasta n). Entonces, un análisis de peor caso de las n secuencias produce tiempo total $O(n^2)$. Sin embargo, un análisis más detallado indica que es imposible que DEQUEUE cueste n operaciones cada vez. Esto significa que, aunque no es incorrecta, la cota $O(n^2)$ para el tiempo total no es ajustada.

Para un análisis más detallado, note que aunque ciertas operaciones DEQUEUE puedan ser costosas, pagar ese costo nos garantiza que las siguientes operaciones serán poco costosas (recuerde el tiempo entre averías del ejemplo de la bicicleta). Intuitivamente, note que si una operación DEQUEUE implica el traspaso de k elementos desde la pila S_1 a S_2 (lo que toma tiempo $\Theta(k)$), eso significa que los siguientes k DEQUEUE serán de tiempo constante, y por lo tanto el tiempo amortizado es constante para todas esas operaciones. Generalizando esto, y para calcular el tiempo de ejecución total $T(n)$, tenga en cuenta que cada elemento es insertado en S_1 una única vez, y luego eliminado de S_1 a lo más una única vez (cuando es traspasado a S_2). De la misma forma, cada elemento es insertado a lo más una única vez en S_2 , y eliminado a lo más una única vez. Esto significa que se paga tiempo constante por elemento, que a lo más pueden ser n . Entonces el tiempo total es $T(n) = \Theta(n)$. El costo amortizado por operación es, entonces:

$$C_{\text{amort}}(n) = \frac{T(n)}{n} = \frac{\Theta(n)}{n} = \Theta(1). \quad (6.12)$$

Esto es, aún cuando ciertas operaciones de la secuencia puedan ser costosas, el costo amortizado es constante por operación.

El método que hemos usado en la Ecuación (6.12) para calcular el costo amortizado es conocido como el *método de agregación*. Existen otros métodos para realizar este tipo de análisis, y que permiten abordar

análisis más complejos. Sin embargo, no serán necesarias para los casos presentados en este documento.

Incrementar un Contador Binario

Consideremos a continuación el problema de incrementar un contador binario de k bits, inicializado en 0. Nuestro contador será el arreglo $A[0..k-1]$, inicializado en 0. En este caso, $A[0]$ será el bit menos significativo del contador, mientras que $A[k-1]$ será el bit más significativo. El Algoritmo 11 ejecuta un incremento sobre un contador.

Algoritmo 11: INCREMENTAR($A[0..k-1]$)

```

1  $i \leftarrow 0$ 
2 while  $i < k$  and  $A[i] = 1$  do
3    $A[i] \leftarrow 0$ 
4    $i \leftarrow i + 1$ 
5 end
6 if  $i < k$  then
7    $A[i] \leftarrow 1$ 
8 end

```

Note que la cantidad de iteraciones que ejecuta el algoritmo depende de la cantidad de bits que cambian su valor ante un incremento. Llamaremos *flip* al cambio de valor de un bit, por lo que el costo del algoritmo estará medido en cantidad de flips ejecutados. Note que en el peor caso, la cantidad de flips realizados es k , por lo que el tiempo de un incremento está acotado por $O(k)$. Como resultado, una secuencia de n incrementos (comenzando con un contador en 0) tiene un costo total de $O(kn)$ flips. Sin embargo, note que el costo de peor caso no puede ocurrir en cada incremento, sino que después de una cierta cantidad de estos (después de todo, hay que lograr poner en 1 los k bits del contador, comenzando con todos los bits en 0, lo cual requiere tiempo). Nuevamente, aunque el análisis no es incorrecto (después de todo, es cierto que el tiempo total está acotado por $O(kn)$), la cota no es ajustada.

$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$
0	0	0	0	0
0	0	0	0	1
0	0	0	1	0
0	0	0	1	1
0	0	1	0	0
0	0	1	0	1
0	0	1	1	0
0	0	1	1	1
0	1	0	0	0

Figura 6.3: Ejemplo para el problema de incrementar un contador binario de 5 bits, sobre el que han ejecutado 8 operaciones de incremento. Los bits marcados en gris son aquellos que cambian de valor al realizar el incremento.

Para ajustar nuestro análisis, hay que notar que no todos los bits hacen flip cuando se hace un incremento. La Figura 6.3 muestra un ejemplo de contador binario de 5 bits y 8 operaciones de incremento (cada fila corresponde al resultado de un incremento, excepto la primera que es

el contador en su estado inicial). Los flips en cada incremento se han marcado en gris. En particular, note que:

- $A[0]$ cambia de valor con cada incremento, lo que suma n flips en total;
- $A[1]$ cambia de valor cada 2 incrementos, lo que suma $\lfloor n/2 \rfloor$ flips en total;
- $A[2]$ cambia de valor cada 4 incrementos, lo que suma $\lfloor n/4 \rfloor$ flips en total;
- En general, $A[i]$ cambia de valor cada 2^i incrementos, lo que suma $\lfloor n/2^i \rfloor$ flips en total, para $i = 0, \dots, \lfloor \lg n \rfloor$. Notar que aunque el contador tiene k bits, n incrementos sólo modificarán los primeros $\lg n$ bits del contador.

Por lo tanto, el número total de flips en la secuencia de n incrementos es:

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n.$$

La última igualdad es válida porque la serie es una geométrica (ver el glosario al final de este capítulo). En conclusión, el costo total de peor caso para los n incrementos es $\Theta(n)$, lo que da costo total amortizado de $\Theta(n)/n = \Theta(1)$ por incremento.

Arreglos Extensibles

Ejercicios

1. Muestre que si se incluye una operación de decremento sobre el ejemplo del contador binario de k bits, n operaciones podrían tener costo $O(kn)$.
2. Una secuencia de n operaciones se llevan a cabo sobre una estructura de datos. La i -ésima operación cuesta i si i es una potencia de 2, y 1 en otro caso. Determine el costo amortizado por operación.
3. Suponga que se quiere no sólo incrementar un contador binario, sino que también resetearlo a 0 (es decir, hacer que sus k bits sean 0). Muestre cómo implementar el contador como un arreglo de bits tal que cualquier secuencia de n operaciones de incremento y reset toman tiempo $O(1)$ amortizado por operación, sobre un contador inicialmente en cero.
4. Suponga una pila con las operaciones típicas de PUSH y POP. Además, se agrega la operación BACKUP, la cual hace una copia de la pila en otra zona de la memoria (con costo lineal respecto al tamaño de la pila). El tamaño de la pila nunca excede los k elementos, para un valor k no necesariamente constante. Además, la operación BACKUP se invoca exactamente cada k operaciones. ¿Cuál es el costo amortizado después que se ejecuta una secuencia de n operaciones que incluyen a PUSH, POP, y BACKUP? Demuéstrelo argumentando adecuadamente su respuesta.
5. Suponga que una aplicación necesita almacenar datos que llegan por ráfagas. No se conoce de antemano la cantidad de datos a recibir. El uso de espacio es importante, por lo que no puede usar una cantidad ilimitada de espacio. Por lo tanto, asuma que usará una tabla dinámica para almacenar los datos, la que crecerá a

medida que se llena. Asuma que dispone de un administrador de memoria dinámica que permite implementar la operación `free` en tiempo $\Theta(1)$ y `realloc` de un arreglo que actualmente tiene k datos en tiempo $\Theta(k)$. Analice cada una de las siguientes alternativas en términos de tiempo amortizado por cada dato recibido, asumiendo que en total se reciben n datos (recuerde que este valor es desconocido al comienzo). Analice también la cantidad total de espacio desperdiciado. Asuma que comienza con una tabla capaz de contener 1 elemento.

- a) Si la tabla llena tiene k elementos, se pide una nueva tabla de $k + 1$ elementos.
 - b) Si la tabla llena tiene k elementos, se pide una nueva tabla de $2k$ elementos.
 - c) Si la tabla llena tiene k elementos, la i -ésima vez que crece se pide una nueva tabla de $k + i$ elementos, para $i \geq 1$.
6. Suponga que necesita almacenar un conjunto de ℓ listas enlazadas, L_1, \dots, L_ℓ , sobre las que realiza las siguientes operaciones:
- `append(i, j)`: agrega el número entero j al final de la lista L_i (en tiempo de peor caso $\Theta(1)$).
 - `suma(i)`: elimina todos los elementos de la lista L_i , y los reemplaza por la suma de los elementos que estaban en la lista (es decir, luego de ejecutar la operación la lista almacena un único entero). Suponga que eliminar un nodo de la lista tiene costo de peor caso $\Theta(1)$.

Dada una secuencia de n operaciones `append` y `suma` sobre listas inicialmente vacías, ¿Cuál es el costo amortizado por operación? Muestre el análisis y argumentos para su respuesta.

7.1. La Complejidad de un Problema

En este punto es adecuado hacerse la siguiente pregunta:

Pregunta: Sobre un modelo de computación arbitrario (e.g., RAM, o similar), ¿Es posible resolver cualquier problema tan eficientemente como quisiéramos?

La respuesta es, obviamente, *no*. La realidad es que para un modelo de computación dado, cada problema tiene una complejidad inherente asociada. La *complejidad de un problema* \mathcal{P} es una medida de la cantidad de trabajo —o pasos discretos— que un algoritmo *necesita* hacer para resolver \mathcal{P} . En otras palabras, es una medida de la dificultad intrínseca para resolver un problema. La complejidad de un problema evita que un algoritmo resuelva el problema en menor tiempo computacional que el necesario para esa complejidad. En otras palabras, es la propia dificultad de un problema la que previene resolverlo tan rápidamente como uno quisiera. Obviamente, esto implica que hay problemas que son más difíciles de resolver que otros ¹.

Normalmente, cuando un problema abstracto es estudiado por primera vez, su complejidad es desconocida. A medida que el problema es estudiado, se descubren nuevos algoritmos que lo resuelven, los cuales mejoran progresivamente el tiempo de ejecución, acercándose (por arriba) a la complejidad del problema. Esto es, con cada nuevo algoritmo que mejora a los anteriores descubrimos que el problema era más fácil de resolver de lo que se pensaba. Pero, ¿Cómo sabemos si, en un punto dado, el tiempo de ejecución de un algoritmo coincide con la complejidad del problema, o no? Saber esto es importante, porque implica que es imposible diseñar algoritmos más eficientes para ese problema. El concepto clave aquí es el de *cotas inferiores*, las que estudiamos a continuación.

Recuerde, de la Definición 3.7.2, el tiempo de ejecución de peor caso $T_{\mathcal{A}}(n)$ de un algoritmo \mathcal{A} para instancias de tamaño n . El concepto de complejidad de un problema del que hemos hablado intuitivamente, puede definirse formalmente de la siguiente manera:

Definición 7.1.1 (Complejidad de Peor Caso de un Problema) *Definimos la complejidad de peor caso de un problema \mathcal{P} sobre instancias de tamaño n como el tiempo de ejecución de peor caso del algoritmo más eficiente que lo resuelve:*

$$T_{\mathcal{P}}(n) \equiv \min_{\mathcal{A} \text{ resuelve } \mathcal{P}} \{T_{\mathcal{A}}(n)\} = \min_{\mathcal{A} \text{ resuelve } \mathcal{P}} \left\{ \max_{|\mathcal{X}|=n} \{T_{\mathcal{A}}(\mathcal{X})\} \right\}.$$

Sin embargo, esta definición deja en claro algo importante: sólo podemos conocer la complejidad de peor caso de un problema si consideramos a *todos* los algoritmos que resuelven el problema, lo cual es imposible

1: Cuidado: “problemas más difíciles de resolver que otros” en este contexto significa que cualquier algoritmo que resuelva el problema necesita un mayor tiempo de computación para obtener una respuesta. No estamos hablando de la dificultad de encontrar un algoritmo para el problema.

en general —posiblemente la mayoría de esos algoritmos aún no han sido descubiertos. Estudiamos a continuación una técnica que permite conocer, en ciertos casos, la complejidad de peor caso de un problema, sin necesariamente tener que conocer todos los algoritmos que lo resuelven.

Mientras la complejidad de peor caso de un problema \mathcal{P} es todavía desconocida, en general nos conformamos con poder acotar dicha complejidad. Hablamos de cotas inferiores y superiores de la complejidad. Dichas cotas nos dan un rango dentro del cual se encuentra la verdadera complejidad del problema. Es común que esas cotas mejoren con el tiempo, como resultado de investigaciones asociadas a los problemas. Esas mejoras dan un mayor conocimiento de la complejidad del problema, ya que se estrecha el rango que la contiene.

Descubrir un nuevo algoritmo A que resuelve el problema \mathcal{P} de forma más eficiente que las conocidas, inmediatamente implica una mejor (i.e., menor) cota superior para $T_{\mathcal{P}}(n)$. Como dijimos, esa nueva cota superior para la complejidad de \mathcal{P} muestra que éste es computacionalmente más fácil de resolver de lo que se pensaba. Por otro lado, demostrar una mejor cota inferior (i.e., más grande) para $T_{\mathcal{P}}(n)$ indica que \mathcal{P} es más difícil de resolver de lo que se pensaba: al demostrar una cota inferior, estamos mostrando que es imposible que un algoritmo que resuelve a \mathcal{P} lo haga en tiempo menor a esa cota inferior. El estudio de cotas inferiores es, en general, más complicado que el de cotas superiores (que ya dijimos que se mejoran con nuevos algoritmos): para probar que $T_{\mathcal{P}}(n) \in \Omega(f(n))$, debemos probar que **todo posible algoritmo** que resuelve a \mathcal{P} tiene un tiempo de ejecución de peor caso que es $\Omega(f(n))$, aún cuando muchos de esos algoritmos no han sido descubiertos todavía. En esta sección estudiaremos algunas técnicas para demostrar cotas inferiores.

La Figura 7.1 ilustra una situación típica para un problema \mathcal{P} , con su complejidad (inicialmente desconocida) mostrada como una línea punteada. La figura muestra, además, las cotas superiores e inferiores actuales para el problema. Al definir algoritmos más eficientes para \mathcal{P} , la cota superior disminuye, acercándose por arriba a $T_{\mathcal{P}}(n)$. Por otro lado, encontrar nuevas cotas inferiores sigue el camino inverso, creciendo respecto a la cota inferior anterior. Así, nos acercamos a $T_{\mathcal{P}}(n)$ por abajo. Eventualmente (aunque no siempre), las cotas superiores e inferiores van a converger en un punto: ese punto es la complejidad del problema. En ese momento hemos descubierto la verdadera complejidad de peor caso $T_{\mathcal{P}}(n)$ para \mathcal{P} . Esto significa que ya no podremos encontrar algoritmos más eficientes para \mathcal{P} (al menos en teoría), y que la cota inferior actual ya no puede mejorarse (de otra manera, el tiempo de ejecución del algoritmo sería menor que la cota, lo que es una contradicción). En este caso diremos que la cota inferior es *ajustada*, y que el algoritmo correspondiente (cuyo tiempo de ejecución de peor caso iguala a $T_{\mathcal{P}}(n)$) es *óptimo*.

Siempre que exista una brecha entre la cota inferior de un problema y su cota superior, puede significar:

- que la cota inferior es igual a $T_{\mathcal{P}}(n)$, pero la cota superior no.
- que la cota superior es igual a $T_{\mathcal{P}}(n)$, pero la cota inferior no.
- que ambas cotas son distintas de $T_{\mathcal{P}}(n)$.

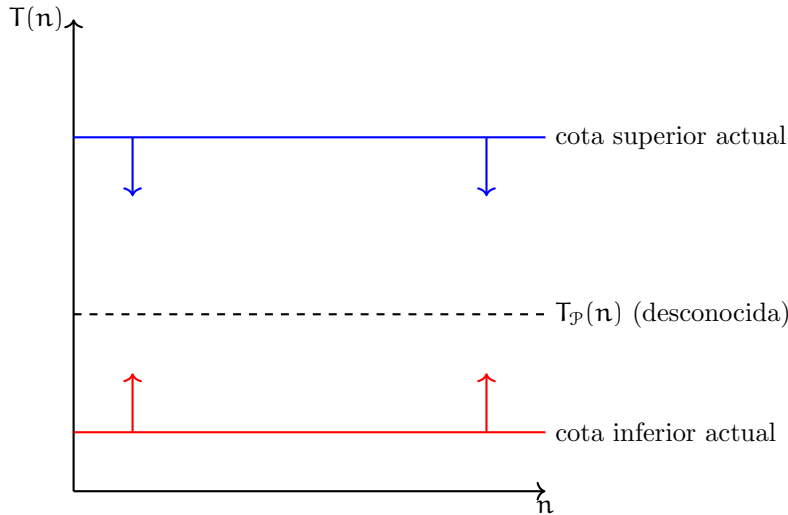


Figura 7.1: Cotas inferiores y superiores a un problema \mathcal{P} . El punto de convergencia de ambas corresponde a la complejidad del problema $T_{\mathcal{P}}(n)$ (mostrada en líneas de puntos).

Lamentablemente, la complejidad de un problema se conoce sólo cuando ambas cotas convergen, por lo tanto no sabemos en cuál de estos tres casos nos encontramos. Por lo tanto, se corre el riesgo de dedicar esfuerzo a mejorar cotas que probablemente sean ajustadas y no puedan ser mejoradas. La búsqueda de la complejidad verdadera del problema concluye cuando ambas cotas convergen; hasta ese punto hay incertidumbre.

Estudiamos a continuación algunas de las técnicas más conocidas para demostrar cotas inferiores. Para ilustrar esas técnicas, usaremos como ejemplo a problemas típicos que seguiremos estudiando en adelante. Desafortunadamente, en general no es fácil demostrar cotas inferiores, ya que hay considerar (de alguna manera) todos los posibles algoritmos para resolver un problema. Algunos de dichos algoritmos, quizás, todavía no han sido descubiertos. Es importante poder modelar todos los algoritmos que resuelven el problema en cuestión, aún cuando sea imposible conocerlos a todos, de forma tal de poder demostrar cotas inferiores sobre ese modelo. De acuerdo a cuán preciso sea ese modelo respecto de los algoritmos, dependerá lo ajustado de la cota inferior obtenida.

7.2. Árboles de Decisión

Uno de los modelos de computación más conocidos para demostrar cotas inferiores es el de *árboles de decisión*. Asumamos que se debe resolver un problema sobre un conjunto S para el que se ha definido una relación de orden total \preceq . Un modelo asociado a los árboles de decisión es el *modelo de comparaciones*, en donde el cómputo principal de un algoritmo puede ser realizado sólo mediante comparaciones (\preceq) entre (o con) los elementos de la entrada. El tiempo de ejecución en este modelo se mide en cantidad de comparaciones hechas por el algoritmo. Muchos algoritmos importantes funcionan de esta manera, comparando elementos, como por ejemplo algoritmos de búsqueda, selección, y ordenamiento.

Un árbol de decisión binario es un modelo de computación en el cual un *algoritmo particular* es representado mediante un árbol con las siguientes características:

- **Nodos internos:** representan las comparaciones de elementos hechas por el algoritmo.
- **Subárbol izquierdo:** representa el cómputo a realizar si la comparación del nodo actual es verdadera.
- **Subárbol derecho:** ídem al anterior, pero si la comparación del nodo actual es falsa.
- **Nodos externos:** representan las posibles salidas del algoritmo.

Note que la cantidad de nodos externos del árbol debe ser *al menos* el número de posibles salidas del algoritmo (aunque pueden ser más, ya que una misma salida puede ser derivada por múltiples caminos en el árbol). La ejecución del algoritmo sobre una entrada en particular corresponde a un camino desde la raíz del árbol hasta el nodo externo correspondiente.

Para una entrada dada, definimos el tiempo de ejecución de un algoritmo en este modelo como el largo del camino realizado desde la raíz del árbol hasta el nodo externo resultante. El tiempo de ejecución de peor caso del algoritmo es, por lo tanto, la altura del árbol de decisión.

La idea central detrás del modelo de árboles de decisión es determinar el número de nodos externos ℓ que debe tener cualquier árbol que resuelve el problema en cuestión. Un árbol binario con ℓ nodos externos (posibles salidas) tiene que ser lo suficientemente alto para tener ese número de nodos externos. En particular, para cualquier árbol binario con ℓ nodos externos y altura h , se cumple que

$$h \geq \lceil \lg \ell \rceil.$$

En otras palabras, para poder tener ℓ nodos externos, el árbol de decisión debe tener *al menos* cierta altura. De esta manera, el número de nodos externos permite definir una cota inferior para la altura del árbol, y por ende para el número de comparaciones de peor caso de cualquier algoritmo que resuelva el problema. A continuación, usamos este modelo para demostrar cotas inferiores de algunos problemas importantes.

El Problema de Ordenamiento

Dado un conjunto de n elementos de cierto tipo representados en un arreglo $A[1..n]$, sobre el que se ha definido una relación de orden total \leq , queremos obtener una permutación i_1, i_2, \dots, i_n de \mathbb{N}_n tal que

$$A[i_1] \leq A[i_2] \leq \dots \leq A[i_n].$$

Esta formulación del problema nos permitirá analizarlo de mejor manera. Sin embargo, en la práctica, los elementos del arreglo original deben ser reorganizados de tal manera que

$$A[1] \leq A[2] \leq \dots \leq A[n].$$

La mayoría de los algoritmos de ordenamiento conocidos —aunque no todos— están basados en comparaciones entre los elementos del arreglo. En este modelo, siempre que podamos definir un orden total entre los elementos del conjunto, podremos ordenarlos por comparaciones. Usaremos árboles de decisión para modelar cualquier algoritmo de ordenamiento basado en comparaciones y obtendremos cotas inferiores para el problema.

Note que un algoritmo de ordenamiento debe encontrar una permutación de los elementos del arreglo en la cual los elementos estén ordenados ascendentemente. Consideremos, como ejemplo, un arreglo de tres elementos, a , b , y c , sobre los que se ha definido una relación de orden total. Un **posible algoritmo de ordenamiento** para estos tres elementos, usando comparaciones, es representado por el árbol de decisión binario de la Figura 7.2. Cualquier algoritmo de ordenamiento

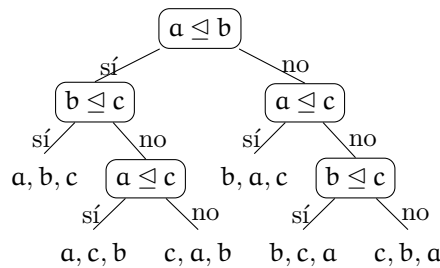


Figura 7.2: Árbol de decisión binario que representa un posible algoritmo para ordenar 3 elementos a , b , y c usando la relación de orden total \leq .

basado en comparaciones puede ser representado con un árbol de decisión como el mostrado. Para $n = 3$ elementos, tenemos $3! = 6$ posibles ordenes (o salidas del algoritmo). Por lo tanto, cualquier algoritmo de ordenamiento válido debe tener al menos esos 6 diferentes ordenes representados por un nodo externo.

Ejemplo 7.2.1 Considere los elementos enteros $a = 5$, $b = 7$, $c = 2$ a ordenar, y la relación de orden total \leq sobre los enteros. De acuerdo al algoritmo de la Figura 7.2, ocurre que $a \leq b$, por lo tanto descendemos por el hijo izquierdo de la raíz. La siguiente comparación $b \leq c$ es falsa, por lo que descendemos por el hijo derecho. Finalmente, $a \leq c$ también es falsa, descendiendo nuevamente por el hijo derecho, alcanzando el nodo externo correspondiente a c, a, b . Esto es, la salida del algoritmo es la secuencia ordenada 2, 5, 7.

En general, para un arreglo de n elementos, hay $n!$ posibles permutaciones. Eso significa que el árbol de decisión debe tener altura

$$h \geq \log_2 n!.$$

Esa es la cota inferior para el problema de ordenamiento. Usando la fórmula de Stirling (ver la Sección 4.3) para $n!$, tenemos que:

$$\begin{aligned}
 \lceil \lg(n!) \rceil &\approx \lg\left(\sqrt{2\pi n} \left(\frac{n}{e}\right)^n\right) \\
 &= n \lg n - n \lg e + \frac{\lg n}{2} + \frac{\lg 2\pi}{2} \\
 &\approx n \lg n.
 \end{aligned} \tag{7.1}$$

Esto significa que, en el peor caso, un algoritmo de ordenamiento

basado en comparaciones de pares de elementos no puede realizar menos de (aproximadamente) $n \lg n$ comparaciones. Es decir, acabamos de descubrir que se cumple

$$T_O(n) \in \Omega(n \lg n),$$

para la complejidad de peor caso $T_O(n)$ del problema de ordenamiento en el modelo de comparaciones.

Cota inferior para el caso promedio

También se podría calcular la cota inferior del caso promedio para el problema de ordenamiento, calculando la profundidad promedio de los nodos externos del árbol de decisión. Esto coincide con la profundidad promedio de un nodo externo en árboles binarios, ya estudiada anteriormente, la cual es $\sim \lg(n!)$. Ese análisis es válido si cualquiera de las $n!$ posibles permutaciones de la entrada son igualmente probables. En ese caso, la cota inferior de caso promedio para el problema de ordenamiento en el modelo de comparaciones es también $\Omega(n \lg n)$.

En capítulos posteriores estudiaremos algoritmos para este problema, y determinaremos si $T_O(n) \in \Theta(n \lg n)$ o no (es decir, si la cota inferior encontrada es ajustada o no). Eso no está claro en este punto.

Búsqueda en un Conjunto Ordenado

Retomemos el problema de búsqueda en conjuntos ordenados, estudiado en la Sección 6.1, para el que hemos planteado dos soluciones: búsqueda secuencial (representando el conjunto en un arreglo ordenado), y árboles binarios de búsqueda. Estudiamos ahora una cota inferior de peor caso del problema en el modelo de comparaciones. Vamos a denotar con S_i al i -ésimo elemento del conjunto ordenado.

Al igual que para los algoritmos de ordenamiento, cualquier algoritmo de búsqueda sobre un conjunto ordenado (en el modelo de comparaciones) puede ser modelado como un árbol de decisión. Para $0 \leq i < n$, los nodos internos del árbol representan 2 tipos distintos de comparaciones:

- $x \leq S_i$: un nodo interno del árbol realiza este tipo de comparación si y sólo si al menos uno de sus hijos es también un nodo interno.
- $x = S_i$: un nodo interno del árbol realiza este tipo de comparación si y sólo si sus dos hijos son nodos externos del árbol.

En otras palabras, el algoritmo “itera”² por ‘ \leq ’, y posterga el chequeo de igualdad hasta el final para decidir si el elemento buscado está o no en el conjunto. Recuerde el Algoritmo 3, que realiza el chequeo de igualdad al final. Cada uno de los nodos que chequea por ‘ $=$ ’ debe tener un hijo externo correspondiente a una búsqueda exitosa (es ese caso, el nodo almacena la posición en donde encontró el elemento), mientras que el otro corresponde a una búsqueda infructuosa. La Figura 7.3 ilustra dos árboles de decisión, correspondientes a dos posibles algoritmos de búsqueda sobre un conjunto de 3 elementos. El árbol de la izquierda

2: Sin embargo, estrictamente hablando, el modelo de árboles de decisión no permite iterar, sino que el cómputo debe ser escrito explícitamente para un n particular usando sentencias de selección tipo if.

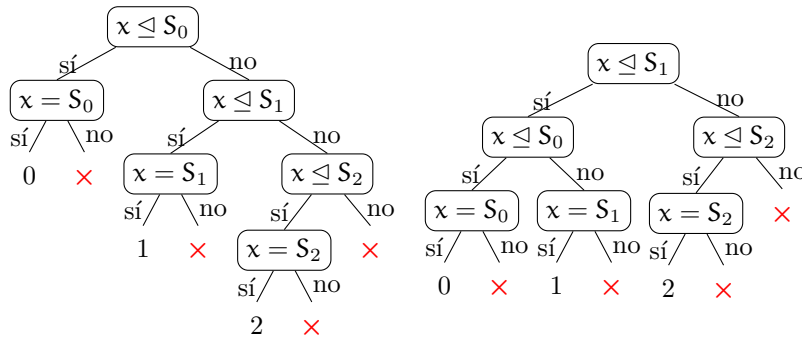


Figura 7.3: Árboles de decisión correspondientes a dos algoritmos de búsqueda de un elemento x en un conjunto ordenado $S = \{S_0, S_1, S_2\}$ de 3 elementos. El árbol de la izquierda corresponde a una búsqueda secuencial, y el de la derecha a búsqueda binaria.

corresponde al Algoritmo 3 de búsqueda secuencial, mientras que el de la derecha es similar al algoritmo de búsqueda binaria (Ejercicio 5 de la Sección 6.1). En la figura, los nodos externos marcados con \times corresponden a las búsquedas infructuosas (recuerde la Figura 6.1), mientras que el resto de nodos externos corresponde a una búsqueda exitosa que finaliza en una posición $1 \leq i \leq n$.

Para pensar...

¿Cuántos algoritmos de búsqueda distintos podríamos llegar a definir?

Pista: acabamos de decir que cada algoritmo de búsqueda puede representarse como un árbol.

Note que los nodos externos del árbol de decisión deben representar, al menos, los n casos posibles de búsqueda exitosa y los $n + 1$ casos posibles de búsqueda infructuosa. En total, $\ell \geq 2n + 1$ nodos externos. Entonces, la altura del árbol es al menos

$$h \geq \lceil \lg(2n + 1) \rceil = \lceil \lg n \rceil + 1. \quad (7.2)$$

Por lo tanto, hemos acotado la complejidad de peor caso del problema de búsqueda en conjunto ordenado, $T_B(n)$, de la siguiente manera:

$$T_B(n) \geq \lceil \lg n \rceil + 1,$$

Esto significa que, en el peor caso, cualquier algoritmo de búsqueda sobre un conjunto ordenado debe realizar $\Omega(\lg n)$ comparaciones.

Para concluir la sección, dos últimas consideraciones. Primero, ¿Es esta cota ajustada? La variante típica de búsqueda binaria, mostrada en el Ejercicio 5 de la Sección 6.1, realiza $O(\lg n)$ comparaciones por búsqueda. En mayor detalle, el algoritmo realiza $2\lceil \lg n \rceil + 1$ comparaciones en el peor caso. Esto es casi el doble que la cota inferior de la Ecuación (7.2). El algoritmo es, de todas formas, óptimo en términos asintóticos para el modelo de comparaciones. Sin embargo, no es claro en este punto si se puede mejorar el algoritmo para realizar exactamente $T_B(n)$ comparaciones en el peor caso, o se debe mejorar la cota inferior. Estudiaremos este problema en detalle en el Capítulo 12, resolviendo esta incertidumbre. Es conveniente considerar, también, las estructuras de datos de tipo árbol balanceado, como por ejemplo los AVL, árboles rojo/negro, árboles 2-3, y árboles B, entre otros. Todos estos permiten buscar en un conjunto ordenado usando $O(\lg n)$ comparaciones, por

lo que son asintóticamente óptimas en el peor caso. Estudiaremos el comportamiento de peor caso de AVLs en el Capítulo 4.4.

Segundo, respecto a la complejidad de caso promedio, si usamos el resultado obtenido en la Sección 2, se puede probar que en promedio la cota inferior de búsqueda en conjunto ordenado también es $\Omega(\lg n)$.

Resumen del problema de búsqueda en el modelo de comparaciones

- La búsqueda binaria es asintóticamente óptima en el peor caso.
- La búsqueda binaria es asintóticamente óptima en el caso promedio (ver Ejercicio 5 de la Sección 6.1).
- Las estructuras de datos del tipo árboles balanceado (e.g., AVL, red/black trees, árboles 2-3, árboles B) son asintóticamente óptimas en peor caso y caso promedio.
- Los árboles binarios de búsqueda (ABB) son asintóticamente óptimos en el caso promedio.

Ejercicios

1. ¿Cuál es la diferencia entre el análisis de mejor caso de un algoritmo y una cota inferior para un problema?
2. Responda a las siguientes situaciones, justificando en cada caso.
 - Usted ha sido contactado por un importante journal en Ciencias de la Computación, y le solicitan ser evaluador de un artículo que ha sido enviado por un reconocido investigador. Le han enviado solamente el abstract del artículo, en donde se dice que se ha descubierto un nuevo algoritmo de búsqueda para arreglos ordenados. Dicho algoritmo funciona bajo el modelo de comparaciones entre elementos del arreglo y el elemento buscado (tal como, por ejemplo, la búsqueda binaria). El autor se jacta de que su algoritmo tiene un costo de búsqueda de peor caso que es $O(\log n / \log \log n)$. ¿Aceptaría revisar el artículo? En caso de revisarlo, ¿Aceptaría el artículo? Para tomar esa decisión, ¿Le haría falta leer el artículo?
 - Usted ha sido contactado nuevamente por el mismo journal de la pregunta anterior. Esta vez, le solicitan ser evaluador de un artículo en donde se presenta una nueva cota inferior de $\Omega(\log n / \log \log n)$ para el problema de búsqueda en un arreglo ordenado. ¿Aceptaría revisar el artículo? En caso de revisarlo, ¿Aceptaría el artículo? Para tomar esa decisión, ¿Le haría falta leer el artículo?
 - Usted ha sido contactado nuevamente por el mismo journal de las preguntas anteriores. Esta vez, le solicitan ser evaluador de un artículo en donde se presenta una nueva cota inferior de $\Omega(\log n \cdot \log \log n)$ para el problema de búsqueda en un arreglo ordenado. ¿Aceptaría revisar el artículo? En caso de revisarlo, ¿Aceptaría el artículo? Para tomar esa decisión, ¿Le haría falta leer el artículo?

3. Considere el problema de encontrar el valor máximo de un arreglo de n elementos.
 - a) Use árboles de decisión para demostrar una cota inferior para dicho problema.
 - b) Demuestre que puede mejorar el resultado del punto anterior, encontrando que $n - 1$ comparaciones son suficientes y necesarias para resolver el problema.
4. Demuestre que $n - 1$ comparaciones son suficientes y necesarias para el problema de buscar un elemento en un arreglo desordenado de n elementos.
5. Considere el problema de mezclar dos arreglos ordenados $A[1 \dots n]$ y $B[1 \dots n]$ de n elementos cada uno, para producir un arreglo ordenado $C[1 \dots 2n]$ que contiene a todos los elementos de A y B (considerando repeticiones). Demuestre (usando un argumento de adversario) que la cota inferior para este problema es de $2n - 1$ comparaciones de elementos.
6. Suponga una generalización del problema de merge (problema anterior), en donde uno quiere mezclar k arreglos ordenados de n/k elementos cada uno, para producir un arreglo ordenado que contiene los n elementos de dichos arreglos. Muestre que $\Omega(n \lg k)$ comparaciones es una cota inferior para dicho problema.
Hint 1: Hay $\frac{n!}{((n/k)!)^k}$ maneras distintas de mezclar k arreglos de tamaño n/k .
Hint 2: $(n/e)^n \leq n! \leq n^n$.
7. Considere la siguiente generalización del problema de búsqueda en un conjunto ordenado. Sea $A[1..n]$ un conjunto ordenado con elementos distintos, almacenado en un arreglo. Dado un arreglo ordenado $X[1..k]$ de $k \geq 1$ elementos distintos, queremos encontrar las posiciones del arreglo A que contienen a cada uno de los elementos $X[i]$. En otras palabras, se quiere obtener el arreglo $I[1..k]$ tal que $I[i] = 0$ si $X[i]$ no está en A , o $A[I[i]] = X[i]$ en caso que $X[i]$ pertenece a A . Use árboles de decisión para demostrar una cota inferior para este problema.
Ayuda: Usando aproximación de Stirling, para cualquier par de números enteros $1 \leq v \leq m$ se puede demostrar que $\binom{m}{v} \leq \left(\frac{em}{v}\right)^v$.

7.3. Argumentos de Adversario

Los *argumentos de adversario* son una técnica que permite demostrar cotas inferiores para ciertos problemas. La misma consiste en un adversario malicioso —aunque honesto— que intenta elegir una entrada para nuestros algoritmos de manera que los mismos ejecuten en el mayor tiempo posible. A lo largo de su ejecución, el algoritmo interactúa con la entrada a través del adversario, quien no está obligado a mostrarla sino hasta el momento en que el algoritmo entrega una respuesta. Para el modelo de comparaciones, por cada comparación realizada por el algoritmo el adversario responde de manera tal que se asegure que ejecutemos la mayor cantidad posible de comparaciones³. Cuando el algoritmo entrega una respuesta, el adversario debe presentar una posible entrada que sea consistente con todas las respuestas que nos

3: Recuerde que el algoritmo no puede ver la entrada, sólo debe confiar en el adversario.

dio ante las comparaciones que hizo el algoritmo, y así convencernos de que ésa fue la entrada que siempre estuvo usando. Note que si hay elementos de la entrada que no fueron comparados por el algoritmo, el adversario puede reemplazarlos por elementos que contradigan la respuesta del algoritmo, aunque sean consistentes con las comparaciones que el algoritmo hizo. Esto permite demostrar que si el algoritmo no realiza la suficiente cantidad de comparaciones, entonces no es capaz de responder correctamente. A continuación, ejemplificamos esta técnica sobre algunos problemas relevantes.

El Problema de Selección

Estudiemos en esta sección el problema de selección, para el que ya hemos estudiado casos particulares en capítulos anteriores. Dado un conjunto no necesariamente ordenado $S = \{s_1, \dots, s_n\}$ de n elementos sobre los que se asume que existe una relación de orden total \leq , y un valor entero k , se quiere encontrar el índice j tal que $\text{rango}(s_j) = k$ (recuerde la Definición 2.2.3). Algunos casos particulares de este problema son los siguientes:

- $k = 1$: corresponde a buscar el mínimo elemento del conjunto.
- $k = n$: corresponde a buscar el máximo elemento del arreglo.
- $k = \lfloor \frac{n+1}{2} \rfloor$: corresponde a buscar la mediana del conjunto.

Este problema tiene numerosas aplicaciones, como la implementación de herramientas estadísticas.

Nos concentramos en esta sección en el caso $k = n$, aunque los resultados son equivalentes para $k = 1$. Dejamos el estudio del caso k general para el Capítulo 12. Asumimos nuevamente el modelo de comparaciones, por lo que debemos buscar el máximo en S sólo usando comparaciones \leq entre elementos del conjunto.

Recuerde el Algoritmo 1 (página 49) para encontrar el máximo de un conjunto, asumiendo el conjunto S almacenado en un arreglo. Dicho algoritmo considera todas las posibles respuestas al problema, quedándose finalmente con la que satisface la condición de máximo. El mismo realiza $n - 1$ comparaciones. ¿Qué tan eficiente es este algoritmo? Para poder responder a esta pregunta, estudiemos su complejidad.

Consideremos primero representar cualquier algoritmo para el máximo en el modelo de árboles de decisión. Cada nodo interno del árbol de decisión es una comparación entre dos elementos del conjunto. Las distintas ramas del árbol mantienen el máximo elemento visto hasta el momento. Finalmente, las hojas corresponden a las posiciones del arreglo que almacenan al máximo. Al estudiar una cota inferior para este problema, es importante que no hagamos ninguna suposición respecto al orden en que los elementos son comparados por el algoritmo. De hecho, podrían ser comparados en el orden que sea necesario. El Algoritmo 1 los compara de izquierda a derecha. Pero podrían existir algoritmos que los verifiquen en un orden distinto, y que sean mejores que éste.

La Figura 8.1 muestra el árbol de decisión correspondiente al Algoritmo 1 para un conjunto de 4 elementos. Note que el árbol tiene 2^{n-1} nodos externos, correspondientes a las posibles respuestas. Sin embargo, en general no podemos asumir que cualquier algoritmo que busque el

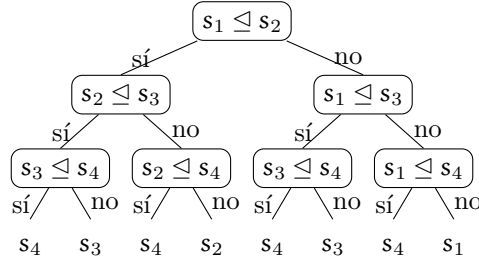


Figura 7.4: Árbol de decisión correspondientes al Algoritmo 1 (página 49), para buscar el máximo de un conjunto de $n = 4$ elementos.

máximo en un conjunto debe tener al menos esta cantidad de nodos externos. Probablemente esto ocurra sólo para este algoritmo, pero no en general. Lo único que podemos asumir en general es que cualquier árbol de decisión que represente un algoritmo para buscar el máximo debe tener al menos n nodos externos, correspondientes a las n posiciones del arreglo en que pudiese encontrarse el máximo. Por lo tanto, mediante árboles de decisión podemos demostrar una cota inferior de $\lg n$ comparaciones para este problema. Si denotamos con $T_M(n)$ a la complejidad de peor caso para el problema del máximo, esta cota inferior nos da la siguiente información:

$$T_M(n) \geq \lg n.$$

Desde ese punto de vista, el tiempo de ejecución del Algoritmo 1 no parece ser eficiente. O, alternatively, puede ocurrir que la cota inferior no es ajustada. O ambas.

Demostramos a continuación que la cota inferior encontrada con árboles de decisión no es ajustada, usando la estrategia del adversario. Al usar esta técnica con este problema, el adversario simula que el conjunto es $S = \{1, 2, \dots, n\}$, aunque no lo revela explícitamente. Supongamos que un algoritmo realiza comparaciones de elementos, en algún orden arbitrario, y que en un paso del mismo se comparan los elementos s_i y s_j , para $i \neq j$, $0 \leq i, j < n$. Si la respuesta es que $s_i < s_j$, el adversario marca que la posición i del conjunto no corresponde al máximo. Esto significa que a lo sumo un elemento es marcado en cada comparación. Dado que $s_n = n$ es el máximo para el arreglo que el adversario pensó, la posición n nunca será marcada.

Probamos a continuación que cualquier algoritmo que resuelve el problema del máximo de forma correcta, no puede hacerlo en menos de $n - 1$ comparaciones. Supongamos, por contradicción, un algoritmo que realiza menos de $n - 1$ comparaciones. Esto significa que alguna posición $k \neq n$ del conjunto $\{1, 2, \dots, n\}$ quedó sin marcar (es decir, s_k nunca fue comparado por el algoritmo). Ante esto, el algoritmo podría responder que el máximo está en alguna de esas posiciones, k o n , porque ya descartó que el máximo estuviese en las restantes. La reacción del adversario sería la siguiente:

- Si el algoritmo responde que el máximo está en la posición k , el adversario muestra el conjunto de entrada $\{1, 2, \dots, n\}$, lo que significa que el algoritmo respondió incorrectamente, ya que el máximo está en la posición n .
- Si, por otro lado, el algoritmo responde que el máximo está en la posición n , eso le da la libertad al adversario de cambiar el valor $s_k \leftarrow n + 1$ (es decir, almacena allí un elemento mayor a todos

los del conjunto). Al presentarnos el arreglo que usó (o simuló usar), veremos también en este caso que hemos respondido de forma incorrecta.

Esto significa que siempre que un algoritmo haga menos de $n - 1$ comparaciones para responder al problema del máximo, el adversario nos puede mostrar que nos hemos equivocado en responder, y por lo tanto nuestro algoritmo es incorrecto.

Esto prueba una mejor cota inferior para el problema del máximo que la que habíamos probado mediante árboles de decisión. Por lo tanto, la complejidad del problema (medido en cantidad de comparaciones es):

$$T_M(n) \geq n - 1.$$

Sin embargo, esta cantidad de comparaciones coincide con la cantidad de comparaciones realizadas por el Algoritmo 1, por lo tanto tenemos que esta cota inferior es ajustada:

$$T_M(n) = n - 1,$$

y dicho algoritmo es óptimo. Esto demuestra que los enfoques de fuerza bruta no son necesariamente ineficientes. En ciertas ocasiones son la solución más eficiente que podemos usar.

Técnicas de Diseño de Algoritmos

Fuerza Bruta y Búsqueda Exhaustiva

8

8.1. Introducción

La *fuerza bruta* es un enfoque simple para resolver un problema mediante un algoritmo que, en general, se basa directamente en el enunciado del problema y la definición de los conceptos involucrados. Con este enfoque, uno resuelve un problema “por la fuerza”, es decir, usando el poder computacional más que un buen diseño del algoritmo. Como podrá imaginar, suele ser la manera más simple de resolver un problema, aunque no necesariamente la más eficiente. Sin embargo, y aunque en muchos casos no constituye una solución final a un problema —generalmente por su ineficiencia— las soluciones de fuerza bruta establecen una solución inicial que permite entender mejor el problema que se está abordando, y cuáles son las operaciones repetitivas que podrían evitarse con un mejor diseño.

Por ejemplo, para el problema de exponenciación, en donde se quiere computar el valor

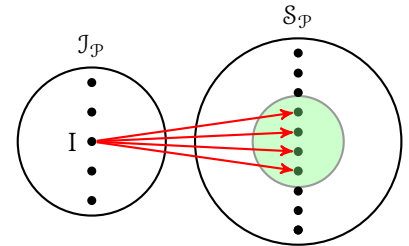
$$a^n = \underbrace{a \times a \times \cdots \times a}_{n-1 \text{ multiplicaciones}},$$

la solución por fuerza bruta usa esta definición para resolverlo, computando directamente las $\Theta(n)$ multiplicaciones.

Aún así, las soluciones de fuerza bruta suelen ser adecuadas para problemas con instancias pequeñas, en las que no vale la pena invertir en el diseño de un algoritmo más eficiente.

La *búsqueda exhaustiva*, por otro lado, es un enfoque de fuerza bruta que resuelve un problema probando exhaustivamente **todas** las posibles soluciones al mismo. Como ya vimos en el Capítulo 3, un problema abstracto \mathcal{P} con conjunto de instancias $\mathcal{I}_{\mathcal{P}}$ y soluciones $\mathcal{S}_{\mathcal{P}}$, puede ser resuelto conceptualmente recorriendo el conjunto de soluciones $\mathcal{S}_{\mathcal{P}}$, chequeando una a una las posibles soluciones, hasta encontrar una adecuada: si el problema es de búsqueda, sólo basta encontrar una solución, mientras que si el algoritmo es de optimización se debe recorrer $\mathcal{S}_{\mathcal{P}}$ completamente. Aunque la idea es bastante simple y directa, y la corrección de los algoritmos resultantes es obvia, en general su implementación requiere un algoritmo para generar objetos combinatorios —por ejemplo, permutaciones. Además, obviamente no es una solución eficiente en muchos casos en que $|\mathcal{S}_{\mathcal{P}}|$ es muy grande.

Estudiaremos en este capítulo estos dos enfoques. Comprenderemos además que, aunque en ciertos casos la fuerza bruta y la búsqueda exhaustiva suelen ser sólo un punto de comienzo para encontrar algoritmos más eficientes ¹, existen ciertos problemas para los que la fuerza bruta —o la búsqueda exhaustiva— es la mejor solución posible.



1: Y, en el camino, entender mejor el problema que queremos resolver.

8.2. El Problema de Selección

Retomemos en esta sección el problema de selección, que ya ha sido estudiado en capítulos anteriores. Dado un conjunto no necesariamente ordenado $S = \{s_1, \dots, s_n\}$ de n elementos sobre los que se asume que existe una relación de orden total \leq , y un valor entero k , se quiere encontrar el índice j tal que s_j tiene rango k en S ². Algunos ejemplos particulares de este problema son los siguientes:

- $k = 1$: corresponde a buscar el mínimo elemento del conjunto.
- $k = n$: corresponde a buscar el máximo elemento del arreglo.
- $k = \lfloor \frac{n+1}{2} \rfloor$: corresponde a buscar la mediana del conjunto.

Esto tiene numerosas aplicaciones, como la implementación de herramientas estadísticas.

Nos concentramos en esta sección en el caso $k = n$, aunque los resultados pueden extrapolarse al caso equivalente $k = 1$. Dejaremos el estudio del caso k general para el Capítulo 12. Asumiremos nuevamente el modelo de comparaciones, por lo que debemos buscar el máximo en S sólo usando comparaciones entre elementos del conjunto.

Recuerde el Algoritmo 1 (página 49) para encontrar el máximo, asumiendo el conjunto S almacenado en un arreglo. Dicho algoritmo es de búsqueda exhaustiva: considera todas las posibles respuestas al problema, quedándose finalmente con la que satisface la condición de máximo. El mismo realiza $n - 1$ comparaciones. ¿Qué tan eficiente es este algoritmo? Después de todo, hemos sugerido que los algoritmos de fuerza bruta y búsqueda exhaustiva son, en general, ineficientes. Para poder concluir, estudiemos cotas inferiores para este problema.

Consideremos primero representar cualquier algoritmo que permite encontrar el máximo en el modelo de árboles de decisión. Cada nodo interno del árbol de decisión es una comparación entre dos elementos del conjunto. Las distintas ramas del árbol mantienen el máximo elemento visto hasta el momento. Finalmente, las hojas corresponden a las posiciones del arreglo que almacenan al máximo. Al estudiar una cota inferior para este problema, es importante que no hagamos ninguna suposición respecto al orden en que los elementos son comparados por el algoritmo. De hecho, podrían ser comparados en el orden que sea necesario. El Algoritmo 1 los compara de izquierda a derecha. Pero podrían existir algoritmos que los verifiquen en un orden distinto, y que sean mejores que éste.

La Figura 8.1 muestra el árbol de decisión correspondiente al Algoritmo 1 para un conjunto de 4 elementos. Note que el árbol tiene 2^{n-1} nodos

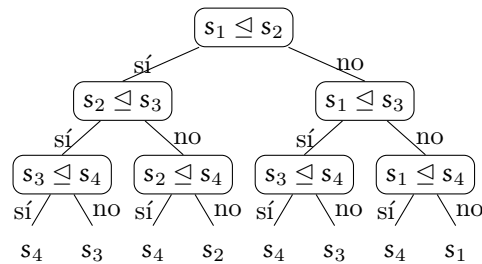


Figura 8.1: Árbol de decisión correspondientes al Algoritmo 1 (página 49), para buscar el máximo de un conjunto de $n = 4$ elementos.

externos, correspondientes a las posibles respuestas. Sin embargo, en

general no podemos asumir que cualquier algoritmo que busque el máximo en un conjunto debe tener al menos esta cantidad de nodos externos. Probablemente esto ocurra sólo para este algoritmo, pero no en general. Lo único que podemos asumir en general es que cualquier árbol de decisión que represente un algoritmo para buscar el máximo debe tener al menos n nodos externos, correspondientes a las n posiciones del arreglo en que pudiese encontrarse el máximo. Por lo tanto, mediante árboles de decisión podemos demostrar una cota inferior de $\lg n$ comparaciones para este problema. Si denotamos con $T_M(n)$ a la complejidad de peor caso para el problema del máximo, esta cota inferior nos da la siguiente información:

$$T_M(n) \geq \lg n.$$

Desde ese punto de vista, el tiempo de ejecución del Algoritmo 1 no parece ser eficiente. O, alternativamente, puede ocurrir que la cota inferior no es ajustada. O ambas.

Demostremos a continuación que la cota inferior encontrada con árboles de decisión no es ajustada, demostrando una mejor cota inferior para el problema. Usaremos una técnica conocida como *argumentos de adversario*. La misma consiste en un adversario malicioso (aunque honesto) que intenta elegir una entrada para nuestros algoritmos, de manera que los mismos ejecuten en el mayor tiempo posible. El adversario no está obligado a mostrarnos la entrada que eligió sino hasta el momento en que el algoritmo entrega una respuesta. Cada vez que nuestro algoritmo realiza una comparación, el adversario responde de manera tal que se asegure que ejecutemos la mayor cantidad posible de comparaciones (recuerde que el algoritmo no puede ver la entrada, sólo debe confiar en el adversario). Una vez que el algoritmo entrega una respuesta, el adversario debe presentar una posible entrada que sea consistente con todas las respuestas que nos dio ante las comparaciones que hizo el algoritmo (recuerde que es un adversario honesto, después de todo), y así convencernos de que ésa fue la entrada que siempre estuvo usando. Note que si hay elementos de la entrada que no fueron comparados por el algoritmo, le damos al adversario la posibilidad de colocar allí el elemento que quiera. Como el algoritmo no verificó nunca ese elemento, podría haber sido cualquiera. De esta manera, el adversario puede colocar allí elementos que contradigan la respuesta del algoritmo, aunque sean consistentes con las comparaciones que el algoritmo sí hizo. Esto permite demostrar que si el algoritmo no realiza la suficiente cantidad de comparaciones, entonces no es capaz de responder correctamente.

Al usar esta técnica con nuestro problema, el adversario simula que el conjunto es $S = \{1, 2, \dots, n\}$, aunque no lo revela explícitamente. Supongamos que el algoritmo realiza comparaciones de elementos, en algún orden arbitrario, y que en un paso del mismo se comparan los elementos s_i y s_j , para $i \neq j$, $0 \leq i, j < n$. Si la respuesta es que $s_i \leq s_j$, el adversario marca que la posición i del conjunto no corresponde al máximo. Esto significa que a lo sumo un elemento es marcado en cada comparación. Dado que $s_n = n$ es el máximo para el arreglo que el adversario pensó, la posición n nunca será marcada.

Probamos a continuación que cualquier algoritmo que resuelve el problema del máximo de forma correcta, no puede hacerlo en menos de

$n - 1$ comparaciones. Supongamos, por contradicción, un algoritmo que realiza menos de $n - 1$ comparaciones. Esto significa que alguna posición $k \neq n$ del conjunto $\{1, 2, \dots, n\}$ quedó sin marcar (es decir, s_k nunca fue comparado por el algoritmo). Ante esto, el algoritmo podría responder que el máximo está en alguna de esas posiciones, k o n , porque ya descartó que el máximo estuviese en las restantes. La reacción del adversario sería la siguiente:

- Si el algoritmo responde que el máximo está en la posición k , el adversario muestra el conjunto de entrada $\{1, 2, \dots, n\}$, lo que significa que el algoritmo respondió incorrectamente, ya que el máximo está en la posición n .
- Si, por otro lado, el algoritmo responde que el máximo está en la posición n , eso le da la libertad al adversario de cambiar el valor $s_k \leftarrow n + 1$ (es decir, almacena allí un elemento mayor a todos los del conjunto). Al presentarnos el arreglo que usó (o simuló usar), veremos también en este caso que hemos respondido de forma incorrecta.

Esto significa que siempre que un algoritmo haga menos de $n - 1$ comparaciones para responder al problema del máximo, el adversario nos puede mostrar que nos hemos equivocado en responder, y por lo tanto nuestro algoritmo es incorrecto.

Esto prueba una mejor cota inferior para el problema del máximo que la que habíamos probado mediante árboles de decisión. Por lo tanto, la complejidad del problema (medido en cantidad de comparaciones es):

$$T_M(n) \geq n - 1.$$

Sin embargo, esta cantidad de comparaciones coincide con la cantidad de comparaciones realizadas por el Algoritmo 1, por lo tanto tenemos que esta cota inferior es ajustada:

$$T_M(n) = n - 1,$$

y dicho algoritmo es óptimo. Esto demuestra que los enfoques de fuerza bruta no son necesariamente ineficientes. En ciertas ocasiones son la solución más eficiente que podemos usar.

8.3. El Problema de Ordenamiento

Estudiamos, por primera vez desde el comienzo, algoritmos para el problema de ordenamiento. Recuerde que en la Sección 7.2 encontramos que la complejidad del problema es $T_O(n) \in \Omega(n \log n)$. Como ya hemos dicho, este problema tiene numerosas aplicaciones, lo que justifica su estudio en profundidad. De hecho, muchas tareas sobre conjuntos de datos pueden llevarse a cabo de manera más eficiente si se realiza un paso previo de ordenamiento. Un ejemplo claro es el de búsqueda en un conjunto. Ya hemos visto que la complejidad del problema de búsqueda ordenada es bastante menor que la complejidad del problema de búsqueda en conjunto desordenado (logarítmica versus lineal). Asumiremos que el conjunto a ordenar está almacenado en un arreglo

A en la memoria, y que sobre el conjunto se ha definido una relación de orden total \triangleleft . Además, asumiremos el modelo de comparaciones, en donde se ordena únicamente mediante comparaciones directas entre elementos del conjunto.

Ordenamiento e Inversiones

Definición 8.3.1 Una inversión en el arreglo A ocurre cuando para dos posiciones $i < j$ ocurre que $A[j] \triangleleft A[i]$.

Por ejemplo, $A[2] = 3$ y $A[5] = 2$ es una inversión. La cantidad mínima de inversiones que puede tener un arreglo es 0: piense en un arreglo ordenado. Por otro lado, para entender la cantidad máxima de inversiones, piense en un arreglo ordenado a la inversa tal como $(8, 7, 6, 5, 4, 3, 2, 1)$. En ese caso, cada elemento $A[i]$, $1 \leq i \leq n$, introduce $n - i$ inversiones, y el total es

$$\sum_{i=1}^n n - i = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \approx \frac{n^2}{2}.$$

De esta forma, la cantidad de inversiones de un arreglo está acotada por $O(n^2)$.

Note que un arreglo está desordenado si y sólo si existe al menos una inversión. Para ordenar un arreglo mediante comparaciones de sus elementos, en general tendremos que deshacer las inversiones del mismo, hasta eventualmente ordenarlo. Para esto, será clave la función de intercambio de los valores almacenados en dos posiciones del arreglo, que se realizará con el procedimiento del Algoritmo 12. Se asume que los

Algoritmo 12: SWAP(x, y)

```

1 temp ← x
2 x ← y
3 y ← temp
```

parámetros son pasados por referencia, a fin de poder intercambiarlos efectivamente.

El Concepto de Estabilidad en el Ordenamiento

La siguiente definición introduce un concepto importante para los algoritmos de ordenamiento: el de *estabilidad*.

Definición 8.3.2 Un algoritmo de ordenamiento es estable si durante el proceso nunca cambia el orden relativo de los elementos que son idénticos.

Este concepto es de importancia cuando los elementos del arreglo a ordenar son compuestos, como por ejemplo un par ordenado (s_j, d_j) . En un caso así, si en el arreglo de entrada tenemos los elementos:

$$(\cdots, (s_i, d_k), \cdots, (s_i, d_l), \cdots, (s_i, d_m), \cdots),$$

un algoritmo de ordenamiento estable obtendrá el arreglo ordenado:

$$(\dots, (s_i, d_k), (s_i, d_l), (s_i, d_m), \dots),$$

lo que respeta el orden relativo que los elementos tenían en la entrada. Muchos de los algoritmos que estudiaremos serán estables, aunque no todos. Algunos otros pueden hacerse estables a partir de pequeños cambios a los algoritmos básicos que estudiemos. Otros pueden hacerse estables usando otro tipo de trucos.

Ejemplo 8.3.1 Un ejemplo de aplicación de algoritmos de ordenamiento estables es el siguiente. Suponga que por alguna razón necesita ordenar un arreglo de puntos en el plano. Los puntos deben estar ordenados por coordenada x , y en caso de haber puntos con la misma coordenada x , estos deben aparecer ordenados por coordenada y . Por ejemplo, si el arreglo es $((3, 5), (4, 2), (3, 2), (1, 7), (2, 5), (3, 1))$, el resultado de ordenarlo debe ser $((1, 7), (2, 5), (3, 1), (3, 2), (3, 5), (4, 2))$. Para resolver este problema, primero se ordenan los puntos por la coordenada y , en el caso del ejemplo obteniendo $((3, 1), (4, 2), (3, 2), (3, 5), (2, 5), (1, 7))$. Luego, en un segundo paso se ordenan los puntos de acuerdo a la coordenada x , usando un algoritmo estable para obtener $((1, 7), (2, 5), (3, 1), (3, 2), (3, 5), (4, 2))$. El uso de un algoritmo estable en el segundo paso es clave para no perder el ordenamiento relativo realizado en el primer paso sobre la coordenada y .

Otro ejemplo típico es un arreglo que almacena pares formados por nombres de ciudades y el país al que corresponden. Uno podría querer ordenar los pares por país, de forma tal que las ciudades de un mismo país queden ordenadas lexicográficamente. Esto se logra ordenando primero lexicográficamente por ciudades, para luego ordenar de forma estable por país. Estas suelen ser consultas típicas sobre una base de datos.

Cómo Comparar Algoritmos de Ordenamiento

Al momento de elegir un algoritmo de ordenamiento, suelen usarse los siguientes criterios de comparación:

- **Cantidad de comparaciones realizadas:** Es la forma más típica de comparar algoritmos en el modelo de comparaciones. Recuerde que hemos medido la complejidad del problema de ordenamiento usando esta medida.
- **Número de intercambios de registros realizados:** Aunque la complejidad del problema no está medida en el número de intercambios de elementos realizados (swaps) por el algoritmo, en la práctica intercambiar dos elementos del arreglo puede ser costoso. Por lo tanto, en ciertos casos es importante enriquecer el análisis con esta medida.
- **Simpleza del algoritmo:** Aunque en general preferimos el análisis asintótico de algoritmos, los términos constantes son muy importantes para el tiempo de ejecución si la entrada es muy pequeña, como por ejemplo, de 1000 elementos. Un algoritmo más simple tiene constantes más pequeñas multiplicando a los términos principales del tiempo de ejecución.

- **Estabilidad:** En ciertas aplicaciones, es necesario que el algoritmo de ordenamiento sea estable, tal como los ejemplos estudiados anteriormente para ordenar arreglos de pares.
- **Requerimientos de uso de memoria:** Ciertos algoritmos usan espacio extra de memoria por sobre el espacio requerido por el arreglo de entrada. Esta es una medida importante para comparar algoritmos, ya que un algoritmo que usa demasiada memoria podría necesitar acceder a memoria secundaria (e.g., el disco) para ordenar arreglos grandes, con la consecuente merma en el rendimiento del algoritmo.

En relación al último punto, definimos el siguiente concepto.

Definición 8.3.3 *Un algoritmo de ordenamiento es in-place si, para ordenar un arreglo de entrada, utiliza una cantidad $O(1)$ de memoria adicional, por sobre la memoria requerida por el arreglo de entrada.*

Con “cantidad $O(1)$ de espacio” nos referimos a una cantidad constante de memoria, i.e., que no dependa del tamaño del arreglo. Típicamente, corresponde a declarar un cierto número (constante) de variables simples, como por ejemplo variables enteras.

Bubble Sort

Como ya lo mencionamos, un arreglo está desordenado siempre y cuando contenga inversiones. Podríamos pensar que ordenar consiste en deshacer, de alguna forma, las inversiones del arreglo. El algoritmo **BubbleSort** (ordenamiento por burbujas) usa el siguiente enfoque de fuerza bruta: deshace todas las inversiones entre elementos consecutivos del arreglo, hasta asegurar que no haya ninguna (y por lo tanto el arreglo está ordenado). Éste es uno de los algoritmos más conocidos de ordenamiento, aunque en general no el más usado ¹.

El algoritmo funciona en $n - 1$ pasadas sobre el arreglo, cada una comenzando desde la primera posición del arreglo. Mantendremos en todo momento un límite superior para el arreglo, que será el punto en donde finaliza cada pasada. Originalmente, ese límite superior está a la derecha del último elemento del arreglo. La invariante que mantendremos es la siguiente: en todo momento, los elementos que están en el segmento del arreglo que está a la derecha del límite superior están ordenados, mientras que los del segmento izquierdo están aún desordenados. En cada paso, haremos crecer el tamaño del segmento derecho en 1, por lo cual el límite superior se irá moviendo desde la última posición a la primera durante la ejecución.

En la primera pasada, se recorre el arreglo A desde la primera posición hacia la última, comparando elementos adyacentes. Cada vez que se cumple $A[j] > A[j + 1]$, tenemos una inversión, por lo que esos objetos deben ser intercambiados. Note que, debido a esto, al finalizar la primera pasada, $A[n]$ almacenará el mayor elemento del arreglo. Esto es, los elementos más grandes “flotan” rápidamente hacia la parte superior del arreglo, como si fueran burbujas (y de ahí el nombre del algoritmo). Esto asegura que al finalizar la pasada i , para $i = 1, \dots, n - 1$, el segmento

¹ https://www.youtube.com/watch?v=k4RRi_ntQc8

Figura 8.2: Las 5 pasadas que BubbleSort realiza sobre el arreglo $A = (3, 4, 1, 14, 8, 2)$. Los valores subrayados en cada paso corresponden a comparaciones realizadas por el algoritmo. La zona del arreglo que está a la derecha de la barra vertical '|' es la parte ordenada del mismo. En cada pasada, un nuevo elemento es colocado en esa zona.

Primera pasada:			
$(\underline{3}, \underline{4}, 1, 14, 8, 2) \rightarrow$	$(3, \underline{4}, \underline{1}, 14, 8, 2) \rightarrow$	$(3, 1, \underline{4}, \underline{14}, 8, 2) \rightarrow$	
$(3, 1, 4, \underline{14}, \underline{8}, 2) \rightarrow$	$(3, 1, 4, 8, \underline{14}, \underline{2}) \rightarrow$	$(3, 1, 4, 8, 2, 14)$	
Segunda pasada:			
$(\underline{3}, \underline{1}, 4, 8, 2, 14) \rightarrow$	$(1, \underline{3}, \underline{4}, 8, 2, 14) \rightarrow$	$(1, 3, \underline{4}, \underline{8}, 2, 14) \rightarrow$	
$(1, 3, 4, \underline{8}, \underline{2}, 14) \rightarrow$	$(1, 3, 4, 2, 8, 14)$		
Tercera pasada:			
$(\underline{1}, \underline{3}, 4, 2, 8, 14) \rightarrow$	$(1, \underline{3}, \underline{4}, 2, 8, 14) \rightarrow$	$(1, 3, \underline{4}, \underline{2}, 8, 14) \rightarrow$	
$(1, 3, 2, 4, 8, 14)$			
Cuarta pasada:			
$(\underline{1}, \underline{3}, 2, 4, 8, 14) \rightarrow$	$(1, \underline{3}, \underline{2}, 4, 8, 14) \rightarrow$	$(1, 2, 3, 4, 8, 14)$	
Quinta pasada:			
$(\underline{1}, \underline{2}, 3, 4, 8, 14) \rightarrow$	$(1, 2, 3, 4, 8, 14)$		

$A[n - i + 1..n]$ está ordenado, lo cual permite mantener la invariante antes mencionada. En particular, luego de $n - 1$ pasadas, el segmento $A[2..n]$ estará ordenado. Pero, además, $A[1]$ va a contener el menor elemento del arreglo, por lo que en realidad $A[1..n]$ estará ordenado. El Algoritmo 13 implementa esta idea. El **while** principal implementa las

Algoritmo 13: BUBBLESORT($A[1..n]$)

```

1  $i \leftarrow 1$ 
2 while  $i < n$  do
3    $j \leftarrow 1$ 
4   while  $j \leq n - i$  do
5     if  $A[j + 1] < A[j]$  then
6        $\text{SWAP}(A[j], A[j + 1])$ 
7     end
8      $j \leftarrow j + 1$ 
9   end
10   $i \leftarrow i + 1$ 
11 end
```

$n - 1$ pasadas sobre el arreglo. El **while** interno implementa el recorrido en cada una de las pasadas. Note que se usa una cantidad $O(1)$ de espacio adicional por sobre el arreglo de entrada: el correspondientes a las variables i y j , usadas como índices para recorrer los arreglos. BubbleSort es, por lo tanto, un algoritmo de ordenamiento in-place.

Ejemplo 8.3.2 La Figura 8.2 muestra, paso a paso, la ejecución del algoritmo BubbleSort para el arreglo $(3, 4, 1, 14, 8, 2)$. En cada paso, se destaca la comparación realizada por el algoritmo (15 en total para este ejemplo, luego de 5 pasadas). También se muestra el segmento ordenado del arreglo, que es el que está a la derecha de '|'.

Análisis de Peor Caso. El **while** interno del Algoritmo 13 se ejecuta $n - i$ veces para cada valor de i , con $1 \leq i < n$. Es decir, la cantidad total de iteraciones es

$$\sum_{i=1}^{n-1} n - i = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

Cada una de esas iteraciones realiza una comparación de elementos, por lo tanto el tiempo de ejecución del algoritmo es $\Theta(n^2)$ comparaciones. Note que estamos introduciendo conocimiento nuevo respecto a la complejidad del problema de ordenamiento: $T_O(n) \in O(n^2)$. Anteriormente sabíamos que $T_O(n) \in \Omega(n \lg n)$. Por lo tanto, ya sabemos que la complejidad real del problema es algo entre $n \lg n$ y n^2 .

Finalmente, en el peor caso el algoritmo realiza un intercambio por iteración. Note que se invoca a SWAP sólo si la condición del **if** es verdadera. De hecho, esa condición se hará verdadera tantas veces como inversiones tenga el arreglo, por lo que la cantidad de intercambios realizados es igual a la cantidad de inversiones que tiene el arreglo de entrada. En resumen la cantidad de intercambios realizados es $O(n^2)$.

Ejercicio

Modifique el Algoritmo 13 para que realice $O(n^2)$ comparaciones.

Selection Sort

Note que en la pasada i , $1 \leq i < n$, **BubbleSort** coloca el mayor elemento del segmento desordenado $A[1..n-i+1]$ en la posición $A[n-i+1]$. Esto es, en la primera pasada, el mayor elemento del segmento $A[1..n]$ se coloca en la posición $A[n]$; en la segunda pasada, el mayor elemento del segmento $A[1..n-1]$ es colocado en la posición $A[n-1]$; y así siguiendo. Durante cada pasada, además, realiza intercambios de elementos que están invertidos, ordenándolos parcialmente. Sin embargo, esto último en general no es de mucha ayuda, ya que no disminuye la cantidad de comparaciones realizadas por el algoritmo. Por lo tanto, esos intercambios realizados no son beneficiosos, y terminan incrementando el tiempo total de ejecución en la práctica.

El algoritmo **SelectionSort** mejora esto, modificando sutilmente el procedimiento de **BubbleSort**. En cada pasada, el máximo del segmento desordenado flota hacia el segmento ordenado, pero esta vez sin hacer intercambios intermedios. Sólo se realiza un intercambio por pasada, haciendo que la cantidad total de intercambios sea $n-1$ (versus $O(n^2)$ de **BubbleSort**). El Algoritmo 14 implementa esta idea. El nombre del

Algoritmo 14: SELECTIONSORT($A[1..n]$)

```

1  $i \leftarrow 1$ 
2 while  $i < n$  do
3    $\text{max} \leftarrow n - i + 1$ 
4    $j \leftarrow 1$ 
5   while  $j \leq n - i$  do
6     if  $A[\text{max}] < A[j]$  then
7        $\text{max} \leftarrow j$ 
8     end
9      $j \leftarrow j + 1$ 
10  end
11  SWAP( $A[n - i + 1], A[\text{max}]$ )
12   $i \leftarrow i + 1$ 
13 end

```

Figura 8.3: Las 5 pasadas que SelectionSort realiza sobre el arreglo $A = (3, 4, 1, 14, 8, 2)$. Los valores subrayados en cada paso corresponden a comparaciones realizadas por el algoritmo. La zona del arreglo que está a la derecha de la barra vertical '|' es la parte ordenada del mismo. En cada pasada, un nuevo elemento es colocado en esa zona.

Primera pasada:					
$(\underline{3}, 4, 1, 14, 8, \underline{2})$	\rightarrow	$(\underline{3}, \underline{4}, 1, 14, 8, 2)$	\rightarrow	$(3, \underline{4}, \underline{1}, 14, 8, 2)$	\rightarrow
$(3, \underline{4}, 1, \underline{14}, 8, 2)$	\rightarrow	$(3, 4, 1, \underline{14}, \underline{8}, 2)$	\rightarrow	$(3, 4, 1, 2, 8 , 14)$	
Segunda pasada:					
$(\underline{3}, 4, 1, 2, \underline{8}, 14)$	\rightarrow	$(3, \underline{4}, 1, 2, \underline{8}, 14)$	\rightarrow	$(3, 4, \underline{1}, 2, \underline{8}, 14)$	\rightarrow
$(3, 4, 1, \underline{2}, \underline{8}, 14)$	\rightarrow	$(3, 4, 1, 2 , 8, 14)$			
Tercera pasada:					
$(\underline{3}, 4, 1, \underline{2}, 8, 14)$	\rightarrow	$(\underline{3}, \underline{4}, 1, 2 , 8, 14)$	\rightarrow	$(3, \underline{4}, \underline{1}, 2 , 8, 14)$	\rightarrow
$(3, 2, 1 , 4, 8, 14)$					
Cuarta pasada:					
$(\underline{3}, 2, \underline{1}, 4, 8, 14)$	\rightarrow	$(\underline{3}, \underline{2}, 1 , 4, 8, 14)$	\rightarrow	$(1, 2 , 3, 4, 8, 14)$	
Quinta pasada:					
$(\underline{1}, \underline{2}, 3, 4, 8, 14)$	\rightarrow	$(1, 2, 3, 4, 8, 14)$			

algoritmo viene de seleccionar, en cada pasada, el máximo elemento restante (recuerde el problema de selección, estudiado en la Sección 8.2). Como se puede ver, éste también es un algoritmo de ordenamiento in-place.

Análisis de Peor Caso. SelectionSort realiza $\Theta(n^2)$ comparaciones, al igual que BubbleSort. Sin embargo, reduce el número de intercambios a $\Theta(n)$. Esto es ventajoso en general, en particular cuando los elementos a ordenar son demasiado grandes y un intercambio requiere varios ciclos del procesador. En la práctica, usualmente SelectionSort es más eficiente que BubbleSort.

Ejercicio

Modifique la implementación de SelectionSort para que haga $O(n)$ intercambios.

8.4. La Fila de Monedas

Considere n monedas dispuestas en una fila sobre la mesa, posiblemente de distintas denominaciones. El problema consiste en tomar la mayor cantidad de dinero desde la fila, sujeto a la restricción de que cada vez que se escoge una moneda, quedan invalidadas las dos monedas contiguas (en la fila original) a la moneda escogida.

Para entender bien el problema, y cómo solucionarlo, estudiemos una solución por fuerza bruta recursiva. Sea $F(n)$ el algoritmo que resuelve el problema para una fila de n monedas. Sea $m[1..n]$ el arreglo que almacena las denominaciones de las n monedas. Definamos a continuación los casos base de la recursión. Note que $F(0) = 0$, ya que si no hay monedas en la fila, no podemos recoger nada. Además, $F(1) = m[1]$, ya que si tenemos una fila con una única moneda, la única alternativa es quedarnos con ella. Una vez definidos los casos base, podemos asumir que $F(i)$ calcula la cantidad óptima de dinero que podemos recoger en una fila de i monedas, para $0 \leq i \leq n - 1$ monedas. Sólo resta mostrar cómo usar este hecho para calcular la solución para $n \geq 2$ monedas.

Note que para una fila de n monedas, tenemos dos alternativas:

1. Recoger la moneda $m[n]$. Esto invalida la moneda $m[n-1]$, y sólo podemos seguir recogiendo entre las $n-2$ primeras monedas. La cantidad óptima que podemos recoger entre las primeras $n-2$ monedas es calculada por $F(n-2)$, por hipótesis inductiva. El total recogido con esta alternativa es $F(n-2) + m[n]$.
2. No recoger la moneda $m[n]$. Por lo tanto, podemos recoger entre las $n-1$ primeras monedas. La cantidad óptima para esta alternativa es calculada por $F(n-1)$, por hipótesis inductiva.

El óptimo es, obviamente, el máximo entre esas dos alternativas. La ecuación de recurrencia resultante para este algoritmo recursivo es la siguiente:

$$F(n) = \begin{cases} \text{máx}\{F(n-2) + m[n], F(n-1)\}, & n \geq 2; \\ 0, & n = 0; \\ m[1], & n = 1. \end{cases} \quad (8.1)$$

El Algoritmo 15 implementa esta idea, usando fuerza bruta. El tiempo

Algoritmo 15: $F(m[1..n])$

```

1 if n = 0 then
2   | return 0
3 else
4   | if n = 1 then
5   |   | return m[1]
6   | else
7   |   | return máx { F(m[1..n-2]) + m[n],
8   |   |               F(m[1..n-1]) }.
9   | end
10 end
```

de ejecución de este algoritmo es $\Theta(\varphi^n)$, porque el proceso recursivo es similar al de Fibonacci: $F(n)$ es calculado en base a $F(n-1)$ y $F(n-2)$.

8.5. Búsqueda en Texto

El problema de búsqueda en texto consiste en encontrar todas las ocurrencias de un patrón $P[1..m]$, de largo m , en un texto $T[1..n]$, de largo n . Esto es, todas las posiciones i tal que $0 \leq i \leq n - m + 1$ y $T[i..i+m-1] = P[1..m]$. Es común que $m \ll n$. Éste es un problema fundamental para muchas aplicaciones, como la función “Buscar” en editores de texto (recuerde la típica secuencia de teclas Ctrl+F para activar esta función), la herramienta `grep` de Unix/Linux, búsqueda en bases de datos textuales, y búsqueda en bases de datos biológicas, entre otras.

Ejemplo 8.5.1 Dado el texto $T[1..20] = \text{“alabar_a_la_alabarda”}$, existen 3 ocurrencias del patrón $P[1..2] = \text{“la”}$:

alabar_a_la_alabarda

Algoritmo de Búsqueda Exhaustiva

El enfoque de búsqueda exhaustiva para este problema consiste en considerar todas las posiciones del texto, y chequear si allí comienza una ocurrencia del patrón P , o no. El Algoritmo 16 ilustra esta idea. Su

Algoritmo 16: BÚSQUEDA_{TEXTOFB}($P[1..m]$)

```

1  $i \leftarrow 1$ 
2 while  $i \leq n - m$  do
3    $j \leftarrow 1$ 
4   while  $j \leq m \wedge P[j] = T[i + j]$  do
5      $j \leftarrow j + 1$ 
6   end
7   if  $j \geq m$  then
8     Reportar la posición  $j$ 
9   end
10   $i \leftarrow i + 1$ 
11 end

```

tiempo de ejecución es $O(nm)$, ya que realiza $O(m)$ comparaciones por cada una de las n posiciones del texto. Esto puede considerarse eficiente en un escenario en el que m es muy pequeño (e.g., $m \geq 2$). Incluso, note que el tiempo total es $\Theta(n)$ si $m \in O(1)$, lo cual no necesariamente puede considerarse eficiente en la práctica porque las constantes ocultas en $O(1)$ podrían ser grandes. En general, este algoritmo es ineficiente si m es grande. El principal problema del algoritmo es que en cada iteración del **while** exterior, el avance (o *shift*) que se hace es de tamaño 1. Aunque es un shift seguro, en el sentido de que nunca perderá ninguna ocurrencia del patrón, puede ser ineficiente.

Algoritmo de Knuth, Morris, y Prat

Estudiaremos a continuación el algoritmo de Knuth, Morris, y Prat (KMP), que permite avanzar más rápidamente sobre el texto, sin perder ninguna ocurrencia del patrón. Básicamente, al momento en que falla una comparación entre un carácter del texto contra un carácter del patrón, es decir $P[j] \neq T[i]$, el algoritmo usa parte del trabajo ya hecho al comparar $P[1..j - 1]$ con los correspondientes caracteres del texto para hacer shifts más grandes.

Considere la Figura 8.4, que ilustra una situación típica durante el proceso de búsqueda del patrón. En algún momento dado, el algoritmo alinea el patrón con una ventana del texto, de tamaño m . La ventana inicial son los primeros m caracteres del texto. En la figura, la ventana está marcada con líneas punteadas rojas. Una vez alineados, los caracteres del patrón son comparados con los de la ventana para determinar si la ventana es una ocurrencia del patrón. Cuando un carácter del patrón es distinto al correspondiente del texto, se avanza la ventana. Analizamos a continuación cuánto podemos avanzar sin perder ninguna ocurrencia de P en T .

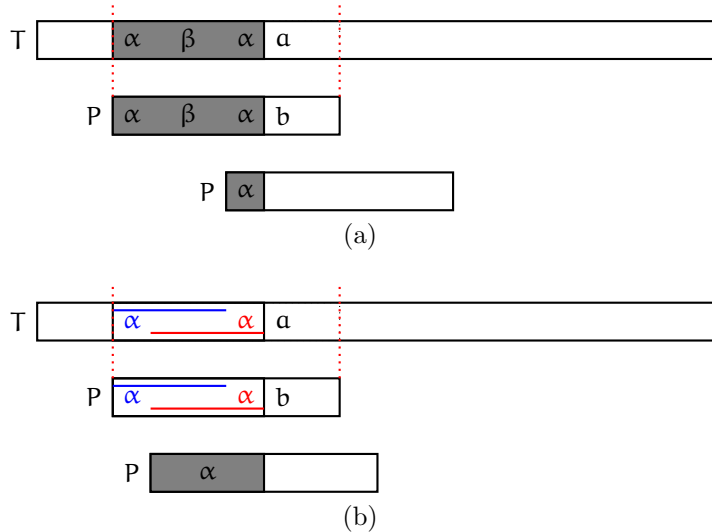


Figura 8.4: Situación típica para el algoritmo KMP, en el que se falla al comparar $T[i] = a \neq P[j] = b$, después de que $P[1..j-1]$ coincidiera con los respectivos caracteres del texto. Sea α el sufijo propio más largo de $P[1..j-1]$ que también es prefijo de $P[1..j-1]$. La parte (a) muestra el caso en que ambas ocurrencias de α son disjuntas, mientras que la parte (b) muestra el caso en que ambas ocurrencias de α se solapan. El shift debe alinear el prefijo α de $P[1..j-1]$ con el sufijo α en el texto, tal como se muestra en cada caso.

Supongamos que, comparando dentro de la ventana, fallamos en encontrar una ocurrencia del patrón ya que $P[j] \neq T[i]$, para algún $1 \leq j \leq m$. Note que $P[1..j-1] = T[i-j+1..i-1]$. A continuación, hay que avanzar la ventana a una posición i' del texto tal que $P[1..i-i'] = T[i'..i-1]$. Note que $P[1..j-1]$ tiene a $T[i'..i-1]$ tanto como prefijo y como sufijo. Para no perder ninguna ocurrencia de P , i' debe ser el mínimo valor en el intervalo $[i-j+1..i]$ para el que $P[1..i-i'] = T[i'..i-1]$. Llamaremos $\alpha = T[i'..i-1]$. En otras palabras, el string α es el sufijo propio más largo de $P[1..j-1]$ que también es prefijo de $P[1..j-1]$ (“sufijo propio” significa que la longitud de α es $0 \leq |\alpha| < j-1$). Esta es la situación que ilustra la Figura 8.4. Tal como se muestra, tenemos dos casos para α :

1. $P[1..j-1] = \alpha\beta\alpha$, tal que β es un string no vacío. Esto significa que las dos ocurrencias de α que nos interesan no se solapan. La Figura 8.4 (a) ilustra esto.
2. $P[1..j-1] = xyxyx$, para strings x, y , tal que $\alpha = xyx$. Esto significa que las dos ocurrencias de α se solapan dentro de $P[1..j-1]$. La Figura 8.4 (b) ilustra esta situación.

Un ejemplo del primer caso es el string “cgctacgc”, en donde $\alpha = \text{“cgc”}$ y $\beta = \text{“ta”}$. Para el segundo caso, un ejemplo es el string “abcdabcdab”, en donde $\alpha = \text{“abcdab”}$.

Dado que $P[j] \neq T[i]$, debemos avanzar la ventana para continuar la búsqueda. Note que si avanzamos la ventana de manera que el sufijo α en el texto se alinee con el prefijo α en el patrón, no perdemos ninguna ocurrencia de P en el texto. Si hubiese alguna ocurrencia que comienza en una posición anterior al sufijo α , entonces α no es el sufijo más largo que también es prefijo de $P[1..j-1]$.

Para poder determinar el largo del string α en cualquier punto de la ejecución del algoritmo, KMP precalcula una tabla, a la que llamaremos $\text{kmpS}[1..m+1]$, con información que permite hacer estos shifts seguros. En particular:

$$\text{kmpS}[j] = \begin{cases} |\alpha| & \text{para } P[1..j-1], \quad j > 1; \\ 0, & j = 1. \end{cases}$$

Eso significa que cuando fallemos al comparar el carácter $P[j]$, la entrada $kmpS[j]$ nos dirá el largo del sufijo α correspondiente. El algoritmo entonces hace $j \leftarrow kmpS[j] + 1$, lo que significa haber alineado el carácter $T[i]$ del texto con el carácter que está inmediatamente después del prefijo α de $P[1..j - 1]$. El algoritmo continua desde esa posición, evitando comparar nuevamente α , que ya sabemos que ocurre tanto en el texto como en el patrón.

La siguiente propiedad es importante para demostrar el tiempo de ejecución del algoritmo KMP:

Lema 8.5.1 *Dado un patrón $P[1..m]$ y su correspondiente tabla $kmpS[1..m + 1]$, $\forall j = 1, \dots, m + 1$ se cumple que*

$$kmpS[j] < j.$$

El Algoritmo 17 muestra el proceso para computar la tabla $kmpS$.

Algoritmo 17: CONSTRUIRKMP($P[1..m]$)

```

1  $i \leftarrow 1$ 
2  $j \leftarrow 0$ 
3 Sea  $kmpS[1..m + 1]$  un arreglo de enteros
4  $kmpS[1] \leftarrow 0$ 
5 while  $i \leq m$  do
6   while  $j \geq 1 \wedge P[i] \neq P[j]$  do
7      $j \leftarrow kmpS[j]$ 
8   end
9    $i \leftarrow i + 1$ 
10   $j \leftarrow j + 1$ 
11   $kmpS[i] \leftarrow j$ 
12 end
13 return  $kmpS$ 
```

Ejemplo 8.5.2 Si usamos el Algoritmo 17 para el string “abcdabcdab”, obtenemos:

	1	2	3	4	5	6	7	8	9	10	11
P:	a	b	c	d	a	b	c	d	a	b	
kmpS:	0	0	0	0	0	1	2	3	4	5	6

Dado esto, si en alguna situación fallamos en la posición 7, $kmpS[7] = 2$ significa que j debe reconfigurarse desde la posición 3, luego del prefijo ab.

Respecto al tiempo de ejecución del Algoritmo 17, podemos demostrar el siguiente Lema.

Lema 8.5.2 *Dado un patrón $P[1..m]$, el Algoritmo 17 calcula la tabla $kmpS[1..m + 1]$ realizando $\Theta(m)$ comparaciones de caracteres.*

Demostración. Note que la variable j del Algoritmo 17 es incrementada en total m veces, en la línea 10 del algoritmo. Dado que inicia en 0,

el máximo valor que puede llegar a tomar j es m . Como el **while** de la línea 6 itera mientras $j \geq 1$, y la asignación de la línea 7 siempre decrementa el valor de j (recuerde la Propiedad 8.5.1), entonces en total ese **while** itera a lo más m veces. Eso significa que la comparación $P[i] \neq P[j]$ se ejecuta $\Theta(m)$ veces. ■

Como primer paso en su ejecución, el algoritmo KMP construye la tabla $kmpS$ para el patrón. Luego, la utiliza para agilizar el proceso de búsqueda, usando un proceso similar al usado para construir la tabla. Esto puede verse en el Algoritmo 18.

Algoritmo 18: KMP($T[1..n]$, $P[1..m]$)

```

1  $i \leftarrow 1$ 
2  $j \leftarrow 1$ 
3  $kmpS[1..m+1] \leftarrow \text{CONSTRUIRKMP}(P)$ 
4 while  $i \leq n$  do
5   while  $j \geq 1 \wedge T[i] \neq P[j]$  do
6      $j \leftarrow kmpS[j] + 1$ 
7   end
8    $i \leftarrow i + 1$ 
9    $j \leftarrow j + 1$ 
10  if  $j = m + 1$  then
11    reportar  $i - j$ 
12     $j \leftarrow kmpS[j] + 1$ 
13  end
14 end

```

Teorema 8.5.3 *Dado un texto $T[1..n]$ y un patrón $P[1..m]$, el Algoritmo 18 permite encontrar todas las ocurrencias de P en T realizando $\Theta(n + m)$ comparaciones de caracteres.*

Demostración. La línea 3 tiene un costo de $\Theta(m)$ comparaciones, como ya vimos. Para demostrar el resto, se procede de forma similar a la demostración del Lema 8.5.2, la única diferencia es que j es incrementada n veces esta vez, lo que provoca que el **while** de la línea 5 ejecute n veces en total, por lo que la comparación $T[i] \neq P[j]$ se ejecuta n veces en total. En total, tenemos $\Theta(n + m)$ comparaciones. ■

La Figura 8.5 muestra la ejecución del algoritmo KMP, paso a paso, para el texto $T = \text{"SASASQQSASNRSAA"}$ y el patrón $P = \text{"SASAD"}$. El ejemplo muestra (en rojo) el carácter en que se falla en cada paso, y sombreado en gris el string α correspondiente.

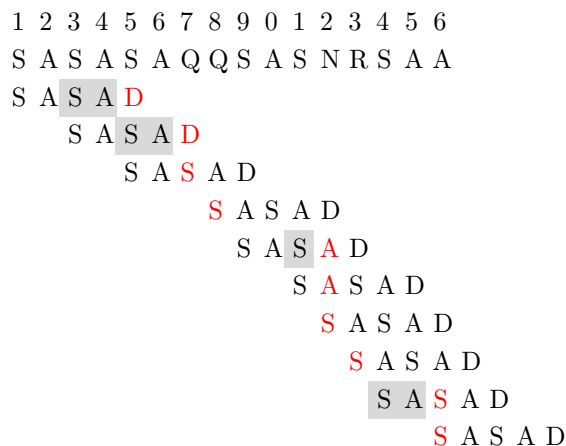


Figura 8.5: Ejemplo de ejecución del algoritmo KMP para el texto $T = \text{"SASASAQQSASNRSA"}$ y el patrón $P = \text{"SASAD"}$.

Ejercicios

1. a) Diseñe un algoritmo de fuerza bruta para calcular el polinomio

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

para un valor dado de x , y determine su eficiencia de peor caso.

- b) ¿Es posible diseñar un algoritmo más eficiente?
- c) ¿Es posible diseñar un algoritmo con eficiencia mejor que lineal en el peor caso para resolver este problema? Justifique su respuesta.
2. Hay n pilas de n monedas idénticas. Todas las monedas en una de esas pilas son falsas, mientras que todas las monedas en el resto de las pilas son genuinas. Cada moneda genuina pesa 10 gramos. Cada moneda falsa pesa 11 gramos. Usted dispone de una balanza que permite determinar el peso exacto de cualquier cantidad de monedas que necesiten pesarse.
 - a) Diseñe un algoritmo de fuerza bruta para identificar la pila con las monedas falsas y determine la eficiencia de peor caso.
 - b) ¿Cuál es el mínimo número de pesos que se deben hacer en el peor caso para identificar la pila con las monedas falsas?
 - c) De acuerdo a su respuesta anterior, ¿es posible mejorar su algoritmo del punto a)?
3. Se tiene una fila de $2n$ discos de color blanco (B) y negro (N), colocadas de manera alternada: negro, blanco, negro, blanco, etc. Se quiere colocar todos los discos negros en el extremo izquierdo de la fila, y los blancos en el extremo derecho. La única operación permitida para llevar a cabo el proceso es el intercambio de dos discos consecutivos en la fila. Por ejemplo, para $n = 4$, se tiene

$$N, B, N, B, N, B, N, B \implies N, N, N, N, B, B, B, B.$$

Diseñe un algoritmo para resolver este problema y determine el número de movimientos que se realizan.

4. El algoritmo de ordenamiento **SelectionSort** estudiado en clases, ¿es estable? Cualquiera sea su respuesta, justifíquela. En caso de

- no serlo, modifique el código visto en clases para que sea estable.
5.
 - a) Pruebe que si `BubbleSort` no hace intercambios en una de sus pasadas, el arreglo está ordenado y puede detenerse.
 - b) Modifique el código fuente del algoritmo visto en clases para que introduzca esta variante.
 - c) Haga el análisis de su algoritmo, y use la notación asintótica que crea más adecuada para acotar el tiempo de ejecución. Compare con la notación asintótica obtenida en clases.
 6. El algoritmo de ordenamiento `BubbleSort` estudiado en clases, ¿es estable? Cualquiera sea su respuesta, justifíquela. En caso de no serlo, modifique el código visto en clases para que sea estable.
 7. Se conoce como *merge* al problema de mezclar dos arreglos ordenados $A[1..n]$ y $B[1..m]$ para producir un único arreglo ordenado de tamaño $n + m$, el cual contiene los elementos de A y de B . Es sabido que esta operación puede implementarse de manera óptima con $n + m - 1$ comparaciones en el peor caso. Suponga que el algoritmo `merge($A[1..n]$, $B[1..m]$)` implementa dicha operación. Estamos interesados en generalizar la solución a k arreglos: el algoritmo `k-Merge(A_1, A_2, \dots, A_k)` recibe k arreglos ordenados de tamaño n cada uno, y los mezcla para producir un único arreglo ordenado de tamaño kn . Diseñe el algoritmo de **fuerza bruta** para resolver este problema, usando como base la operación `merge` mencionada anteriormente. Analice su algoritmo para obtener $T(k)$, el tiempo de ejecución para mezclar k arreglos.
 8. Sean $A[1 \dots n]$ y $B[1 \dots n]$ dos arreglos ordenados de números enteros positivos, de tamaño n cada uno. Se quiere encontrar la mediana del arreglo de tamaño $2n$ que contiene los elementos de A y los elementos de B . Recuerde que la mediana de un arreglo de tamaño n es el elemento que ocuparía la posición $\lfloor (n + 1)/2 \rfloor$ si el arreglo estuviera ordenado. Los arreglos A y B han sido almacenados en la memoria principal de una RAM, y se tiene la restricción de que la cantidad de espacio extra disponible para resolver el problema es $\Theta(1)$. Diseñe el algoritmo `mediana($A[1 \dots n]$, $B[1 \dots n]$)`, el cual resuelve el problema por fuerza bruta en tiempo $\Theta(n)$.
 9. Un string de caracteres s de largo $|s|$ tiene período k si puede ser formado concatenando una o más repeticiones de otro string de largo k . Por ejemplo, el string “`abcabcabcabc`” tiene período 3, ya que es formado por 4 repeticiones del string “`abc`”. Ese mismo string también tiene períodos 6 (2 repeticiones del string “`abcabc`”) y 12 (1 repetición del string `abcabcabcabc`). Diseñe un algoritmo que determine el menor período de un string. El tiempo de ejecución debería ser $\Theta(|s|)$.

9.1. Introducción

La programación dinámica fue utilizada en los años 50 por Richard Bellman, como un método general para optimizar procesos de decisión de múltiples etapas. Por lo tanto la palabra “programación” se refiere a “planificación”, y no a la programación de computadores. Nosotros la usaremos como técnica de diseño de algoritmos. La programación dinámica permite diseñar algoritmos que buscan de manera sistemática sobre todas las posibles soluciones a un problema dado (por lo tanto inmediatamente garantizando correctitud, como la búsqueda exhaustiva). Al igual que la técnica de dividir y conquistar, la programación dinámica busca resolver un problema dividiéndolo en subproblemas más pequeños. Sin embargo, y a diferencia de dividir y conquistar, la programación dinámica es de especial interés para resolver problemas con subproblemas que se solapan (es decir, subproblemas no totalmente independientes). Si se utiliza simplemente fuerza bruta para resolver un problema de esa naturaleza (por ejemplo, usando un algoritmo recursivo), observaremos que por diferentes ramas del proceso de cómputo, un mismo subproblema es resuelto una y otra vez, repetidamente. Esto es, sin dudas, ineficiente, y la causa de los altos tiempos de ejecución en esos casos. Si por una de las ramas del cómputo resolvemos un subproblema que luego debe ser resuelto en otras de las ramas, no deberíamos tener que resolverlo nuevamente: podemos usar la respuesta obtenida en el proceso anterior, y podar el árbol de cómputo en esa rama (ahorrando tiempo!). Muchos autores consideran a la programación dinámica como un enfoque de fuerza bruta con almacenamiento de resultados parciales. La clave de la técnica está en determinar el orden en que los subproblemas son resueltos, para hacerlo de forma eficiente. En las secciones siguientes ilustramos el uso de esta técnica, aplicada a diversos problemas de interés.

9.2. Números de Fibonacci

Un ejemplo típico de problema con subproblemas que se solapan es el cálculo del n -ésimo número en la serie de Fibonacci, definida por la recurrencia:

$$F(n) = \begin{cases} F(n-1) + F(n-2), & n \geq 2; \\ 0, & n = 0; \\ 1, & n = 1. \end{cases} \quad (9.1)$$

Más allá de que este ejemplo es bastante simple, y existen maneras más eficientes de calcular el n -ésimo número de Fibonacci, podremos ilustrar de forma sencilla muchos de los aspectos de la programación dinámica.

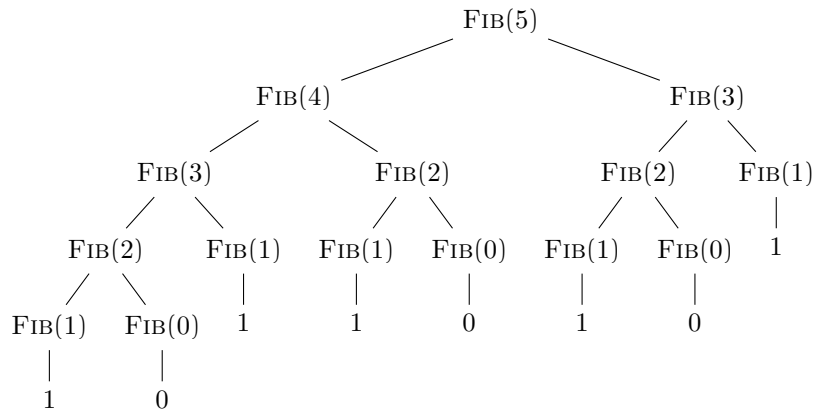


Figura 9.1: Árbol de recursión para el cálculo de FIB(5).

La Ecuación (9.1) puede implementarse directamente de forma recursiva como ya lo estudiamos en capítulos anteriores, y se muestra nuevamente en el Algoritmo 19. Al ejecutar este algoritmo para algún valor particular

Algoritmo 19: FIB(n)

```

1 if  $n \leq 1$  then
2   | return  $n$ 
3 else
4   | return FIB( $n - 1$ ) + FIB( $n - 2$ )
5 end

```

de n se puede observar el solapamiento y repetición de subproblemas. Considere calcular FIB(5). La Figura 9.1 muestra el árbol de recursión generado por el algoritmo. Como se puede observar, para calcular FIB(5), necesitamos calcular FIB(4) y FIB(3). A su vez, para resolver FIB(4), necesitamos resolver FIB(3) y FIB(2). En particular, note cómo en dos ramas distintas de la ejecución necesitamos resolver FIB(3), ilustrando el solapamiento de subproblemas. Cada una de esas ramas va a repetir exactamente el mismo cómputo.

Pero, ¿Qué tanto influye la resolución repetitiva de subproblemas en el tiempo de ejecución del algoritmo para Fibonacci? Primero, note que el número de Fibonacci a calcular es exactamente igual a la cantidad de 1s en las hojas del árbol de recursión. Esos 1s son sumados a lo largo de las distintas ramas para terminar conformando el resultado final. Por ejemplo, FIB(5) = 5, lo que corresponde a los 5 1s en las hojas del árbol de la Figura 9.1. Segundo, recuerde que FIB(n) $\approx \varphi^n$, en donde $\varphi = (1 + \sqrt{5})/2 = 1,618 \dots$ es la conocida razón (o proporción) áurea. Esto significa que el árbol de cómputo de FIB(n) tiene al menos φ^n hojas con valor 1. Dado que cada nodo interno del árbol tiene 2 hijos, la cantidad total de nodos en el árbol es $\Theta(\varphi^n)$. Esto es, una cantidad exponencial de nodos, por lo que el tiempo de ejecución del Algoritmo 19 es exponencial. Resolver el mismo subproblema repetidas veces tiene sus consecuencias.

A continuación, estudiamos un enfoque que sólo resuelve un subproblema la primera vez que es necesario hacerlo, y luego almacena el resultado para futuros usos. De esta forma, antes de resolver un subproblema debemos chequear si ya ha sido resuelto anteriormente o no, y así reusar el resultado si ya ha sido almacenado. La idea es mantener

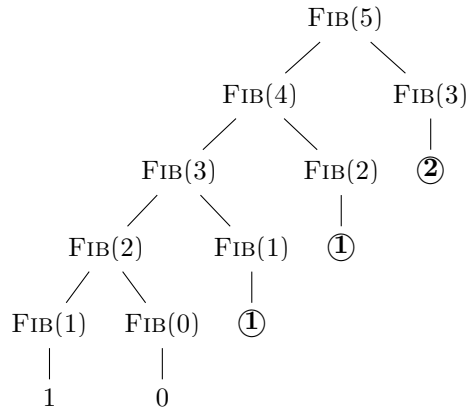


Figura 9.2: Árbol de recursión para FIB(5), con chequeo de subproblemas ya resueltos y almacenamiento de soluciones parciales.

un arreglo `memo[0..n]`, tal que `memo[i]` almacenará el resultado de FIB(i) cuando éste haya sido calculado. Para poder determinar si un subproblema FIB(i) ya ha sido computado anteriormente, necesitamos una marca especial. En este caso particular, podemos usar el valor -1 como marca, dado que los resultados de FIB son ≥ 0 . Inicialmente, todas las posiciones del arreglo `memo` almacenan esta marca. A medida que el cómputo avanza, esas marcas son reemplazadas por los resultados correspondientes. Esta técnica en la que se van almacenando las soluciones de los subproblemas ya resueltos es conocida como *memoization* (sí, está bien escrita!). El Algoritmo 20 muestra la implementación correspondiente. El árbol de recursión resultante es mostrado en la

Algoritmo 20: FIB-MEMO(n)

```

1 if  $n \leq 1$  then
2   return  $n$ 
3 else
4   if memo[n] = -1 then
5     memo[n] ← FIB-MEMO(n - 1) + FIB-MEMO(n - 2)
6   end
7   return memo[n]
8 end

```

Figura 9.2. Es fácil ver que la cantidad de nodos del árbol ahora es $\Theta(n)$, dado que únicamente se calcula la rama de más a la izquierda. El resto de los nodos usa los resultados ya almacenados para podar el cómputo (han sido marcados en negrita y encerrados en un círculo en el árbol de la Figura 9.2). El tiempo de ejecución del algoritmo es, por lo tanto, lineal. Note las consecuencias en el tiempo de ejecución al evitar resolver subproblemas repetidamente, permitiendo reducir el tiempo de exponencial a lineal. Para lograrlo hemos incrementado el uso de espacio a $\Theta(n)$ celdas de memoria.

Sin embargo, y más allá de la importante reducción de tiempo de ejecución que hemos logrado, el algoritmo puede ser implementado más eficientemente. Aquí no nos referimos a una mejora en términos asintóticos, sino más bien a una mejora de las constantes que multiplican el término lineal en el tiempo de ejecución. Para lograr esto, note que las dos implementaciones que hemos usado construyen el árbol de cómputo de forma top-down (i.e., desde la raíz hacia las hojas). Es decir, comienzan con el problema original (la raíz del árbol), y van resolviendo

subproblemas cada vez más pequeños para resolver el problema original. Como ya vimos, esas soluciones parciales se almacenan en una tabla, que es llenada a medida que se van resolviendo los subproblemas. Para lograr una implementación más práctica de Fibonacci, estudiaremos una técnica conocida como programación dinámica bottom-up. La idea es construir el árbol de cómputo desde las hojas hacia la raíz. En particular, las hojas corresponden a los casos base de la recursión. Luego se usa esto para resolver los problemas más pequeños, los cuales son luego usados para construir la solución a problemas cada vez más grandes, hasta llegar a resolver el problema completo. Este proceso será implementado iterativamente, recorriendo el arreglo f en un orden especificado. Dado que en este caso particular tenemos que una entrada $f[i]$ del arreglo depende de $f[i - 1]$ y $f[i - 2]$ (para $i = 2, \dots, n$), eso indica un orden de llenado del arreglo de izquierda a derecha. De esta manera, no hay necesidad de usar una marca para indicar qué subproblemas ya han sido resueltos. El Algoritmo 21 muestra una posible implementación del algoritmo. El tiempo de ejecución es, claramente, $\Theta(n)$. El espacio

Algoritmo 21: FIB-PD(n)

```

1  $f[0] \leftarrow 0$ 
2  $f[1] \leftarrow 1$ 
3 for  $i \leftarrow 2, \dots, n$  do
4    $f[i] \leftarrow f[i - 1] + f[i - 2]$ 
5 end
```

usado, por otro lado, es $\Theta(n)$ celdas de memoria.

Para finalizar, es importante notar que en muchos casos no será necesario almacenar el arreglo f usado para el cómputo de los números de Fibonacci. En particular, para calcular $f[i]$ sólo necesitamos las dos posiciones anteriores del arreglo, $f[i - 1]$ y $f[i - 2]$. El Algoritmo 22 muestra una implementación de Fibonacci que usa $\Theta(1)$ celdas de espacio, y calcula $\text{fib}(n)$ en tiempo $\Theta(n)$.

Algoritmo 22: FIB-EC(n)

```

1 if  $n \leq 1$  then
2   return  $n$ 
3 else
4    $f_2 \leftarrow 0$ 
5    $f_1 \leftarrow 1$ 
6   for  $i \leftarrow 2, \dots, n - 1$  do
7      $f \leftarrow f_1 + f_2$ 
8      $f_2 \leftarrow f_1$ 
9      $f_1 \leftarrow f$ 
10  end
11  return  $f_1 + f_2$ 
12 end
```

9.3. Elementos Generales de la Programación Dinámica

El ejemplo estudiado en la sección anterior ilustra muchos de los elementos y principales conceptos relacionados a la técnica de programación dinámica. Mencionamos a continuación los puntos más importantes:

1. Primero, se debe definir la ecuación de recurrencia que resuelva el problema por fuerza bruta. Esta ecuación será la base para construir, más tarde, el algoritmo de programación dinámica.
2. Definir una tabla de programación dinámica M , que se usará para la memoization. Relacionar las entradas de la tabla con los subproblemas generados recursivamente por la ecuación de recurrencia del punto anterior. La tabla debe tener tantas entradas como distintos subproblemas existan.
3. Comenzar llenando las entradas de la tabla correspondientes a los casos base (recordar que esas son las hojas del árbol de recursión).
4. Definir la dependencia entre entradas de M , indicando para cada entrada qué otras entradas son necesarias para calcularla. Usar la ecuación de recurrencia para esto.
5. Definir el orden en que las entradas de M serán llenadas. Esto queda claro al determinar la dependencia entre entradas de la tabla del punto anterior.
6. Especificar cómo obtener la solución final luego de haber llenado la tabla.

Tal como se estudió en el caso de Fibonacci, en muchas situaciones no es necesario almacenar toda la tabla a medida que se lleva a cabo el cómputo, sino que únicamente las entradas de la tabla que sean necesarias para calcular la siguiente entrada de la misma.

9.4. La Fila de Monedas

Consideremos nuevamente el problema de las n monedas dispuestas en una fila sobre la mesa, de la que podemos recoger tanto dinero como sea posible, sujetos a la restricción de que cada vez que se recoge una moneda, se invalidan las dos monedas contiguas. El objetivo es, por supuesto, tomar la mayor cantidad de dinero respetando esas reglas.

Para resolver este problema usando programación dinámica, pensemos primero en la solución de fuerza bruta (recursiva). Sea $F(n)$ el algoritmo que resuelve el problema para una fila de n monedas. Sea $m[1..n]$ el arreglo que almacena las denominaciones de las n monedas. Definamos a continuación los casos base de la recursión. Note que $F(0) = 0$, ya que si no hay monedas en la fila, no podemos recoger nada. Además, $F(1) = m[1]$, ya que si tenemos una fila con una única moneda, debemos quedarnos con ella. Una vez definidos los casos base, podemos asumir que $F(i)$ calcula la respuesta correctamente para $0 \leq i \leq n-1$ monedas. Sólo resta mostrar cómo usar este hecho para calcular la solución para $n \geq 2$ monedas.

Note que para considerar la n -ésima moneda de la fila tenemos dos alternativas:

1. Recoger la moneda $m[n]$. Esto invalida la moneda $m[n-1]$, y sólo podemos seguir recogiendo entre las $n-2$ primeras monedas. La cantidad óptima que podemos recoger allí es $F(n-2)$, por lo que el total para esta alternativa es $F(n-2) + m[n]$.
2. No recoger la moneda $m[n]$. Por lo tanto, podemos recoger entre las $n-1$ primeras monedas. La cantidad óptima para esta alternativa es calculada por $F(n-1)$.

El óptimo es, obviamente, el máximo entre esas dos alternativas. La ecuación de recurrencia resultante es la siguiente:

$$F(n) = \begin{cases} \text{máx}\{F(n-2) + m[n], F(n-1)\}, & n \geq 2; \\ 0, & n = 0; \\ m[1], & n = 1. \end{cases} \quad (9.2)$$

El Algoritmo 23 muestra una posible implementación usando fuerza bruta recursiva. El tiempo de ejecución de este algoritmo es $\Theta(\varphi^n)$,

Algoritmo 23: $F(m[1..n])$

```

1 if n = 0 then
2   return 0
3 else
4   if n = 1 then
5     return m[1]
6   else
7     return máx { F(m[1..n-2]) + m[n],
8                  F(m[1..n-1]) }.
9   end
10 end
```

porque el proceso recursivo es similar al de Fibonacci: $F(n)$ es calculado en base a $F(n-1)$ y $F(n-2)$.

En base a esta primera solución, planteamos a continuación el algoritmo de programación dinámica correspondiente. Sea $f[0..n]$ el arreglo de programación dinámica tal que $f[i]$ representará el valor de $F(i)$. Para implementar la solución bottom-up, debemos comenzar con las hojas del árbol de recursión, que corresponden a los casos bases. Eso significa que asignamos $f[0] = 0$ y $f[1] = m[1]$. Luego, calculamos el resto de las entradas del arreglo. Dado que para $i = 2, \dots, n$, $f[i]$ es calculado en base a $f[i-1]$ y $f[i-2]$, esto sugiere que el arreglo sea llenado en un recorrido de izquierda a derecha del mismo. El Algoritmo 24 muestra la implementación usando programación dinámica bottom-up. La respuesta buscada es el valor $f[n]$. Nuevamente, hemos transformado una solución de tiempo exponencial en una de tiempo lineal mediante programación dinámica. El espacio usado es $\Theta(n)$ celdas de memoria.

La Figura 9.3 muestra un ejemplo de ejecución para las monedas 1, 5, 10, 1, 5, 10, 50, 10, 1, 1. La figura muestra el arreglo m con esas denominaciones y el arreglo f usado por el algoritmo de programación dinámica.

Sin embargo, note que el algoritmo definido sólo permite calcular la cantidad óptima de dinero que se puede recoger, pero no nos da la estrategia a seguir para lograrla. Para este problema en particular, de

Algoritmo 24: $F(m[1..n])$

```

1 if  $n = 0$  then
2   return 0
3 else
4    $f[0] \leftarrow 0$ 
5    $f[1] \leftarrow m[1]$ 
6   for  $i \leftarrow 2, \dots, n$  do
7      $f[i] \leftarrow \max \begin{cases} f[i-2] + m[i], \\ f[i-1]. \end{cases}$ 
8   end
9   return  $f[n]$ 
10 end

```

poco sirve saber cuánto dinero se puede recoger si no sabemos cómo lograrlo. A continuación discutimos cómo almacenar de forma eficiente el árbol de recursión que nos permita saber qué monedas recoger. El ejemplo de la Figura 9.3 también muestra una flecha indicando a partir de qué otra entrada del arreglo se obtuvo el valor óptimo para cada entrada¹. En otras palabras, para $i = 2, \dots, n$, si $f[i-1] < f[i-2] + m[i]$, existe una flecha desde $f[i]$ hacia $f[i-2]$; en otro caso, la flecha va desde $f[i]$ hacia $f[i-1]$. Estas flechas permiten conocer qué monedas recoger para obtener la cantidad de dinero indicada por $f[i]$: si la flecha va desde $f[i]$ hacia $f[i-1]$, significa que la moneda $m[i]$ no es recogida; si, en cambio, la flecha va desde $f[i]$ hacia $f[i-2]$, la moneda $m[i]$ debe ser recogida. En particular, para obtener la cantidad óptima de dinero $f[10] = 67$ se deben recoger las monedas 10 (\$1), 7 (\$50), 5 (\$5), 3 (\$10), y 1 (\$1). Las entradas del arreglo correspondientes a esas monedas han sido sombreadas.

En resumen, las flechas permiten almacenar la historia de cómputo para cada elemento del arreglo f . De alguna manera, almacena (usando una cantidad lineal de espacio) el árbol de recursión para calcular $F(10)$. La Figura 9.3 también muestra parte del árbol de recursión para $F(10)$ (el árbol completo tiene $\Theta(\varphi^{10})$ nodos). En rojo se muestra la rama del árbol que maximiza la cantidad de dinero a recoger. Cada vez que el cómputo continúa por el “hijo derecho” de un nodo, se debe recoger la moneda correspondiente. Esto es, la rama del árbol del ejemplo nos indica que debemos tomar las monedas con denominación $m[10]$, $m[7]$, $m[5]$, $m[3]$, y $m[1]$, que coinciden con las entradas sombreadas del arreglo.

Este procedimiento de indicar en base a qué otra entrada (o entradas) se calcula el valor de cada entrada de la tabla de programación dinámica, permite obtener la estrategia a seguir para optimizar. Usaremos esta misma técnica para otros problemas en este capítulo.

¹ Para poder implementarlas es necesario un arreglo adicional.

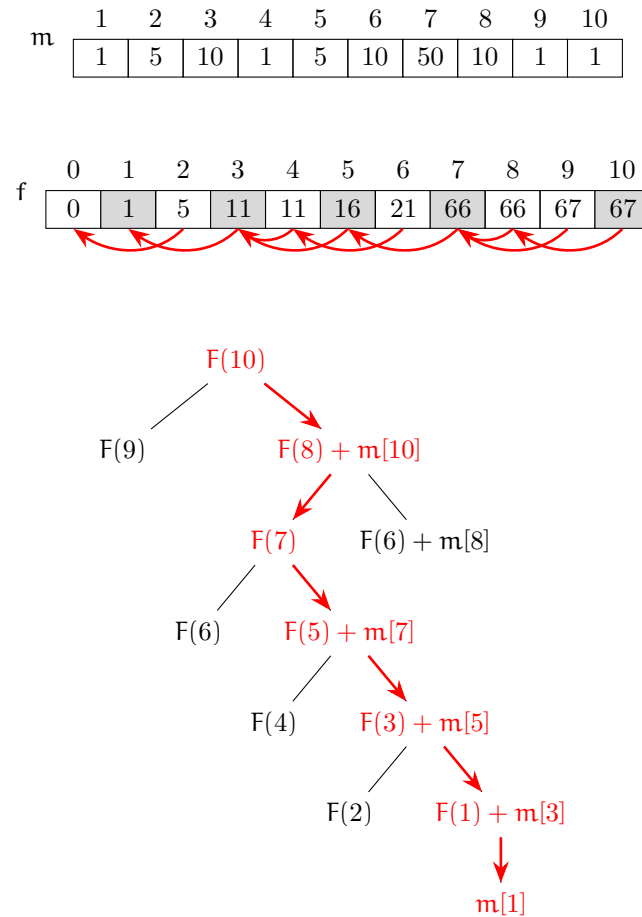


Figura 9.3: Ejemplo para el problema de la fila de monedas.

9.5. Viajes en Canoa por el Río

Suponga que está viajando río abajo en canoa, desde una estación origen e_1 hasta una estación destino e_n . Suponga, de hecho, que existe un total de n estaciones e_1, \dots, e_n a lo largo del río, en cada una de las cuales se puede arrendar una canoa. Antes de comenzar el viaje, para cada $1 \leq i \leq j \leq n$, se informa la tarifa $f_{i,j}$ de arrendar una canoa desde la estación e_i hasta la estación e_j . Esas tarifas son arbitrarias, ya que cada estación está manejada por distintos concesionarios. Por ejemplo, puede pasar que $f_{1,3} = 10$ y $f_{1,4} = 5$. El viaje comienza en la estación e_1 y se debe llegar a la estación e_n usando canoas arrendadas. Sólo se puede viajar río abajo. El objetivo es minimizar el costo total de arriendo.

Como siempre, comencemos resolviendo este problema usando fuerza bruta recursiva. Para $1 \leq i \leq n$, sea $m(i)$ el costo total óptimo para viajar desde la estación e_i hasta la estación e_n , arrendando canoas. Para el caso base, note que $m(n) = 0$, ya que no necesitamos arrendar ninguna canoa. Para el caso recursivo, debemos considerar todas las maneras de arrendar una canoa desde la estación e_i a alguna estación e_j , para $i < j \leq n$, y luego considerar el costo óptimo $m(j)$ para viajar desde e_j a e_n . En resumen, hay que considerar todas las maneras de arrendar una canoa desde e_i a e_j , y luego resolver el problema recursivamente desde e_j a e_n . La opción a elegir es la que minimice la

suma de esos dos términos. Formalmente, tenemos:

$$m(i) = \begin{cases} \min_{i < j \leq n} \{f_{i,j} + m(j)\}, & \text{si } i < n; \\ 0, & \text{si } i = n. \end{cases}$$

Note cómo esta definición toma en cuenta todas las alternativas posibles. Obviamente, $m(1)$ nos dará la respuesta al problema.

A continuación, usamos programación dinámica para implementar este algoritmo. Note que los subproblemas son de la forma $m(i)$, para $1 \leq i \leq n$, por lo que definiremos una tabla de una dimensión $M[1..n]$, de manera que $M[i]$ corresponderá a la solución del subproblema $m(i)$. A continuación, mostramos cómo llenar dicha tabla. Primero, comenzamos con el caso base, por lo que hacemos $M[n] \leftarrow 0$. Luego, hay que determinar para una entrada $M[i]$, $1 \leq i < n$, de qué otras entradas de la tabla depende. Esto nos ayudará a determinar el orden de llenado de la tabla. Note, en la definición anterior, que $m(i)$ depende de otros subproblemas $m(j)$, con $i < j < n$. Llevando eso a la tabla, y cambiando $m(\cdot)$ por $M[\cdot]$, nos indica que para calcular $M[i]$ es necesario haber calculado todos los $M[j]$, para $i < j \leq n$. En otras palabras, el llenado de la matriz se hace de derecha a izquierda. Por lo tanto, para $i = n - 1, \dots, 1$, se hace:

$$M[i] \leftarrow \min_{i < j \leq n} \{f_{i,j} + M[j]\}.$$

Dado que el problema se resuelve mediante $m(1)$, la respuesta se encuentra en $M[1]$.

El tiempo de ejecución de este algoritmo es $\Theta(n^2)$, ya que la minimización ejecuta en total $\sum_{i=1}^n i \approx \frac{n^2}{2}$ operaciones. El espacio adicional usado por la tabla M , por otro lado, es $\Theta(n)$.

9.6. Coeficientes Binomiales

El coeficiente binomial $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, para $0 \leq k \leq n$, cuenta la cantidad de subconjuntos distintos de tamaño k que pueden formarse usando objetos de un conjunto de tamaño n . Este es un concepto fundamental en combinatoria, con diversas aplicaciones. Aunque la fórmula para calcular el coeficiente binomial está claramente definida y el problema parece ser trivial, es importante notar que la fórmula involucra el cálculo de factoriales, lo que produce números enteros muy grandes aún para valores de n relativamente pequeños. En consecuencia, los cálculos intermedios pueden causar overflows aritméticos, incluso cuando el coeficiente final obtenido sea relativamente pequeño.

Una manera más estable de calcular el coeficiente binomial, que no involucra calcular factoriales, usa la propiedad del triángulo de Pascal:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

Notar que en el primer término de la suma, cuando se reemplaza de

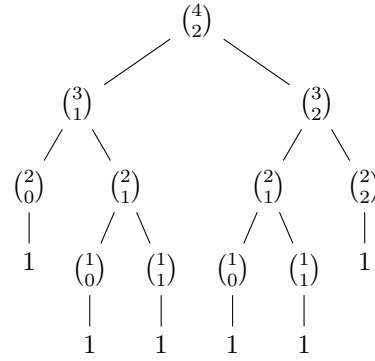


Figura 9.4: Árbol de recursión para el cálculo de $\binom{4}{2}$ mediante fuerza ruta recursiva.

forma recurrente por su definición, nos lleva a:

$$\binom{n-k}{0} = 1,$$

mientras que el segundo término nos lleva a:

$$\binom{k}{k} = 1.$$

La Figura 9.4 muestra el árbol de recursión para el cálculo de $\binom{4}{2}$, usando esta fórmula.

Si reescribimos esta propiedad como ecuación de recurrencia, tenemos:

$$CB(n, k) = \begin{cases} CB(n-1, k-1) + CB(n-1, k), & 0 < k < n; \\ 1, & k = 0; \\ 1, & n = k. \end{cases} \quad (9.3)$$

Note que, a diferencia de las anteriores, ésta es una ecuación de recurrencia de 2 variables. Como lo hemos hecho antes, este será el punto de partida para nuestra solución que usa programación dinámica. La tabla de programación dinámica usada será de 2 dimensiones, una por cada variable de la recurrencia. Para definir dicha tabla, tengamos en cuenta que debemos almacenar todos los posibles subproblemas $CB(i, j)$, para $0 \leq i \leq n$ y $0 \leq j \leq k$. Para esto, definimos la tabla bidimensional $cb[0..n, 0..k]$, de $(n+1) \times (k+1)$ entradas. Sin embargo, note que no todas las entradas de la matriz corresponden a un subproblema válido. Esto es porque en realidad debemos almacenar la respuesta a los subproblemas $CB(i, j)$ tal que $0 \leq j \leq i \leq n$ y $j \leq k$. Esto significa que sólo necesitamos la matriz que está por debajo de la diagonal principal de cb . La Figura 9.5 muestra la matriz cb para el ejemplo de calcular $\binom{4}{2}$. Para llenar la matriz, comenzamos con los casos de borde. Primero, tenemos que $cb[i, i] = 1$, para $i = 0, \dots, k$. Esto corresponde al segundo caso base de la Ecuación (9.3), y son los 1s de la diagonal de la matriz. Luego, el primer caso base de la Ecuación (9.3) especifica que $cb[i, 0] = 1$, para $i = 0, \dots, n$. Eso corresponde a los 1s de la primera columna de la matriz. Luego, dado que hemos dicho que $cb[i, j] = CB(i, j)$, la definición recurrente de la Ecuación (9.3) define $cb[i, j] = cb[i-1, j-1] + cb[i-1, j]$, para $0 < j < k$, y $0 < j < i < n$. En la Figura 9.5, se usan flechas para indicar la dependencia entre las entradas de la tabla. Finalmente, notar que $\binom{n}{k} = CB(n, k) = cb[n, k]$.

cb	0	1	2
0	1	0	0
1	1	1	0
2	1	2	1
3	1	3	3
4	1	4	6

Figura 9.5: Matriz cb para el cómputo de $\binom{4}{2}$.

En nuestro ejemplo particular, tenemos que $\binom{4}{2} = cb[4, 2] = 6$. Dado que la matriz tiene $\Theta(n \times k)$ entradas y cada una puede ser calculada en tiempo $O(1)$, el tiempo de ejecución del algoritmo es $\Theta(n \times k)$. El espacio ocupado es $\Theta(n \times k)$ celdas de memoria.

9.7. Distancia de Edición

El problema de búsqueda en texto estudiado en capítulos anteriores (recuerde el algoritmo KMP), es relevante en muchas aplicaciones. Dado un texto $T[1..n]$ y un patrón de búsqueda $P[1..m]$, usualmente para $m \ll n$, queremos encontrar todas las posiciones de T en donde comienza una ocurrencia de P . Este problema es conocido como búsqueda exacta en texto. La búsqueda aproximada en texto, por otro lado, es una variante que permite encontrar ocurrencias de P en T permitiendo ciertos “errores”. De esta manera se pueden encontrar ocurrencias aún cuando quien busca comete un error al escribir el patrón, lo que suele ser común. Incluso, también permite encontrar ocurrencias de P cuando hay errores en el texto, lo que tiene aplicaciones en búsqueda sobre documentos obtenidos mediante escaneo (en donde el proceso de reconocimiento óptico de caracteres es propenso a errores), búsqueda en textos antiguos (por ejemplo, en inglés antiguo se escribía “Thou shalt not”, mientras que hoy buscaríamos “You should not”, que significa lo mismo), búsqueda en bases de datos biológicas (debido a mutaciones, un mismo gen va cambiando con el tiempo, por lo que un biólogo no sólo puede estar interesado en encontrar una ocurrencia exacta de un gen dentro de un genoma, sino que también encontrar posibles ocurrencias de ese gen que hayan mutado y ya no sean exactamente el mismo). Otra aplicación importante es la corrección ortográfica: cuando se comete un error al tipear una palabra, no sólo se debe marcar ese error, sino que además se deben proponer palabras correctas para reemplazar la errónea. Para determinar qué palabras sugerir, es importante poder buscar las palabras del diccionario que más se le parezcan.

Todas estas aplicaciones comparten el hecho de que es necesario medir el “error” (o diferencia) entre strings. Una de las medidas de error más comunes se conoce como Distancia de Edición, o Distancia de Levenshtein, la que permite medir qué tan “lejanas” o diferentes son dos strings.

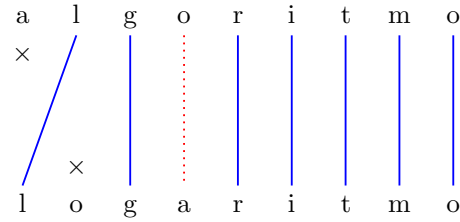


Figura 9.6: Número mínimo de operaciones de edición necesarias para transformar “algoritmo” en “logaritmo”.

Definición 9.7.1 *Dados dos strings $s[1..n]$ y $t[1..m]$ sobre algún alfabeto Σ , la distancia de edición $ed(s, t)$ está definida como el mínimo número de cambios (u operaciones de edición) que se deben hacer a s para transformarla en t .*

En su definición clásica, los tipos de operaciones de edición permitidos son tres:

Sustitución: reemplazar un carácter por otro. Por ejemplo, cambiar ‘h’ por ‘p’ en “shot” \rightarrow “spot”. La operación tiene costo 1.

Inserción: insertar un carácter, como por ejemplo “ago” \rightarrow “agog”. Esta operación tiene costo 1.

Borrado: borrar un carácter, como por ejemplo “hour” \rightarrow “our”. El costo de esta operación es 1.

La sustitución podría no ser una operación *per se*, y ser implementada mediante un borrado seguido de una inserción. Esto tendría costo 2 en nuestro esquema. Sin embargo, la sustitución de caracteres es un error común al escribir, entonces se agrega como operación elemental con costo 1. Existen definiciones alternativas en donde además se agrega, por razones similares, la operación de transposición, en la que invertir el orden de un par de caracteres consecutivos tiene costo 1 (por ejemplo, “ab” y “ba” tendrían distancia 1 en ese esquema, mientras que en el nuestro la distancia es 2).

En nuestra definición, cada operación de edición es penalizada con costo 1. Sin embargo, estos costos pueden cambiarse dependiendo del contexto. Por ejemplo, uno podría penalizar a una operación por sobre las otras en ciertos escenarios. Note que $ed(s, s) = 0$ para todo string s , ya que no hay que hacer ninguna operación para transformar un string en sí mismo. Finalmente, $ed(s, t) = ed(t, s)$, para cualquier par de strings s y t . Esto significa que ed es simétrica, propiedad que puede ser demostrada muy simplemente: (1) las sustituciones son, obviamente, simétricas; (2) una inserción al calcular $ed(s, t)$ es un borrado al calcular $ed(t, s)$; (3) un borrado al calcular $ed(s, t)$ es una inserción al calcular $ed(t, s)$.

Ejemplo 9.7.1 Por ejemplo, $ed(\text{“Thou shalt not”}, \text{“You should not”}) = 5$, mientras que $ed(\text{“book”}, \text{“back”}) = 2$, y $ed(\text{“algoritmo”}, \text{“logaritmo”}) = 3$. La Figura 9.6 muestra el proceso de edición para este último ejemplo, que transforma el string “algoritmo” en el string “logaritmo”. Las 3 operaciones necesarias son: (1) borrar la primera ‘a’, indicado con \times en la figura; (2) insertar ‘o’ luego de la primera ‘l’, indicado con \times en la figura; (3) sustituir la primera ‘o’ por ‘a’, indicado con línea punteada en la figura.

A continuación, definimos la ecuación de recurrencia $\text{ed}(s[1..n], t[1..m])$ que nos permita calcular la distancia de edición entre s y t usando fuerza bruta. Primero, consideremos los casos base de la recursión. Note que si ambos strings son vacíos, tenemos $\text{ed}(\varepsilon, \varepsilon) = 0$. Por otro lado, si sólo uno de los strings es vacío tenemos $\text{ed}(s[1..n], \varepsilon) = n$ (hay que borrar los n símbolos de s) y $\text{ed}(\varepsilon, t[1..m]) = m$ (hay que agregar m símbolos a ε para obtener t).

Los casos recurrentes necesitan algo más de análisis. Luego de definir los casos base de la recursión, podemos asumir que:

- $\text{ed}(s[1..n-1], t[1..m-1])$,
- $\text{ed}(s[1..n-1], t[1..m])$, y
- $\text{ed}(s[1..n], t[1..m-1])$

calculan las respectivas distancias de edición correctamente. Mostramos a continuación cómo en base a estos se puede extender el cálculo para computar $\text{ed}(s[1..n], t[1..m])$. Para transformar $s[1..n]$ en $t[1..m]$ debemos considerar exhaustivamente las 3 operaciones de edición permitidas. Entonces, $\text{ed}(s[1..n], t[1..m])$ es igual al mínimo de las siguientes 3 alternativas:

1. Calcular $\text{ed}(s[1..n-1], t[1..m-1]) = k$, lo que nos indica la cantidad mínima de operaciones necesarias para transformar $s[1..n-1]$ en $t[1..m-1]$ (recuerde que hemos asumido que sabemos cómo calcular $\text{ed}(s[1..n-1], t[1..m-1])$ correctamente). Luego, nos falta considerar los últimos símbolos de ambos strings. Si $s[n] \neq t[m]$, hay que sustituir $s[n]$ por $t[m]$, y entonces la cantidad de operaciones de esta alternativa es $k + 1$. Si, por otro lado, $s[n] = t[m]$, la cantidad de operaciones es k .
2. Calcular $\text{ed}(s[1..n], t[1..m-1]) = k'$, lo que calcula la cantidad mínima de operaciones necesarias para transformar $s[1..n]$ en $t[1..m-1]$ (recuerde que hemos asumido que sabemos cómo calcular $\text{ed}(s[1..n], t[1..m-1])$ correctamente). Luego, falta insertar el símbolo $t[m]$ para obtener el string completo $t[1..m]$. La cantidad de operaciones realizadas por esta alternativa es $k' + 1$.
3. Calcular $\text{ed}(s[1..n-1], t[1..m]) = k''$, lo que calcula la cantidad mínima de operaciones necesarias para transformar $s[1..n-1]$ en $t[1..m]$ (recuerde que hemos asumido que $\text{ed}(s[1..n-1], t[1..m])$ es correcta). Luego, debemos borrar el símbolo $s[n]$, ya que k'' es el costo de obtener el string t completo. La cantidad de operaciones realizadas por esta alternativa es $k'' + 1$.

Como hemos dicho, la mínima cantidad de operaciones realizadas por alguna de esas 3 alternativas es la que define la distancia de edición. Resumiendo, tenemos la ecuación de recurrencia:

$$\text{ed}(s[1..n], t[1..m]) = \begin{cases} \min \begin{cases} \text{ed}(s[1..n-1], t[1..m-1]) + [s[n] \neq t[m]], \\ \text{ed}(s[1..n], t[1..m-1]) + 1, \\ \text{ed}(s[1..n-1], t[1..m]) + 1. \end{cases} \\ n, \\ m. \end{cases} \quad (9.4)$$

El primer caso es para $m > 0$ y $n > 0$. El segundo caso es para $t = \varepsilon$. El tercer caso es para $s = \varepsilon$. No hemos incluido el caso en que ambos strings son vacíos, $s = \varepsilon$ y $t = \varepsilon$, ya que está incluido en el segundo (también en el tercer) caso. Asumimos que $s[n] \neq t[m]$ vale 1 cuando $s[n] \neq t[m]$, y 0 en otro caso. Note cómo los casos recurrentes llevan, de alguna u otra manera, a los casos base.

Ejemplo 9.7.2 Para comprender de mejor manera esta definición, estudiemos el primer nivel de recursión al calcular $\text{ed}(\text{"alg"}, \text{"log"})$. La primera alternativa considera el costo de transformar "al" en "lo", y luego sustituir 'g' por 'g' (que tiene costo 0, porque son iguales). Aquí tenemos que $\text{ed}(\text{"al"}, \text{"lo"}) = 2$, por lo tanto esta opción tiene costo total 2. La segunda alternativa es transformar "alg" en "lo", y luego agregar 'g'. Dado que $\text{ed}(\text{"alg"}, \text{"lo"}) = 2$, el costo de esta alternativa es 3. Finalmente, hay que considerar transformar "al" en "log", y luego borrar 'g' (del primer string). Como $\text{ed}(\text{"al"}, \text{"log"}) = 3$, el costo de esta alternativa es 4. Dado que el mínimo de esas tres alternativas es 2, tenemos que $\text{ed}(\text{"alg"}, \text{"log"}) = 2$. Queda como ejercicio para el lector ejecutar la recursión completa para este ejemplo.

La Ecuación (9.4) puede implementarse de forma directa para obtener una solución por fuerza bruta. Para calcular su tiempo de ejecución, note que la recursión tiene 3 ramas. En cada una de las ramas, al menos uno de los strings reduce su largo en 1, por lo que en todas las ramas al menos uno de los strings se consumirá por completo. De hecho, el primer string en consumirse es el más corto entre s y t . En conclusión, el árbol de recursión es ternario, y tiene altura $\min\{n, m\}$, por lo que tiene $\Theta(3^{\min\{n, m\}})$ nodos. Como cada nodo toma tiempo constante, el tiempo de ejecución es $\Theta(3^{\min\{n, m\}})$. Esta solución es útil sólo para strings relativamente cortos.

Para reducir el tiempo de ejecución, evitaremos repetir subproblemas usando programación dinámica. Note que todos los subproblemas involucrados en el cálculo de $\text{ed}(s[1..n], t[1..m])$ tienen la forma $\text{ed}(s[1..i], t[1..j])$, para $0 \leq i \leq n$ y $0 \leq j \leq m$. Aquí estamos asumiendo $s[1..0] = \varepsilon$ y $t[1..\varepsilon] = \varepsilon$. En consecuencia, tenemos $(n+1) \times (m+1)$ subproblemas distintos. Definimos la tabla de programación dinámica $M[0..n, 0..m]$, de manera que $M[i, j] = \text{ed}(s[1..i], t[1..j])$. Usando la definición dada en la Ecuación (9.4), tenemos que:

- $M[0, 0] = \text{ed}(\varepsilon, \varepsilon) = 0$;
- $M[i, 0] = \text{ed}(s[1..i], \varepsilon) = i$, para $i = 1, \dots, n$.
- $M[0, j] = \text{ed}(\varepsilon, t[1..j]) = j$, para $j = 1, \dots, m$.

Esto completa los casos base. Para los casos recurrentes, tenemos que calcular $M[i, j]$ en base a $\text{ed}(s[1..i-1], t[1..j-1])$, $\text{ed}(s[1..i], t[1..j-1])$, y $\text{ed}(s[1..i-1], t[1..j])$. Respectivamente, esos 3 valores están almacenados en $M[i-1, j-1]$, $M[i, j-1]$, y $M[i-1, j]$, las celdas de la tabla que están en la diagonal, a su izquierda, y arriba de $M[i, j]$, quedando:

$$M[i, j] = \min \begin{cases} M[i-1, j-1] + [s[i] \neq t[j]], \\ M[i, j-1] + 1, \\ M[i-1, j] + 1. \end{cases}$$

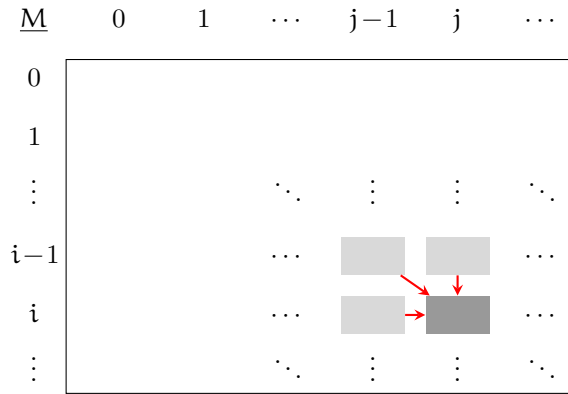


Figura 9.7: Matriz M usada para el cálculo de la distancia de edición. Las flechas indican la dependencia de la entrada $M[i, j]$ con respecto a otras entradas de la tabla.

La Figura 9.7 muestra, usando flechas, cómo la entrada $M[i, j]$ depende de las otras 3 entradas para su cómputo. Es importante conocer esta dependencia entre entradas de la tabla, porque nos ayuda a definir el orden en que deben llenarse. Note que en este caso sirve tanto un llenado por columnas (de izquierda a derecha), por filas (de arriba hacia abajo), o incluso por diagonales. Finalmente, note que $\text{ed}(s[1..n], t[1..m]) = M[n, m]$, por lo que la esquina inferior derecha de la tabla contiene el resultado final luego de ejecutar el algoritmo.

Ejemplo 9.7.3 La Figura 9.8 muestra la tabla de programación dinámica M , resultante de ejecutar $\text{ed}(\text{"algoritmo"}, \text{"logaritmo"})$. Por simplicidad, se han colocado los strings en los bordes de la tabla, en lugar de los índices de la tabla, como es usual. Por cada entrada de la tabla, se muestra en base a qué otras entradas se calculó su valor (pueden ser varias en caso de empate). Esa información permite conocer qué operaciones se pueden hacer para transformar un string en otro. Una flecha hacia abajo implica un borrado en s , una flecha hacia la derecha indica inserción en s , y una flecha en diagonal indica una sustitución (o no, en caso de que los símbolos sean iguales). Las entradas sombreadas indican la secuencia de operaciones necesarias para transformar "algoritmo" en "logaritmo". Comenzando desde la entrada $M[0, 0]$, primero tenemos que borrar la 'a' de s (\downarrow , recuerde la Figura 9.6), luego calzar la 'l' en ambos strings, insertar la 'o' en s (\rightarrow), calzar la 'g' en ambos strings, sustituir la 'o' por 'a' en s , y finalmente calzar cada uno de los símbolos restantes en ambos strings.

Para finalizar, el tiempo de ejecución del algoritmo es $\Theta(n \times m)$. El espacio necesario para lograr dicho tiempo de ejecución es $\Theta(n \times m)$. El mismo puede reducirse a $\Theta(\min\{n, m\})$ manteniendo únicamente 2 filas (o columnas) a la vez. Note que eso es suficiente para calcular cada entrada de la matriz. Sin embargo, de esta manera no podremos almacenar las operaciones de edición necesarias para transformar un string en otro.

Figura 9.8: Matriz de programación dinámica usada para calcular $\text{ed}(\text{"algoritmo"}, \text{"logaritmo"})$.

M	ε	l	o	g	a	r	i	t	m	o
ε	0	1	2	3	4	5	6	7	8	9
a	1	1	2	3	3	4	5	6	7	8
l	2	1	2	3	4	4	5	6	7	8
g	3	2	2	2	3	4	5	6	7	8
o	4	3	2	3	3	4	5	6	7	7
r	5	4	3	3	4	3	4	5	6	7
i	6	5	4	4	4	4	3	4	5	6
t	7	6	5	5	5	5	4	3	4	5
m	8	7	6	6	6	6	5	4	3	4
o	9	8	7	7	7	7	6	5	4	3

9.8. Multiplicación de Secuencias de Matrices

En muchas aplicaciones, como por ejemplo el procesamiento de imágenes o animación 3D, es necesario multiplicar secuencias de matrices. Formalmente, sea M_1, M_2, \dots, M_n una secuencia de n matrices de números reales, de dimensiones $d_0 \times d_1, d_1 \times d_2, \dots, d_{n-1} \times d_n$, respectivamente. El objetivo es calcular $M_1 \times M_2 \times \dots \times M_n$, minimizando la cantidad de multiplicaciones de escalares ejecutadas. Recuerde que si multiplica dos matrices M_1 y M_2 de dimensiones $d_0 \times d_1$ y $d_1 \times d_2$, respectivamente, la cantidad total de multiplicaciones de escalares necesarias para obtener $M_1 \times M_2$ es $d_0 \cdot d_1 \cdot d_2$. Recuerde, además, que la multiplicación de matrices no es conmutativa, por lo que el orden original de la secuencia debe ser respetado. Afortunadamente, la multiplicación de matrices sí es asociativa. El objetivo será encontrar la parentización de la secuencia que produzca la cantidad óptima (mínima) de multiplicaciones de escalares.

Ejemplo 9.8.1 Considere la secuencia de matrices $M_1 \times M_2 \times M_3 \times M_4$, de dimensiones:

- M_1 : 50×20 ,
- M_2 : 20×1 ,
- M_3 : 1×10 , y
- M_4 : 10×100 .

Algunas alternativas de parentización son:

- $M_1 \times ((M_2 \times M_3) \times M_4)$: 120.200 multiplicaciones.
- $(M_1 \times (M_2 \times M_3)) \times M_4$: 60.200 multiplicaciones.
- $(M_1 \times M_2) \times (M_3 \times M_4)$: 7.000 multiplicaciones.

Diseñemos a continuación un esquema de fuerza bruta para resolver

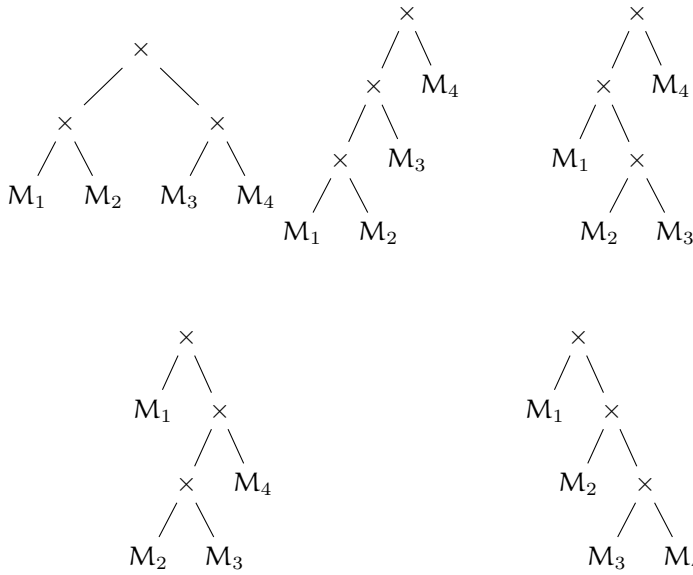


Figura 9.9: Árboles de expresión para la secuencia de matrices $M_1 \times M_2 \times M_3 \times M_4$, correspondientes a (arriba, de izquierda a derecha): $(M_1 \times M_2) \times (M_3 \times M_4)$, $((M_1 \times M_2) \times M_3) \times M_4$, y $(M_1 \times (M_2 \times M_3)) \times M_4$; (abajo, de izquierda a derecha): $M_1 \times ((M_2 \times M_3) \times M_4)$ y $M_1 \times (M_2 \times (M_3 \times M_4))$.

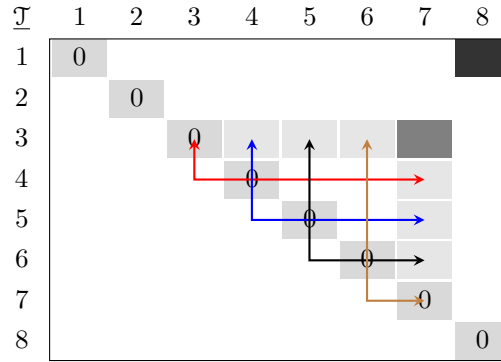
este problema. La idea es probar exhaustivamente todas las parentizaciones posibles, buscando la que produzca la menor cantidad total de multiplicaciones de escalares. Desafortunadamente, este esquema no es práctico, de acuerdo al siguiente análisis. Pensemos en el árbol de expresión de una parentización cualquiera de la secuencia, que tiene $n - 1$ nodos internos correspondientes a los $n - 1$ operadores \times de la secuencia, y cuyos nodos externos son las matrices M_1, \dots, M_n , dispuestas de izquierda a derecha en los nodos externos. La Figura 9.9 muestra todos los posibles árboles de expresión correspondientes a las parentizaciones de la secuencia $M_1 \times M_2 \times M_3 \times M_4$. En consecuencia, la cantidad total de parentizaciones coincide con el Número de Catalan para un árbol de $n - 1$ nodos internos $C_{n-1} = \frac{1}{n} \binom{2n-2}{n-1}$, lo cual es $\approx 4^{n-1}$ para n grande. Por lo tanto, la fuerza bruta no es viable. Sin embargo, y tal como hemos hecho en secciones anteriores, definiremos un esquema de fuerza bruta recursiva al que luego implementaremos usando programación dinámica.

Sea $M(\ell, \ell')$, para $\ell \leq \ell'$, el costo óptimo de multiplicar la subsecuencia de matrices $M_\ell \times \dots \times M_{\ell'}$. Obviamente estaremos interesados en obtener $M(1, n)$. Para definir los casos base de la recursión, note que $M(i, i) = 0$, para $i = 1, \dots, n$. Esto es porque una secuencia de 1 matriz (y por ende $0 \times$) necesita 0 multiplicaciones de escalares. Luego, podemos asumir que sabemos cómo multiplicar de manera óptima secuencias de hasta $n - 1$ matrices. Mostraremos cómo, en base a eso, obtener la forma óptima de multiplicar una secuencia de n matrices. La idea es considerar las $n - 1$ maneras de particionar la secuencia en dos subsecuencias, calcular el costo óptimo de cada una de esas particiones, y quedarnos con la óptima. Dado que cada una de esas particiones tiene tamaño a lo más $n - 1$, por hipótesis de inducción sabemos cómo multiplicarlas de forma óptima.

Ejemplo 9.8.2 Por ejemplo, para $M_1 \times M_2 \times M_3$, el valor $M(1, 3)$ se define considerando las alternativas

- $(M_1 \times M_2) \times M_3$: costo $M(1, 2) + M(3, 3) + d_0 \cdot d_2 \cdot d_3$;

Figura 9.10: Matriz de programación dinámica para una secuencia $M_1 \times \dots \times M_8$. Las flechas indican las entradas que deben ser combinadas para calcular la entrada $\mathcal{T}[3, 7]$. El resultado buscado está almacenado en la entrada $\mathcal{T}[1, 8]$, la cual ha sido sombreada.



■ $M_1 \times (M_2 \times M_3)$: costo $M(1, 1) + M(2, 3) + d_0 \cdot d_1 \cdot d_3$.

La alternativa que produzca el menor costo es la que define el valor de $M(1, 3)$.

En general, para $i < j$, tenemos que $M(i, j)$ se calcula particionando la secuencia $M_i \times \dots \times M_j$ en cada posición $k = i, \dots, j-1$, y determinando $M(i, k)$ (note que $M_i \times \dots \times M_k$ genera una matriz de dimensión $d_{i-1} \times d_k$), y $M(k+1, j)$ (note que $M_{k+1} \times \dots \times M_j$ genera una matriz de dimensión $d_k \times d_j$):

$$\underbrace{M_i \times \dots \times M_k}_{M(i, k)} \times \underbrace{M_{k+1} \times \dots \times M_j}_{M(k+1, j)}.$$

Luego, el costo de multiplicar esas dos matrices es $d_{i-1} \cdot d_k \cdot d_j$, lo que debe ser sumado a $M(i, k)$ y a $M(k+1, j)$ para obtener el costo total de la partición. De todas esas posibles particiones, nos quedamos con la que minimice la cantidad de multiplicaciones de escalares.

Resumiendo, tenemos que nuestra ecuación de recurrencia es:

$$M(i, j) = \begin{cases} 0, & \text{si } i = j; \\ \min_{i \leq k < j} \{M(i, k) + M(k+1, j) + d_{i-1} \cdot d_k \cdot d_j\}, & \text{si } i < j. \end{cases} \quad (9.5)$$

Como vimos anteriormente, si implementamos directamente esta ecuación obtendremos tiempo de ejecución exponencial en n .

Para resolver el problema usando programación dinámica, definimos la matriz $\mathcal{T}[1..n, 1..n]$ de dos dimensiones, en la que almacenaremos las soluciones a todos los subproblemas involucrados en la resolución de nuestro problema. En particular, definimos $\mathcal{T}[i, j] = M(i, j)$. Note que sólo son válidos los subproblemas $M(i, j)$ para $i \leq j$, por lo que sólo necesitamos usar la matriz triangular superior. El algoritmo comienza llenando las entradas de la tabla correspondientes a los casos base, es decir, $\mathcal{T}[i, i] = 0$, para $i = 1, \dots, n$. Luego, hay que continuar llenando el resto de las entradas de \mathcal{T} . Para determinar el orden de llenado, estudiemos las dependencias entre entradas de la tabla, de manera de saber qué entradas deben haber sido ya calculadas antes de calcular otra. La Figura 9.10 muestra un ejemplo de matriz de programación dinámica para una secuencia de 8 matrices $M_1 \times \dots \times M_8$. La figura muestra las entradas que deben ser combinadas para obtener $\mathcal{T}[3, 7] =$

$M(3, 7)$. Recuerde que debemos considerar todas las particiones de $M_3 \times \cdots \times M_7$:

- $M(3, 3)$ y $M(4, 7)$, correspondientes a las entradas $\mathcal{T}[3, 3]$ y $\mathcal{T}[4, 7]$ (flecha roja en la figura).
- $M(3, 4)$ y $M(5, 7)$, correspondientes a las entradas $\mathcal{T}[3, 4]$ y $\mathcal{T}[5, 7]$ (flecha azul en la figura).
- $M(3, 5)$ y $M(6, 7)$, correspondientes a las entradas $\mathcal{T}[3, 5]$ y $\mathcal{T}[6, 7]$ (flecha negra en la figura).
- $M(3, 6)$ y $M(7, 7)$, correspondientes a las entradas $\mathcal{T}[3, 6]$ y $\mathcal{T}[7, 7]$ (flecha marrón en la figura).

Básicamente, para calcular $\mathcal{T}[i, j]$ necesitamos haber calculado toda la fila i y columna j de la matriz. Por lo tanto, un llenado por diagonales es el necesario en este caso. Finalmente, el resultado que estamos buscando es $M(1, n)$, la cantidad óptima de multiplicaciones de escalares necesarias para multiplicar la secuencia $M_1 \times \cdots \times M_8$. Eso corresponde a la entrada superior derecha $\mathcal{T}[1, n]$ de la matriz. Respecto al tiempo de ejecución, note que para calcular cada entrada, hay que recorrer toda su fila y columna, lo cual toma tiempo $O(n)$. Dado que hay que llenar $\Theta(n^2)$ entradas, el tiempo total es $\Theta(n^3)$. Aunque este tiempo pueda parecer algo elevado, recuerde que n es la cantidad de matrices a multiplicar. Probablemente, para la mayoría de las aplicaciones n no es un valor elevado (por ejemplo, multiplicar cientos o miles de matrices). Las matrices pueden ser de grandes dimensiones, por lo que el costo cúbico necesario para determinar la parentización óptima se justifica ampliamente en muchos casos. Para finalizar, el espacio utilizado por la solución de programación dinámica es $\Theta(n^2)$.

Ejercicios

1. ¿Cuáles son las características comunes entre las técnicas de diseño *Divide y Vencerás* y *Programación Dinámica*? ¿Cuál es la principal diferencia entre ellas?
2. Suponga una matriz A de dimensión $n \times m$, tal que en algunas de las entradas $A[i, j]$ se han dejado depositadas monedas. Hay a lo más una moneda por entrada. Un robot, que inicialmente está posicionado en la entrada superior izquierda $A[1, 1]$ de la matriz, necesita recolectar tantas monedas como sea posible, y llevarlas hasta la entrada $A[n, m]$. En cada paso, el robot sólo puede moverse hacia la derecha o hacia abajo, de a una entrada por vez. Cuando el robot visita una entrada que contiene una moneda, la recoge. Diseñe un algoritmo de programación dinámica para encontrar el máximo número de monedas que el robot puede recolectar y dar la ruta necesaria para lograrlo.
3. Modifique el algoritmo del punto anterior si ahora pueden haber celdas de la matriz a las que el robot no puede acceder. Dada la siguiente matriz, en donde las entradas con \times son inaccesibles y las entradas con \bigcirc contienen una moneda, ¿Cuántas rutas óptimas existen para esta matriz?

	×		○		
○			×	○	
	○		×	○	
			○		○
×	×	×		○	

4. Suponga un tablero que tiene n casillas ordenadas en secuencia, y enumeradas desde 1 a n . El objetivo del juego es moverse desde la casilla 1 a la n , dando la menor cantidad posible de saltos para lograrlo. La dificultad está en que cada casilla i tiene asociado un valor $s_i \geq 0$ que indica el largo del salto máximo que se puede dar desde esa casilla. Es decir, si se está en la casilla i , sólo se puede avanzar a alguna de las siguientes casillas (si es que existe): $i + 1, i + 2, \dots, i + s_i$. Escriba un algoritmo de programación dinámica que permita obtener la cantidad mínima de saltos que hay que dar para recorrer un tablero dado. Describa cada una de las partes de su algoritmo, e indique su tiempo de ejecución.

Ejemplo: Para el caso $s[1..11] = \{1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9\}$, la respuesta es 3 saltos.

5. Una mina de oro se ha organizado como un rectángulo de $n \times m$ secciones. Cada sección tiene una cantidad determinada de oro, indicada por un entero ≥ 0 . Un minero puede recorrer la mina comenzando desde la columna 1, desde cualquiera de las filas. Si actualmente el minero se encuentra en la sección (i, j) de la mina, tiene permitido moverse a las secciones $(i - 1, j + 1)$, $(i, j + 1)$, o $(i + 1, j + 1)$, siempre y cuando $i - 1 \geq 1$, $i + 1 \leq n$, $j + 1 \leq m$. Escriba un algoritmo de programación dinámica que permita obtener la máxima cantidad de oro que un minero pudiese recolectar en la mina. Escriba todo los detalles necesarios para el correcto funcionamiento de su algoritmo.
6. Dada una secuencia de símbolos (los cuales pueden ser, indistintamente, caracteres o números enteros), una subsecuencia consiste de un subconjunto de elementos de la secuencia, respetando el orden original de esos elementos. Por ejemplo, dada la secuencia $[8, 4, 12, 2, 10, 6, 1, 9, 5, 13, 2, 3, 11]$, una posible subsecuencia sería $[4, 1, 9, 13, 3, 11]$.

Sean $s[1..n]$ y $t[1..m]$ dos secuencias de símbolos. Diseñar un algoritmo de programación dinámica para calcular la *subsecuencia común más larga* entre s y t . Por ejemplo, si $s = [8, 4, 12, 2, 10, 6, 1, 9, 5, 13, 2, 3, 11]$ y $t = [7, 1, 6, 10, 5, 8, 13, 3, 15]$, entonces la subsecuencia común más larga entre ellas es $[10, 5, 13, 3]$, como se indica a continuación:

$$s = [8, 4, 12, 2, \underline{10}, 6, 1, 9, \underline{5}, \underline{13}, \underline{2}, \underline{3}, 11]$$

y

$$t = [7, 1, 6, \underline{10}, \underline{5}, 8, \underline{13}, \underline{3}, 15].$$

Note que $[6, 5, 13, 3]$ y $[1, 5, 13, 3]$ son también subsecuencias comunes más largas alternativas entre s y t .

Defina formalmente el algoritmo de programación dinámica indicando **todos** los detalles necesarios para su entendimiento. Explique también el razonamiento que llevó a definir su algoritmo.

7. Defina un algoritmo de programación dinámica para encontrar la subsecuencia monótona creciente más larga de un string $s[1..n]$. Por ejemplo, si $s = [7, 1, 6, 10, 5, 8, 13, 3, 15]$, una posible subsecuencia monótona más larga es $[1, 5, 8, 13, 15]$. Su algoritmo debería tener un tiempo de ejecución $\Theta(n^2)$.
8. Se quiere ordenar un arreglo de n números enteros positivos usando el mínimo número de movimientos. Cada movimiento consiste en transportar un elemento del arreglo desde su posición original a su posición final. Escriba un algoritmo de programación dinámica que cuente la **cantidad mínima** de elementos que deben ser movidos para producir un arreglo ordenado.

Ejemplos:

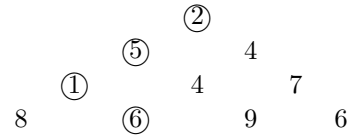
- Para el arreglo $[1, 8, 9, 2]$, la respuesta es 1 elemento (correspondiente al elemento 2).
- Para el arreglo $[9, 8, 2, 1]$, la respuesta es 3 elementos (correspondiente a los elementos 8, 2, y 1; o alternativamente los elementos 9, 2, y 1; o alternativamente 9, 8, y 1; o, finalmente, 9, 8, y 2).
- Para el arreglo $[50, 3, 10, 7, 40, 80]$, la respuesta es 2 elementos (correspondiente a los elementos 50 y 10; o alternativamente 50 y 7).
- Para el arreglo $[10, 22, 9, 33, 21, 50, 41, 60]$, la respuesta es 3 elementos (correspondiente a los elementos 9, 21, 41; o alternativamente 9, 21, y 50).

Hint: Relacione este problema con algún problema conocido.

9. Suponga que se quiere resolver el problema de la *supersecuencia común más corta*: dados dos strings s_1 y s_2 , encontrar el string más corto que tiene a s_1 y s_2 como subsecuencias.

Ejemplos:

- para $s_1 = \text{"geek"}$ y $s_2 = \text{"eke"}$, el resultado es *"geeke"*.
 - para $s_1 = \text{"AGGTAB"}$ y $s_2 = \text{"GXTXAYB"}$, el resultado es *"AGGTXAYB"*.
- a) Explique cómo resolvería este problema usando un algoritmo de programación dinámica. Puede relacionarlo con algún problema conocido visto en clases o guías de ejercicios. No es necesaria una explicación formal, sino más bien la idea principal y los argumentos necesarios. Puede usar figuras para ilustrar.
 - b) Escriba un algoritmo de programación dinámica que permita calcular la longitud de la supersecuencia común más corta entre dos strings s_1 y s_2 . Puede relacionarlo con algún problema conocido visto en clases o guías de ejercicios, pero tiene que escribir el algoritmo.
10. Dada una secuencia de números enteros a_1, a_2, \dots, a_n , diseñar un algoritmo de programación dinámica que permita encontrar la subsecuencia contigua de suma máxima (una subsecuencia de longitud cero tiene suma cero). Por ejemplo, si la secuencia de entrada es 5, 15, -30, 10, -5, 40, 10, entonces la respuesta es la subsecuencia 10, -5, 40, 10, cuya suma es 55.
 11. Dado un conjunto de números enteros positivos, colocados en un triángulo equilátero con n números en su base, tal como se muestra en el siguiente ejemplo para $n = 4$:



El problema consiste en encontrar la suma más pequeña en un descenso desde el vértice superior del triángulo hasta su base, a través de números adyacentes (para el ejemplo anterior, dichos números que conforman la suma mínima han sido marcados con círculos). Diseñe un algoritmo de programación dinámica para este problema, e indique su eficiencia tanto en tiempo como en espacio.

12. El Profesor Oblivious es un poco descuidado, y se le han perdido los paréntesis de una expresión matemática fundamental para su investigación. Lo único que le ha quedado es una expresión matemática sin paréntesis, de la forma:

$$x_1 \diamond_1 x_2 \diamond_2 \cdots \diamond_{n-1} x_n,$$

donde x_1, x_2, \dots, x_n son números enteros positivos, y $\diamond_1, \diamond_2, \dots, \diamond_{n-1}$ son operadores, cada uno de los cuales puede ser $+$ (suma) o \times (multiplicación). Para poder continuar con su investigación, el Profesor se conforma con conocer un rango de valores que su expresión pudiese alcanzar. En particular, le interesa saber los valores máximo y mínimo que se pueden alcanzar, obtenidos mediante distintas formas de colocar paréntesis a la expresión.

Ejemplo: Para la expresión $2 + 3 \times 4 + 5 \times 6$, el valor máximo es 270 y corresponde a la parentización $(2 + 3) \times (4 + 5) \times 6$, mientras que el valor mínimo es 44 y corresponde a $2 + (3 \times 4) + (5 \times 6)$.

- Defina las ecuaciones de recurrencia para obtener el mínimo y máximo valor para una expresión dada.
- Defina un algoritmo de programación dinámica que permita resolver de forma más eficiente el problema.

10.1. Introducción

En el diseño de algoritmos, el *backtracking* (“vuelta atrás”) proporciona una manera sistemática y ordenada de generar el conjunto —o un subconjunto— $S_{\mathcal{P}}$ de las soluciones de un problema \mathcal{P} , en casos en que dichas soluciones puedan construirse incrementalmente. En general, esta técnica está asociada a algoritmos de búsqueda exhaustiva. Con la técnica de backtracking intentaremos generar el conjunto $S_{\mathcal{P}}$ eficientemente, podando el espacio de soluciones siempre que sea posible. Eso significa que un algoritmo de backtracking intenta descartar de antemano soluciones candidatas que no correspondan a la instancia que estamos resolviendo, reduciendo el tiempo de ejecución. Estudiaremos en este capítulo la versión más típica de la técnica: el backtracking recursivo.

Sea $\mathcal{X}_N = (x_1, \dots, x_N)$ la N -tupla que representa una solución del problema \mathcal{P} . Cada uno de los componentes x_i son los posibles valores de los que está compuesta la solución \mathcal{X} . Por ejemplo, para el problema de la mochila cada x_i sería un elemento a agregar en la mochila. La idea principal de la técnica es construir las soluciones incrementalmente, un elemento x_i a la vez, de forma recursiva. Conceptualmente, se realiza un recorrido en profundidad (DFS) del árbol de recursión., al que llamaremos *árbol de estado del problema*. Las soluciones se construyen agregando elementos a medida que descendemos en el árbol, de la siguiente manera:

- La raíz del árbol de recursión corresponde a una solución vacía, es decir

$$\mathcal{X}_0 = ().$$

- Un nodo de nivel $k \geq 0$ del árbol representa una solución parcial

$$\mathcal{X}_k = (x_1, \dots, x_k),$$

es decir, construida correctamente hasta una etapa k . Para el problema de la mochila, \mathcal{X}_k podría indicar los k elementos que estamos considerando cargar en la mochila. Los nodos hijo corresponden a todas las posibles prolongaciones a esa solución parcial,

$$\mathcal{X}_{k+1} = (x_1, \dots, x_k, x_{k+1}),$$

completa ahora hasta una etapa $k + 1$. Nuevamente, en el ejemplo de la mochila, debemos intentar agregar todos los posibles elementos x_{k+1} a la mochila actual (x_1, \dots, x_k) , uno a la vez, y continuar el llenado desde allí.

- Los nodos externos del árbol corresponden a las soluciones del problema, o a nodos de fracaso: es decir, nodos tal que si son expandidos, no conducen a ninguna solución del problema.

Para examinar el conjunto de posibles soluciones al problema, es suficiente generar (y por lo tanto recorrer) el árbol de recursión, construyendo soluciones parciales a medida que se avanza en el recorrido.

Asumamos que estamos en un nodo de nivel k del árbol de recursión, correspondiente a la solución parcial $\mathcal{X}_k = (x_1, \dots, x_k)$. En este punto, hay que considerar 3 posibles casos:

1. **Que \mathcal{X}_k sea una solución completa válida:** Esto indica que hemos llegado a una hoja del árbol, y en general no hay posibilidad de seguir expandiendo esta solución. Si el algoritmo buscaba una posible solución al problema, entonces el algoritmo finaliza en este punto. Si, en cambio, se buscan todas las soluciones que satisfagan al problema, o se busca la solución que optimice una función dada, el algoritmo seguirá explorando el árbol en busca de soluciones alternativas.
2. **Que \mathcal{X}_k sea una solución parcial válida:** significa que la solución parcial \mathcal{X}_k aún satisface las restricciones del problema (por ejemplo, que los k elementos que conforman la mochila aún no exceden su peso). En ese caso, se debe extender esta solución recursivamente.
3. **Que \mathcal{X}_k sea una solución parcial inválida:** significa que la solución parcial \mathcal{X}_k no satisface las restricciones del problema (por ejemplo, que los k elementos que conforman la mochila excedieron su peso). En ese caso, se dice que hemos alcanzado un *nodo de fracaso*, y el algoritmo vuelve atrás en su recorrido (de ahí el nombre de la técnica), eliminando la componente x_k de la solución parcial, retornando al nodo padre en el árbol de recursión, y considerando una nueva alternativa (distinta) para x_k . Para la mochila, significa sacar el elemento x_k (que fue el que excedió la capacidad de la mochila) y considerar otro. Si todos los hijos de un nodo x del árbol de recursión son nodos de fracaso, entonces el nodo x también es un nodo de fracaso.

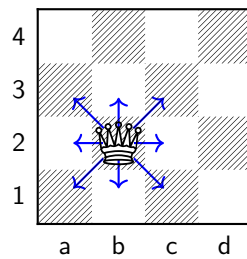
Note cómo en el tercer caso, al momento en que detectamos que \mathcal{X}_k no puede llevarnos a una solución válida, detenemos el proceso de generar soluciones a partir de ésta. Esto significa que el árbol de recursión se poda en los nodos fracaso, reduciendo el tiempo de ejecución. Por ejemplo, si al agregar un elemento a la mochila excedemos su capacidad, es obvio que no deberíamos intentar agregar más elementos a partir de allí. La filosofía de estos algoritmos no sigue reglas fijas en la búsqueda de las soluciones. Se podría hablar de un proceso de prueba y error en el que se trabaja por etapas, construyendo las soluciones gradualmente. Cuando se detecta que una decisión del algoritmo no puede llevarnos a alguna solución válida, se deshace esa decisión y se considera otra alternativa.

Para muchos problemas, esta prueba en cada etapa puede llevar a un árbol de tamaño exponencial. Para lograr un algoritmo (relativamente) eficiente, hay que considerar un árbol de recursión con el menor número posible de nodos. Sin embargo, después de podar el árbol en los nodos

fracaso, debemos asegurar que aún contiene todas las soluciones al problema.

10.2. El Problema de las N Reinas

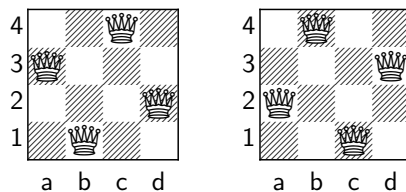
El problema consiste en colocar N reinas en un tablero de ajedrez de $N \times N$ casillas, de manera que ninguna de las reinas pueda atacar a otra, o determinar que no es posible hacerlo. Este puzzle en su versión original de 8×8 fue introducido por Max Bezzel, ajedrecista alemán, en 1848. La versión general, en un tablero de $N \times N$, fue introducido posteriormente por François-Joseph Eustache Lionnet en 1969. Matemáticos de la talla de Carl Friedrich Gauss trabajaron para encontrar soluciones, hasta que Edsger Dijkstra mostró en 1972 la solución de backtracking recursivo que estudiaremos más adelante en esta sección. Recuerde que la reina del ajedrez puede atacar en cualquier dirección a partir de la casilla en la que está ubicada, ya sea verticalmente, horizontalmente, o en diagonal:



Esto significa que en una solución válida no podemos tener dos reinas en la misma fila, columna, ni diagonal. Para $N = 1$, tenemos obviamente una única solución:



Para $N = 2$ y $N = 3$, sin embargo, no hay soluciones posibles. Para $N = 4$, tenemos las dos siguientes soluciones:



Note que, debido a la forma de ataque de una reina, no podemos colocar más de una reina por fila del tablero. Por lo tanto, vamos a denotar las posibles soluciones usando una tupla $\mathcal{X}_N = (x_1, \dots, x_N)$, en la que cada posición representa una fila del tablero. El valor x_i indica la columna

en que se posiciona la reina en la fila i . Por ejemplo, para $N = 4$, las soluciones mencionadas anteriormente tienen las representaciones de tupla:

$$(b, d, a, c) \text{ y } (c, a, d, b).$$

A partir de esto, podemos construir la siguiente solución de búsqueda exhaustiva para resolver este problema: generar las N^N posibles tuplas, chequeando cada una para determinar si las reinas en esa configuración están en posición de ataque o no. Para $N = 4$, un posible orden de generación y chequeo podría ser:

$(a, a, a, a),$
 $(a, a, a, b),$
 $(a, a, a, c),$
 $(a, a, a, d),$
 $(a, a, b, a),$
 $(a, a, b, b),$
 $(a, a, b, c),$
 \vdots
 $(d, d, d, d).$

Son $4^4 = 256$ distintas configuraciones. Para determinar si una tupla tiene reinas en posición de ataque (o no), por cada una de las entradas x_i de la tupla, determinamos si está en posición de ataque con cada una de las reinas de las filas $i - 1, i - 2, \dots, 1$. Para determinar si una reina en la fila F y columna C del tablero (representado por una tupla $T[1..N]$) están en posición de ataque con otras reinas en filas menores a F , usamos el Algoritmo 25. El algoritmo retornar true si encontramos

Algoritmo 25: ENATAQUE(tupla $T[1..N]$, fila F , columna C)

```

1  $i \leftarrow F - 1$ 
2 while  $i \geq 1$  do
3   if  $T[i] = T[F]$  or  $i - T[i] = F - C$  or  $i + T[i] = F + C$  then
4     return true
5   end
6    $i \leftarrow i - 1$ 
7 end
8 return false

```

reinas en ataque.

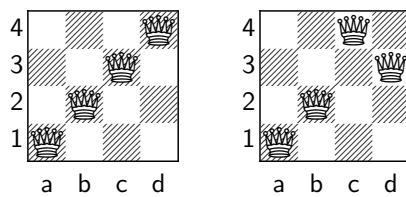
Una mejora a este proceso consiste en chequear sólo las $N!$ posibles permutaciones de $1, \dots, N$, ya que valores repetidos dentro de una tupla indican dos reinas en la misma columna. Para $N = 4$, eso significa 24 posibles configuraciones a chequear, una décima parte del trabajo del algoritmo anterior. Aquí sólo modificamos la forma en que generamos las tuplas, el resto del proceso es similar al explicado anteriormente.

Sin embargo, este algoritmo todavía chequea tuplas que podríamos

evitar. Supongamos que el orden de chequeo es el siguiente:

(a, b, c, d)
 (a, b, d, c)
 (a, c, b, d)
 (a, c, d, b)
 (a, d, b, c)
 (a, d, c, b)
 ⋮

Las primeras dos tuplas corresponden a los tableros:



Luego de chequear la primera tupla (tablero de la izquierda) y determinar que las reinas de las filas 1 y 2 se atacan entre ellas, podríamos descartar la siguiente tupla sin chequear, ya que tiene las reinas que se atacan en las mismas posiciones.

Estudiamos a continuación un algoritmo de backtracking que genera las posibles soluciones de forma ordenada, descartando de forma eficiente tuplas que no pueden ser soluciones. Para simplificar el discurso, denotaremos con Q_i a la reina correspondiente a la fila i del tablero, para $1 \leq i \leq N$. El algoritmo recursivo funcionará considerando una fila por vez, comenzando desde la fila 1. En el nivel 1 de la recursión, colocará la reina Q_1 en todas las columnas de la fila 1, una a la vez, en order (al ser la primera fila, y al no haber otras reinas en el tablero, en la fila 1 tenemos total libertad). Luego de ubicar a Q_1 , se invoca recursivamente para considerar a Q_2 . Sólo en caso de poder ubicar a la reina Q_2 sin que se ataque con Q_1 , avanzaremos a la fila 3 recursivamente. Eso significa que mantendremos la invariante de que al avanzar hacia la fila i , las reinas Q_1, \dots, Q_{i-1} están en posición de no ataque entre ellas.

La Figura 10.1 muestra el árbol de recursión para resolver el problema de las 4 reinas. El nodo raíz del árbol corresponde al tablero sin reinas. El algoritmo comienza considerando la fila 1, por lo que ubica a Q_1 en la columna a1, y se invoca recursivamente para resolver la fila 2. Dado que Q_1 está ubicada en a1, Q_2 no puede ser ubicada en la columna a2, ni tampoco en la b2. La primera alternativa segura es la columna c2. Luego, se invoca recursivamente para considerar la fila 3, aunque sin éxito: en cualquiera de las columnas en que se ubique Q_3 , queda en posición de ataque con Q_2 y Q_1 . Esto permite podar el árbol de recursión en este punto, evitando chequear todas las soluciones que derivan de la actual.

En este momento retornamos al nivel 2 (hacemos backtracking), y avanzamos Q_2 a d2, para nuevamente invocarse recursivamente para ubicar a Q_3 . Esta vez tendremos éxito al ubicar a Q_3 en b3, y se invocas

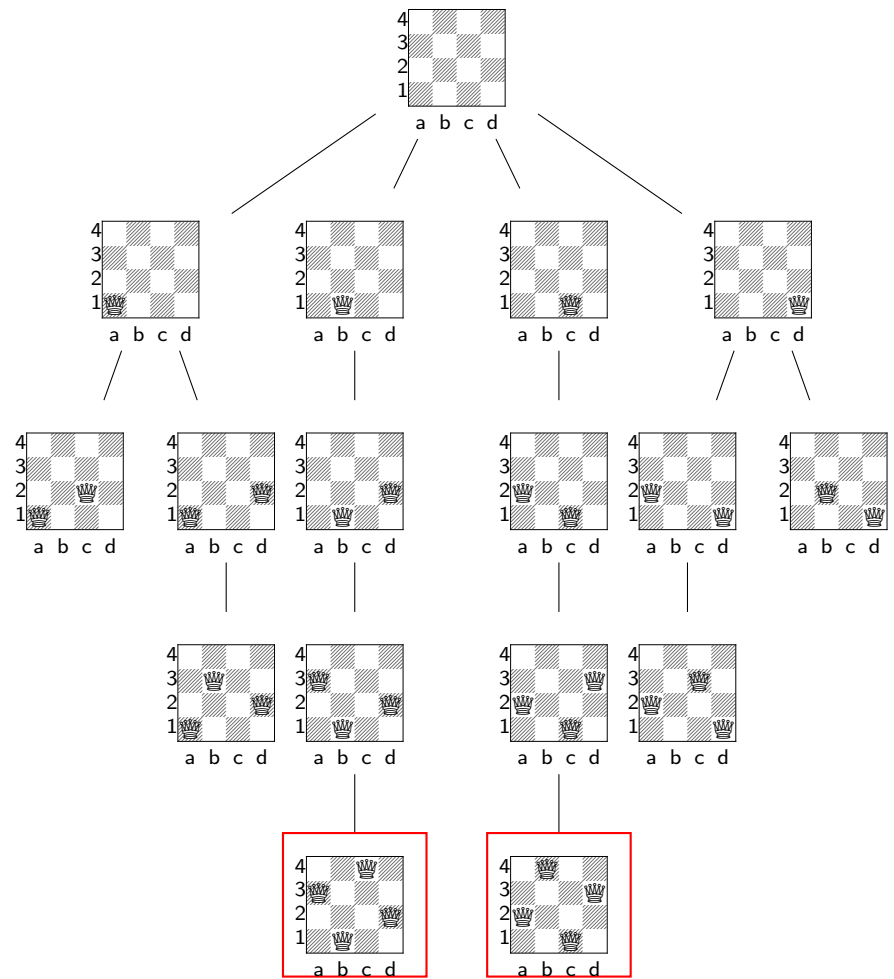


Figura 10.1: Árbol de recursión y backtracking para el problema de las 4 reinas. Las dos soluciones posibles son los nodos externos marcados.

recursivamente para ubicar a Q_4 , algo que no podremos hacer dadas las ubicaciones de las demás reinas. Retornamos al nivel 3, y al no poder ubicar a Q_3 en ninguna otra columna sin ataques, retornamos al nivel 2. Dado que Q_2 estaba en d2, no tenemos más alternativas para ella y regresamos a la fila 1. Con todo este trabajo, acabamos de descubrir que colocar a Q_1 en a1 no produjo ningún resultado, por lo que debemos avanzar Q_1 a b1, y pasamos a considerar recursivamente la fila 2. En esta fila, la única ubicación posible para Q_2 es d2, y continuamos recursivamente en la fila 3. Aquí, Q_3 puede ubicarse en a3, y continuamos recursivamente en el nivel 4. Iterando desde a4, la primera columna en que podemos ubicar a Q_4 es c4, y continuamos recursivamente en el nivel 5. En este punto, dado que estamos en un tablero de 4×4 , el algoritmo detecta que hemos ubicado correctamente a las 4 reinas, por lo que reporta la solución encontrada: (b, d, a, c). Luego, el proceso continúa buscando las siguientes soluciones.

El proceso para encontrar todas las soluciones al problema de las N reinas es implementado por el Algoritmo 26. Aunque la cantidad de configuraciones completas que deben ser probadas todavía es $O(N!)$, la Figura 10.1 permite ver que para el caso $N = 4$ son menos que $N!$: sólo 16 configuraciones completas (es decir, con las 4 reinas ubicadas) son chequeadas, correspondientes a los 4 nodos de nivel 3 del árbol de la figura. Dichas configuraciones son: (a, d, b, a), (a, d, b, b), (a, d, b, c),

Algoritmo 26: NREINAS($T[1..N]$, nivel)

```

1 if nivel =  $N + 1$  then
2   | Print  $T$ 
3 else
4   |  $T[\text{nivel}] \leftarrow 1$  // Primera columna
5   | for col  $\leftarrow 1$  to  $N$  do
6     | if not ENATAQUE(nivel, col) then
7       |   NREINAS(nivel + 1)
8     | end
9     |  $T[\text{nivel}] \leftarrow T[\text{nivel}] + 1$  // Avanza la reina
10  | end
11 end

```

(a, d, b, d), (b, d, a, a), (b, d, a, b), (b, d, a, c), (b, d, a, d), (c, a, d, a), (c, a, d, b), (c, a, d, c), (c, a, d, d), (d, a, c, a), (d, a, c, b), (d, a, c, c), y (d, a, c, d). El algoritmo anterior (que no usa backtracking, que chequea todas las permutaciones posibles), chequeaba 24, que hubiese correspondido a 6 nodos de nivel 3 en el caso del árbol. Note, sin embargo, cómo 2 de 6 nodos son podados en el segundo nivel del árbol, permitiendo ahorrar tiempo.

La Figura 10.2 muestra una posible implementación de la solución, empleando lenguaje C.

```

1  int soluciones = 0;
2  int N;
3  int fila[1024];
4
5  void printSolucion() {
6      printf("[");
7      for (i = 0; i < N; i++)
8          printf(" %d", fila[i]);
9      printf("]\n");
10 }
11
12 int enAtaque(int fAct, int cAct) {
13     int i;
14     for (i = fAct - 1; i >= 0; i--)
15         if (fila[i]==fila[fAct]
16             || i-fila[i]==fAct-cAct
17             || i+fila[i]==fAct+cAct)
18             return 1;
19     return 0;
20 }
21
22 void nReinas(int fAct) {
23     int col;
24     if (fAct == N) {
25         soluciones++;
26         printSolucion();
27     }
28     else {
29         fila[fAct] = 0;
30         for (col = 0; col < N; col++) {
31             if (!enAtaque(fAct, col))
32                 nReinas(fAct+1);
33             fila[fAct]++;
34         }
35     }
36 }

```

Figura 10.2: Algoritmo de backtracking recursivo para el problema de las N reinas.

Ejercicios

1. Sea S un conjunto ordenado de n enteros positivos, y K un valor entero positivo. Use la técnica de backtracking recursivo para diseñar un algoritmo que permita encontrar todos los subconjuntos de S tal que la suma de sus elementos sea K .
2. (Problema de la mochila 0-1) Dados n elementos e_1, \dots, e_n , los cuales pesan p_1, \dots, p_n y valen v_1, \dots, v_n , y dada una mochila capaz de almacenar hasta un máximo de peso W , se quiere encontrar un vector binario x_1, \dots, x_n (es decir, $x_i = 0$ o $x_i = 1$, para $1 \leq x_i \leq n$) que maximice el valor

$$\sum_{i=1}^n x_i \cdot v_i,$$

sujeto a la restricción

$$\sum_{i=1}^n x_i \cdot p_i \leq W.$$

Diseñe un algoritmo de backtracking recursivo que resuelva este problema.

3. Diseñe un algoritmo de backtracking recursivo tal que dado un número natural $x > 0$, imprima todas las formas de obtener x mediante la suma de una secuencia ascendente de números naturales mayores a 0. Por ejemplo, para $x = 6$ la respuesta debe ser

- $1 + 2 + 3$,
- $1 + 5$,
- $2 + 4$,
- 6 .

Use todos los criterios que permitan detener la ejecución y hacer backtracking en un nodo infactible del árbol de recursión. Justifique esos criterios brevemente.

4. Sea $n \in \mathbb{N}^+$ un número par. Diremos que una permutación de los números $1, 2, \dots, n$ es válida si cumple con las siguientes propiedades:
 - El primer elemento de la secuencia siempre es 1.
 - La suma de cualquier par de números consecutivos en la permutación produce un número primo. Considere que la permutación es circular, de tal manera que el último elemento es consecutivo al primer elemento (y viceversa).

Por ejemplo, para $n = 6$ hay dos posibles permutaciones válidas: $1, 4, 3, 2, 5, 6$ y $1, 6, 5, 2, 3, 4$. Diseñe un algoritmo de backtracking recursivo que muestre todas las permutaciones válidas para un valor entero positivo par n dado como parámetro.

11.1. Introducción

La técnica de diseño de algoritmos conocida como *dividir y conquistar* permite resolver un problema mediante su división en *subproblemas que no se solapan* entre sí. Cada subproblema es una instancia más pequeña del problema original, y puede ser resuelto recursivamente usando el mismo algoritmo. Al ser más pequeños, los subproblemas son más “fáciles” de resolver. El proceso se repite recursivamente, dividiendo cada vez en subproblemas más y más pequeños, hasta que estos sean lo suficientemente pequeños para resolverlos de manera simple. Una vez que todos los subproblemas han sido resueltos, esas soluciones son usadas para construir la solución al problema original. Esta idea ya ha sido estudiada anteriormente en el Capítulo 4.4. Sin embargo, en este capítulo haremos énfasis en los casos en los que subproblemas no se solapan. Esto es, casos en que un mismo subproblema es resuelto en una única rama del árbol de recursión.

Un algoritmo del tipo dividir y conquistar consiste de las siguientes etapas:

Dividir: el problema original de tamaño n es dividido en a subproblemas no solapantes de tamaño n_1, n_2, \dots, n_a .

Conquistar: los subproblemas generados en la etapa anterior se resuelven recursivamente. Si un subproblema es de tamaño suficientemente pequeño, la recursión se detiene y se resuelve directamente.

Combinar: una vez que los subproblemas han sido resueltos, sus soluciones se combinan para formar la solución al problema original.

Al igual que en capítulos anteriores, el pensamiento recursivo (y por inducción), es clave para dominar esta técnica de diseño.

11.2. El Problema de Ordenamiento

En capítulos anteriores hemos estudiado dos algoritmos para resolver el problema de ordenar un conjunto A de n elementos, en particular los algoritmos *BubbleSort* y *SelectionSort*. Ambos resuelven el problema con $O(n^2)$ comparaciones. Además, en la Sección 7.2 demostramos una cota inferior $\Omega(n \lg n)$ para este problema en el modelo de comparaciones. Todo esto permite acotar la complejidad $T_O(n)$ del problema de ordenamiento de la siguiente manera:

$$T_O(n) \in \Omega(n \lg n) \text{ y } T_O(n) \in O(n^2).$$

No es claro hasta este punto si se puede diseñar un algoritmo más eficiente para resolver el problema, o si la cota inferior demostrada no es ajustada. Mostramos a continuación que la primera alternativa es la

correcta, definiendo algoritmos de ordenamiento más eficientes que los estudiados anteriormente.

MergeSort

El algoritmo conocido como **MergeSort** fue propuesto en el año 1945 por John von Neumann, y fue el primero en ejecutarse bajo el modelo que hoy en día se conoce como modelo de von Neumann. Sea $A[1..n]$ el conjunto de elementos a ordenar, representado como un arreglo. Asumiendo que existe una relación de orden total sobre A , diseñaremos un algoritmo recursivo para ordenarlo en el modelo de comparaciones. Note que si $n = 1$, el arreglo está trivialmente ordenado. Este es el caso base de la recursión para **MergeSort**. Luego, como ya lo hemos explicado anteriormente, podemos asumir (como hipótesis inductiva) que **MergeSort** permite ordenar correctamente arreglos de tamaño $< n$. En base a esto, mostramos cómo ordenar un arreglo de n elementos, para cualquier $n > 1$, completando la inducción. En particular, dividimos el arreglo de tamaño n en dos subarreglos $A[1..\frac{n}{2}]$ y $A[\frac{n}{2} + 1..n]$. Dado que esos subarreglos son de tamaño $\frac{n}{2}$ cada uno, por hipótesis inductiva sabemos cómo ordenarlos usando **MergeSort**, de forma recursiva. Finalmente, usamos el proceso especificado en el Algoritmo 27 sobre esas mitades ordenadas para ordenar el arreglo completo.

Algoritmo 27: MERGE($A_1[1..n], A_2[1..m], A[1..n + m]$)

```

1  $i_1 \leftarrow 1$ 
2  $i_2 \leftarrow 1$ 
3  $i \leftarrow 1$ 
4 while  $i_1 \leq n$  and  $i_2 \leq m$  do
5   if  $A_1[i_1] \leq A_2[i_2]$  then
6      $A[i++] \leftarrow A_1[i_1++]$ 
7   else
8      $A[i++] \leftarrow A_2[i_2++]$ 
9   end
10 end
11 while  $i_1 \leq n$  do  $A[i++] \leftarrow A_1[i_1++]$ 
12 while  $i_2 \leq m$  do  $A[i++] \leftarrow A_2[i_2++]$ 

```

Las etapas descritas anteriormente para este algoritmo en particular son las siguientes:

Dividir: Dividir el arreglo a ser ordenado en dos mitades. Es importante notar que la división es lógica y no física, ya que simplemente hay que determinar la mitad del arreglo para realizar la división. Este proceso, por lo tanto, no realiza comparaciones de elementos.

Conquistar: Ordenar las dos mitades por separado, de manera recursiva. El punto de parada son arreglos de tamaño 1, los cuales están trivialmente ordenados.

Combinar: Mezclar ambas mitades (ya ordenadas) del arreglo para ordenar el arreglo original. Como se estudió en capítulos anteriores, la cantidad óptima de comparaciones necesarias para mezclar dos arreglos de tamaño $n/2$ es $n - 1$, en el peor caso.

Algoritmo 28: MergeSort($A[1..n]$)

```

1 if  $n = 1$  then
2   return
3 else
4   MergeSort( $A[1..\frac{n}{2}]$ )
5   MergeSort( $A[\frac{n}{2} + 1..n]$ )
6   for  $i \leftarrow 1$  to  $n$  do
7      $temp[i] = A[i]$ 
8   end
9   MERGE( $temp[1..\frac{n}{2}]$ ,  $temp[\frac{n}{2} + 1..n]$ ,  $A[1..n]$ )
10 end

```

Este proceso recursivo completo se muestra en el Algoritmo 28. Note que el algoritmo utiliza un arreglo temporal **temp**, necesario para realizar la mezcla de las mitades ordenadas. Esto implica una cantidad $\Theta(n)$ de espacio adicional, lo que es una desventaja de este algoritmo de ordenamiento: duplica el espacio requerido por el arreglo original.

La Figura 11.1 muestra el árbol de recursión para las operaciones Dividir y Conquistar del algoritmo MergeSort, aplicado sobre el arreglo (55, 4, 58, 90, 56, 30, 85, 80, 50, 35). A partir de esto, la etapa Combinar procede de la siguiente manera:

1. Se invoca al algoritmo Merge (Algoritmo 27) con los subarreglos ordenados $A[1] = (55)$ y $A[2] = (4)$, para obtener el subarreglo ordenado $A[1..2] = (4, 55)$.
2. Se invoca al algoritmo Merge con los subarreglos ordenados $A[4] = (90)$ y $A[5] = (56)$, para obtener el subarreglo ordenado $A[4..5] = (56, 90)$.
3. Se invoca al algoritmo Merge con los subarreglos ordenados $A[3] = (58)$ y $A[4..5] = (56, 90)$, para obtener el subarreglo ordenado $A[3..5] = (56, 58, 90)$.
4. Se invoca al algoritmo Merge con los subarreglos ordenados $A[1..2] = (4, 55)$ y $A[3..5] = (56, 58, 90)$, para obtener el subarreglo ordenado $A[1..5] = (4, 55, 56, 58, 90)$.
5. Se invoca al algoritmo Merge con los subarreglos ordenados $A[6] = (30)$ y $A[7] = (85)$, para obtener el subarreglo ordenado $A[6..7] = (30, 85)$.
6. Se invoca al algoritmo Merge con los subarreglos ordenados $A[9] = (50)$ y $A[10] = (35)$, para obtener el subarreglo ordenado $A[9..10] = (35, 50)$.
7. Se invoca al algoritmo Merge con los subarreglos ordenados $A[8] = (80)$ y $A[9..10] = (35, 50)$, para obtener el subarreglo ordenado $A[8..10] = (35, 50, 80)$.
8. Se invoca al algoritmo Merge con los subarreglos ordenados $A[6..7] = (30, 85)$ y $A[8..10] = (35, 50, 80)$, para obtener el subarreglo ordenado $A[6..10] = (30, 35, 50, 80, 85)$.
9. Se invoca al algoritmo Merge con los subarreglos ordenados $A[1..5] = (4, 55, 56, 58, 90)$ y $A[6..10] = (30, 35, 50, 80, 85)$, para obtener el arreglo ordenado $A[1..10] = (4, 30, 35, 50, 55, 56, 58, 80, 85, 90)$.

Si denotamos con $T(n)$ al tiempo de ejecución de MergeSort para un arreglo de n elementos, la etapa Conquistar realiza $2T(n/2)$ comparaciones, dado que se invoca 2 veces a MergeSort con arreglos de tamaño

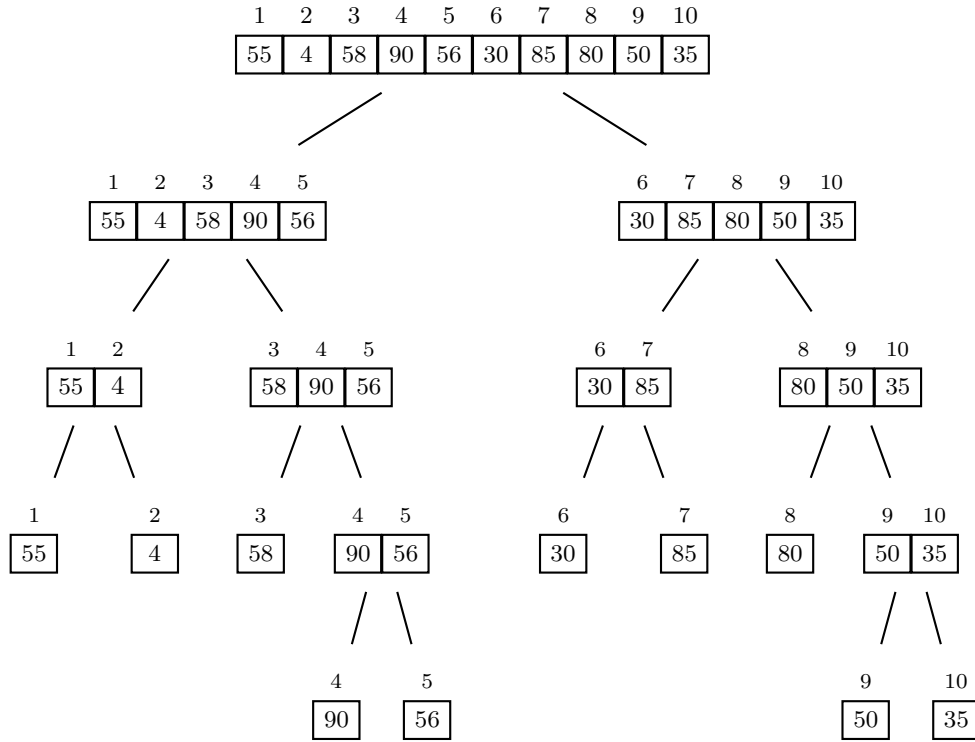


Figura 11.1: Árbol de recursión para las etapas *Dividir* y *Conquistar* del algoritmo MergeSort sobre el arreglo (55, 4, 58, 90, 56, 30, 85, 80, 50, 35).

$n/2$. Finalmente, la etapa Combinar necesita $n - 1$ comparaciones en el peor caso, y $n/2$ comparaciones en el mejor caso. Esto es, el tiempo de la etapa combinar puede acotarse con $\Theta(n)$ comparaciones. La ecuación de recurrencia para el tiempo de ejecución de peor caso es, entonces:

$$T(n) = \begin{cases} 0, & n = 1; \\ 2T(n/2) + n - 1; & n > 1, \end{cases} \quad (11.1)$$

que es similar a la Ecuación (4.6) (en la página 81). Aplicando las técnicas de resolución de ecuaciones de recurrencia del Capítulo 4.4, llegamos a $T(n) = n \lg n - n + 1$ comparaciones. Esto significa que MergeSort es asintóticamente óptimo para el problema de ordenamiento en el modelo de comparaciones. Otra forma de entender este resultado puede verse en el árbol de recursión de la Figura 11.2. En cada nodo del árbol se muestra el tamaño del problema a resolver: la raíz resuelve el problema completo de tamaño n , a nivel 1 se tienen 2 subproblemas de tamaño $\frac{n}{2}$, y así siguiendo hasta el nivel $\lg n$, en donde se tienen $2^{\lg n} = n$ subproblemas de tamaño $\frac{n}{2^{\lg n}} = 1$ elementos cada uno. A la derecha del árbol se muestra la cantidad total de comparaciones de peor caso realizadas en cada nivel. En el primer nivel, se hacen $n - 1$ comparaciones en el peor caso para producir el arreglo completo ordenado. En el segundo nivel, en total se hacen $2(\frac{n}{2} - 1)$ comparaciones, en el tercer nivel $4(\frac{n}{4} - 1)$ comparaciones, y así hasta el nivel $\lg n$ en el que se realizan $2^{\lg n}(\frac{n}{2^{\lg n}} - 1) = 0$ comparaciones en total (recuerde que en este nivel, los subproblemas tienen tamaño 0, por lo que no es necesario hacer comparaciones para ordenarlos). De esta manera, la

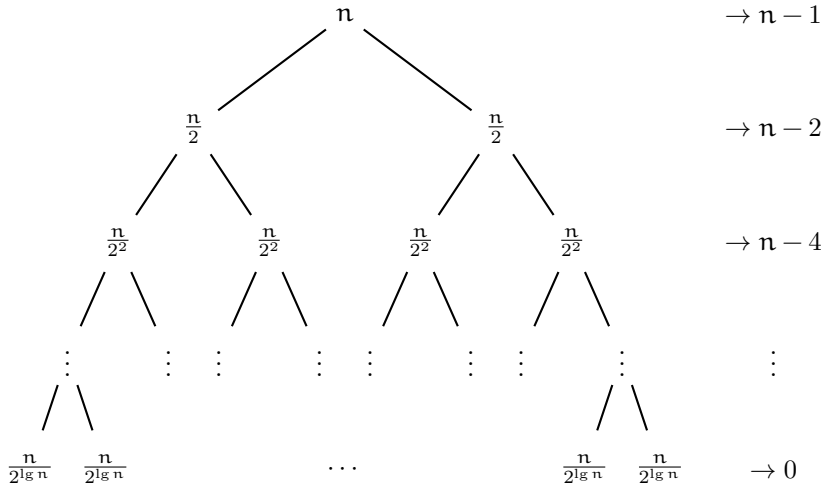


Figura 11.2: Árbol de recursión para el algoritmo MergeSort. En cada nodo del árbol se muestra el tamaño del arreglo correspondiente, y a la derecha se muestra la cantidad total de comparaciones realizadas en cada nivel en el peor caso.

cantidad total de comparaciones realizadas por el algoritmo es:

$$T(n) = \sum_{i=0}^{\lg n} 2^i \left(\frac{n}{2^i} - 1 \right) = n(1 + \lg n) - \sum_{i=0}^{\lg n} 2^i = n \lg n - n + 1.$$

Recuerde que este análisis es en el peor caso, en donde se asume que cada vez se realiza la máxima cantidad de operaciones para la mezcla de dos arreglos ordenados. Esto nos entrega más conocimiento sobre el problema de ordenamiento: su complejidad de peor caso es $T_O(n) \in \Theta(n \lg n)$ comparaciones. Por otro lado, note que en el mejor caso, en el nivel 0 del árbol se realizan $\frac{n}{2}$ comparaciones para producir el arreglo ordenado completo, en el nivel 1 se realizan $2 \frac{n}{4} = \frac{n}{2}$ comparaciones, luego $4 \frac{n}{8} = \frac{n}{2}$ comparaciones, y así siguiendo hasta el nivel $\lg n$, en donde se hacen 0 comparaciones. La suma total es $\frac{n}{2} \lg n$ comparaciones en el mejor caso. En conclusión, podemos decir que el tiempo de ejecución de MergeSort es $\Theta(n \lg n)$.

QuickSort

El algoritmo QuickSort fue propuesto por C. A. R. Hoare en 1959 y publicado en 1961¹. El algoritmo fue seleccionado entre los top-10 algoritmos del siglo XX². Este es uno de los algoritmos de ordenamiento más conocidos y usados. Es la base de la función `sort` provista por muchas bibliotecas de varios lenguajes de programación.

El algoritmo estudiado en la sección anterior, MergeSort, utiliza la forma mas obvia de Dividir y Conquistar: dividir siempre el arreglo en dos mitades, luego ordenarlas, y finalmente mezclarlas para generar el arreglo ordenado. Como ya vimos, esto genera un árbol de recursión (casi) perfectamente balanceado, y mostramos que esto es óptimo en el modelo de comparaciones. Sin embargo, no es la única manera de dividir el arreglo. Por ejemplo, uno podría dividirlo en dos partes desiguales pero de tamaño fijo (1/3 versus 2/3, por ejemplo), o usando particiones de tamaño no fijo, que varíen a lo largo del proceso. En ambos casos,

¹ C. A. R. Hoare. Algorithm 64: Quicksort. Communications of the ACM. 4 (7): 321. 1961. doi:10.1145/366622.366644

² Computing in Science & Engineering, Vol. 2, número 1, 2000.

se podría generar un árbol de recursión no balanceado. Vimos, además, que MergeSort usa un arreglo adicional para realizar la mezcla, lo cual consume espacio de memoria y hace que el algoritmo no sea in-place. Veremos que QuickSort implementa una estrategia más relajada en términos de división del arreglo, que resulta en general más práctica que MergeSort.

Para entender cómo funciona QuickSort, pensemos en cómo podemos ordenar usando un árbol binario de búsqueda (ABB). La idea es insertar los elementos en un ABB (inicialmente vacío) uno por uno para, finalmente, recorrerlo en inorden, obteniendo los elementos ordenados. Esto requiere espacio extra para los punteros del árbol, y tiempo adicional no despreciable en la práctica para manejar la memoria dinámica para los n nodos. Este proceso es conocido como TreeSort³, y tiene tiempo de ejecución promedio $\Theta(n \lg n)$ si las $n!$ permutaciones de los n elementos del arreglo son igualmente probables⁴. Aunque en general no es una solución eficiente en la práctica, y por lo tanto no se usa, introduce otra manera de pensar el problema: la raíz del árbol divide el problema en dos partes (no necesariamente del mismo tamaño), implementando Dividir y Conquistar de manera implícita. QuickSort implementa esta idea aunque de forma más eficiente, reduciendo el espacio extra usado y sin manejar memoria dinámica para los nodos: el árbol será implícito, implementado por la recursión.

QuickSort ejecuta las tres etapas Dividir y Conquistar conceptualmente de la siguiente manera:

Dividir: Si el arreglo tiene tamaño ≤ 1 , está trivialmente ordenado.

Sino, se elige un elemento arbitrario p del arreglo A , el *pivote*, y se construyen dos arreglos que representan los conjuntos $A_{\min} = \{x \in A \mid x < p\}$ y $A_{\max} = \{x \in A \mid x > p\}$, que contienen los elementos menores y mayores que p , respectivamente.

Conquistar: Se aplica QuickSort recursivamente sobre los arreglos A_{\min} y A_{\max} , generados en el paso anterior, con lo cual los ordenamos.

Combinar: Luego de ordenar A_{\min} y A_{\max} , se concatena A_{\min} con p y con A_{\max} , en ese orden, para producir el arreglo ordenado.

A continuación, implementamos esas tres etapas eficientemente, con un único recorrido del subarreglo correspondiente a cada nodo del árbol de recursión, y sin usar espacio adicional. La etapa Dividir particionará el arreglo de forma tal que todos los elementos menores o iguales al pivote queden en el extremo izquierdo del arreglo, mientras que los elementos mayores al pivote quedarán en el extremo derecho. Esto es, implementaremos los arreglos A_{\min} y A_{\max} de forma implícita dentro del mismo arreglo A . Primero, elegimos el pivote $p = A[j]$, para una posición $1 \leq j \leq n$ del arreglo. Por ejemplo, suponga el siguiente arreglo, en donde el pivote es $p = A[5] = 56$:

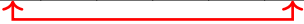
³ R. Floyd. Algorithm 245: Treesort. Communications of the ACM. 7 (12): 701. 1964.

⁴ El tiempo puede hacerse $\Theta(n \lg n)$ en el peor caso, usando un AVL en lugar de un ABB.

1	2	3	4	5	6	7	8	9	10
55	4	58	90	56	30	85	80	50	35



Luego, el pivote es reservado en la última posición del arreglo, intercambiando su posición con el elemento que estaba allí mediante la operación $\text{SWAP}(A[j], A[n])$. En el ejemplo hacemos:

1	2	3	4	5	6	7	8	9	10
55	4	58	90	35	30	85	80	50	56


 SWAP



A continuación, usamos dos variables índice $i \leftarrow 1$ y $d \leftarrow n - 1$:

1	2	3	4	5	6	7	8	9	10
55	4	58	90	35	30	85	80	50	56



i *d*


La variable i es usada para iterar de izquierda a derecha, mientras $A[i] \leq p$. La variable d , por su lado, es usada para iterar de derecha a izquierda, mientras $A[d] > p$. En nuestro ejemplo, luego de ejecutar ambas iteraciones tenemos:

1	2	3	4	5	6	7	8	9	10
55	4	58	90	35	30	85	80	50	56



i *d*



Ahora ejecutamos $\text{SWAP}(A[i], A[d])$, para intercambiar los elementos, obteniendo:

1	2	3	4	5	6	7	8	9	10
55	4	50	90	35	30	85	80	58	56


 SWAP
i *d*

Repetimos el proceso, avanzando con i y retrocediendo con d , con las mismas condiciones que antes, obteniendo:

1	2	3	4	5	6	7	8	9	10
55	4	50	90	35	30	85	80	58	56



i *d*

Luego, nuevamente intercambiamos los elementos correspondientes:

1	2	3	4	5	6	7	8	9	10
55	4	50	30	35	90	85	80	58	56

i d

Continuamos con este proceso hasta que eventualmente ambos índices, i y d , coincidan en una misma posición. Dado que la iteración que hace avanzar a i se ejecuta antes que la iteración para d , el punto de encuentro entre ambos índices corresponde necesariamente a un elemento mayor al pivote ⁵. Además, todos los elementos que están a la izquierda de i son elementos menores (o iguales) al pivote, y los que están a su derecha son mayores. En otras palabras, A_{\min} es el subarreglo que está a la izquierda de i , mientras que A_{\max} está a la derecha. En nuestro ejemplo tenemos:

1	2	3	4	5	6	7	8	9	10
55	4	50	30	35	90	85	80	58	56

i, d

Ya que el elemento $A[i]$ es mayor al pivote, podemos desplazarlo al final del arreglo, intercambiando su posición con el pivote. Como todos los elementos a la izquierda de i son menores o igual al pivote, y los de la derecha son mayores, se cumple lo que buscábamos: separar los elementos de A_{\min} y A_{\max} dentro del mismo arreglo. En nuestro ejemplo tenemos:

1	2	3	4	5	6	7	8	9	10
55	4	50	30	35	56	85	80	58	90

SWAP

Finalmente, debemos resolver recursivamente el problema de ordenar los segmentos izquierdo y derecho del arreglo:

1	2	3	4	5	6	7	8	9	10
55	4	50	30	35	56	85	80	58	90

QUICKSORT QUICKSORT

Es importante notar que al finalizar la 56 partición del arreglo, el pivote se encuentra en su posición final del arreglo ordenado. Ninguna otra operación del algoritmo va a tocarlo. En nuestro ejemplo, significa que el pivote $p = 56$ tiene ranking 6 (esto es, estará en la sexta posición en el arreglo ordenado). Esto implica que la etapa de combinación no necesita realizar ninguna operación: luego de ordenar A_{\min} y A_{\max} recursivamente, el arreglo estará ordenado, sin necesidad de realizar acciones adicionales de combinación.

El Algoritmo 29 formaliza este proceso. Luego de realizar la partición,

⁵ El único caso en que esto no así es cuando el pivote es mayor que todos los elementos del arreglo, lo que ocurre cuando $i = d = n - 1$.

Algoritmo 29: PARTICIÓN(A, i, d, p)

```

1  $i \leftarrow i - 1$ 
2  $d \leftarrow d + 1$ 
3 repeat
4    $i \leftarrow i + 1$ 
5   while  $A[i] < A[p]$  do  $i \leftarrow i + 1$ 
6    $d \leftarrow d - 1$ 
7   while  $i < d$  and  $A[d] > A[p]$  do  $d \leftarrow d - 1$ 
8   if  $i < d$  then SWAP( $A[i], A[d]$ )
9 until  $i \geq d$ ;
10 return  $i$ 

```

el algoritmo finaliza devolviendo la posición final del índice i (recuerde que es igual a d en ese punto). El Algoritmo 30 muestra el proceso completo para QuickSort. La Figura 11.3 muestra el árbol de recursión correspondiente a aplicar QuickSort sobre el arreglo: El mismo elige el

Algoritmo 30: QUICKSORT(A, i, d)

```

1  $n \leftarrow d - i + 1$ 
2 if  $n \leq 1$  then
3   return
4 else
5    $p \leftarrow \frac{i+d}{2}$ 
6   SWAP( $A[p], A[d]$ )
7    $k \leftarrow$  PARTICIÓN( $A, i, d, p$ )
8   SWAP( $A[k], A[d]$ )
9   if  $k = d - 1$  then  $k \leftarrow d$ 
10  QUICKSORT( $A, i, k - 1$ )
11  QUICKSORT( $A, k + 1, d$ )
12 end

```

elemento ubicado en la mitad del arreglo como pivote, aunque puede ser cualquier otro. Como seguramente lo ha notado, la elección del pivote es clave para la eficiencia del algoritmo. Recuerde que, conceptualmente, el pivote es equivalente al elemento insertado en la raíz del ABB que almacena al conjunto. Si elegimos como pivote al elemento más pequeño del conjunto (o al más grande), el árbol de recursión estará completamente desbalanceado hacia uno de los lados. Si repetimos esta mala elección en cada uno de los nodos del árbol, vamos a tener un árbol de recursión de altura $n - 1$. En ese caso, la cantidad de comparaciones realizadas en el nodo de profundidad j , para $0 \leq j \leq n - 1$, es $n - j - 1$. Entonces, la cantidad de comparaciones en el peor caso es:

$$\sum_{j=0}^{n-1} n - j - 1 = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2},$$

por lo que el tiempo de ejecución del algoritmo es $O(n^2)$. Esto significa que, en el peor caso, QuickSort puede ser tan ineficiente como BubbleSort o SelectionSort. Aún cuando el proceso de partición es in-place en cada nodo de la recursión, el uso de espacio adicional de QuickSort en este caso es proporcional a n , debido a la profundidad de la recursión. En conclusión, QuickSort requiere espacio adicional $O(n)$.

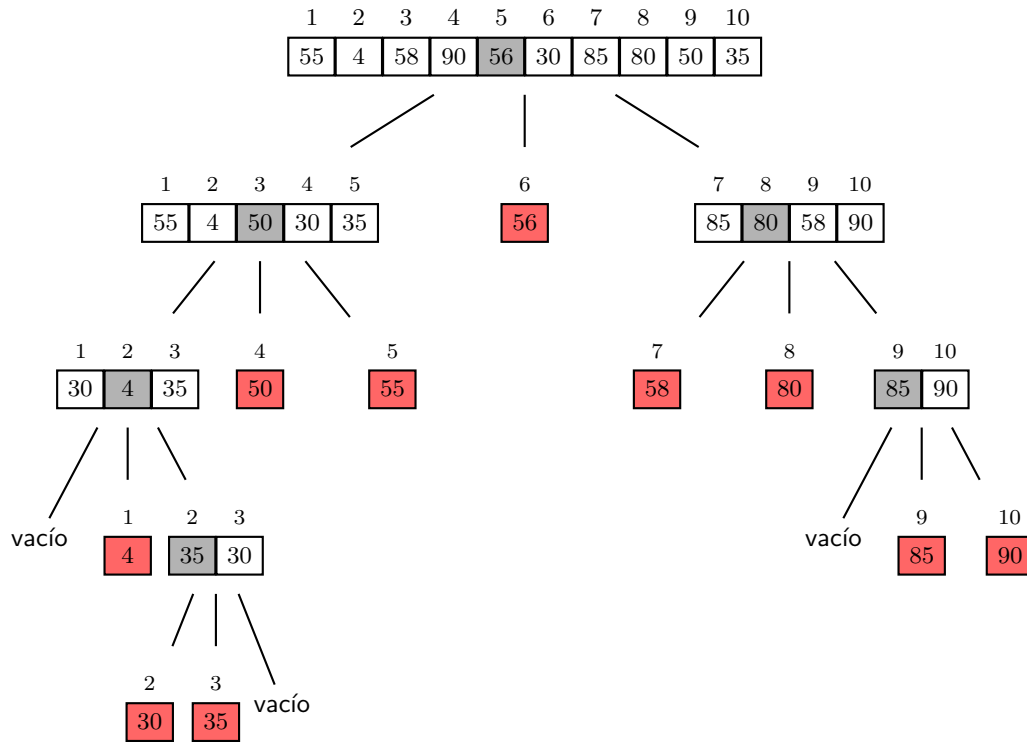


Figura 11.3: Árbol de recursión para el algoritmo QuickSort sobre el arreglo (55, 4, 58, 90, 56, 30, 85, 80, 50, 35). El hijo central de cada nodo no corresponde a una invocación recursiva del algoritmo, sino que representa conceptualmente al pivote de cada nodo. Esto permite una mejor visualización del arreglo ordenado (marcado en nodos en color rosa).

Sin embargo, a pesar de esto, QuickSort suele ser más rápido en la práctica que MergeSort, en particular cuando el arreglo de entrada tiene orden aleatorio. Aunque nos hemos centrado exclusivamente en la cantidad de comparaciones al analizar algoritmos de ordenamiento, en la práctica hay que considerar otras operaciones necesarias para el proceso. Por ejemplo, la asignación de elementos del arreglo, necesaria tanto para traspasar elementos entre arreglos (MergeSort) como para intercambiar elementos que están invertidos usando la función SWAP (QuickSort). Hay dos razones principales que explican por qué QuickSort suele ser más práctico que MergeSort:

- Primero, consideremos un nodo del árbol de recursión correspondiente a un arreglo de n' elementos. En este nodo, QuickSort realiza $n' - 1$ comparaciones y $3(\lfloor \frac{n'-1}{2} \rfloor + 2)$ asignaciones de elementos en el peor caso, debido a SWAP. Esto es porque en el peor caso hay que intercambiar $\lfloor \frac{n'-1}{2} \rfloor$ elementos, y cada SWAP necesita 3 asignaciones. MergeSort, por otro lado, hace $n' - 1$ comparaciones en el peor caso, y *exactamente* $2n'$ asignaciones de elementos para traspasarlos (n' para traspasarlos al arreglo `temp` del Algoritmo 28, para luego volver a traspasarlos, ordenados, al arreglo original `A`). Esto es, aún cuando ambos algoritmos hacen la misma cantidad de comparaciones en el peor caso dentro de un nodo del árbol de recursión (y MergeSort podría realizar menos en algunos casos), MergeSort realiza siempre $2n'$ asignaciones, mientras que QuickSort realiza $3\lfloor \frac{n'-1}{2} \rfloor$ en el peor caso. Esto podría incrementar el tiempo de ejecución de MergeSort en la práctica.

- La segunda razón (que, además, refuerza el punto anterior) es que si la entrada tiene orden aleatorio, el tiempo de ejecución promedio de QuickSort es proporcional a $n \lg n$, lo que indica que el árbol de recursión es, en promedio, un árbol balanceado. Para entender esto intuitivamente, recuerde que la profundidad promedio de un nodo externo en un árbol binario de búsqueda es $\sim 1,386 \lg n$, si los elementos son insertados en orden aleatorio. Esto indica que la profundidad promedio de la recursión es $\sim 1,386 \lg n$ si asumimos que el pivote es elegido aleatoriamente. Esto último significa que todos los elementos tienen la misma probabilidad de ser elegidos como pivotes. Dada esa profundidad promedio, el costo promedio debería ser $\sim 1,386n \lg n$ comparaciones (de forma similar a MergeSort, en cada nivel de la recursión se hacen $\sim n$ comparaciones). Esto significa que QuickSort es óptimo en caso promedio, y lo demostramos formalmente a continuación.

Teorema 11.2.1 *Dado un arreglo aleatorio $A[1..n]$ de elementos sobre los que se ha definido una relación de orden total, el algoritmo QuickSort permite ordenar A realizando en promedio $1,386n \lg n - n$ comparaciones.*

Demostración. Asumimos un arreglo de entrada aleatorio, esto es, las $n!$ distintas permutaciones de n elementos tienen la misma probabilidad de ocurrir. Esto también significa que cualquier pivote que escojamos será aleatorio. Para simplificar, asumimos que se toma el primer elemento del arreglo como pivote. Denotaremos con $T(n)$ el tiempo de ejecución promedio para ordenar el arreglo $A[1..n]$ usando QuickSort. Para calcular el tiempo de ejecución promedio, debemos considerar que en la raíz del árbol de recursión se realizan $n - 1$ comparaciones para particionar el arreglo, y luego tenemos que considerar los n posibles casos en que el pivote particiona el arreglo:

- A_{\min} tiene 0 elementos y A_{\max} tiene $n - 1$ elementos, lo cual tomaría tiempo $T(0) + T(n - 1)$.
- A_{\min} tiene 1 elemento y A_{\max} tiene $n - 2$ elementos, lo que tomaría tiempo $T(1) + T(n - 2)$.
- \vdots
- A_{\min} tiene $n - 1$ elementos y A_{\max} tiene 0 elementos, lo cual tomaría tiempo $T(n - 1) + T(0)$.

Note la simetría en esos casos: por ejemplo, el primer y último casos son equivalentes. Para calcular el tiempo promedio, debemos sumar el tiempo de todos esos n posibles casos, y luego dividir entre n . Recordando que $T(0) = T(1) = 0$, tenemos la siguiente ecuación de recurrencia:

$$T(n) = \begin{cases} n - 1 + \frac{1}{n} \left(\sum_{k=0}^{n-1} T(k) + T(n - 1 - k) \right), & n > 1; \\ 0, & n = 0; \\ 0, & n = 1. \end{cases} \quad (11.2)$$

Aprovechando la simetría de los casos de partición, la parte recurrente

puede escribirse de forma más simple como:

$$T(n) = n - 1 + \frac{2}{n} \sum_{k=0}^{n-1} T(k).$$

Para poder manipular la suma recurrente, multiplicamos por n a ambos lados y obtenemos:

$$nT(n) = n^2 - n + 2 \sum_{k=0}^{n-1} T(k).$$

Esta recurrencia es similar a la de la Ecuación (6.7) (página 130), usada en el análisis de costo promedio de búsqueda en ABBs, por lo que vamos a resolverla de forma similar, considerando el término $n - 1$ de la recurrencia:

$$(n-1)T(n-1) = (n-1)^2 - n + 1 + 2 \sum_{k=0}^{n-2} T(k),$$

luego restando las dos últimas ecuaciones a ambos lados de la igualdad, para llegar a:

$$nT(n) - (n-1)T(n-1) = 2n - 1 + 2T(n-1).$$

Note cómo pudimos eliminar la suma recurrente, manteniendo únicamente a $T(n-1)$, el último término de la suma de la primera ecuación. Ahora sólo resta manipular esta ecuación para llegar al resultado. En particular, despejamos $T(n)$ y obtenemos

$$T(n) = \frac{n+1}{n}T(n-1) + 2 - \frac{1}{n} \leq \frac{n+1}{n}T(n-1) + 2.$$

Si reemplazamos la parte recurrente por su definición, repetidamente, obtenemos:

$$\begin{aligned} T(n) &= \frac{n+1}{n}T(n-1) + 2 \\ &= \frac{n+1}{n} \left(\frac{n}{n-1}T(n-2) + 2 \right) + 2 = \frac{n+1}{n-1}T(n-2) + 2\frac{n+1}{n} + 2 \\ &= \frac{n+1}{n-2}T(n-3) + 2 \left(\frac{n+1}{n-1} + \frac{n+1}{n} + 1 \right) \\ &\vdots \\ &= \frac{n+1}{n-(i-1)}T(n-i) + 2(n+1) \sum_{k=0}^{i-2} \frac{1}{n-k} + 2. \end{aligned}$$

Esto se detiene cuando $n-i = 1$, que corresponde al caso base $T(1) = 0$. Por lo tanto, tenemos que $i = n-1$, y por lo tanto:

$$T(n) = 2(n+1) \sum_{k=0}^{n-3} \frac{1}{n-k} = 2(n+1) \sum_{k=3}^n \frac{1}{k}.$$

Note que $\sum_{k=3}^n \frac{1}{k} = H_n - \frac{3}{2}$, en donde $H_n = \sum_{k=1}^n \frac{1}{k}$ es el n -ésimo número armónico. Dado que $\ln(n+1) \leq H_n \leq \ln(n) + 1$, tenemos

que:

$$T(n) = 2(n+1) \left(\ln(n) - \frac{1}{2} \right) + 2 \approx 1,386n \lg n - n.$$

■

Para concluir, al parecer la estrategia usada por QuickSort, que tiene una etapa de división más costosa que la de MergeSort, le permite obtener un tiempo total de ejecución que es más eficiente en la práctica, ya que posteriormente no necesita la etapa de combinación (que es la más cara de MergeSort). A todo esto le ayuda el hecho de que, en promedio, el árbol de recursión es balanceado. Podemos concluir que, en promedio, QuickSort requiere espacio adicional logarítmico, lo que corresponde a la profundidad de la recursión en promedio.

Ejercicios

1. Dado el arreglo con los siguientes números enteros:

(8, 9, 7, 9, 3, 2, 3, 8, 4, 6),

ejecutar los algoritmos de ordenamiento mencionados más abajo, tal como se implementaron en clases, mostrando el arreglo resultante en cada paso.

- a) MergeSort.
 - b) QuickSort (asumiendo que el pivote es el elemento que está en la mitad del arreglo en cada caso).
2. Recuerde que un algoritmo de ordenamiento se llama *estable* si el orden original de los valores repetidos del arreglo se preservan luego de ordenarse. De los algoritmos de ordenamiento estudiados basados en dividir y conquistar, MergeSort y QuickSort, ¿Cuáles son estables, y cuáles no? Justifique. En los casos en que el algoritmo podría hacerse estable con un cambio menor en su implementación, explique ese cambio.
 3. Suponga un conjunto de elementos enteros. Cualquier algoritmo de ordenamiento puede hacerse estable si alteramos esos valores enteros de manera que los valores duplicados se hagan únicos. Es decir, modificar los valores originales de manera que la primera ocurrencia de un valor original repetido sea menor que la segunda ocurrencia de ese mismo, el cual a su vez sea menor que la tercera, y así siguiendo. En el peor caso, es posible que los n valores de entrada tengan el mismo valor. Diseñar un algoritmo para modificar los valores del arreglo de manera que los valores resultantes nos dan el mismo ordenamiento que los valores originales, el resultado sea estable (en el sentido que valores repetidos conservan su orden original) y el proceso de alterar los valores tome tiempo $\Theta(n)$ y usa sólo una cantidad $\Theta(1)$ de espacio adicional (es decir, sea un algoritmo in-place).
 4. Dada la implementación de QuickSort vista en clases, que toma como pivote al elemento que está en la mitad del arreglo a ordenar, mostrar una permutación de $1, \dots, 8$ que produce el peor caso para QuickSort.

5. Asuma que L es un arreglo, que $\text{length}(L)$ retorna el número de elementos del arreglo, y que $\text{qsort}(L, i, j)$ ordena los elementos de L desde la posición i a la posición j usando el algoritmo QuickSort. ¿Cuál es el tiempo de *caso promedio* para cada uno de los siguientes fragmentos de código?
 - a)

```
for (i=0; i<length(L); i++)
    qsort(L, 0, i);
```
 - b)

```
for (i=0; i<length(L); i++)
    qsort(L, 0, length(L)-1);
```
6. Imagine un conjunto de n tornillos (todos de distinto tamaño) almacenados en un cajón. Imagine otro cajón en donde se han almacenado las n tuercas correspondientes a los tornillos (cada tornillo tiene su tuerca). Se quiere encontrar la tuerca correspondiente a cada tornillo. Las diferencias de tamaño entre cada tuerca o cada tornillo puede ser muy pequeña para detectarlas a simple vista, así que no podemos resolver el problema mediante comparación de tuercas entre sí o tornillos entre sí. La única posibilidad para llevar a cabo las comparaciones entre tuercas y tornillos es intentar atornillar un tornillo en una tuerca. Esta operación nos permite determinar si el tornillo corresponde a la tuerca (es decir, ambos son del mismo tamaño), si el tornillo es más pequeño que la tuerca, o que el tornillo es más grande que la tuerca. Diseñe un algoritmo eficiente, basado en dividir y conquistar, para encontrar el tornillo que corresponde a cada tuerca. Analice su algoritmo, dando cotas de peor caso y de caso promedio.
7. Sea $A[0..n-1]$ un arreglo de n números reales. Un par $(A[i], A[j])$ se llama una *inversión* si $i < j$ y $A[i] > A[j]$. Diseñe un algoritmo de tiempo $\Theta(n \log n)$ para contar el número de inversiones en el arreglo A .
Hint: piense en un procedimiento similar al de MergeSort.
8. Sea $A[1..n]$ un arreglo tal que las primeras $n - \sqrt{n}$ posiciones están ordenadas, mientras que los \sqrt{n} elementos restantes no están necesariamente ordenados. Diseñar un algoritmo que ordene el arreglo usando $o(n \log n)$ comparaciones. Muestre el tiempo de ejecución de su algoritmo.
9. Diseñe un algoritmo que permita reorganizar los elementos de un arreglo $A[1..n]$ de números reales, de manera que todos los números negativos en A preceden a todos sus números positivos. El tiempo de ejecución de su algoritmo debe ser $\Theta(n)$. Su algoritmo debe ser in-place: sólo se puede usar una cantidad $\Theta(1)$ de espacio extra.

11.3. El Teorema Maestro

Tal como lo hemos visto hasta este momento, los algoritmos del tipo Dividir y Conquistar resuelven un problema de tamaño n resolviendo a subproblemas de tamaño n/b cada uno (asumiendo que todos los subproblemas son del mismo tamaño, algo que no siempre se cumple, tal como vimos para QuickSort), y combinando esas respuestas en tiempo, digamos, $O(n^d)$, para constantes $a, b > 0$, $d \geq 0$. Por ejemplo, para

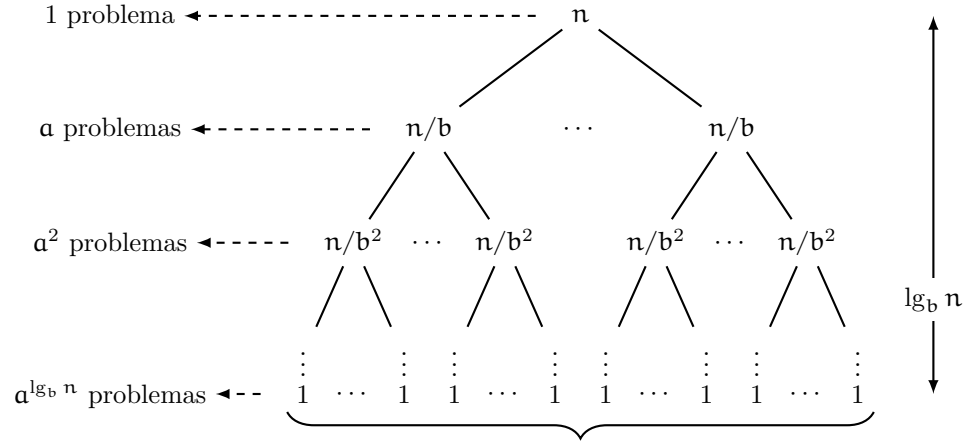


Figura 11.4: Árbol de recursión para un algoritmo dividir y conquistar que divide el problema original en a subproblemas de tamaño n/b cada uno. Los nodos del árbol muestran el tamaño del problema a resolver en cada caso.

MergeSort se tiene $a = 2$, $b = 2$, y $d = 1$. Es normal, entonces, que el tiempo de ejecución de estos algoritmos sea expresado por ecuaciones de recurrencia de la forma:

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + O(n^d), & n > 1; \\ 1, & n = 1. \end{cases} \quad (11.3)$$

Tal como lo hemos estudiado, esto genera un árbol de recursión que tendrá una cierta anchura y altura. En particular, la anchura está dada por la constante a , la cantidad de subproblemas que se generan, mientras que la altura depende de b (mientras más grande sea b , más bajo será el árbol). La Figura 11.4 muestra uno de dichos árboles de recursión.

Dado este patrón general exhibido por los algoritmos del tipo Dividir y Conquistar, mostraremos ahora un resultado que permite acotar asintóticamente las ecuaciones de recurrencia de la forma de la Ecuación (11.3). Este resultado es conocido como el *Teorema Maestro* (Master Theorem, en inglés). La demostración se hará estudiando el árbol de recursión generado por el algoritmo. Note que el árbol de la Figura 11.4 tiene $a^0 = 1$ problema en el nivel 0, a^1 subproblemas en el nivel 1, a^2 subproblemas en el nivel 2, y así siguiendo. En general, a^k subproblemas en el nivel k . Respecto al tamaño de los subproblemas, estos son de tamaño n en el nivel 0 del árbol, n/b en el nivel 1, n/b^2 en el nivel 2, y así siguiendo. El algoritmo continuará dividiendo en subproblemas hasta un nivel i en el que $n/b^i = 1$ (es decir, los subproblemas se vuelven de tamaño 1, que es el caso base de la recurrencia). Despejando i , la altura del árbol es $i = \lg_b n$. La cantidad de nodos externos del árbol es, de acuerdo a lo que hemos dicho antes, $a^{\lg_b n}$. Note que $a^{\lg_b n} = n^{\lg_b a}$, por propiedades de logaritmos. Básicamente, el Teorema Maestro indica que el tiempo de ejecución $T(n)$ se puede acotar asintóticamente comparando d (el exponente de n^d , el costo de combinar soluciones) con $\lg_b a$ (el exponente de $n^{\lg_b a}$, la cantidad de nodos externos del árbol de recursión).

Teorema 11.3.1 Dado un algoritmo del tipo Dividir y Conquistar,

cuyo tiempo de ejecución está dado por la ecuación de recurrencia $T(n) = aT(\frac{n}{b}) + O(n^d)$, para constantes $a, b > 0$, $d \geq 0$, entonces

$$T(n) = \begin{cases} O(n^d), & \text{si } d > \lg_b a; \\ O(n^d \lg n), & \text{si } d = \lg_b a; \\ O(n^{\lg_b a}), & \text{si } d < \lg_b a. \end{cases}$$

Demostración. Asumamos que n es potencia de b . El k -ésimo nivel del árbol consiste de a^k subproblemas, cada uno de tamaño n/b^k (vea la Figura 11.4). El trabajo total en el nivel k es, por lo tanto:

$$a^k \times O\left(\left(\frac{n}{b^k}\right)^d\right) = O(n^d) \left(\frac{a}{b^d}\right)^k.$$

El trabajo total en todos los niveles es, entonces:

$$T(n) = \sum_{k=0}^{\lg_b n} O(n^d) \left(\frac{a}{b^d}\right)^k = O(n^d) \sum_{k=0}^{\lg_b n} \left(\frac{a}{b^d}\right)^k. \quad (11.4)$$

Esta última es una serie geométrica con ratio $\frac{a}{b^d}$, para la cual tenemos los siguientes 3 casos:

$\frac{a}{b^d} < 1$: esto implica que $d > \lg_b a$. En este caso, la serie de la Ecuación (11.4) suma valores decrecientes, por lo que prevalece $O(n^d)$, por lo tanto $T(n) = O(n^d)$.

$\frac{a}{b^d} = 1$: esto implica $d = \lg_b a$. En este caso, la serie de la Ecuación (11.4) suma

$$T(n) = O(n^d) \underbrace{[1 + \dots + 1]}_{\lg_b n + 1} = O(n^d \lg n).$$

$\frac{a}{b^d} > 1$: esto implica $d < \lg_b a$. En este caso, la serie de la Ecuación (11.4) suma valores crecientes, y el total está dado por el último término, siendo:

$$T(n) = n^d \left(\frac{a}{b^d}\right)^{\lg_b n} = n^d \frac{a^{\lg_b n}}{(b^{\lg_b n})^d} = n^d \frac{a^{\lg_b n}}{n^d} = a^{\lg_b n} = n^{\lg_b a}.$$

Esos 3 casos corresponden a los que propone el teorema, quedando entonces demostrado. ■

Observación 11.3.2 Si el tiempo de ejecución de la etapa de combinación es $\Theta(n^d)$, entonces se puede demostrar una versión del teorema con los mismos cotas, pero reemplazando $\Theta(\cdot)$ por $O(\cdot)$.

11.4. El Problema de Determinar si Existe Mayoría Absoluta

Asumamos un sistema que permite realizar votaciones, en donde n usuarios pueden votar por una de entre k opciones posibles. Al finalizar

la votación, el sistema no sólo debería informar cuál fue la opción más votada, sino que en muchos casos también es necesario saber si hubo o no mayoría absoluta (esto es, la opción ganadora recibió al menos $\lfloor n/2 \rfloor + 1$ votos). De forma abstracta, el problema puede plantearse como a continuación. Dado un arreglo $A[1..n]$ de elementos de algún tipo, no necesariamente ordenado, se quiere determinar si A posee una mayoría absoluta (o elemento mayoritario) o no: es decir, si existe un elemento que se repite al menos $\lfloor \frac{n}{2} \rfloor + 1$ veces. Por ejemplo, el arreglo de enteros $A = (2, 8, 2, 2, 1, 8, 2, 6, 2, 2)$ tiene una mayoría absoluta (el 2), mientras que $A = (2, 8, 2, 8, 1, 8, 2, 6, 2, 2)$ no tiene mayoría.

Si las alternativas a votar están representadas por elementos de tipo entero en el rango $[1..k]$, una alternativa simple es definir un arreglo de contadores $C[1..k]$. De esta manera, se recorre el arreglo A , incrementando $C[A[i]]$ por cada $i = 1, \dots, n$. Finalmente, se recorre C para determinar si alguna de las opciones es mayoría absoluta o no. El tiempo total es $\Theta(n + k)$, mientras que el espacio adicional usado es $\Theta(k)$. Sin embargo, este enfoque no funciona si las alternativas a votar no son enteros en $[1..k]$.

Para objetos sobre los que existe una relación de orden total pero no están necesariamente en el rango $[1..k]$ (por ejemplo, strings), una alternativa simple sería ordenar el arreglo A . Note que si A tiene una mayoría, éste sería el elemento $A[n/2]$ después de ordenar: si un elemento es mayoría, sus $\geq \lfloor n/2 \rfloor + 1$ ocurrencias siempre van a ocupar más de la mitad del arreglo ordenado, por lo que $A[n/2]$ siempre contendrá la mayoría en caso de existir. Para determinar si $A[n/2]$ es mayoría o no, sólo debemos recorrer el arreglo contando la cantidad de ocurrencias de éste. Si usamos MergeSort para ordenar A , el tiempo de ejecución será $\Theta(n \lg n)$, y el espacio adicional usado es $\Theta(n)$.

Una alternativa diferente es representar el arreglo de contadores C usando un arreglo asociativo, de manera tal que pueda ser indexado con objetos que no necesariamente son enteros. Por ejemplo, se podría usar una estructura de tipo map de C++, que son implementados usando árboles balanceados. Más formalmente, un arreglo asociativo es una instancia del problema de búsqueda en arreglo ordenado de tamaño k . Si usamos un AVL para representar este arreglo asociativo, en donde los nodos se ordenan de acuerdo a las k alternativas, y cada una tiene un contador asociado inicializado en 0. Nuevamente hay que recorrer A , y por cada $A[i]$ hay que incrementar el contador adecuado en el árbol, lo que implica una búsqueda que toma tiempo $O(\lg k)$ (recuerde la Sección ??). Esto permite resolver el problema en tiempo $O(n \lg k)$, usando espacio adicional $\Theta(k)$. Sin embargo, esta solución asume el modelo de comparaciones y la existencia de un orden total sobre los elementos de A .

Todas las soluciones propuestas tiene las siguientes desventajas:

- No son in-place: se requiere espacio adicional no constante, ya sea para el arreglo C o para ordenar usando MergeSort.
- Asume el modelo de comparaciones, y la existencia de una relación de orden total sobre los elementos del arreglo. En algunas aplicaciones, podríamos tener un conjunto parcialmente ordenado, es decir, un conjunto en el que los elementos pueden compararse entre sí por $=$ (y \neq), pero no existe un orden total definido sobre

ellos (es decir, no todos los pares de elementos se pueden comparar por $<$ y $>$). Un ejemplo es que cada elemento de A sea un conjunto de números enteros. No es fácil ordenar un conjunto de conjuntos: un posible orden podría ser el de inclusión de conjuntos, es decir, un conjunto $X \leq Y$ si $X \subseteq Y$. Note que si X no es subconjunto de Y , no hay un orden definido entre ellos, por lo que son incomparables, y el arreglo no se podría ordenar.

Estudiemos a continuación una solución Dividir y Conquistar que usa espacio adicional $\Theta(\lg n)$ y que sólo necesita comparar elementos por $=$, por lo tanto no necesitando una definición de orden total. La propiedad clave para la solución es la siguiente:

Lema 11.4.1 *Si a es una mayoría en el arreglo $A[1..n]$, entonces a es también una mayoría en al menos una de las dos mitades del arreglo, $A[1..\lfloor \frac{n}{2} \rfloor]$ o $A[\lfloor \frac{n}{2} \rfloor + 1..n]$.*

Esta propiedad puede observarse fácilmente, notando que si a es mayoría, entonces se repite al menos $\lfloor \frac{n}{2} \rfloor + 1$ veces. Eso significa que si dividimos el arreglo en dos mitades, entonces podría darse el caso en que a ocurre $\lfloor \frac{n}{4} \rfloor$ veces en una de las mitades, y $\lfloor \frac{n}{4} \rfloor + 1$ veces en la otra, lo que lo haría una mayoría en esa mitad. Note que el recíproco no es necesariamente cierto: si un elemento a es mayoría en una de las mitades del arreglo, no necesariamente será mayoría para todo el arreglo.

Usando todo esto, nuestro algoritmo funciona de la siguiente manera. Si $n = 1$, entonces el arreglo tiene, obviamente, una mayoría. Podemos asumir entonces que sabemos determinar si un arreglo de $< n$ elementos tiene una mayoría o no. Mostramos ahora cómo, en base a esta hipótesis inductiva, determinar si un arreglo de n elementos tiene o no una mayoría, para todo $n \geq 2$, completando la inducción. La idea es resolver recursivamente el problema de la mayoría en cada una de las mitades. Por hipótesis inductiva, eso resuelve el problema correctamente en las mitades, ya que esas mitades tienen $\frac{n}{2}$ elementos. Digamos que esas invocaciones recursivas producen, respectivamente, dos mayorías, m_1 y m_2 . Esas son las dos candidatas a ser mayoría absoluta del arreglo. El siguiente paso para producir la respuesta es contar cuántas veces ocurren m_1 y m_2 : recuerde que el recíproco de la Propiedad 11.4.1 no necesariamente se cumple. Si uno de esos valores ocurre al menos $\lfloor \frac{n}{2} \rfloor + 1$ veces, entonces se reporta como mayoría. En otro caso, se indica que no existe mayoría (por ejemplo, devolviendo algún valor inválido). El Algoritmo 31 formaliza este proceso. La función CONTAR es formalizada en el Algoritmo 32. La Propiedad 11.4.1 es la que garantiza que, de existir una mayoría en A , será encontrada por al menos una de las dos invocaciones recursivas que hace el algoritmo, asegurando que el algoritmo es correcto.

Respecto al tiempo de ejecución del algoritmo, note que CONTAR tiene tiempo de ejecución $O(n)$ (medido en cantidad de comparaciones de igualdad). Por lo tanto, el tiempo de ejecución del Algoritmo 31 es:

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + O(n), & n \geq 2; \\ 1, & n = 1. \end{cases}$$

Algoritmo 31: MAYORÍA($A[1..n]$)

```

1 if  $n = 1$  then
2   | return  $A[1]$ 
3 else
4   |  $m_1 \leftarrow \text{MAYORÍA}(A[1..\lfloor \frac{n}{2} \rfloor])$ 
5   |  $m_2 \leftarrow \text{MAYORÍA}(A[\lfloor \frac{n}{2} \rfloor + 1..n])$ 
6   |  $c_1 \leftarrow \text{máx}\{\text{CONTAR}(A[1..n], m_1), 0\}$ 
7   |  $c_2 \leftarrow \text{máx}\{\text{CONTAR}(A[1..n], m_2), 0\}$ 
8   | if  $c_1 \geq \lfloor \frac{n}{2} \rfloor + 1$  then
9     | | return  $m_1$ 
10  | else
11    | if  $c_2 \geq \lfloor \frac{n}{2} \rfloor + 1$  then
12      | | return  $m_2$ 
13    | else
14      | | return NoHayMayoría
15    | end
16  | end
17 end

```

Algoritmo 32: CONTAR($A[1..n], m$)

```

1 if  $m = \text{NoHayMayoría}$  then
2   | return  $-\infty$ 
3 else
4   |  $c \leftarrow 0$ 
5   | for  $i \leftarrow 1$  to  $n$  do
6     | if  $A[i] = m$  then
7       | |  $c++$ 
8     | end
9   | end
10  | return  $c$ 
11 end

```

Por el Teorema Maestro, esto significa que $T(n) \in O(n \lg n)$. Esto es, el mismo tiempo que al ordenar el arreglo. El espacio adicional usado esta vez es $\Theta(\lg n)$, que equivale a la altura del árbol de recursión. El algoritmo además compara elementos del arreglo usando sólo = (ver línea 6 del Algoritmo 32).

Para finalizar: ¿Se podrá resolver este problema en menor tiempo? En el Capítulo 12 mostraremos un algoritmo de tiempo $\Theta(n)$ para este problema, aunque asumiendo el modelo de comparaciones.

11.5. El Problema de Intercalar Elementos de dos Conjuntos

Dadas $A = \langle a_1, a_2, \dots, a_n \rangle$ y $B = \langle b_1, b_2, \dots, b_n \rangle$, dos secuencias de elementos de algún tipo, representadas por un arreglo de la forma

$$C[1..2n] = (a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_n),$$

se quiere obtener el arreglo:

$$(a_1, b_1, a_2, b_2, \dots, a_n, b_n),$$

que intercala los elementos de ambas secuencias. Por ejemplo, para el arreglo de elementos enteros $C = (5, -7, 3, 1, 9, 12, -3, 6)$, la salida debería ser $(5, 9, -7, 12, 3, -3, 1, 6)$.

Tal como para el problema de determinar una mayoría, la solución es simple si usamos un arreglo temporal. Sin embargo, eso implica un espacio adicional $\Theta(n)$. Para hacer el problema más interesante, estudiaremos una solución que emplea espacio adicional $\Theta(\lg n)$.

El caso base de la recursión es para $n = 1$: para un arreglo de la forma (a_i, b_i) , el problema está trivialmente resuelto. Asumimos entonces que el algoritmo es correcto para $\frac{n}{2}$. A partir de esto, mostramos cómo resolver el problema para un arreglo de tamaño n , para $n > 1$.

La idea es dividir el arreglo de entrada C en dos mitades, y proceder recursivamente en cada mitad. Sin embargo, hay un detalle importante a resolver: una instancia (i.e., entrada) de este problema es un arreglo que en la primera mitad contiene elementos a_i , mientras que la segunda mitad contiene elementos b_i . Entonces, simplemente dividir el arreglo en dos y proceder recursivamente sobre esas mitades no es suficiente, ya que esas no corresponden a instancias del problema: note que contienen sólo un tipo de elementos. Esto significa que antes de proceder recursivamente, debemos preparar esas instancias de tamaño $\frac{n}{2}$ para que contengan mitad de a_i s y mitad de b_i s. Note que en el resultado final, la primera mitad de la secuencia contiene los elementos $(a_1, b_1, \dots, a_{\frac{n}{2}}, b_{\frac{n}{2}})$. Por otro lado, la segunda mitad contendrá los elementos $(a_{\frac{n}{2}+1}, b_{\frac{n}{2}+1}, \dots, a_n, b_n)$. Eso significa que las entradas para las llamadas recursivas deben ser $(a_1, a_2, \dots, a_{\frac{n}{2}}, b_1, b_2, \dots, b_{\frac{n}{2}})$ y $(a_{\frac{n}{2}+1}, a_{\frac{n}{2}+2}, \dots, a_n, b_{\frac{n}{2}+1}, b_{\frac{n}{2}+2}, \dots, b_n)$. De esa forma, tienen la forma que el algoritmo espera, con cada mitad conteniendo elementos de cada tipo. El Algoritmo 33 resume este proceso. Note que, al igual que

Algoritmo	33:	INTERCALAR($C[1..2n]$	=
$(a_1, \dots, a_n, b_1, \dots, b_n)$			
1	if $n = 1$ then		
2	return		
3	else		
4	$i \leftarrow \frac{n}{2}$		
5	$j \leftarrow n + 1$		
6	while $i \leq n$ do		
7	SWAP($C[i], C[j]$)		
8	$i \leftarrow i + 1$		
9	$j \leftarrow j + 1$		
10	end		
11	INTERCALAR($C[1..n]$)		
12	INTERCALAR($C[n + 1..2n]$)		
13	end		

QuickSort, el trabajo real lo hace la etapa de dividir. De hecho, luego de las invocaciones recursivas, no es necesario combinar las respuestas. El tiempo de ejecución del algoritmo, medido en cantidad de ciclos que

ejecuta es

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \frac{n}{2}, & n > 1; \\ 0, & n = 1. \end{cases}$$

Usando el Teorema Maestro, podemos concluir que $T(n) \in \Theta(n \lg n)$.

La resolución de este problema ilustra un concepto importante: en un algoritmo recursivo, las instancias recursivas deben ser instancias del mismo problema. En otras palabras, los argumentos usados en las invocaciones recursivas deben ser consistentes con las entradas que asume el algoritmo.

Ejercicios

1. Dado un arreglo $A[1..n]$ de n strings de largo máximo k , use la técnica Dividir y Conquistar para diseñar un algoritmo que permita encontrar de forma eficiente el largo del prefijo común más largo entre dichos strings.
Por ejemplo, si $A[4] = \{\text{"atroz"}, \text{"atribular"}, \text{"atril"}, \text{"atributo"}\}$, el prefijo común más largo es **"atr"**. Escriba su solución usando pseudo código. Su algoritmo no puede modificar el arreglo de entrada A , y debe usar una cantidad $\Theta(\lg n)$ de espacio adicional. ¿Cuál es el tiempo de ejecución de su algoritmo?

11.6. El Problema de Multiplicar Números Enteros

Estudiamos a continuación un problema fundamental y conocido por todos: el de multiplicar dos números enteros positivos, x e y , cada uno representado con n bits. Ésta es una operación aritmética fundamental, enseñada desde la escuela básica, y es incluida como instrucción en la mayoría de los procesadores. Esto significa que en esos procesadores es posible multiplicar en tiempo constante dos números enteros de una cantidad limitada de bits, típicamente $n = 32$ o $n = 64$ bits. Sin embargo, el problema se vuelve no trivial cuando n es grande, lo que es típico en aplicaciones como criptografía y el cálculo de constantes clásicas como π o e ^{6 7}. En esos casos, se necesita implementar la aritmética para números de miles de millones de dígitos decimales. Explicamos a continuación cómo resolver multiplicaciones de forma eficiente cuando la cantidad de dígitos es muy grande.

El algoritmo clásico para multiplicar enteros (el que se enseña en la escuela básica), consiste en multiplicar cada uno de los dígitos de uno de los números por cada uno de los dígitos del otro número, y luego

⁶ D. V. Chudnovsky y G. V. Chudnovsky. The computation of classical constants. En *Proceedings of the National Academy of Sciences of the United States of America*. Vol. 86 21, 8178–82. 1989.

⁷ D. V. Chudnovsky y G. V. Chudnovsky. Classical Constants and Functions: Computations and Continued Fraction Expansions. In: Chudnovsky D.V., Chudnovsky G.V., Cohn H., Nathanson M.B. (eds) *Number Theory*. Springer, New York, NY. 1991.

sumar todos esos resultados. Por ejemplo, para multiplicar 45×53 , cuya representación en binario es 101101 y 110101, hacemos:

$$\begin{array}{r}
 \begin{array}{cccccccc}
 & & & & & & 1 & 0 & 1 & 1 & 0 & 1 \\
 & & & & & & \times & 1 & 1 & 0 & 1 & 0 & 1 \\
 \hline
 & & & & & & 1 & 0 & 1 & 1 & 0 & 1 \\
 & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & 1 & 0 & 1 & 1 & 0 & 1 & & & \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & & & \\
 & & & 1 & 0 & 1 & 1 & 0 & 1 & & \\
 + & 1 & 0 & 1 & 1 & 0 & 1 & & & & \\
 \hline
 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1
 \end{array}
 \end{array}$$

El resultado son $\Theta(n^2)$ multiplicaciones y sumas de dígitos, lo que hace un tiempo total $\Theta(n^2)$. Para n grande (por ejemplo, piense en un número de más de 10 mil bits), el tiempo se vuelve no despreciable.

Estudiamos a continuación un algoritmo más eficiente para multiplicar números enteros, descubierto por el matemático ruso Anatoly Karatsuba en 1960. Éste fue el primer algoritmo de multiplicación con tiempo de ejecución $\mathcal{O}(n^2)$, indicando que multiplicar dos números era un problema más fácil (en términos de complejidad) que lo que se pensaba⁸. El algoritmo de Karatsuba se base en el siguiente truco de Gauss para multiplicar dos números complejos:

$$(a + bi) \times (c + di) = ac - bd + (ad + bc)i.$$

En principio, esa multiplicación de números complejos parece involucrar 4 multiplicaciones de números reales: ac , bd , ad , y bc . Si embargo, Gauss notó lo siguiente:

$$ad + bc = (a + b)(c + d) - ac - bd.$$

Por lo tanto, si $p_1 = ac$, $p_2 = bd$, entonces

$$(a + bi) \times (c + di) = p_1 - p_2 + ((a + b)(c + d) - p_1 - p_2)i,$$

que necesita sólo 3 multiplicaciones: ac , bd , y $(a + b)(c + d)$. Aunque no parece mucho *per se*, cuando apliquemos esta regla recursivamente lograremos reducir el tiempo de ejecución necesario para multiplicar enteros.

Denotemos con $x[n..1]$ y $y[n..1]$ a los dos números que queremos multiplicar. Aquí, $x[n]$ e $y[n]$ son los bits más significativos de ambos números, mientras que $x[1]$ e $y[1]$ son los bits menos significativos. Vamos a denotar con $x_S = x[n.. \frac{n}{2} + 1]$ y $x_I = x[\frac{n}{2}..1]$, a la mitad superior (S) e inferior (I) de x , respectivamente. Similarmente, definimos $y_S = y[n.. \frac{n}{2} + 1]$ e $y_I = y[\frac{n}{2}..1]$. Esto significa que:

$$x[n..1] = \underbrace{x[n]x[n-1] \cdots x\left[\frac{n}{2} + 1\right]}_{x_S} \underbrace{x\left[\frac{n}{2}\right] x\left[\frac{n}{2} - 1\right] \cdots x[1]}_{x_I},$$

⁸ ¡Uno debería enterarse antes de estas cosas!

y

$$y[n..1] = \underbrace{y[n]y[n-1] \cdots y\left[\frac{n}{2}+1\right]}_{y_S} \underbrace{y\left[\frac{n}{2}\right]y\left[\frac{n}{2}-1\right] \cdots y[1]}_{y_I}.$$

Por ejemplo, si $x = 101100$ e $y = 110101$, entonces:

$$x_S = 101 \quad y \quad x_I = 100,$$

y

$$y_S = 110 \quad y \quad y_I = 101.$$

Note que

$$x = 2^{n/2}x_S + x_I, \quad (11.5)$$

y

$$y = 2^{n/2}y_S + y_I. \quad (11.6)$$

Para el ejemplo anterior, $x = 2^3x_S + x_I = 101000 + 100 = 101100$, mientras que $y = 2^3y_S + y_I = 110000 + 101 = 110101$.

Note que el propósito de $2^{n/2}x_S$ es agregar $\frac{n}{2}$ ceros por la derecha a x_S . Luego, la suma con x_I se realiza entre esos $\frac{n}{2}$ ceros y los correspondientes $\frac{n}{2}$ bits de x_I , lo que permite obtener x . Recuerde que una multiplicación por una potencia 2^k puede implementarse como una operación que haga k shifts a la izquierda (agregando k ceros por la derecha).

Aprovechando las Ecuaciones (11.5) y (11.6) podemos escribir la multiplicación de la siguiente manera:

$$\begin{aligned} x \times y &= (2^{n/2}x_S + x_I) \times (2^{n/2}y_S + y_I) \\ &= 2^n x_S \times y_S + 2^{n/2}(x_S \times y_I + y_S \times x_I) + x_I \times y_I. \end{aligned}$$

Hemos transformado el problema original de multiplicar dos enteros de n bits en 4 multiplicaciones de enteros de $\frac{n}{2}$ bits, cuyos resultados son combinados posteriormente usando multiplicaciones por potencias de 2 (esto es, shifts a nivel del procesador) y sumas (todo lo cual toma tiempo $\Theta(n)$) para obtener el resultado de la multiplicación original. El caso base corresponde a enteros de 1 bit, los cuales deben multiplicarse. Esto corresponde a un algoritmo Dividir y Conquistar, cuyo tiempo de ejecución es expresado por la ecuación de recurrencia:

$$T(n) = \begin{cases} 4T(\frac{n}{2}) + \Theta(n), & n > 1; \\ 1, & n = 1. \end{cases}$$

Usando el Teorema Maestro, llegamos a $T(n) \in \Theta(n^2)$, por lo que no hemos ganado nada respecto del algoritmo original. Sin embargo, si usamos el método de Gauss para multiplicar números complejos, podemos resolver la multiplicación $x_S \times y_I + y_S \times x_I$ usando una única multiplicación, de la siguiente forma:

$$x_S \times y_I + y_S \times x_I = (x_S + x_I) \times (y_S + y_I) - x_S \times y_S - x_I \times y_I.$$

Entonces, en total necesitamos calcular sólo 3 multiplicaciones recursivamente:

- $p_1 = x_S \times y_S$,
- $p_2 = x_I \times y_I$,
- $p_3 = (x_S + x_I) \times (y_S + y_I)$.

A partir de ellos podemos calcular:

$$x \times y = 2^n p_1 + 2^{n/2} (p_3 - p_1 - p_2) + p_2.$$

El Algoritmo 34 muestra el proceso completo. El tiempo total es,

Algoritmo 34: KARATSUBA($x[n..1]$, $y[n..1]$)

```

1 if  $n = 1$  then
2   | return  $x[1] \cdot y[1]$ 
3 else
4   |  $x_S \leftarrow x[n.. \frac{n}{2} + 1]$ 
5   |  $x_I \leftarrow x[\frac{n}{2}..1]$ 
6   |  $y_S \leftarrow y[n.. \frac{n}{2} + 1]$ 
7   |  $y_I \leftarrow y[\frac{n}{2}..1]$ 
8   |  $p_1 \leftarrow \text{KARATSUBA}(x_S, y_S)$ 
9   |  $p_2 \leftarrow \text{KARATSUBA}(x_I, y_I)$ 
10  |  $p_3 \leftarrow \text{KARATSUBA}(x_S + x_I, y_S + y_I)$ 
11  | return  $2^n p_1 + 2^{n/2} (p_3 - p_1 - p_2) + p_2$ 
12 end
```

entonces:

$$T(n) = \begin{cases} 3T(\frac{n}{2}) + \Theta(n), & n > 1; \\ 1, & n = 1. \end{cases}$$

Por el Teorema Maestro, tenemos $T(n) \in \Theta(n^{\lg 3}) = \Theta(n^{1.585})$, lo cual es notablemente más rápido que el algoritmo clásico.

11.7. El Problema de Multiplicar Matrices

Estudiamos ahora el problema de multiplicar dos matrices M_1 y M_2 , ambas de dimensión $n \times n$, para obtener una matriz M , también de dimensión $n \times n$. El algoritmo tradicional de multiplicación de matrices obtiene el elemento $M[i][j]$ recorriendo la fila i de M_1 y la columna j de M_2 , multiplicando los elementos correspondientes y sumando. Esto implica tiempo $\Theta(n)$ para calcular $M[i][j]$. Dado que M tiene $\Theta(n^2)$ entradas, y cada una de ellas es calculada en tiempo $\Theta(n)$, el tiempo total del algoritmo es $\Theta(n^3)$. Es difícil imaginar que uno puede mejorar este tiempo de ejecución, de hecho, todos creían que era el caso hasta 1969, cuando el matemático alemán Volker Strassen mostró que podía resolver el problema en tiempo $o(n^3)$, usando un algoritmo del tipo Dividir y Conquistar. Este problema es central para muchas aplicaciones que necesitan manipular matrices muy grandes, por lo que el descubrimiento tuvo mucha repercusión.

Para obtener un enfoque Dividir y Conquistar, la idea es dividir a las matrices M_1 y M_2 en 4 sub matrices cada una, de dimensiones $\frac{n}{2} \times \frac{n}{2}$. Es particular, digamos que M_1 es dividida en las sub matrices $A = M_1[1..\frac{n}{2}][1..\frac{n}{2}]$, $B = M_1[1..\frac{n}{2}][\frac{n}{2} + 1..n]$, $C = M_1[\frac{n}{2} + 1..n][1..\frac{n}{2}]$, y $D = M_1[\frac{n}{2} + 1..n][\frac{n}{2} + 1..n]$. De forma similar, M_2 es dividida en las sub matrices $E = M_2[1..\frac{n}{2}][1..\frac{n}{2}]$, $F = M_2[1..\frac{n}{2}][\frac{n}{2} + 1..n]$, $G = M_2[\frac{n}{2} + 1..n][1..\frac{n}{2}]$, y $H = M_2[\frac{n}{2} + 1..n][\frac{n}{2} + 1..n]$. Gráficamente, la subdivisión se ve de la siguiente manera:

$$M_1 = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array} \right] \quad \text{y} \quad M_2 = \left[\begin{array}{c|c} E & F \\ \hline G & H \end{array} \right].$$

Es fácil ver, entonces, que el problema puede resolverse con un algoritmo Dividir y Conquistar de la siguiente manera:

$$\left[\begin{array}{c|c} A & B \\ \hline C & D \end{array} \right] \times \left[\begin{array}{c|c} E & F \\ \hline G & H \end{array} \right] = \left[\begin{array}{c|c} A \times E + B \times G & A \times F + B \times H \\ \hline C \times E + D \times G & C \times F + D \times H \end{array} \right]$$

Sin embargo, implica resolver recursivamente 8 multiplicaciones de matrices de $\frac{n}{2} \times \frac{n}{2}$: $A \times E$, $B \times G$, $A \times F$, $B \times H$, $C \times E$, $D \times G$, $C \times F$, y $D \times H$. La combinación de esos subproblemas para formar el resultado implica sumas de matrices, como puede verse, lo que toma tiempo $\Theta(n^2)$. Entonces, el tiempo de ejecución de este algoritmo es dado por la siguiente ecuación de recurrencia:

$$T(n) = \begin{cases} 8T(\frac{n}{2}) + \Theta(n^2), & n > 1; \\ 1, & n = 1. \end{cases}$$

Desafortunadamente, el Teorema Maestro nos dice que $T(n) \in \Theta(n^3)$, el mismo tiempo de ejecución del algoritmo tradicional.

Strassen resolvió este problema reduciendo la cantidad de subproblemas recursivos a 7, basándose en las siguientes matrices de dimensión $\frac{n}{2} \times \frac{n}{2}$:

- $P_1 = A \times (F - H)$,
- $P_2 = (A + B) \times H$,
- $P_3 = (C + D) \times E$,
- $P_4 = D \times (G - E)$,
- $P_5 = (A + D) \times (E + H)$,
- $P_6 = (B - D) \times (G + H)$, y
- $P_7 = (A - C) \times (E + F)$.

Luego, se tiene

$$M_1 \times M_2 = \left[\begin{array}{c|c} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ \hline P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{array} \right].$$

El Algoritmo 35 muestra el procedimiento explicado. El tiempo de ejecución del algoritmo, medido en cantidad de operaciones aritméticas realizadas sobre escalares corresponde a la ecuación de recurrencia:

$$T(n) = \begin{cases} 7T(\frac{n}{2}) + \Theta(n^2), & n > 1; \\ 1, & n = 1. \end{cases}$$

Algoritmo 35: STRASSEN($M_1[1..n][1..n], M_2[1..n][1..n]$)

```

1 if  $n = 1$  then
2   return  $M_1[1][1] \cdot M_2[1][1]$ 
3 else
4   Subdividir  $M_1 = \left[ \begin{array}{c|c} A & B \\ \hline C & D \end{array} \right]$ , siendo A, B, C, y D matrices
      de  $\frac{n}{2} \times \frac{n}{2}$ 
5   Subdividir  $M_2 = \left[ \begin{array}{c|c} E & F \\ \hline G & H \end{array} \right]$ , siendo E, F, G, y H matrices de
       $\frac{n}{2} \times \frac{n}{2}$ 
6    $P_1 \leftarrow \text{STRASSEN}(A, F - H)$ 
7    $P_2 \leftarrow \text{STRASSEN}(A + B, H)$ 
8    $P_3 \leftarrow \text{STRASSEN}(C + D, E)$ 
9    $P_4 \leftarrow \text{STRASSEN}(D, G - E)$ 
10   $P_5 \leftarrow \text{STRASSEN}(A + D, E + H)$ 
11   $P_6 \leftarrow \text{STRASSEN}(B - D, G + H)$ 
12   $P_7 \leftarrow \text{STRASSEN}(A - C, E + F)$ 
13  return  $\left[ \begin{array}{c|c} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ \hline P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{array} \right]$ 
14 end

```

Por el Teorema Maestro, $T(n) \in \Theta(n^{\lg_2 7}) = \Theta(n^{2,8073})$, notablemente mejor que $\Theta(n^3)$ para $n > 100$.

Ejercicios

1. Escriba pseudo código (similar a lenguaje C) para el algoritmo de multiplicación de Strassen visto en clases.
2. Suponga que quiere multiplicar dos matrices X e Y, de dimensiones $kn \times n$ y $n \times kn$, respectivamente. Escriba pseudocódigo para llevar a cabo dicha multiplicación, usando el algoritmo de Strassen como subrutina. ¿Cuál es el tiempo de ejecución de su algoritmo?
3. Responda a la misma pregunta anterior, pero ahora asumiendo que las matrices tienen dimensión $n \times kn$ y $kn \times n$, respectivamente.
4. Se quiere diseñar un algoritmo para multiplicar matrices de $n \times n$ que sea asintóticamente más rápido que el algoritmo de Strassen. El algoritmo se basará en el enfoque dividir y conquistar, dividiendo cada matriz en submatrices de tamaño $\frac{n}{4} \times \frac{n}{4}$, mientras que las etapas divide y combine tomarán en conjunto tiempo $\Theta(n^2)$. ¿Cuántos subproblemas debe crear el algoritmo para ser asintóticamente más rápido que el algoritmo de Strassen? Es decir, si el algoritmo crea a subproblemas, ¿Cuál es el valor entero a más grande para el cual el nuevo algoritmo es más rápido que el de Strassen?

11.8. Convex Hull de un Conjunto de Puntos

Dado un conjunto $P = \{p_1, \dots, p_n\}$ de n puntos en el plano, el *convex hull* (cierre convexo, o cápsula convexa) $\mathcal{CH}(P)$ es el conjunto convexo

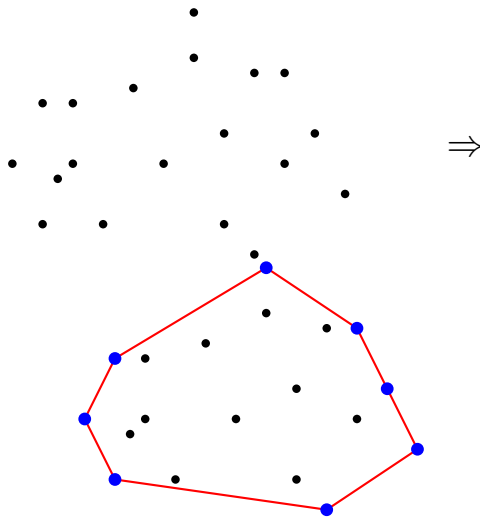


Figura 11.5: Un conjunto de puntos en el plano (izquierda) y su correspondiente convex hull (derecha).

de área mínima que contiene a P . Un conjunto de infinitos puntos S en el plano es convexo si para cualquier par de puntos $p, q \in S$, el segmento \overline{pq} está completamente contenido dentro de S . Dado que hay infinitos conjuntos convexos que contienen a P , encontrar aquel que tenga área mínima parece ser un problema complejo. Sin embargo, veremos que no es tan complejo como parece.

Imagine que los puntos en el plano son clavos sobre una madera, de manera que sus cabezas sobresalen (es decir, no están totalmente clavados). Imagine también una banda elástica que se expande con una mano para que contenga a todos los clavos, y luego se suelta para que el elástico se mueva hacia los clavos. Lo que ocurrirá es que el elástico se va a apoyar sobre los clavos “extremos” del conjunto, definiendo un polígono que se apoya sobre esos puntos extremos. Este hecho puede demostrarse: el conjunto convexo de área mínima que contiene a P debe ser un polígono convexo cuyos lados están formados por puntos de P . La Figura 11.5 ilustra un conjunto de puntos y su convex hull.

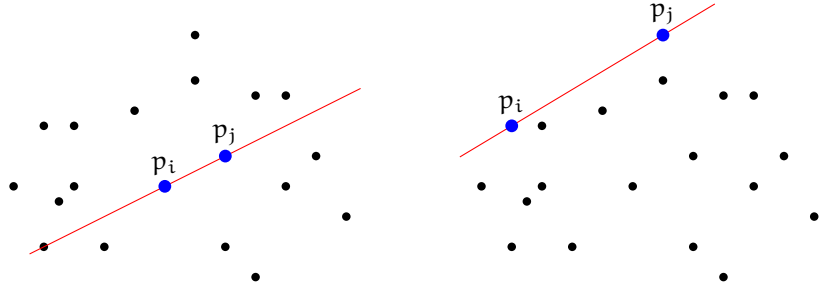
Esto último simplifica bastante el problema de calcular el convex hull, al restringir las posibles soluciones entre las que buscar: el problema consiste en determinar cuáles de los puntos de P definen el convex hull (que en el caso del diagrama que está del lado derecho en la Figura 11.5 serían los puntos azules).

La siguiente propiedad es clave para calcular $\mathcal{CH}(P)$:

Lema 11.8.1 *Dado un conjunto de puntos $P = \{p_1, \dots, p_n\}$ en el plano, el segmento $\overline{p_i p_j}$ (para cualquier par de puntos $p_i \neq p_j$) pertenece a $\mathcal{CH}(P)$ si y sólo si al considerar los dos semiplanos determinados por la recta infinita que pasa por p_i y p_j , todos los puntos de P se ubican en un mismo semiplano.*

La Figura 11.6 muestra dos pares de puntos, y la línea infinita que pasa por ellos. En la figura de la izquierda, los puntos de P quedan distribuidos en los dos semiplanos que define la línea; de esta manera, el segmento $\overline{p_i p_j}$ no pertenece al convex hull. En la figura de la derecha, por otro lado, todos los puntos se ubican en uno de los dos semiplanos, por lo que $\overline{p_i p_j}$ sí es parte del convex hull en este caso.

Figura 11.6: Un conjunto de puntos en el plano y ejemplos de puntos p_i y p_j . En el ejemplo de la izquierda, el segmento $\overline{p_i p_j}$ no pertenece al convex hull del conjunto de puntos, mientras que en el ejemplo de la derecha sí pertenece.



Esta propiedad nos permite definir el siguiente enfoque de fuerza bruta: por cada posible par de puntos p_i y p_j del conjunto, chequear que el resto de los puntos de P se ubiquen en el mismo semiplano respecto a la recta infinita que pasa por p_i y p_j . Si esto es así, agregar el segmento $\overline{p_i p_j}$ al convex hull. Ahora sólo resta saber cómo determinar de qué lado de una recta se encuentra un punto, lo cual estudiamos a continuación.

Sean $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, y $p_3 = (x_3, y_3)$. El área del triángulo formado por esos tres puntos puede calcularse con el determinante:

$$\text{Área}(\Delta(p_1, p_2, p_3)) = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1 y_2 + x_2 y_3 + x_3 y_1 - x_1 y_3 - x_2 y_1 - x_3 y_2.$$

De hecho, esta fórmula entrega el área del triángulo $\Delta(p_1, p_2, p_3)$ pero con signo:

- $\text{Área}(\Delta(p_1, p_2, p_3)) > 0$: significa que al movernos entre los puntos con dirección $p_1 \rightarrow p_2 \rightarrow p_3$, en p_2 hacemos un giro a la izquierda (o anti horario). Implica que p_3 está a la izquierda del segmento $\overrightarrow{p_1 p_2}$.
- $\text{Área}(\Delta(p_1, p_2, p_3)) < 0$: significa que al movernos entre los puntos con dirección $p_1 \rightarrow p_2 \rightarrow p_3$, en p_2 hacemos un giro a la derecha (u horario). Implica que p_3 está a la derecha del segmento $\overrightarrow{p_1 p_2}$.
- $\text{Área}(\Delta(p_1, p_2, p_3)) = 0$: por propiedades de determinantes, significa que p_1 , p_2 , y p_3 son puntos colineales.

La Figura 11.7 muestra dos puntos, p_1 y p_2 , junto a otros seis puntos. Aplicando la fórmula del determinante, se puede determinar que p_3 , p_4 ,

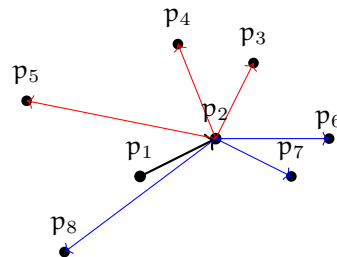


Figura 11.7: Dos puntos $p_1 = (3, 2)$ y $p_2 = (4, 2, 5)$ en el plano, junto a seis puntos $p_3 = (4, 5, 3, 5)$, $p_4 = (3, 5, 3, 75)$, $p_5 = (1, 5, 3)$, $p_6 = (5, 5, 2, 5)$, $p_7 = (5, 2)$, $p_8 = (2, 1)$.

y p_5 están a la izquierda de $\overrightarrow{p_1 p_2}$, mientras que p_6 , p_7 , y p_8 están a la derecha. Queda como ejercicio chequear esto realizando los cálculos usando la fórmula.

Volviendo a nuestro algoritmo de fuerza bruta, por cada uno de los $\Theta(n^2)$ posibles pares de puntos distintos p_i y p_j , hay que determinar si los $n - 2$ puntos restantes están del mismo lado de $\overrightarrow{p_i p_j}$. Esto significa

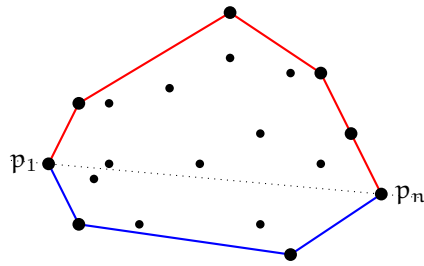


Figura 11.8: Un conjunto de puntos en el plano y su convex hull separado en la cadena poligonal superior (rojo) e inferior (azul).

que al aplicar la fórmula del determinante, se obtienen todos resultados positivos o negativos. Apenas uno de los chequeos dé un signo distinto al de los chequeados hasta el momento, se puede detener el proceso y descartar el par de puntos p_i y p_j actual. Dado que cada chequeo toma tiempo $O(1)$, el tiempo de ejecución total es $O(n^3)$.

Estudiamos ahora un algoritmo del tipo Dividir y Conquistar conocido como QuickHull, por su similitud con el algoritmo de ordenamiento QuickSort. Asuma que el conjunto de puntos ha sido ordenado respecto de la primera coordenada. Sean $p_1 = (x_1, y_1), p_2 = (x_2, y_2), \dots, p_n = (x_n, y_n)$, tal que $x_1 < x_2 < \dots < x_n$. Para comenzar, note que los puntos extremos respecto al eje x , p_1 y p_n , siempre pertenecen al convex hull de P . Cualquier convex hull puede ser separado en dos cadenas poligonales: la cadena poligonal superior y la cadena poligonal inferior. La Figura 11.8 muestra un ejemplo de dichas cadenas poligonales. El algoritmo QuickHull va a calcular dichas cadenas por separado, para finalmente concatenarlas. Esto significa que antes de comenzar, el conjunto de puntos debe ser dividido en dos: aquellos puntos que están por arriba del segmento $\overline{p_1 p_n}$, y los que están por debajo. Luego, aplicaremos QuickHull sobre ambos conjuntos, y ambos resultados serán concatenados para obtener el resultado final. Una vez notado esto, nos concentramos sólo en la cadena poligonal superior. el primer paso, es determinar cuál es el punto más lejano del segmento $\overline{p_1 p_n}$, al que llamaremos p_{\max} . La Figura 11.9 muestra dicho punto para nuestro conjunto de puntos de ejemplo. La obtención de p_{\max} puede hacerse en

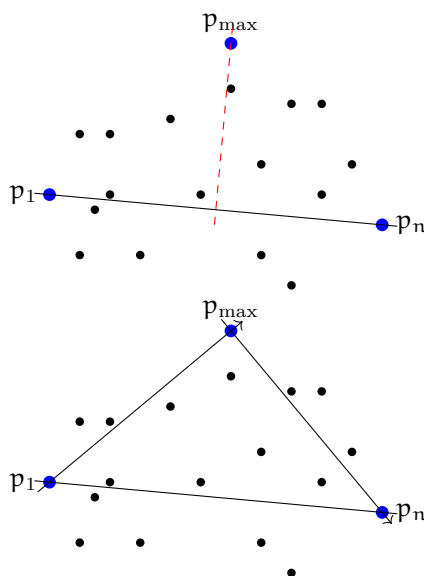


Figura 11.9: Punto extremo p_{\max} respecto del segmento $\overline{p_1 p_n}$ (lado izquierdo). El algoritmo se ejecuta recursivamente primero sobre los puntos que están a la izquierda de $\overrightarrow{p_1 p_{\max}}$, y luego sobre los puntos que están a la izquierda del segmento $\overrightarrow{p_{\max} p_n}$ (lado derecho).

tiempo lineal respecto a la cantidad de puntos del conjunto actual de puntos. Luego, el algoritmo se aplica recursivamente sobre los puntos

que quedan a la izquierda del segmento $\overrightarrow{p_1 p_{\max}}$, y luego recursivamente sobre los puntos que están a la izquierda del segmento $\overrightarrow{p_{\max} p_n}$. Esas dos cadenas obtenidas serán concatenadas para formar la cadena superior completa. Esto concluye el proceso, el cual es formalizado en el Algoritmo 36. El algoritmo representa el convex hull como una lista de

Algoritmo 36: QuickHull(puntos P , extremo p_1 , extremo p_n)

```

1 if  $n = 1$  then
2   return  $\langle \rangle$                                 // Lista vacía
3 else
4    $p_{\max} \leftarrow$  punto extremo respecto a  $\overrightarrow{p_1 p_n}$ 
5    $P_1 \leftarrow$  puntos de  $P$  a la izquierda de  $\overrightarrow{p_1 p_{\max}}$ 
6    $P_2 \leftarrow$  puntos de  $P$  a la izquierda de  $\overrightarrow{p_{\max} p_n}$ 
7   return
8   QuickHull( $P_1, p_1, p_{\max}$ ) +  $\langle p_{\max} \rangle$  + QuickHull( $P_2, p_{\max}, p_n$ )
9 end

```

puntos, de manera que dos puntos consecutivos en la lista corresponden a un lado del polígono correspondiente. De esta forma, el algoritmo retorna una lista enlazada representando el hull superior (o inferior, dependiendo con qué puntos se lo invoca en un comienzo). La operación “+” de la línea 7 corresponde a la concatenación de listas.

Respecto al tiempo de ejecución de QuickHull, lamentablemente en el peor caso, el proceso de división respecto a p_{\max} genera que todos los puntos queden a la izquierda de $\overrightarrow{p_1 p_{\max}}$ (o, equivalentemente, todos a la izquierda de $\overrightarrow{p_{\max} p_1}$). Dado que el proceso de dividir el conjunto toma tiempo lineal, y sólo reducimos el tamaño del problema en una cantidad constante, el tiempo total de ejecución total es $O(n^2)$. Aunque esto es bastante mejor que el algoritmo inicial de tiempo $O(n^3)$, todavía no es un algoritmo práctico en el peor caso. Por otro lado, se puede demostrar que el tiempo de ejecución promedio para QuickHull es $O(n)$, lo cual es mucho más eficiente (no estudiaremos este caso aquí).

Ejercicios

1. Diseñar el algoritmo MergeHull, el cual computa el cierre convexo (convex hull) de un conjunto $S = \{p_1, \dots, p_n\}$ de n puntos en el plano. Su algoritmo debe funcionar de manera eficiente y de forma similar a MergeSort. Defina claramente las etapas Divide, Conquer y Combine de su algoritmo. Proponga la ecuación de recurrencia que modele el tiempo de ejecución de su algoritmo, y use el Teorema Maestro para acotarla asintóticamente.
2. Para el algoritmo QuickHull, ¿Se puede elegir el punto “pivote” en la partición de manera aleatoria, tal como se hace en el algoritmo QuickSort? Argumente, sea cual sea su respuesta.

12.1. Introducción

Decrecer y Conquistar es una técnica de diseño de algoritmos que algunos autores consideran como un tipo particular de Dividir y Conquistar, en donde no existe la etapa final de combinar los resultados. Sin embargo, existen diferencias entre esos enfoques que justifican un estudio separado de las técnicas.

Al usar esta técnica, un algoritmo disminuye el tamaño del problema en cada paso y luego lo resuelve. La idea es explotar la relación entre la solución al problema más pequeño y el problema original: la solución del problema más pequeño es usada para construir la solución al problema original.

La técnica puede ejecutarse típicamente de dos formas:

Top down: se comienza con el problema original, el que se va haciendo más pequeño a cada paso. Generalmente lleva a una solución recursiva, que generalmente conviene implementar iterativamente (que suele ser más eficiente en la práctica).

Bottom up: se resuelve el problema comenzando por soluciones muy pequeñas a partes del problema original. En cada paso se incrementa el tamaño de la solución. Es un enfoque incremental, en donde la solución se construye paso a paso.

Las siguientes son las tres variantes principales del método:

Decrecer por una constante En cada paso, el algoritmo hace decrecer el tamaño del problema en un valor constante. Por ejemplo, calcular a^n , para una constante a , usando el siguiente algoritmo:

$$a^n = a^{n-1} \times a.$$

Note cómo el tamaño del problema original (a^n) es reducido en cada paso a problemas más pequeños (a^{n-1}). En particular, en este ejemplo se disminuye en 1 en cada paso.

Decrecer por un factor constante En cada paso, el algoritmo hace decrecer el tamaño del problema en un factor constante, por ejemplo, a la mitad del tamaño del problema original, o a un tercio, etc. Por ejemplo, calcular a^n usando el siguiente algoritmo:

$$a^n = (a^{n/2})^2.$$

Aquí, el tamaño del problema original disminuye a la mitad en cada paso.

Decrecer por un factor variable En cada paso, el algoritmo hace decrecer el tamaño del problema en un factor variable, es decir, no necesariamente decrece por el mismo factor en cada paso. Por ejemplo, en un paso se podría decrecer el problema a la mitad, mientras que en otro paso posterior se podría decrecer un tercio del tamaño actual.

12.2. El Problema de Ordenamiento: InsertionSort

En esta sección resolvemos, nuevamente, el problema de ordenamiento, estudiando un nuevo algoritmo: **InsertionSort**. Dicho algoritmo utiliza la técnica decrecer y conquistar de forma bottom-up (incremental). Sea $A[1..n]$ el arreglo a ordenar. Lo ordenaremos incrementalmente, paso a paso. En todo momento, el algoritmo mantiene el segmento izquierdo del arreglo ordenado, mientras que el segmento por ordenar es almacenado en la parte derecha del arreglo. Al comenzar el paso i del algoritmo, el segmento $A[1..i-1]$ está ordenado, para $i \geq 1$. Esta es una invariante que el algoritmo mantiene en todo momento. En este punto, el elemento $A[i]$, el primero del segmento aún sin ordenar, es insertado (y de aquí el nombre del algoritmo) en el segmento ordenado $A[1..i-1]$, en la posición adecuada para mantener la invariante. Luego de insertar este elemento, el segmento ordenado es $A[1..i]$, lo cual hace decrecer el problema (i.e., el segmento aún no ordenado) en 1 en cada paso. Si repetimos este proceso n veces, el arreglo A quedará completamente ordenado. Note el caso especial en el primer paso del algoritmo: se debe insertar $A[1]$ dentro del segmento inicial vacío, que está trivialmente ordenado al no contener elementos. En una implementación, usualmente se asume que $A[1]$ está trivialmente ordenado, y se comienza a iterar insertando desde $A[2]$ en adelante.

Dado que en cada paso el algoritmo disminuye el tamaño del problema en 1, **InsertionSort** es un algoritmo del tipo decrecer por un valor constante. El Algoritmo 37 implementa **InsertionSort**. El **while** principal

Algoritmo 37: InsertionSort($A[1..n]$)

```

1  $i \leftarrow 2$ 
2 while  $i \leq n$  do
3    $j \leftarrow i$ 
4   while  $j > 1$  and  $A[j-1] > A[j]$  do
5     SWAP( $A[j], A[j-1]$ )
6      $j \leftarrow j - 1$ 
7   end
8    $i \leftarrow i + 1$ 
9 end
```

itera $n-1$ veces. El **while** interno itera tantas veces como sea necesario hasta dejar el elemento en la posición adecuada, manteniendo el orden del segmento izquierdo $A[1..i]$ del arreglo. El algoritmo usa una cantidad constante de espacio extra, para las variables i y j . Por lo tanto, **InsertionSort** es un algoritmo de ordenamiento in-place.

Análisis de Peor Caso En el peor caso, cada elemento debe ser insertado en la primera posición del arreglo. Esto ocurre cuando los elementos están originalmente en orden inverso al deseado. Por ejemplo, $A[1..8] = \langle 8, 7, 6, 5, 4, 3, 2, 1 \rangle$. Cada iteración del **while** principal produce $i - 1$ comparaciones en el **while** interno. En total, para las $n - 1$ iteraciones del **while** principal tenemos $\sum_{i=2}^n (i - 1) \approx \frac{n^2}{2}$ comparaciones. Es decir, en el peor caso, el algoritmo realiza una cantidad cuadrática de comparaciones. Notar que la cantidad de intercambios de elementos (función SWAP) es igual, en el peor caso, a la cantidad de comparaciones realizadas.

Análisis de Mejor Caso Ya hemos dicho que este tipo de análisis no suele ser interesante en general, porque es optimista y no entrega demasiada información respecto a la complejidad del problema. Sin embargo, en este caso es conveniente hacerlo para entender mejor cómo funciona InsertionSort. Si pensamos en un arreglo de entrada que está ordenado, InsertionSort realiza una comparación en cada iteración del **while** principal, para determinar que el elemento ya está ordenado (es decir, el **while** interno no itera nunca en este caso). En total, tenemos $\sum_{i=2}^n 1 = n - 1$ comparaciones para determinar que el arreglo está ordenado.

Pregunta ¿Cómo es posible ordenar con sólo $n - 1$ comparaciones, cuando la cota inferior para el problema son $\Omega(n \lg n)$ comparaciones?

Note, además, que en el mejor caso el algoritmo no hace intercambio de elementos (es decir, nunca se invoca a SWAP).

Análisis de Caso Promedio Es interesante, además, hacer un análisis de caso promedio de InsertionSort. Aquí debemos considerar, en cada paso, todas las posibilidades de inserción del elemento actual. En el análisis de peor caso, sólo consideramos el caso en que cada elemento debe ser insertado en la posición inicial del segmento ordenado del arreglo. Por otro lado, en el análisis de mejor caso, consideramos sólo el caso en que cada elemento ya está en la posición adecuada respecto al orden. Ahora, vamos a considerar que el elemento $A[i]$ podría ser insertado en cualquiera de las posiciones dentro del segmento ordenado $A[1..i]$ ¹. Tenemos i casos posibles por cada elemento $A[i]$. Si el elemento está en la posición adecuada, entonces se hace 1 comparación; si el elemento $A[i]$ es menor que $A[i - 1]$ y mayor que $A[i - 2]$, entonces necesitamos 2 comparaciones; y así siguiendo. Asumiendo que cada uno de los casos para $A[i]$ tiene la misma probabilidad de ocurrir (es decir, probabilidad $1/i$), tenemos que el costo promedio para el elemento $A[i]$ es $\frac{1}{i} \sum_{j=1}^i j$. Considerando todos los elementos del arreglo, la cantidad promedio de comparaciones realizadas es:

$$\sum_{i=1}^{n-1} \left(\frac{1}{i} \sum_{j=1}^i j \right) = \sum_{i=1}^{n-1} \frac{1}{i} \frac{i(i+1)}{2} = \sum_{i=1}^{n-1} \frac{(i+1)}{2} \approx \frac{n^2}{4}.$$

Esto es, la cantidad promedio de comparaciones realizadas por InsertionSort es también cuadrática.

¹ Note que esto asume, implícitamente, que todas las permutaciones de los elementos del arreglo tienen la misma probabilidad de ocurrir.

Como puede verse, en general `InsertionSort` realiza $O(n^2)$ comparaciones. Esto podría hacerlo de poco interés, sin embargo este algoritmo es eficiente para casos en que el arreglo de entrada está casi ordenado. Por ejemplo, podría usarse para podar el árbol de recursión de algoritmos como `MergeSort` y `QuickSort`: en lugar de que la recursión continúe hasta arreglos de tamaño 1, el caso base podría definirse para arreglos de tamaño ≤ 15 (por ejemplo), y ordenarlos con `InsertionSort`. Esto suele producir buenos resultados en la práctica. Por ejemplo, la implementación de la función `qsort` de las bibliotecas estándar del lenguaje C implementa `QuickSort`, ordenando subarreglos suficientemente pequeños con `InsertionSort`².

12.3. El problema de Búsqueda en un Conjunto Ordenado

En esta sección consideramos el problema de búsqueda en un conjunto ordenado, el cual asumimos que está representado por un arreglo ordenado $A[1..n]$ (asumimos que la relación de orden es una función total). El problema consiste en determinar si un elemento x dado pertenece al conjunto o no. En otras palabras, se quiere determinar si existe una posición i tal $A[i] = x$. Vamos a considerar el modelo de comparaciones, es decir, sólo podemos buscar usando comparaciones entre x y elementos del arreglo. En capítulos anteriores del curso ya se determinó una cota inferior $\Omega(\lg n)$ para este problema. Estudiamos en esta sección maneras de resolver el problema, para el cual demostramos (en la Sección 7.2) la siguiente cota inferior para su complejidad de peor caso:

$$T_B(n) \geq \lceil \lg n \rceil + 1,$$

medida en cantidad de comparaciones.

Búsqueda Binaria de Tres Ramas

La búsqueda binaria es, quizás, uno de los algoritmos más conocidos en ciencias de la computación. Dado un arreglo ordenado $A[1..n]$, permite buscar un elemento x (o determinar que éste no está en A) de manera eficiente, usando el hecho de que A está ordenado.

La variante de búsqueda binaria de tres ramas es la más conocida, en general³. La idea es dividir el arreglo en tres partes: (1) el elemento del centro del arreglo, (2) la mitad izquierda del arreglo, y (3) la mitad derecha. En cada iteración, el algoritmo decide quedarse con una de esas partes, descartando las restantes. En caso de quedarse con el elemento del centro, el algoritmo se detiene, como veremos a continuación. Se compara x con el elemento $A[\frac{n}{2}]$ (el centro de A), para determinar cuál de los siguientes casos corresponde:

- Si $A[\frac{n}{2}] = x$, la búsqueda es exitosa y el algoritmo finaliza.

² Jon Louis Bentley, M. Douglas McIlroy. *Engineering a Sort Function*. Software: Practice & Experience, 23(11): 1249-1265. 1993.

³ Es probable que sea ésta la que aprendió en cursos más básicos.

- Si $x < A[\frac{n}{2}]$, entonces por el orden del arreglo, tenemos que x es menor que toda la segunda mitad de A . La búsqueda continúa en $A[1..\frac{n}{2} - 1]$.
- Si $x > A[\frac{n}{2}]$, entonces por el orden del arreglo, tenemos que x es mayor que toda la primera mitad de A . La búsqueda continúa en $A[\frac{n}{2} + 1..n]$.

Éstas son las tres ramas que dan nombre al algoritmo. El Algoritmo 38 implementa esta idea. Note que en cada iteración el algoritmo realiza

Algoritmo 38: BBINARIA3($A[1..n]$, x)

```

1  izq  $\leftarrow$  1
2  der  $\leftarrow$  n
3  while izq  $\leq$  der do
4      med  $\leftarrow$  (izq + der)/2
5      if  $x < A[\text{med}]$  then
6          | der  $\leftarrow$  med - 1
7      else
8          | if  $x > A[\text{med}]$  then
9              | izq  $\leftarrow$  med + 1
10         | else
11             | return med
12         | end
13     end
14 end
15 return n + 1    // Si alcanza este punto, x no está en A

```

dos comparaciones: $A[\text{med}] < x$ (en el **if**) y $A[\text{med}] > x$ (en el primer **else**). Aunque esto no modifica el tiempo de ejecución del algoritmo en términos asintóticos, es un hecho importante en la práctica. Veremos más adelante que si consideramos el peor caso, la cantidad total de comparaciones realizadas puede reducirse (aunque no asintóticamente, sí en términos de la constante que acompaña al tiempo de ejecución asintótico).

Tiempo de Ejecución de Peor Caso Para analizar la cantidad de comparaciones realizadas en el peor caso, pensemos el algoritmo de forma recursiva (aunque, como vimos, una búsqueda binaria se implementa típicamente de forma iterativa). Sea $T_3(n)$ el tiempo de ejecución de peor caso para una búsqueda exitosa (medido en cantidad de comparaciones) para un arreglo de tamaño n . Si $n = 1$, entonces $T_3(1) = 2$ comparaciones son suficientes para resolver el problema. Si $n > 1$, el algoritmo realiza 2 comparaciones en la mitad del arreglo, y luego continúa buscando en una de las mitades del arreglo (lo que toma tiempo $T_3(n/2)$, de acuerdo a nuestra notación). Por lo tanto, la ecuación de recurrencia que modela el tiempo de ejecución de peor caso de la búsqueda binaria de tres ramas es la siguiente:

$$T_3(n) = \begin{cases} T_3(n/2) + 2, & \text{si } n > 1. \\ 2, & \text{si } n = 1. \end{cases} \quad (12.1)$$

Usando el Teorema Maestro, tenemos que $T_3(n) \in O(\lg n)$, lo que nos indica que el algoritmo es óptimo en términos asintóticos para el

problema de búsqueda en conjuntos ordenados. Si usamos las técnicas para resolver ecuaciones de recurrencia de la Sección 4.5, obtenemos $T_3(n) = 2 \lg n + 2$, para $n \geq 1$. Ese es el tiempo de ejecución de peor caso del algoritmo, en donde son necesarias $\lg n$ iteraciones hasta encontrar el elemento buscado. Note que el algoritmo realiza el doble de comparaciones respecto a la cota inferior de $\lceil \lg n \rceil + 1$ comparaciones de la Sección 7.2. Sin embargo, aún no es claro si la cota inferior es no ajustada, o la búsqueda binaria de tres ramas realiza el doble de comparaciones respecto a la cota inferior. Sí podemos decir, por el momento, que la búsqueda binaria es asintóticamente óptima.

Análisis de Caso Promedio Para obtener la cantidad promedio de comparaciones realizadas por el algoritmo de búsqueda binaria, debemos considerar todos los casos de búsqueda. Vamos a considerar únicamente búsquedas exitosas (se deja como ejercicio el estudio de las búsquedas no exitosas), y suponemos $n = 2^k - 1$, para $k \geq 1$, para simplificar el análisis. Note, en particular, que hay 1 elemento del arreglo cuyo costo de búsqueda es 2 comparaciones (el elemento que está en el centro del arreglo). Además, hay 2 elementos del arreglo cuyo costo de búsqueda es 4 comparaciones (los elementos $A[\frac{n}{4}]$ y $A[\frac{3}{4}n]$). De la misma forma, hay 4 elementos del arreglo cuyo costo es 6 comparaciones. En general, hay 2^{i-1} elementos en el arreglo cuyo costo de búsqueda es $2i$ comparaciones, para $i = 1, \dots, \lg n$. Asumiendo que cualquier elemento del arreglo es buscado con igual probabilidad, el costo promedio es:

$$\frac{1}{n} \sum_{i=1}^{\lg n} 2^{i-1} \cdot 2i = \frac{1}{n} \sum_{i=1}^{\lg n} 2^i \cdot i = \frac{1}{n} (2n \lg n + 2 - 2n) \approx 2 \lg n - 2.$$

La suma anterior es conocida como “*lineal exponencial*”, cuya fórmula general puede verse en la Sección ?? (ver apartado “Serie Geométrica”). En conclusión, el costo promedio de búsqueda binaria exitosa es también logarítmico. En particular, se hacen en promedio 4 comparaciones menos que en el peor caso, lo que indica que en promedio el algoritmo itera una cantidad logarítmica de veces.

Búsqueda Binaria de Dos Ramas

Estudiamos a continuación una variante de búsqueda binaria que es más eficiente que la anterior en el peor caso. La idea es posponer el chequeo de igualdad que hace la búsqueda binaria de tres ramas (Algoritmo 38). En consecuencia, el único punto de detención del algoritmo es que el subarreglo sobre el que está buscando tenga tamaño 1 (recuerde que la búsqueda binaria de tres ramas chequea la igualdad en cada iteración, pudiendo detenerse apenas encuentra el elemento buscado). Al no chequear la igualdad en cada iteración, no sabremos si hemos encontrado el elemento buscado o no hasta que el algoritmo haya finalizado. Recuerde el árbol de decisión de la Figura 7.3 (lado derecho) (página 145), usado para ilustrar la demostración de la cota inferior para el problema de búsqueda en arreglo ordenado. En dicho árbol, posponíamos el chequeo de igualdad al final de la ejecución, mientras

que “iterábamos” por \leq y $>$. Éste corresponde a una búsqueda binaria de dos ramas, la cual definimos a continuación.

A diferencia de la búsqueda binaria de tres ramas, dividimos el arreglo en dos partes en cada iteración. El algoritmo compara el elemento buscado x con el elemento $A[\frac{n}{2}]$ (el centro de A), suponiendo que este elemento queda en la mitad izquierda del arreglo, y procede como a continuación:

- Si $x \leq A[\frac{n}{2}]$, entonces por el orden x es menor que toda la segunda mitad de A . La búsqueda continúa en $A[1..\frac{n}{2}]$. Notar que en este caso, se sigue buscando en la primera mitad del arreglo, sin descartar el elemento medio $A[\frac{n}{2}]$, a diferencia de la búsqueda de tres ramas. Esto es porque esta rama incluye la igualdad (la anterior comparación sólo chequeaba por $<$). Dado que esa igualdad no será chequeada aún, no podemos perder el elemento del centro, que puede ser el elemento buscado.
- Si $x > A[\frac{n}{2}]$, entonces por el orden del arreglo x es mayor que toda la primera mitad de A . La búsqueda continúa en $A[\frac{n}{2} + 1..n]$.

El Algoritmo 39 implementa este proceso. Note que las dos ramas del

Algoritmo 39: BBINARIA2($A[1..n]$, x)

```

1  izq  $\leftarrow$  1
2  der  $\leftarrow$  n
3  while izq < der do
4      med  $\leftarrow$  (izq + der)/2
5      if  $x \leq A[\text{med}]$  then
6          | der  $\leftarrow$  med
7      else
8          | izq  $\leftarrow$  med + 1
9      end
10 end
    // Pospone el chequeo de igualdad hasta el
    final
11 if  $x = A[\text{izq}]$  then
12     | return izq
13 else
14     | return n + 1
15 end

```

algoritmo pueden implementarse usando una única comparación por iteración. Si $T_2(n)$ denota el tiempo de ejecución de la búsqueda binaria de dos ramas, la correspondiente ecuación de recurrencia es:

$$T_2(n) = \begin{cases} T_2(n/2) + 1, & \text{si } n > 1. \\ 1, & \text{si } n = 1. \end{cases} \quad (12.2)$$

Resolviendo tenemos que $T_2(n) = \lg n + 1$, para $n \geq 1$. Hay dos aspectos a destacar en este punto. Primero, en el peor caso la búsqueda binaria de dos ramas realiza la mitad de las comparaciones realizadas por el algoritmo de tres ramas. Segundo, la cantidad de comparaciones realizadas por la búsqueda binaria de dos ramas es la misma en cualquier caso, sin importar el elemento buscado, por lo que $T_2(n) \in \Theta(\lg n)$. Recordar que $T_3(n) \in O(\lg n)$, por lo que en algunos casos la búsqueda

binaria de tres ramas puede ser más rápida. Sin embargo, como ya se dijo, en el peor caso $T_2(n)$ es menor a $T_3(n)$. Finalmente, podemos concluir que la búsqueda binaria de 2 ramas es óptima para el problema de búsqueda en conjunto ordenado, ya que la cantidad de comparaciones que realiza es exactamente igual a la cota inferior para la complejidad $T_B(n)$. Además, hemos encontrado la complejidad exacta del problema de búsqueda en arreglo ordenado:

$$T_B(n) = \lceil \lg n \rceil + 1 \text{ comparaciones.}$$

Búsqueda en Multiconjuntos Ordenados Un multiconjunto se define como un conjunto que puede contener elementos repetidos. Un ejemplo de multiconjunto ordenado es $A = \langle 1, 2, 3, 3, 3, 4, 5, 5, 5, 5, 5, 6, 7, 8, 8 \rangle$. En estos casos, uno puede estar interesado no sólo en determinar si un elemento x pertenece o no al multiconjunto, sino que además encontrar, por ejemplo, la primera (o última) ocurrencia de x en el multiconjunto, o alternatively determinar la cantidad de veces que ocurre x en el multiconjunto. La búsqueda binaria de dos ramas es útil en estos casos. Observe que si ejecuta el código del algoritmo para el ejemplo anterior, buscando por $x = 5$, se encontrará la primera ocurrencia de 5 en A (haga esa ejecución como ejercicio). El código puede ser modificado de forma muy simple, para que las comparaciones por igualdad hagan que el algoritmo siga en la mitad derecha del arreglo, lo que permitiría encontrar la última ocurrencia del elemento buscado (dicha implementación queda como ejercicio).

Ejercicio

Diseñe un algoritmo que permita determinar la cantidad de ocurrencias de un elemento x en un multiconjunto A de n elementos, en tiempo $\Theta(\lg n)$.

Hint: Use las dos versiones de búsqueda binaria de dos ramas, para encontrar la primera y última ocurrencia de x en el arreglo que almacena al multiconjunto A .

Búsqueda Interpolada

Este tipo de búsqueda es una variante de la búsqueda binaria en la que las comparaciones no necesariamente se hacen siempre en el centro del arreglo, sino que el punto de acceso puede variar. Es un caso típico de decrecer por un factor variable. Piense la manera en que una persona busca en una guía telefónica ⁴. Más allá de que la búsqueda binaria es el algoritmo óptimo para el problema de búsqueda en el modelo de comparaciones. Difícilmente al buscar en una guía uno aplique exactamente una búsqueda binaria. La realidad es que si uno está buscando el número de una persona cuyo apellido comienza con una letra cercana a la A, entonces uno abrirá la guía en alguna posición cercana al extremo inferior. Esto ilustra la manera en que funciona una búsqueda interpolada: trata de “adivinar” la posición del arreglo en donde debería estar almacenado el elemento buscado, y luego actúa

⁴ Una guía telefónica era un listado ordenado alfabéticamente, que se usaba en la antigüedad para conocer el número de teléfono de las personas.

de forma similar a la búsqueda binaria, descartando partes que no contienen el elemento buscado.

La idea es usar los valores almacenados en los extremos del arreglo para determinar aproximadamente dónde debería estar almacenado el valor buscado. Por ejemplo, si el arreglo es $A[1..10] = \langle 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 \rangle$, entonces al buscar $x = 20$ el algoritmo debería comparar con algún elemento cerca de la segunda posición del arreglo: dado que el menor elemento del arreglo es 10 y el máximo es 100, en una distribución uniforme de los elementos sobre el arreglo eso indicaría que los elementos incrementan de 10 en 10. En efecto, el método que estudiaremos será efectivo en arreglos cuyos elementos están distribuidos uniformemente dentro del intervalo $[A[1]..A[n]]$. En otras palabras, asume que los valores del arreglo crecen linealmente con los índices del arreglo. Esto es porque haremos una interpolación lineal en base a $A[1]$ y $A[n]$.

Sean i y d los límites inferior y superior actuales del arreglo A , respectivamente. La posición m en donde se debe hacer la comparación se calcula mediante interpolación lineal de la siguiente manera:

$$m = i + \left\lfloor \frac{(x - A[d])(d - i)}{A[d] - A[i]} \right\rfloor.$$

Como se dijo anteriormente, la búsqueda procede de manera similar a la búsqueda binaria, usando el valor de m para acceder al arreglo y descartar partes del mismo.

El algoritmo es más eficiente en la práctica que la búsqueda binaria en casos en que la distribución uniforme de los elementos es cierta. Sin embargo, en general puede llegar a tener un comportamiento bastante malo. Por ejemplo, piense en el arreglo $A[1..10] = \langle 10, 11, 12, 13, 14, 15, 16, 17, 18, 100 \rangle$. Al buscar $x = 18$, se comenzará accediendo inicialmente a la segunda posición. En la siguiente iteración el punto de acceso es el cuarto elemento del arreglo, y así siguiendo. En general, el peor caso son n comparaciones, lo que hace que el algoritmo tenga tiempo de ejecución $O(n)$. Puede demostrarse, sin embargo, que el tiempo de ejecución promedio es $O(\lg \lg n)$.

Búsqueda Exponencial

Antes de estudiar la siguiente estrategia de búsqueda, deténgase a resolver el siguiente problema.

Ejercicio

Pídale a algún conocido que piense (y anote en un papel) un número natural $n \geq 1$. Note que no hay cota superior para n . Su objetivo es adivinar n en la menor cantidad de intentos, sólo permitiendo comparaciones por mayor, menor, o igual.

La búsqueda exponencial permite buscar de manera eficiente en espacios no acotados en uno de sus extremos, tal como ocurre en el ejercicio anterior. Una estrategia podría ser avanzar consultando por números que son múltiplos de algún valor constante C . Sin embargo, son necesarias

$\frac{n}{C} + C = O(n)$ comparaciones para encontrar el número buscado, dado que C es constante.

Algo similar, pero más eficiente, es avanzar dando pasos de tamaño variable creciente (y no constantes como antes). En particular, el algoritmo consulta por valores que son potencia de 2: $1, 2, 4, 8, 16, \dots, 2^d$, tal que $2^{d-1} < n \leq 2^d$. Por lo tanto, también se cumple $\lg 2^{d-1} < \lg n \leq \lg 2^d$, y entonces $d-1 < \lg n \leq d$. Es decir, son necesarias $\lceil \lg n \rceil$ comparaciones para encontrar 2^d , la potencia de 2 más pequeña que acota superiormente al número buscado. Luego, se realiza una búsqueda binaria entre los valores $2^{d-1}+1$ y 2^d . Note que hay $2^d - (2^{d-1}+1) + 1 = 2^{d-1}$ elementos en ese intervalo, por lo que la búsqueda binaria necesita $\lg 2^{d-1} = d-1 = \lceil \lg n \rceil - 1$ comparaciones. En total, son necesarias $2\lceil \lg n \rceil - 1$ comparaciones para encontrar n .

Aplicando la misma técnica sobre un arreglo $A[1..n]$, el algoritmo consulta en las posiciones que son potencia de 2: $A[1], A[2], A[4], A[8], A[16], \dots, A[2^d]$, hasta que $A[2^{d-1}] < x \leq A[2^d]$. Luego, se realiza una búsqueda binaria en el subarreglo $A[2^{d-1} + 1..2^d]$. Si $x = A[i]$ (es decir, el elemento buscado está almacenado en la posición i del arreglo), entonces $2^{d-1} < i \leq 2^d$, por lo tanto $d-1 < \lg i \leq d$. En particular, $d = \lceil \lg i \rceil$. Luego, el subarreglo $A[2^{d-1} + 1..2^d]$ tiene 2^{d-1} elementos, por lo que la búsqueda binaria allí requiere $\lg i \leq d$ comparaciones. El tiempo total es, por lo tanto, $\Theta(\lg i)$ ⁵.

Pregunta

¿Por qué el tiempo de ejecución del algoritmo para un arreglo de n elementos es $\Theta(\lg i)$, lo que es menor o igual a la cota inferior $\Omega(\lg n)$ del problema? ¿Cómo es esto posible?

Note que el algoritmo tiene un tiempo de ejecución que depende de la posición que ocupa el elemento buscado: si está más cerca del origen, se lo encuentra más rápido que si está más alejado. Es un ejemplo de algoritmo adaptativo, ya que su tiempo de ejecución se adapta a la “dificultad” del problema. Aquí la dificultad estaría medida en la distancia del elemento buscado respecto a la primera posición del arreglo. Este algoritmo de búsqueda es preferible (por sobre la búsqueda binaria) si es probable que el elemento buscado esté cerca del origen de la búsqueda, o en casos en que no se conozca un límite superior del espacio de búsqueda.

Ejercicios

1. Dado el arreglo $A = \langle 3, 14, 27, 31, 39, 42, 55, 70, 74, 81, 85, 93, 98 \rangle$, resuelva los siguientes puntos.
 - a) ¿Cuál es la mayor cantidad de comparaciones de elementos realizados por el algoritmo de búsqueda binaria de 3 ramas cuando se busca un elemento en dicho arreglo?

⁵ Note que aún usando una búsqueda binaria de tres ramas en el subarreglo $A[2^{d-1} + 1..2^d]$, el tiempo de ejecución sigue siendo $\Theta(\lg i)$ y no $O(\lg i)$. Esto es por la primera etapa del algoritmo, que requiere exactamente $\lceil \lg i \rceil$ comparaciones.

- b) Muestre todos los elementos del arreglo que requieren el mayor número de comparaciones para ser encontrados.
 - c) Calcule el número promedio de comparaciones hechas por el algoritmo de búsqueda binaria de 3 ramas, para una búsqueda exitosa en el arreglo dado. Asuma que cada uno de los elementos del arreglo tiene la misma probabilidad de ser buscado.
 - d) Calcule el número promedio de comparaciones hechas por el algoritmo de búsqueda por trisección, para una búsqueda no exitosa en el arreglo dado. Asuma que cada uno de los 14 puntos de fracaso tienen la misma probabilidad de producir un fracaso.
2. Al implementar `InsertionSort`, se podría usar una búsqueda binaria para localizar la posición dentro de los primeros $i - 1$ elementos del arreglo en la cual el i -ésimo elemento del arreglo debería ser insertado. ¿Cómo afectaría esto al número de comparaciones realizadas? ¿Cómo afectaría el uso de una búsqueda binaria al tiempo de ejecución de `InsertionSort`?
 3. Suponga que se cuenta con $n > 2$ monedas idénticas (indistinguibles entre ellas). Suponga además que exactamente una de las monedas es falsa, y su peso es distinto al de las monedas legítimas (aunque no se sabe si la moneda falsa pesa más o menos que las monedas legítimas). Se tiene acceso a una balanza de dos platos, la cual permite colocar objetos en cada uno de sus platos, indicando de qué lado está el mayor peso. Cada uso de la balanza se cobra, por lo que debe usarse la menor cantidad de veces posible. Diseñe un algoritmo eficiente para encontrar la moneda falsa. Analice su algoritmo y determine su costo usando la notación asintótica más adecuada.
 4. Dada una secuencia ordenada de enteros (sin elementos repetidos) $A = \langle a_1, a_2, \dots, a_n \rangle$, tal que $1 \leq a_i \leq m$, para $i = 1, \dots, n$, y $n < m$. Diseñe un algoritmo de tiempo de peor caso $O(\log n)$, que permita encontrar el entero más pequeño $j \in [1..m]$ tal que $j \notin A$.
 5. Dada una secuencia ordenada de enteros (sin elementos repetidos) $A = \langle a_1, a_2, \dots, a_n \rangle$, tal que $1 \leq a_i \leq m$, para $i = 1, \dots, n$, y $n < m$. Diseñe un algoritmo de tiempo de peor caso $O(\log j)$, que permita encontrar el entero más pequeño $j \in [1..m]$ tal que $j \notin A$.
 6. Suponga una matriz booleana infinita hacia abajo y hacia la derecha. Se sabe que la matriz almacena valores 0 tanto en el borde izquierdo como en el borde superior. El resto de la matriz, es decir, la esquina inferior derecha (infinita) almacena valores 1.

Un ejemplo de dicha matriz es la siguiente:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & \dots & 0 & \dots \\ 0 & \ddots & 0 & 0 & 0 & \dots & 0 & \dots \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \dots \\ 0 & \ddots & 0 & 0 & 0 & \dots & 0 & \dots \\ 0 & \ddots & 0 & 1 & 1 & \dots & 1 & \dots \\ 0 & \ddots & 0 & 1 & 1 & \ddots & 1 & \dots \\ 0 & \ddots & 0 & 1 & 1 & \ddots & 1 & \dots \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots \end{pmatrix}$$

Proponga un algoritmo eficiente que permita encontrar la fila i y la columna j en donde está almacenado el 1 superior izquierdo. Analice su algoritmo y determine el número de comparaciones realizadas.

7. Sea M una matriz de $n \times n$ en la cual las entradas de cada fila están ordenadas de manera creciente (de izquierda a derecha) y las entradas en cada columna están ordenadas de manera creciente (de arriba hacia abajo). Diseñe un algoritmo eficiente para encontrar la posición de un entero x en M , o que determine que dicho x no está en la matriz. Analice su algoritmo usando el Teorema Maestro.
8. Suponga que se tiene un arreglo ordenado $A[0..n-1]$ de enteros, en donde los elementos han sido movidos (“shifteados”) k posiciones a la derecha, de manera circular. Por ejemplo, $A = \langle 35, 42, 5, 15, 27, 29 \rangle$ es un arreglo ordenado que ha sido shifteado circularmente $k = 2$ posiciones, mientras que $A = \langle 27, 29, 35, 42, 5, 15 \rangle$ ha sido shifteado $k = 4$ posiciones.
 - a) Suponga que se conoce k . Dar un algoritmo de tiempo $\Theta(1)$ que permita encontrar el mayor valor almacenado en A .
 - b) Suponga que no se conoce el valor de k . Dar un algoritmo que permita encontrar el mayor elemento almacenado en A en tiempo $O(\log n)$.
 - c) Suponga que no se conoce el valor de k . Dar un algoritmo que permita encontrar cualquier elemento almacenado en A en tiempo $O(\log n)$.
9. La búsqueda por interpolación realiza en promedio $O(\log \log n)$ comparaciones, mientras que en peor caso ese número es $O(n)$.
 - a) Encontrar el valor más pequeño de n para el cual $\log \log n$ es mayor a 6.
 - b) Dé un ejemplo de un arreglo de enteros ordenado y un valor a buscar en el arreglo, de manera que produzca el peor caso para el algoritmo de búsqueda interpolada.
 - c) Suponga que A es un arreglo ordenado. Proponga un algoritmo de tiempo $O(\log n)$ en el peor caso para determinar si A tiene una mayoría absoluta o no.
 - d) En el modelo de comparaciones ¿Cómo usaría el algoritmo QuickSelect para resolver el problema en tiempo $\Theta(n)$ en el peor caso?

Usos Alternativos de la Búsqueda Binaria

El proceso aplicado para la búsqueda binaria (o también la búsqueda exponencial) puede ser utilizado en escenarios que no necesariamente corresponden al problema de búsqueda en un arreglo ordenado. En esta sección ilustramos usos prácticos de la búsqueda binaria.

Búsqueda Binaria de las Posibles Soluciones En ciertos casos uno puede buscar la respuesta a un problema usando búsqueda binaria sobre el espacio de posibles soluciones. Considere el siguiente problema: Dados dos enteros positivos n y m , diseñe un algoritmo que calcule $\lfloor n/m \rfloor$, asumiendo un modelo de computación que no tiene la operación aritmética de división de enteros (de otra manera el problema sería trivial), pero sí tiene disponible otras operaciones como multiplicación, suma, resta, shift, etc. Note que en el caso en que n es divisible por m , el algoritmo debe encontrar el entero z tal que $n = m \times z$. Aunque en general n no será necesariamente divisible por m , esto permite pensar en una solución. Hay que asumir también que los enteros positivos n y m están representados, como es usual, en binario. Recuerde que $\frac{x}{2^k} = x \gg k$ (para números enteros x y k), en donde ' \gg ' es el operador de shift a la izquierda, usando notación de lenguaje C.

Para resolver este problema, usaremos búsqueda binaria para encontrar la respuesta z dentro del intervalo inicial $[1..n]$, en tiempo $O(\lg n)$. Esto es, considere el valor $z = (n + 1)/2$, el elemento central del intervalo $[1..n]$. Note que si $z \times m > n$, entonces para cualquier valor z' dentro del intervalo $[z..n]$ también ocurre $z' \times m > n$ (esto es porque $z' \geq z$). Dado que ninguno de esos z' puede ser la respuesta, ese subintervalo se descarta y se continúa buscando en el subintervalo $[1..z - 1]$. Si, por otro lado, $z \times m < n$, la búsqueda debe continuar en el subintervalo $[z + 1..n]$. Sí, finalmente, $z \times m = n$, hemos encontrado la respuesta. El tiempo de ejecución es $O(\lg n)$. El algoritmo es el siguiente:

```
int dividir(int n, int m) {
    int inf = 1;
    int sup = n;
    int z;

    while ((sup - inf + 1) > 1) {
        z = (sup+inf) >> 1;    // es lo mismo que (sup+inf)/2
        if (z*m == n) return z;
        else if (z*m > n)
            sup = z-1;
        else
            inf = z+1;
    }
    return z;
}
```

El algoritmo se puede mejorar para que tome tiempo $O(\lg m)$. ¿Cómo? ¿Por qué?

Búsqueda Binaria en Arreglos no Ordenados Aún cuando la premisa principal de la búsqueda binaria es que el arreglo de entrada debe estar ordenado, éste tipo de proceso de búsqueda puede ser empleado (en ciertos casos) sobre arreglos no ordenados. Considere el siguiente problema: dado un arreglo $A[1..n]$ de números enteros positivos, no necesariamente ordenado y sin elementos repetidos, el elemento $A[i]$, para $1 < i < n$, es un mínimo local si cumple con:

- $A[i - 1] > A[i]$; y
- $A[i] < A[i + 1]$.

Para $i = 1$ es suficiente $A[1] < A[2]$, y para $i = n$ es suficiente $A[n - 1] > A[n]$. Por ejemplo:

- $A[1..7] = \{9, 6, 3, 14, 5, 7, 4\}$. En este caso, $A[3]$, $A[5]$, y $A[7]$ son mínimos locales.
- $A[1..5] = \{23, 8, 15, 2, 3\}$. En este caso, $A[2]$ y $A[4]$ son mínimos locales.
- $A[1..3] = \{1, 2, 3\}$. En este caso, $A[1]$ es el único mínimo local.
- $A[1..3] = \{3, 2, 1\}$. En este caso, $A[3]$ es el único mínimo local.

Diseñaremos un algoritmo que permita encontrar un mínimo local (cualquiera) en A en tiempo $O(\lg n)$. Note que, de acuerdo a la definición, cualquier arreglo de enteros positivos tiene al menos un mínimo local. Luego, se aplica búsqueda binaria de la siguiente manera.

```
int localMin(int A[], int izq, int der, int n) {
    int m = (izq + der)/2;
    while (der-izq + 1 > 0) {
        if ((m == 0 || A[m-1] > A[m]) && (m == n-1 || A[m+1] > A[m]))
            return m;
        else
            if (m > 0 && A[m-1] < A[m]) // parte izq. contiene mínimo local
                der = m - 1;
            else
                izq = m + 1; // parte derecha contiene mínimo local
    }
}
```

Buscando Dos Valores de Suma Dada Dados un arreglo $A[1..n]$ de valores enteros y un entero x , se quiere determinar si existen dos valores en A cuya suma sea x . Mostraremos un algoritmo que lo resuelve usando $\Theta(n \lg n)$ comparaciones.

El primer paso es ordenar A usando MergeSort, para luego hacer búsqueda binaria del valor $x - A[i]$ por cada elemento $A[i]$ del arreglo. El algoritmo es el siguiente:

```
bool suma(int *A, int n, int x) {
    MergeSort(A, n);
    for (i = 0; i < n; i++)
        if (busquedaBinaria(A, x-A[i])) return true;
    return false;
}
```

Buscando la Mediana de Dos Arreglos Ordenados Sean $A[1 \dots n]$ y $B[1 \dots n]$ dos arreglos ordenados de números enteros positivos, de tamaño n cada uno. Se quiere encontrar la mediana del arreglo de tamaño $2n$ que contiene los elementos de A y los elementos de B . Recuerde que la mediana de un arreglo de tamaño n es el elemento que ocuparía la posición $\lfloor (n+1)/2 \rfloor$ si el arreglo estuviera ordenado.

Diseñaremos el algoritmo $\text{mediana}(A[1 \dots n], B[1 \dots n])$, que resuelva el problema planteado en tiempo $O(\log n)$. La idea es buscar la mediana de ambos arreglos sin combinarlos, es decir, sin usar la operación *merge* estudiada en otras secciones, lo que tomaría tiempo $\Theta(n)$. La idea es, en base a la comparación de las medianas de cada uno de los arreglos (A y B), descartar mitades de estos arreglos y proceder recursivamente.

Sea $m_1 = A[\lfloor \frac{n+1}{2} \rfloor]$ y $m_2 = B[\lfloor \frac{n+1}{2} \rfloor]$. En base a la comparación de m_1 y m_2 , tenemos los siguientes 3 casos:

- $m_1 = m_2$: en este caso, notar que m_1 o m_2 (que tienen el mismo valor) ocupan la posición n al combinar los elementos de A y B . Dado que esa es la posición de la mediana que se busca, el algoritmo finaliza.
- $m_1 < m_2$: notar que al combinar A y B , en el resultado habría al menos $n/2$ elementos menores a m_1 . Esos corresponden a los elementos menores a m_1 en A . Ninguno de esos elementos puede ser la mediana buscada, por lo tanto pueden descartarse. De la misma forma, hay al menos $n/2$ elementos mayores a m_2 . Esos corresponden a los elementos mayores a m_2 en B (los que pueden descartarse por la misma razón que antes). Entonces, el algoritmo continúa con la invocación recursiva

$$\text{mediana}\left(A\left[\left\lfloor \frac{n+1}{2} \right\rfloor .. n\right], B\left[1 .. \left\lfloor \frac{n+1}{2} \right\rfloor\right]\right).$$

- $m_1 > m_2$: este caso es similar al anterior, pero invocando a

$$\text{mediana}\left(A\left[1 .. \left\lfloor \frac{n+1}{2} \right\rfloor\right], B\left[\left\lfloor \frac{n+1}{2} \right\rfloor .. n\right]\right).$$

De esta manera, cada vez se divide el tamaño del problema a la mitad. El tiempo de ejecución está dado por una ecuación de recurrencia del tipo $T(n) = T(n/2) + 1$, lo cual es $O(\lg n)$.

Colocando k Elementos que Maximicen la Distancia Mínima. Esta interesante, el planteo y solución del problema aquí: <https://www.geeksforgeeks.org/place-k-elements-such-that-minimum-distance-is-maximized/>

Ejercicios

1. Diseñe un algoritmo de tipo decrecer por la mitad para computar $\lfloor \log_2 n \rfloor$, para un valor de n dado, y determine el tiempo de ejecución del mismo.
2. Suponga el problema de calcular a^n , para una constante a y $n > 0$.

- a) Diseñe un algoritmo para resolver el problema, basado en decrecer por un valor constante. Plantear la ecuación de recurrencia para su algoritmo, y acotarla asintóticamente.
- b) Diseñe un algoritmo basado en decrecer y conquistar para resolver el problema, basado en decrecer por un factor constante. Plantear la ecuación de recurrencia para su algoritmo, y acotarla asintóticamente.

En cada uno de los casos, su algoritmo debe resolver el problema para cualquier valor de n (tanto par como impar).

3. Dados un multiconjunto (es decir, un conjunto que puede contener elementos repetidos) ordenado $A[1..n]$, y un rango $[i..j]$, para $i \leq j$, se quiere conocer cuántos valores distintos hay en $A[i..j]$. Por ejemplo, para $A[1..20] = (1, 1, 1, 2, 2, 2, 3, 3, 3, 3, 3, 4, 5, 6, 7, 7, 8, 8, 8, 9)$ y el rango $[6..17]$, la cantidad de valores distintos es 7. Escriba un algoritmo basado en Decrecer y Conquistar para resolver este problema. Si hay k valores distintos en el rango, el tiempo de ejecución de su algoritmo debería ser $O(k \lg n)$ o, alternativamente, $\Theta(k \lg n)$.
4. Dada una función $F: \mathbb{N}^+ \mapsto \{\text{true}, \text{false}\}$, tal que

$$F(j) = \begin{cases} \text{true} & \text{si } j < k, \\ \text{false} & \text{si } j \geq k, \end{cases}$$

en donde k es un número entero que define a F de forma única. Escriba un algoritmo tal que dada una función de ese tipo (con k desconocido), permita encontrar k en tiempo $O(\lg k)$.

12.4. El Problema de Selección

Tal como en la Sección 8.2 (página 154), el *problema de selección* se define de la siguiente manera: dado un conjunto $S = \{s_1, \dots, s_n\}$ no necesariamente ordenado de n elementos, sobre el que se ha definido una relación de orden total \leq , se quiere encontrar el elemento de rango k . Esto es, el elemento $s_j \in S$ tal que s_j ocuparía la posición k si el conjunto S estuviese ordenado. Estudiamos a continuación cómo resolver este problemas para casos particulares de k , asumiendo el modelo de comparaciones.

Ordenando el Conjunto

Una manera simple de resolver el problema es ordenar S , y luego reportar $S[k]$. El tiempo de ejecución es $\Theta(n \lg n)$ comparaciones si se usa MergeSort, para cualquier valor de k . Esto requiere una cantidad de espacio adicional $\Theta(n)$. Dado esto, ¿Será posible resolver el problema en menor tiempo de ejecución (es decir, $o(n \lg n)$ comparaciones), además de usar una menor cantidad de espacio adicional?

Los Casos $k = 1$ y $k = n$

Estos dos casos ya han sido estudiados anteriormente, por ejemplo en la Sección 8.2. Consiste en encontrar el mínimo del conjunto ($k = 1$) o

el máximo ($k = n$). Ya sabemos que la complejidad de este problema es $n - 1$ comparaciones (en ambos casos), por lo que el Algoritmo 1 (página 49) es óptimo para $k = n$ (se puede definir un algoritmo análogo para $k = 1$). Esto indica que para $k = 1$, ordenar el conjunto no es una alternativa eficiente. ¿Esto será cierto sólo para $k = 1$ y $k = n$, o habrá otros valores de k para los cuales también se puede resolver el problema en tiempo $o(n \lg n)$? ¿Será posible resolver el problema en tiempo $o(n \lg n)$ para cualquier valor de k , o existe algún k a partir del cual $\Theta(n \lg n)$ es la alternativa más eficiente? Resolvemos estas preguntas en lo que sigue.

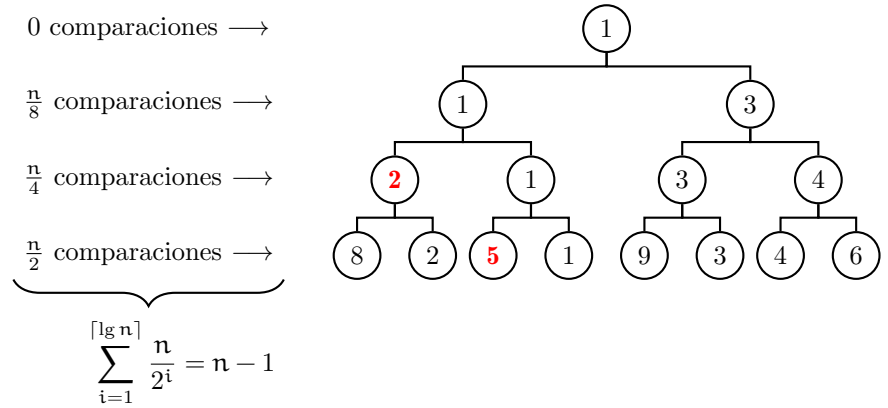
El Caso $k = 2$

Para este caso tenemos varias alternativas de tiempo lineal, que mostraremos en orden creciente de eficiencia y dificultad. En conclusión, para $k = 2$ (y, de forma equivalente, $k = n - 2$) el problema puede resolverse más eficientemente sin ordenar.

Alternativa 1: Fuerza Bruta La manera más simple de encontrar el segundo elemento más pequeño en S es con un algoritmo de dos pasadas sobre S . En la primera pasada, buscamos el elemento mínimo, digamos que está en la posición m_1 , con un total de $n - 1$ comparaciones. Luego, en una segunda pasada, buscamos el mínimo en $A[1..m_1 - 1]$ y $A[m_1 + 1..n]$ (es decir, sin considerar el elemento $A[m_1]$). Esto requiere de dos iteraciones, una por cada uno de esos dos segmentos. La cantidad de comparaciones es $n - (m_1 + 1) + 1 + m_1 - 2 = n - 2$. El total para las dos pasadas es $2n - 3$ comparaciones.

Alternativa 2: Decrecer y Conquistar Esta alternativa usa un esquema del tipo decrecer y conquistar. La idea es dividir (de forma lógica, no física) el arreglo en dos mitades, de tamaños $\lceil \frac{n}{2} \rceil$ y $\lfloor \frac{n}{2} \rfloor$, respectivamente. Luego, se busca el mínimo de cada una de las mitades, digamos que se encuentran en las posiciones m_1 y m_2 del arreglo (asumamos $m_1 < m_2$). Este proceso toma $\lceil \frac{n}{2} \rceil + \lfloor \frac{n}{2} \rfloor - 2 = n - 2$ comparaciones. Luego, se comparan $A[m_1]$ y $A[m_2]$ para decidir cuál de los elementos es el mínimo, con lo que el total de comparaciones hasta el momento es $n - 1$. Sin pérdida de generalidad, asumamos que $A[m_1] < A[m_2]$. Luego, para buscar el segundo mínimo, note que $A[m_2]$ es un candidato a serlo. Sin embargo, podría existir un elemento menor que $A[m_2]$ en la primera mitad del arreglo el cual, a causa de que $A[m_1]$ está en esa mitad, no pudo encontrarse. A continuación, buscamos el mínimo dentro de los segmentos $A[1..m_1 - 1]$ y $A[m_1 + 1..\lceil \frac{n}{2} \rceil]$, de forma similar a la segunda pasada de la alternativa 1. Esto requiere $\lceil \frac{n}{2} \rceil - 2$ comparaciones. Digamos que encontramos ese mínimo en la posición m_3 . Finalmente, el mínimo entre $A[m_3]$ y $A[m_2]$ es la respuesta al problema, lo cual agrega una última comparación. Sumando, la cantidad de comparaciones en total son $3\lceil \frac{n}{2} \rceil - 2$ comparaciones, bastante mejor que la Alternativa 1: de las dos pasadas originales sobre el conjunto, ahora necesitamos sólo una pasada y media.

Figura 12.1: Algoritmo del torneo de tenis para buscar el segundo elemento más pequeño del conjunto $A = (8, 2, 5, 1, 9, 3, 4, 6)$. Se hacen $n-1$ comparaciones para encontrar el mínimo (1). Luego, se busca el mínimo entre 3 y los $\lg n - 1 = 2$ elementos que “perdieron” con 1 (en negrita y rojo). El total es $n + \lg n - 2$ comparaciones.



Alternativa 3: El Torneo de Tenis A continuación mostramos que para $k = 2$ el problema puede resolverse básicamente con una única pasada, más una cantidad logarítmica de trabajo. La idea es realizar un proceso similar a lo que sería un *torneo de tenis*. El algoritmo procede por rondas. La primera de ellas compara elementos consecutivos del arreglo original, y los ganadores pasan a la segunda ronda. La Figura 12.1 muestra un ejemplo del proceso. El algoritmo construirá un árbol casi perfecto, en el que las hojas serán los mismos elementos del arreglo. Los “ganadores” de cada comparación pasan a la siguiente ronda, representada por un nuevo nivel en el árbol. En cada ronda, la cantidad de elementos se reduce a la mitad, por lo que se necesitan $\lceil \lg n \rceil$ rondas para encontrar el mínimo. En la ronda i , para $1 \leq i \leq \lceil \lg n \rceil$, se hacen $\frac{n}{2^i}$ comparaciones. Esto es, a nivel de las hojas, se hacen $\frac{n}{2}$ comparaciones, en el siguiente nivel se hacen $\frac{n}{4}$ comparaciones, y así siguiendo. En total, se realizan

$$\sum_{i=1}^{\lceil \lg n \rceil} \frac{n}{2^i} = n - 1$$

comparaciones para encontrar el mínimo del arreglo. Nada nuevo hasta ahora. Además, al igual que en la Alternativa 2, hemos encontrado al mínimo de la otra mitad del arreglo. La novedad está en que al buscar nuevamente el mínimo en la mitad que contiene al mínimo (sin considerar al mínimo ya encontrado), no haremos búsqueda exhaustiva como antes, sino que usaremos el árbol para buscarlo: la razón por la que ese elemento no llegó a la final del torneo es que en algún momento se cruzó con el mínimo. Esto significa que sólo debemos buscar el mínimo entre los elementos que en algún momento se compararon con el mínimo, que en total son $\lceil \lg n \rceil - 1$. Por lo tanto, la cantidad total de comparaciones es $n + \lceil \lg n \rceil - 2$. En el ejemplo de la Figura 12.1, luego de encontrar que 1 es el mínimo del arreglo, debemos buscar el mínimo entre 3 y los dos elementos que se compararon con el 1: en este caso, 5 y 2, que han sido marcados con color en la figura.

El Caso General: QuickSelect

Finalmente, mostramos que para cualquier $k \leq n$, el problema de selección se puede resolver en tiempo lineal, lo que implica que la solución de ordenar el arreglo planteada inicialmente no es eficiente en ningún

caso. Lamentablemente, las ideas estudiadas para $k = 2$ no generalizan de buena manera para todo k . El algoritmo que estudiaremos a continuación es conocido como **QuickSelect**, por su similitud con **QuickSort**. La idea principal es que luego de elegir el pivote y hacer la partición del arreglo en menores y mayores a éste, el mismo pivote queda ubicado en la posición que ocupará en el arreglo ordenado. Revisemos el siguiente ejemplo, ya visto anteriormente, en donde se elige el pivote $p = A[5] = 56$:

1	2	3	4	5	6	7	8	9	10
55	4	58	90	56	30	85	80	50	35

Luego de realizar la partición respecto al pivote, la situación es la siguiente:

1	2	3	4	5	6	7	8	9	10
55	4	50	30	35	56	85	80	58	90

El pivote $p = 56$ queda ubicado en la posición 6 del arreglo, lo que significa que es el sexto elemento del arreglo ordenado. Si aplicamos esta misma idea para el problema de selección, y $k < 6$, entonces debemos continuar buscando el k -ésimo elemento recursivamente en el segmento a la izquierda de la posición final del pivote. Si, por otro lado, $k \geq 6$, debemos buscar de forma recursiva el $(k - 6)$ -ésimo elemento en el segmento derecho del arreglo. Por ejemplo, si $k = 7$, entonces hay que buscar por $k = 1$ (es decir el mínimo) en el segmento derecho. Finalmente, si $k = 6$, el algoritmo finaliza ya que el pivote es la respuesta.

El Algoritmo 40 formaliza el proceso realizado por **QuickSelect**. Esto

Algoritmo 40: QuickSelect(A, i, d, k)

```

1   $p \leftarrow \frac{d+i}{2}$ 
2  SWAP( $A[p], A[d]$ )
3   $p \leftarrow \text{PARTITION}(A, i, d, p)$ 
4  if  $p \neq d - 1$  then
5    | SWAP( $A[p], A[d]$ )
6  end
7  if  $p - i = k$  then
8    | return  $A[p]$ 
9  else
10   | if  $k < p - i$  then
11     | return QuickSelect( $A, i, p - 1, k$ )
12   | else
13     | return QuickSelect( $A, p + 1, d, k - p$ )
14   | end
15 end

```

es una adaptación de **QuickSort**, ahora con un enfoque Decrecer y Conquistar: luego de hacer la partición, dividimos el arreglo en tres partes y continuamos recursivamente en alguna de ellas (en el caso de quedarnos con el elemento pivote, en realidad el algoritmo finaliza). Como ya lo vimos intuitivamente en su momento, la situación ideal sería que el pivote divida al arreglo en dos mitades exactas (o casi)

cada vez. En ese caso, la cantidad de comparaciones en el peor caso podría expresarse por una ecuación de recurrencia de la forma $T(n) = T(\frac{n}{2}) + \Theta(n)$. El Teorema Maestro indica que $T(n) \in \Theta(n)$, nuevamente mejor que ordenar el arreglo. Sin embargo, ya sabemos que no es posible garantizar (al menos hasta ahora) que el pivote sea la mediana del arreglo. Sin embargo, note cómo el hecho de descartar una de las mitades del arreglo (a diferencia de cuando ordenábamos), permite hacer que el tiempo total sea lineal, y no $n \lg n$. En el peor caso, si embargo, el pivote solo permite descartar una cantidad constante de elementos cada vez, y el tiempo de ejecución sería cuadrático.

A continuación mostramos cómo elegir el pivote de manera tal que se pueda garantizar descartar una fracción de n elementos cada vez, obteniendo tiempo lineal. Aunque la partición no será exactamente balanceada, será lo suficientemente buena como para garantizar tiempo lineal. La estrategia que usaremos se conoce como *mediana de medianas*.

Usemos el siguiente ejemplo para entender el proceso:

12	6	2	11	20	5	25	10	26	13	22	4	18	17	1	21	16	7	14	3	15	23	8	24	19
----	---	---	----	----	---	----	----	----	----	----	---	----	----	---	----	----	---	----	---	----	----	---	----	----

Luego, se divide el arreglo en $\frac{n}{5}$ bloques de 5 elementos cada uno (el último bloque puede tener menos elementos, eso no será un problema). Esos bloques de tamaño 5 se ordenan, usando un algoritmo de ordenamiento simple, por ejemplo `SelectionSort` o `InsertionSort`, produciendo un tiempo total de $\Theta(n)$ comparaciones para ordenar los bloques. Para el ejemplo, el resultado sería:

2	6	11	12	20	5	10	13	25	26	1	4	17	18	22	3	7	14	16	21	8	15	19	23	24
11					13					17					14					19				

Los bloques han sido sombreados alternadamente para una mejor lectura. Las $\frac{n}{5}$ medianas de los bloques se muestran debajo de cada bloque. Consideremos ahora esos mismos bloques organizados verticalmente y ordenados de acuerdo a las medianas:

2	5	3	1	8
6	10	7	4	15
11	< 13	< 14	< 17	< 19
12	25	16	18	23
20	26	21	22	24

Considere la mediana de las $\frac{n}{5}$ medianas. En el ejemplo sería el 14. Ese elemento, al ser la mediana de medianas, es mayor al menos que $\frac{n}{10}$ elementos (las medianas de bloques que son menores a ella). Eso también significa que la mediana de medianas es (al menos) mayor que los otros dos elementos de esos (al menos) $\frac{n}{10}$ bloques, por lo tanto podemos asegurar que la mediana de medianas es mayor que al menos $\frac{3n}{10}$ elementos. Esos son los elementos del cuadrante superior izquierdo

en el diagrama anterior. Note que podemos asegurar que la mediana es mayor al menos que esa cantidad de elementos, pero podrían ser más. De manera similar, podemos asegurar que al menos $\frac{3n}{10}$ elementos son mayores a la mediana de medianas.

Por lo antes expuesto, si elegimos a la mediana de mediana como pivote, podemos asegurar que luego de hacer la partición de **QuickSort**, los segmentos del arreglo a la izquierda y derecha del pivote tendrán al menos $\frac{3n}{10}$ elementos. Note que para poder seleccionar la mediana de mediana, debemos invocar recursivamente a **QuickSelect** con un arreglo de $\frac{n}{5}$ elementos. Luego de encontrar la mediana de medianas y hacer la partición, en el peor caso debemos continuar trabajando recursivamente sobre un arreglo de tamaño $\frac{7n}{10}$. El tiempo de ejecución de peor caso, entonces, está dado por una ecuación de recurrencia de la forma:

$$T(n) = \underbrace{T\left(\frac{n}{5}\right)}_{\text{Mediana de medianas}} + \underbrace{T\left(\frac{7n}{10}\right)}_{\text{Trabajo recursivo}} + \underbrace{\Theta(n)}_{\text{Ordenar bloques, particionar arreglo}} \quad (12.3)$$

A continuación, mostramos que $T(n) \in \Theta(n)$. De hecho, basta mostrar que $T(n) \in O(n)$. Esto significa que hay que probar que $\exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}$ tal que

$$T(n) \leq cn, \quad \forall n \geq n_0. \quad (12.4)$$

Sea $\alpha > 0$ la constante del término $\Theta(n)$ en la Ecuación (12.3), por lo que queremos demostrar que

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + \alpha n \leq cn.$$

De forma similar a lo mencionado en la Ecuación (12.4), tenemos que demostrar que $T\left(\frac{n}{5}\right) \leq c\frac{n}{5}$ y $T\left(\frac{7n}{10}\right) \leq c\frac{7n}{10}$, por lo tanto tenemos que probar que

$$T(n) = c\frac{n}{5} + c\frac{7n}{10} + \alpha n \leq cn.$$

Manipulando esta expresión, es equivalente a demostrar que

$$c\frac{9}{10} + \alpha \leq c \Rightarrow \frac{c}{10} \geq \alpha \Rightarrow c \geq 10\alpha.$$

Dado que $\alpha > 0$, entonces cualquier constante c tal que $c \geq 10\alpha$ es válida para probar que $T(n) \in O(n)$.

Una vez demostrado esto, estudiemos ahora la razón por la que elegimos inicialmente dividir en $\frac{n}{5}$ bloques de tamaño 5. Primero, el hecho de usar un tamaño impar simplifica la selección de la mediana de cada bloque. Segundo, ¿por qué no elegir dividir en $\frac{n}{3}$ bloques de 3 elementos cada uno? Después de todo, 3 también es un número impar. En ese caso, el tiempo de ejecución sería

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + \Theta(n).$$

Ahora, para probar que $T(n) \leq cn$, tenemos que

$$c \frac{n}{3} + c \frac{2n}{3} + an \leq cn.$$

Esto nos lleva a que

$$cn + an \leq cn,$$

lo que significa que $c + a \leq c$, que es lo mismo que $a \leq 0$. Esto es, para que $T(n) \in O(n)$ debe cumplirse que $a \leq 0$. Sin embargo, esta es una contradicción, ya que hemos dicho que $a > 0$. Podemos concluir que dividir en $\frac{n}{3}$ bloques de 3 elementos no permite obtener tiempo lineal. Eso implica que 3 es la menor constante impar para el cual se puede probar tiempo lineal.

Para concluir, hemos podido demostrar que el problema de selección puede resolverse en tiempo $\Theta(n)$ para cualquier $k \leq n$. Esto es más eficiente que ordenar el arreglo para responder. Finalmente, el método de mediana de medianas podría ser usado para elegir el pivote de QuickSort, y garantizar de esa manera $\Theta(n \lg n)$ comparaciones en el peor caso para ese algoritmo de ordenamiento. Sin embargo, las constantes involucradas son muy grandes —por ejemplo, $c \geq 10a$ —, lo que neutraliza la principal ventaja de QuickSort: su practicidad.

Ejercicios

1. Muestre que si un arreglo $A[1..n]$ no tiene elementos repetidos, QuickSelect puede ser usado para implementar QuickSort de manera que la cantidad de comparaciones es $\Theta(n \log n)$.
2. Suponga que dispone de un algoritmo `mediana(A)`, el cual permite encontrar la mediana del arreglo $A[1..n]$ en $O(n)$ comparaciones en el peor caso. Usando esa rutina, diseñe el algoritmo `select(A, k)`, el cual permite encontrar el k -ésimo menor elemento del arreglo A usando $O(n)$ comparaciones.

13.1. Introducción

En muchos casos, un problema puede ser resuelto de forma mucho más sencilla si se explota algún tipo de estructura subyacente de los datos o, alternativamente, se construye una estructura de datos explícita que permita manipular los datos de forma más eficiente. Estudiaremos en este capítulo problemas que pueden ser resueltos de forma eficiente usando algún tipo de estructura de datos.

13.2. El Problema de Ordenamiento: HeapSort

El problema de ordenamiento puede ser resuelto de forma eficiente usando estructuras de datos que ayuden en el proceso. Definimos a continuación una estructura de datos que será clave para lograr un algoritmo de ordenamiento óptimo in-place.

Estructuras de Datos de Tipo Heap

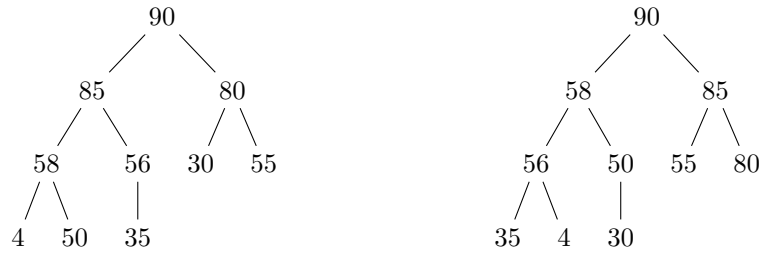
Definición 13.2.1 Sea S un conjunto de n elementos de algún tipo, sobre los que se ha definido una función de orden total. Una *heap de máximos* (máx-heap) para el conjunto S es un árbol binario que tiene las siguientes propiedades:

1. Es un árbol binario casi perfecto, y
2. todo elemento del árbol es mayor o igual que sus dos hijos.

Alternativamente, se puede definir un heap de mínimos (mín-heap) modificando la propiedad 2 de esta definición, tal que todo elemento del árbol es menor o igual a sus dos hijos. A lo largo de toda esta sección nos enfocaremos en máx-heaps, pero teniendo en cuenta que muchas de las definiciones y operaciones tienen su análogo en un mín-heap.

Debido a la propiedad 1 de la definición anterior, un heap que representa un conjunto de n elementos tiene altura $\lceil \lg n \rceil$. Note además que, de acuerdo a la definición, el mayor elemento de un conjunto sólo puede almacenarse en la raíz de un máx-heap del conjunto. La Figura 13.1 muestra dos ejemplos de heaps para el conjunto de números enteros $S = \{55, 4, 58, 90, 56, 30, 85, 80, 50, 35\}$. Otra propiedad importante (y que es resultado directo de la propiedad 2 de la Definición 13.2.1) es que cada una de las ramas de un máx-heap está ordenada de forma descendente desde la raíz. Por ejemplo, la rama de más a la izquierda del máx-heap de la izquierda en la Figura 13.1 contiene los elementos 90, 85, 58, y 4.

Figura 13.1: Dos máx-heaps distintos para el conjunto $S = \{55, 4, 58, 90, 56, 30, 85, 80, 50, 35\}$.



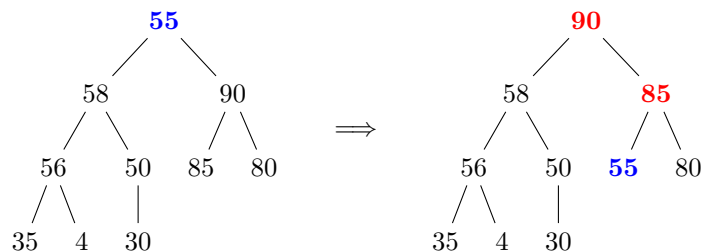
El hecho de que la raíz de un máx-heap siempre almacena el máximo del conjunto que representa, permite calcular la operación $\text{MÁX}(S)$ en tiempo $O(1)$. Un máx-heap también permite operaciones de inserción (de un nuevo elemento) y borrado del elemento máximo de forma eficiente. La siguiente es una propiedad que es importante comprender para resolver correctamente algunas de las operaciones.

Lema 13.2.1 *Dado un árbol casi lleno que no necesariamente es un máx-heap, y tal que los subárboles derecho e izquierdo de la raíz sí son máx-heaps, entonces el árbol puede ser convertido en un máx-heap usando $O(\lg n)$ comparaciones de elementos.*

Demostración. Hay que notar que el elemento raíz x es el único que podría estar violando la propiedad de orden del heap. Para reponer el orden de todas las ramas del árbol (y, por lo tanto, convertir al árbol en un máx-heap), es suficiente considerar una única rama, como se explica a continuación. Se comienza comparando a x con el mayor de sus hijos, que denotamos y (esto requiere 2 comparaciones). Si $x < y$, significa que y es el mayor elemento del conjunto y debe ocupar la raíz, por lo que se intercambian de posición. Es posible asegurar esto porque, por hipótesis, los dos hijos de la raíz son máx-heaps, por lo tanto x o alguno de sus hijos originales es el elemento máximo. Luego se procede recursivamente hacia abajo en el árbol, desde la nueva posición de x . En cada posición, el mismo argumento aplica. El proceso se detiene cuando x alcance una posición en la que sea mayor (o igual) a sus hijos, o eventualmente x se vuelva un nodo externo del árbol. Al modificar una única rama del heap, y en cada nivel hacer 2 comparaciones de elementos, el tiempo total es $O(\log n)$. ■

La Figura 13.2 ilustra esta propiedad, y la operación realizada para

Figura 13.2: Un árbol binario casi lleno que no es un máx-heap, tal que los dos hijos de la raíz sí son máx-heaps (izquierda). La propiedad de orden del máx-heap es restablecida hundiendo la raíz del árbol (55) por la rama correspondiente al mayor de los hijos en cada nivel (elementos marcados en rojo en el heap de la derecha).



que todo el árbol ahora sea un máx-heap. Llamaremos *heapify* a esta operación, tal como se conoce en la literatura.

A continuación mostramos cómo resolver las operaciones de inserción y borrado del máximo en un máx-heap.

Insertar(H, x): Para mantener la propiedad 1 de la Definición 13.2.1 después de haber insertado el nuevo elemento x , el árbol debe llenarse por niveles. Además, dentro de cada nivel el llenado es de izquierda a derecha. Es decir, x debe insertarse inicialmente lo más a la derecha posible en el último nivel del árbol, o como primer elemento de un nuevo nivel cuando el último nivel está lleno. Luego de insertar x en el punto indicado, se podría romper el orden descendente de los elementos en la rama de H que va desde la raíz a x . Para mantener la propiedad 2 de la Definición 13.2.1, se compara x con su padre actual y . Si $x > y$, entonces estos elementos intercambian su posición, pasando x a ser el padre de y . El proceso continúa, comparando a x con su nuevo padre hasta que se cumpla la condición de orden indicada por la propiedad 2, o hasta eventualmente alcanzar la raíz del árbol (al insertar un elemento que es mayor a todos los del conjunto).

BorrarMáx(H): Nuevamente, para mantener la propiedad 1 de la Definición 13.2.1, el elemento x que se encuentra más a la derecha en el último nivel de H es colocado en la raíz y eliminado de su anterior posición (el árbol tiene ahora un nodo menos). Esto produce como resultado un árbol casi lleno que no necesariamente es un máx-heap, y tal que los dos hijos de la raíz sí son máx-heaps. Por lo tanto, se procede como se indica en la Propiedad 13.2.1, usando la operación heapify para recuperar el orden en todas las ramas del máx-heap.

El tiempo de ejecución de ambas operaciones es, claramente, proporcional a la altura del árbol. Para lograr tiempo total $O(\lg n)$, sólo debemos ser capaces de navegar en tiempo constante no sólo a los hijos de un nodo, sino que también al padre. Eso puede lograrse trivialmente si se utiliza una representación de punteros. Sin embargo, aún cuando esa es la manera clásica de representar árboles, no es la usada para implementar heaps: no sólo porque utiliza más espacio (por los punteros, incluyendo al padre), sino que además introduce complicaciones al llenar el árbol por niveles ¹.

El llenado por niveles de un heap hace que sea natural representarlo usando un recorrido por niveles (o BFS, primero a lo ancho). La idea es usar un arreglo $H[1..n]$ tal que $H[i]$ almacena el valor del i -ésimo nodo en un recorrido por niveles. De esta forma, $H[1]$ representa al nodo raíz del heap, $H[2]$ y $H[3]$ representan a los dos hijos de la raíz, y así siguiendo, tal como lo muestra la Figura 13.3 para un heap particular. Este arreglo no sólo almacena los elementos del heap, sino que también

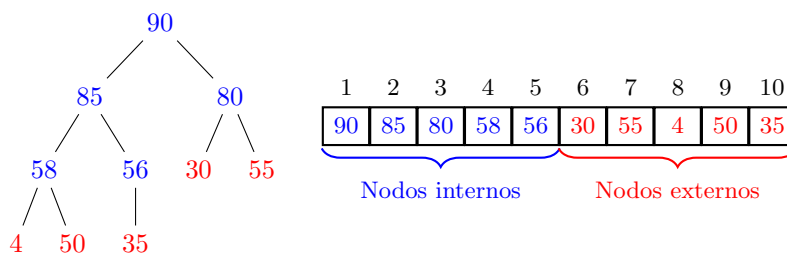


Figura 13.3: Un máx-heap para el conjunto $S = \{55, 4, 58, 90, 56, 30, 85, 80, 50, 35\}$ (izquierda) y su correspondiente representación por niveles usando un arreglo (derecha).

¹ Básicamente, se debe mantener una estructura de datos de tipo cola que almacena los nodos externos del árbol en un recorrido por niveles. Esto permite conocer los próximos puntos de inserción en el llenado por niveles.

la estructura del árbol de forma implícita, sin necesidad de almacenar punteros:

- Dado un nodo del heap almacenado en $H[i]$, sus dos hijos están almacenados en $H[2i]$ y $H[2i + 1]$. Esta propiedad se cumple porque el árbol es casi lleno.
- De la misma forma, un elemento almacenado en $H[i]$ tiene su padre representado en $H[\lfloor i/2 \rfloor]$.

Así, la estructura permite navegación hacia padres e hijos, ocupando espacio sólo para almacenar los elementos del conjunto. Este tipo de estructuras de datos recibe el nombre de *estructuras de datos implícitas*: requieren espacio adicional $O(1)$ por sobre el espacio necesario para almacenar el conjunto de datos.

Construcción de un Heap a Partir de un Arreglo de Elementos

Como vimos anteriormente, la operación de inserción en un heap de n elementos toma tiempo $O(\lg n)$. Para un conjunto S de n elementos, esa operación nos permite construir un máx-heap que lo represente en tiempo $O(n \lg n)$ (mediante sucesivas inserciones). Sin embargo, esto requiere espacio adicional para el heap. A continuación, mostramos que la construcción no sólo se puede hacer de forma in-place, sino que también en tiempo $\Theta(n)$.

Note que en un máx-heap representado en un arreglo H , los nodos almacenados en $H[1..\lfloor n/2 \rfloor]$ son nodos internos (todos tienen dos hijos, excepto $H[\lfloor n/2 \rfloor]$ que puede tener un único hijo), mientras que los nodos en el segmento $H[\lfloor n/2 \rfloor + 1..n]$ son nodos externos. Por ejemplo, para el heap de la Figura 13.3 se tiene que los elementos $H[1..5]$ son nodos internos del heap, mientras que $H[6..10]$ son externos.

La idea de la construcción de tiempo lineal es repetir, para cada uno de los nodos internos $H[1..\lfloor n/2 \rfloor]$, la operación heapify de la demostración de la Propiedad 13.2.1. Obviamente, este proceso debe hacerse en un orden que garantice que la operación recibe cada vez dos máx-heaps y un nodo que es padre de ambos heaps. El Algoritmo 41 implementa la operación heapify. El arreglo recibe un máx-heap $H[1..n]$, y un nodo

Algoritmo 41: HEAPIFY($H[1..n]$, i)

```

1 mayor ← i
2 izq ← 2i
3 der ← 2i + 1
4 if izq ≤ n and H[izq] > H[mayor] then
5   | mayor ← izq
6 end
7 if der ≤ n and H[der] > H[mayor] then
8   | mayor ← der
9 end
10 if mayor ≠ i then
11   | SWAP(H[i], H[mayor])
12   | HEAPIFY(H[1..n], mayor)
13 end
```

i tal que $1 \leq i \leq n$. Además, asume que los dos hijos del nodo i son máx-heaps. El procedimiento es el mismo explicado en la demostración de la Propiedad 13.2.1, y se realiza recursivamente.

Para la construcción del máx-heap, supongamos que el conjunto ha sido almacenado en un arreglo $S[1..n]$, en algún orden. Note que el arreglo por sí mismo puede ser considerado un árbol casi lleno almacenado por niveles, como ya hemos discutido. La idea es ordenar las ramas de este árbol, para que cumpla con las propiedades de un máx-heap. Note que cada uno de los nodos externos del máx-heap es un máx-heap por se. Por lo tanto, si invocamos al Algoritmo 41 con los padres de esos nodos externos (que están en el penúltimo nivel del árbol), lograremos que cada nodo en los últimos dos niveles del árbol sea la raíz de un máx-heap. El algoritmo procede de esta manera, retrocediendo por niveles, con la garantía de que sus nodos descendientes ya conforman máx-heaps. El proceso concluye una vez que llegamos a la raíz del árbol. El Algoritmo 42 muestra este proceso.

Algoritmo 42: CONSTRUIRMÁXHEAP($S[1..n]$)

```

1 for  $i \leftarrow \lfloor n/2 \rfloor$  downto 1 do
2   | HEAPIFY( $S[1..n], i$ )
3 end

```

Demostramos a continuación que la construcción de un heap con este algoritmo toma tiempo lineal.

Teorema 13.2.2 *Para un conjunto S de n elementos sobre los que se ha definido una función de orden total, el Algoritmo 42 permite construir un máx-heap en tiempo $\Theta(n)$.*

Demostración. Para simplificar la demostración, asumamos que $n = 2^k - 1$, para $k \geq 1$. Esto significa que el árbol está lleno en sus k niveles. Obviamente, la altura del árbol es $h = \lfloor \lg(2^k - 1) \rfloor$. Note que el Algoritmo 42 invoca al Algoritmo 41 $\lfloor n/2 \rfloor$ veces. En cada invocación de este último algoritmo, en el peor caso el elemento correspondiente va a moverse hasta el nivel h del árbol. En cada nivel se necesitan 2 comparaciones para determinar la rama por donde bajar (líneas 4 y 7 del Algoritmo 41). Eso significa que para un elemento que está en el nivel i del árbol, $0 \leq i < h$, el Algoritmo 41 realiza $2(h - i)$ comparaciones en el peor caso. Como el árbol está lleno en todos sus niveles, hay 2^i nodos en el nivel $i = 0, \dots, h - 1$. Por todo esto, el costo total de peor caso es:

$$\begin{aligned}
 \sum_{i=0}^{h-1} 2(h-i)2^i &= \sum_{i=0}^{h-1} h2^{i+1} - \sum_{i=0}^{h-1} i2^{i+1} \\
 &= 2n \lg n - \lg n - (-4n + 2n \lg n + 4) \\
 &= 4(n-1) - \lg n.
 \end{aligned}$$

Esto prueba que el tiempo total del Algoritmo 42 es $O(n)$. Para probar que en realidad es $\Theta(n)$, solo hace falta notar que realiza exactamente $\lfloor n/2 \rfloor$ iteraciones. ■

Modificando un Valor en un Heap

Aquí mostrar el algoritmo que permite modificar un valor en el heap. Luego esa función se podrá usar en capítulos posteriores para implementar Prim.

HeapSort

Mostramos a continuación cómo usar heaps para ordenar eficientemente un conjunto de n elementos representado por un arreglo $A[1..n]$. El algoritmo se conoce como **HeapSort**, y fue definido originalmente por J. W. J. Williams en 1964, cuya idea inicial es:

1. Traspasar los elementos de A a un arreglo $H[1..n]$;
2. construir un mín-heap sobre el arreglo H ; y
3. para $i = 1, \dots, n$, extraer el mínimo de H y almacenarlo en la posición $A[i]$.

El tiempo total para extraer el mínimo es $\sum_{i=1}^n O(\lg i) = O(n \lg n)$, por lo que logramos otro algoritmo de ordenamiento óptimo en el peor caso. Sin embargo, tenemos la desventaja de que no es in-place (al igual que **MergeSort**), ya que necesita el arreglo $H[1..n]$.

Estudiamos a continuación una variante de este algoritmo introducida por R. Floyd también en 1964, que tiene la ventaja de ser óptimo e in-place², lo que podría darle ventajas por sobre **MergeSort**. La idea principal es la siguiente:

1. Usar el Algoritmo 42 para construir un máx-heap (y no un mín-heap como sugerimos antes) sobre el mismo arreglo A en tiempo $O(n)$. Esto permite construir el máx-heap de forma in-place, y ahorrarnos el arreglo H ;
2. para $i = n, \dots, 2$, intercambiar los elementos $A[1]$ y $A[i]$;
3. invocar el Algoritmo 41 con el heap $A[1..i-1]$ y el nodo $A[1]$. Volver al paso 2.

Conceptualmente, para $i = n, \dots, 2$, este algoritmo divide al arreglo A en dos regiones: (a) la región izquierda que almacena un heap de tamaño i ; y (b) la región derecha, que almacena los $n-i$ elementos más grandes del conjunto de forma ordenada. De esta forma, la parte izquierda tiene inicialmente tamaño n , y la derecha tamaño 0. Para cada i , el paso 2 del algoritmo intercambia el i -ésimo mayor elemento del arreglo (que está en la raíz del heap), con el elemento que está en la posición i . Esto agrega un nuevo elemento a la parte derecha del arreglo. Para recuperar el orden del heap, en el paso 3 se usa la operación **heapify**, teniendo en cuenta que el heap ahora tiene $i-1$ elementos. Luego de repetir este proceso $n-1$ veces, el arreglo es ordenado. El Algoritmo 43 formaliza este proceso. Al igual que antes, el tiempo para **heapify** en cada paso es $O(\lg i)$, por lo que la suma total es $O(n \lg n)$.

² Floyd presenta este algoritmo como una variante in-place de su anterior algoritmo **TreeSort** (el cual mencionamos en la Sección 11.2. Vea en particular la nota al pie 3, en la página 208).

Algoritmo 43: *HeapSort*($A[1..n]$)

```
1 CONSTRUIRMÁXHEAP( $A[1..n]$ )
2 for  $i \leftarrow n$  downto 2 do
3   | SWAP( $A[1], A[i]$ )
4   | HEAPIFY( $A[1..i - 1], 1$ )
5 end
```

HeapSort en la Práctica

Aunque *HeapSort* es un algoritmo de ordenamiento óptimo e in-place, en general en la práctica es menos eficiente que *QuickSort* (más allá de que éste realiza $O(n^2)$ comparaciones) y *MergeSort* (que es óptimo pero no in-place). Una razón importante para esto es el patrón de acceso de *HeapSort* sobre el conjunto no es secuencial. Note cómo las distintas operaciones sobre el heap acceden a posiciones del arreglo que no necesariamente son contiguas. Eso es un mal patrón de acceso para las memorias cache que tienen los computadores actuales, en donde los accesos secuenciales a un arreglo son mucho más beneficiosos. *QuickSort* y *MergeSort* recorren los elementos de forma secuencial durante el proceso de ordenamiento y, por lo tanto, toman ventaja de ese tipo de memorias. El resultado es que ordenan de forma más rápida en la práctica.

Este tipo de consideraciones agrega otro punto a tener en cuenta en el diseño y análisis de algoritmos. Un algoritmo de ordenamiento óptimo respecto a la cantidad de comparaciones que realiza, no necesariamente implica un algoritmo más rápido en la práctica. Ya vimos esto mismo al comparar *QuickSort* con *MergeSort*, en donde el primero suele ser más rápido en la práctica porque en promedio realiza una cantidad de comparaciones óptima, y además realiza menos intercambios de elementos que *MergeSort*. Ahora, la razón de la diferencia en la práctica es el patrón de acceso a los elementos del arreglo. Todo esto debe tenerse en cuenta al momento de elegir un algoritmo de ordenamiento (aunque la lección se puede extrapolar a algoritmos en general). Sin embargo, y más allá de estos comentarios respecto al comportamiento en la práctica, siempre existirán situaciones en que se necesite ordenar un conjunto garantizando tiempo óptimo y sin usar espacio adicional. En tal caso, *HeapSort* debería ser elegido, por sobre *QuickSort* y *MergeSort*.



Figura 13.4: Los dos primeros pasos en la ejecución del algoritmo HeapSort, para el arreglo $A[1..10] = (55, 4, 58, 90, 56, 30, 85, 80, 50, 35)$. Al finalizar estos dos pasos, se puede ver cómo la parte derecha del arreglo contiene a los dos elementos mayores del arreglo.

14.1. Introducción

A menudo, los problemas de optimización se resuelven mediante algoritmos que toman decisiones a lo largo de una secuencia de pasos. Los algoritmos greedy (en castellano: avaro, codicioso) construyen la solución a un problema mediante una secuencia de pasos, cada uno de los cuales expande la solución parcial construida hasta el momento, hasta llegar a la solución completa. En cada paso, el algoritmo greedy toma una decisión (o hace una elección) que debe cumplir con las siguientes propiedades:

Factible: es decir, debe satisfacer las restricciones del problema.

Localmente óptima: debe ser la mejor elección entre todas las elecciones factibles en ese paso.

Irrevocable: una vez tomada una decisión, no puede ser cambiada en los siguientes pasos del algoritmo.

Es muy importante destacar el segundo punto, el de la decisión localmente óptima. Esto sugiere que el algoritmo toma la que parece ser la mejor decisión en cada paso, sin tener en cuenta pasos posteriores. Sólo usa información que posee al momento de tomar la decisión, lo que suele llevar a soluciones no óptimas.

Un ejemplo típico de estrategia greedy ocurre cuando compramos un producto como, por ejemplo, una bicicleta. Una decisión greedy típica es elegir la más barata de todas las bicicletas. Aquí no se piensa en, por ejemplo, la calidad o durabilidad, sino que sólo en minimizar la cantidad a pagar. En muchos casos, la bicicleta más barata suele dañarse más rápidamente que las otras, por lo que la decisión puede no haber sido la óptima. Note cómo privilegiamos minimizar el dinero gastado al momento de hacer la compra, y no el dinero gastado durante el tiempo en que necesitaremos usar la bicicleta. Eso es un ingrediente típico de un algoritmo greedy.

Al usar este tipo de estrategias, en general buscamos diseñar algoritmos eficientes basados en decisiones que son rápidas de tomar y cuyas soluciones son suficientemente buenas en general, aunque no necesariamente óptimas. En otras palabras, se sacrifica la garantía de optimalidad en las soluciones a cambio de algoritmos eficientes. Dado que normalmente este tipo de algoritmos necesita minimizar o maximizar en cada paso, es común el uso de estructuras de datos de tipo heap (de mínimo o máximo).

Un ejemplo que ilustra claramente una estrategia greedy es el siguiente. Considere n monedas dispuestas en una fila sobre la mesa. Las monedas podrían ser de distintas denominaciones. El problema consiste en tomar la mayor cantidad de dinero de la mesa, sujeto a la restricción de que cada vez que se tome una moneda, quedan invalidadas las dos

monedas que estaban inmediatamente contiguas a la moneda escogida. Por ejemplo, considere la siguiente fila de monedas:

$$1, 5, 10, 1, 5, 10, 50, 10, 1, 1.$$

Una estrategia greedy toma, sin dudas, la moneda de mayor denominación, en este caso la de 50 pesos. Eso invalida las dos monedas contiguas de 10 pesos. La situación es la siguiente:

$$1, 5, 10, 1, 5, \underbrace{10}_{\times}, \underbrace{50}_{\checkmark}, \underbrace{10}_{\times}, 1, 1.$$

A continuación, la estrategia greedy escoge la tercera moneda de la fila, de 10 pesos, invalidando las dos monedas contiguas. Luego se escoge la quinta moneda de la fila, de 5 pesos (cuyas monedas contiguas ya estaban invalidadas). Finalmente, los últimos dos pasos escogen las monedas de 1 peso de los extremos. El resultado final es el siguiente:

$$\underbrace{1}_{\checkmark}, \underbrace{5}_{\times}, \underbrace{10}_{\checkmark}, \underbrace{1}_{\times}, \underbrace{5}_{\checkmark}, \underbrace{10}_{\times}, \underbrace{50}_{\checkmark}, \underbrace{10}_{\times}, \underbrace{1}_{\times}, \underbrace{1}_{\checkmark}.$$

El total recogido es de 67 pesos, lo cual es la cantidad óptima para este ejemplo.

Parece ser que esta estrategia greedy es, después de todo, óptima. Sin embargo, no lo es en general. El hecho de tomar siempre la moneda de mayor denominación, sin considerar el valor de las monedas contiguas que está invalidando, llevan al algoritmo a obtener soluciones sub óptimas. Considere el siguiente ejemplo, que de forma muy simple ilustra este hecho:

$$\underbrace{25}_{\times}, \underbrace{50}_{\checkmark}, \underbrace{26}_{\times}.$$

La estrategia greedy escoge la moneda de 50 pesos e invalida las otras dos, mientras que la decisión óptima consiste en tomar las dos monedas de menor denominación, pero de mayor valor final: 51 pesos.

Ejercicio 14.1 Implemente esta estrategia greedy de forma eficiente. Use las estructuras de datos que considere necesarias para evitar recorrer la fila de monedas buscando el máximo cada vez (que tomaría tiempo total $\Theta(n^2)$). Compare la eficiencia de su solución con la obtenida en el capítulo de fuerza bruta, en donde se usó una estrategia de tiempo exponencial para encontrar el óptimo.

En el próximo capítulo, dedicado a la técnica de programación dinámica, estudiaremos cómo encontrar la cantidad óptima de dinero para este problema, en tiempo $\Theta(n)$.

14.2. El Problema de Dar Cambio

Considere el problema de dar cambio para una cantidad de dinero n usando la menor cantidad posible de monedas con denominaciones $d_1 > d_2 > \dots > d_m$. Por ejemplo, $d_1 = 25$, $d_2 = 10$, $d_3 = 5$, y $d_4 = 1$. ¿Cómo se daría cambio para, por ejemplo, 48 pesos, usando

la menor cantidad posible de monedas? Una estrategia greedy intenta dar monedas de 25 pesos primero, tantas como pueda. Luego sigue con las monedas de 10 pesos, y así hasta dar completamente el cambio. El resultado es: 1 moneda de 25, 2 monedas de 10, 3 monedas de 1, un total de 6 monedas. Esta estrategia es óptima para esas denominaciones, aunque no es óptima en general: para las denominaciones $d_1 = 25$, $d_2 = 10$, $d_3 = 1$, suponga que necesita dar cambio de 30 pesos. La estrategia usa 6 monedas (1 de 25, 5 de 1). Lo óptimo para estas denominaciones son 3 monedas (simplemente, 3 monedas de 10).

14.3. División de un Texto en Líneas

Dado un texto de n palabras p_1, p_2, \dots, p_n , de longitudes l_1, l_2, \dots, l_n , y dados un ancho de línea L y una amplitud óptima b para los espacios en blanco que separan las palabras, se quiere dividir el texto en líneas para su impresión. Los espacios de amplitud óptima b pueden ampliarse o reducirse, para acomodar las palabras en una línea. Sin embargo, modificar la amplitud de los espacios tiene una penalización: Fijar los espacios en un valor b' para la línea que contiene las palabras p_i, p_{i+1}, \dots, p_j tiene una penalización $(j-i)|b'-b|$ siendo $b' = (L - l_i - l_{i+1} - \dots - l_j)/(j-i)$. El objetivo es dividir el texto en líneas de manera que la penalización total sea mínima.

Una solución greedy a este problema comienza agregando las palabras p_1, p_2, \dots , en la línea actual usando el espacio óptimo de tamaño b para separarlas, hasta que una palabra p_a no quepa en la línea actual. Es decir:

$$l_1 + l_2 + \dots + l_a + (a-1)b > L.$$

En este punto se debe tomar una decisión:

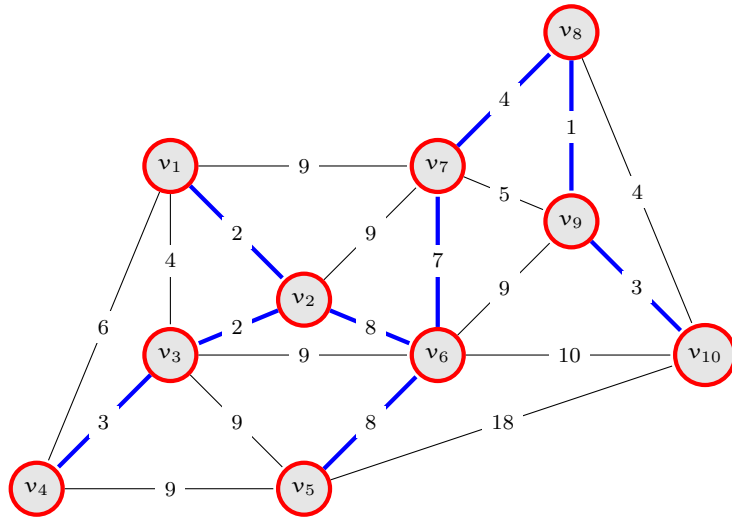
1. Imprimir p_a en la línea actual, reduciendo la amplitud de los espacios en blanco.
2. Imprimir p_a en la línea siguiente, ampliando la amplitud de los espacios en blanco.

La solución greedy elige la alternativa que produce la menor penalización. Aunque esta solución es eficiente (toma tiempo $\Theta(n)$), produce soluciones sub óptimas.

14.4. Árboles Recubridores Mínimos en Grafos

En un grafo, es común encontrar caminos redundantes, los que permiten alcanzar un nodo de diferentes formas. Si en un grafo conexo se eliminan caminos redundantes, el grafo sigue siendo conexo. En aplicaciones como, por ejemplo, el diseño de redes de comunicación, uno quiere mantener la conectividad del grafo reduciendo las conexiones redundantes. Esto podría significar una reducción de costos al instalar una menor cantidad de conexiones en la red.

Figura 14.1: Un ejemplo de árbol recubridor mínimo (arcos en azul) para un grafo dado.



Recordemos que en teoría de grafos un árbol se define como un grafo conexo y sin ciclos.

Definición 14.4.1 Dado un grafo conexo $G = (V, E)$, un árbol recubridor es un subgrafo de G con el mismo conjunto de vértices V tal que es un árbol.

Definición 14.4.2 Dado un grafo conexo con pesos $G = (V, E)$, un árbol recubridor mínimo es un árbol recubridor de G cuya suma de pesos asociados a los arcos es mínimo.

La Figura 14.1 muestra un grafo y su árbol recubridor mínimo. Para un grafo en particular, pueden existir varios árboles recubridores mínimos diferentes. Obviamente, todos esos árboles tendrán el mismo peso mínimo. Las siguientes dos observaciones serán importantes para diseñar algoritmos eficientes que resuelvan el problema.

Observación 14.4.1 Dado un árbol recubridor mínimo T de un grafo G , considere cualquier arco (u, v) del árbol. Sean T_1 y T_2 dos árboles incluidos en T y generados si eliminamos el arco (u, v) . De entre todos los arcos que podrían unir a T_1 y T_2 , el arco (u, v) es el de menor peso.

Verifique esta observación con el grafo de la Figura 14.1. Por ejemplo, considere el único arco de peso 1 del árbol recubridor mínimo. Al eliminar ese arco, el árbol recubridor mínimo se divide en dos árboles: uno que contiene un único arco, al que llamamos T_1 , y otro árbol de 7 arcos, al que llamamos T_2 . Note que hay 6 arcos que podrían unir a T_1 y T_2 , con pesos: 4, 1, 5, 9, 10, y 18. De todos esos, el de peso 1 es el único que pertenece al árbol recubridor mínimo. Esto puede repetirse para cada uno de los arcos del árbol.

Observación 14.4.2 Dado un árbol recubridor mínimo T de un grafo G , considere cualquier arco (u, v) de G que no haya sido incluido en T . Si (u, v) se agrega a T , se forma exactamente un ciclo, y el peso de (u, v) es mayor o igual a todos los pesos de los arcos que conforman el ciclo.

Verifique esta observación nuevamente usando el grafo de la Figura 14.1. Por ejemplo, considere los únicos dos arcos de peso 2 del grafo, que también son parte del árbol recubridor mínimo. Si agregamos el arco de peso 4 que une a dos de los extremos de esos arcos, se forma un único ciclo, y todos los arcos de dicho ciclo son de peso menor a 4.

Una solución simple es buscar exhaustivamente el árbol de peso mínimo dentro del espacio de posibles árboles recubridores. Esto implica generar todos los árboles posibles, que es una cantidad exponencial y por lo tanto no práctico. A continuación estudiamos dos algoritmos greedy eficientes, basados en las Observaciones 14.4.1 y 14.4.2.

El Algoritmo de Prim

Consideremos primero el algoritmo de Prim, que fue originalmente definido por el matemático Vojtěch Jarník en 1930, y que fue redescubierto de forma independiente por Robert Prim en 1957, y luego por Edsger Dijkstra en 1959. El método usa una estrategia greedy para encontrar el árbol recubridor de peso mínimo de un grafo dado. El árbol es construido de forma incremental por el algoritmo, comenzando por un árbol vacío, al que se agrega un nuevo arco (y por ende un nuevo vértice) en cada paso. El algoritmo requiere como entrada un grafo con pesos G , y un vértice r de G . Ese vértice es a partir del cual se construye el árbol recubridor mínimo. El mismo puede ser cualquier vértice del grafo, no hay ninguna restricción para ello.

En términos generales, el algoritmo funciona de la siguiente manera. Digamos que T denota el árbol recubridor mínimo que es construido por el algoritmo. Inicialmente, T contiene únicamente el vértice r , y no contiene arcos. En cada paso, el algoritmo considera todos los arcos que son adyacentes a un vértice que ya está en T , tal que el otro vértice aún no está en T . Esta restricción garantiza que no se formen ciclos al agregar arcos al árbol. De todos los posibles arcos que cumplen con esa condición, el algoritmo agrega a T el de menor peso. Esta es una decisión greedy que lleva al algoritmo a encontrar el árbol recubridor mínimo, tal como se demuestra a continuación.

La correctitud del algoritmo puede ser probada usando la Observación 14.4.1 de la siguiente manera: suponga un arco (u, v) del árbol recubridor mínimo encontrado por el algoritmo de Prim. Considere los dos árboles recubridores mínimos T_1 y T_2 formados si se quita el arco (u, v) de T . Sin pérdida de generalidad, supongamos que al momento en que el algoritmo agregó el arco (u, v) a T , el árbol recubridor mínimo calculado hasta ese punto era T_1 (podría ser T_2 , el análisis es simétrico). Note que si el algoritmo agregó el arco (u, v) a T en este paso, eso significa que ese era el arco de menor peso de entre todos los que unen a T_1 con T_2 . Por lo tanto, (u, v) debe pertenecer a T (recuerde la Observación 14.4.1). Dado que esto ocurre con cada uno de los arcos agregados al árbol, el algoritmo calcula el árbol recubridor mínimo de forma correcta.

Estudiamos a continuación la implementación del algoritmo. Para poder ejecutar de forma eficiente el proceso antes descrito, los pasos a seguir no serán exactamente como se explicaron. Sin embargo, el proceso es conceptualmente el mismo. Sea $\text{key}[1..|V|]$ un arreglo que almacena,

para cada vértice u de G , el menor peso de un arco incidente en u que se haya visto desde el inicio del algoritmo hasta el paso actual, tal que el otro extremo (digamos v) de ese arco ya pertenezca al árbol recubridor mínimo. Sobre este arreglo construimos un heap de mínimos, al que llamamos Q . Además, almacenaremos un arreglo $\pi[1..|V|]$, tal que si en un paso dado del algoritmo tenemos que $\pi[u] = v$, esto significa que el arco (u, v) (cuyo peso es $\text{key}[u]$) pertenece a T .

Inicialmente, el algoritmo asigna $\text{key}[u] \leftarrow +\infty$, para todo $u = 1, \dots, |V|$, excepto para el vértice inicial, para el cual $\text{key}[r] \leftarrow 0$. En cada paso, el algoritmo extrae de Q el vértice u cuyo valor actual $\text{key}[u]$ sea el mínimo entre todos los vértices que aún están en Q . Dada la inicialización antes mencionada, el primer vértice en ser extraído de Q es r , el vértice de inicio. Luego de extraer el vértice u de Q , se consideran todos los arcos (u, v) del grafo (esto es, todos los arcos incidentes en u). Por cada vecino v de u tal que $v \in Q$ (lo cual es importante para evitar generar ciclos al agregar arcos al árbol), se compara el peso $w(u, v)$ del arco (u, v) con el valor $\text{key}[v]$. Si ocurre que $w(u, v) < \text{key}[v]$, hemos encontrado un arco de menor peso con el que alcanzamos a v . Por lo tanto, actualizamos $\pi[v] \leftarrow u$ y $\text{key}[v] \leftarrow w(u, v)$. Esto último implica reorganizar el heap Q , ya que $\text{key}[v]$ ha cambiado. El pseudocódigo puede verse en el Algoritmo 44.

Algoritmo 44: PRIM(grafo $G(V, E)$, nodo r)

```

1 for  $u \in V$  do
2    $\text{key}[u] \leftarrow +\infty$ 
3    $\pi[u] = \text{NULL}$ 
4 end
5  $\text{key}[r] \leftarrow 0$ 
6  $Q \leftarrow \text{BUILDHEAP}(V, \text{key})$ 
7 while  $Q \neq \emptyset$  do
8    $u \leftarrow \text{EXTRACTMIN}(Q)$ 
9   for  $v \in \text{ADJ}(u)$  do
10    if  $v \in Q$  and  $w(u, v) < \text{key}[v]$  then
11       $\pi[v] \leftarrow u$ 
12       $\text{key}[v] \leftarrow w(u, v)$     // Debe actualizar Q
13    end
14  end
15 end

```

Para analizar el tiempo de ejecución del algoritmo, primero hay que notar que el heap Q tiene inicialmente $|V|$ elementos. Entonces, cada invocación a las funciones `EXTRACTMIN` y la actualización de la línea 12 del Algoritmo 44 toma tiempo $O(\lg |V|)$. La construcción de Q en la línea 6, por su parte, toma tiempo $O(|V|)$. Luego, el **while** de la línea 7 se ejecuta $|V|$ veces, ya que en cada iteración se extrae un vértice de Q , y dentro de la iteración no se agregan vértices a Q . Por lo tanto, el tiempo total de invocación a la función `EXTRACTMIN` es $O(|V| \lg |V|)$. El **for** de la línea 9 se ejecuta, en total, $|E|$ veces, dado que su función es recorrer uno a uno los arcos del grafo. En el peor caso, la condición del **if** de la línea 10 es verdadera. Esto significa que, por cada arco del grafo, la línea 12 debe actualizar el heap en tiempo $O(\lg |V|)$ cada vez. El tiempo total de dicho **for** es $O(|E| \lg |V|)$, por lo que el tiempo total de ejecución del algoritmo es $O((|V| + |E|) \lg |V|)$.

Al finalizar la ejecución del algoritmo, los arcos del árbol recubridor mínimo está representado por el conjunto $\{(v, \pi[v]) \mid v \in V - \{r\}\}$.

El Algoritmo de Kruskal

Estudiamos a continuación el algoritmo definido por Joseph Kruskal en 1956, para calcular el árbol recubridor mínimo de un grafo $G = (V, E)$. Éste también es un algoritmo greedy que, al igual que el de Prim, encuentra la solución óptima al problema. La idea es construir un heap de mínimos Q que almacena los $|E|$ arcos del grafo. Note la diferencia con el heap del algoritmo de Prim. En cada paso, el algoritmo extrae de Q el arco de menor peso, y lo agrega al árbol recubridor mínimo si y sólo si no se forma un ciclo. El siguiente teorema, que está asociado a la Observación 14.4.2, permite demostrar la correctitud del algoritmo.

Teorema 14.4.3 *Sea $G = (V, E)$ un grafo conexo con pesos. Si el peso de un arco $e \in E$ es mayor o igual que el peso de cualquier otro arco en un ciclo que contenga a e , entonces e no puede pertenecer a ningún árbol recubridor mínimo de G .*

Si al considerar un arco durante la ejecución del algoritmo, éste forma un ciclo, note que su peso es mayor o igual al peso de cada uno de los arcos que forman ese ciclo. Esto es porque los arcos son considerados en orden creciente de sus pesos. El teorema permite demostrar que dicho arco no puede ser parte del árbol recubridor mínimo y, por lo tanto, debe ser descartado.

Un punto clave en la implementación del algoritmo de Kruskal es el chequeo que determine si un nuevo arco agrega un ciclo o no. Conceptualmente, el algoritmo de Kruskal comienza dividiendo los vértices del grafo en $|V|$ clases de equivalencia (cada vértice es una clase en sí misma). Luego, se procesan los arcos en orden de peso. Un arco es agregado al árbol recubridor mínimo si une dos vértices que están en clases de equivalencia diferentes. Esto indicará que no se forma un ciclo. Luego de agregar un arco, las dos clases de equivalencia a las que pertenecían los vértices del arco se transforman en una única clase de equivalencia. Este proceso se repite hasta que existe una única clase de equivalencia, la cual contendrá a todo el grafo (nuevamente, asumimos un grafo conexo).

Ejemplo 14.4.1 Considere el grafo $G = (V, E)$ tal que $V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$ y $E = \{(v_1, v_2, 6), (v_1, v_3, 8), (v_1, v_4, 6), (v_1, v_6, 3), (v_2, v_3, 5), (v_3, v_4, 7), (v_3, v_6, 6), (v_4, v_5, 2), (v_4, v_6, 5), (v_5, v_6, 4)\}$, en donde la tercera componente de cada arco corresponde a su peso. Inicialmente, las clases de equivalencia definidas por el algoritmo son las siguientes:

$\{v_1\}; \quad \{v_2\}; \quad \{v_3\}; \quad \{v_4\}; \quad \{v_5\}; \quad \{v_6\}.$

Se han usado diferentes colores para diferenciar las clases de equivalencia. Luego, el algoritmo comienza a extraer los arcos ordenados por sus pesos, de menor a mayor. El primer arco obtenido es (v_4, v_5) ,

de peso 2. Dado que los extremos v_4 y v_5 tienen diferente color (es decir, están en distintas clases de equivalencias), el arco es agregado al árbol recubridor mínimo, y ambos vértices ahora pertenecen a la misma clase de equivalencia. Las clases de equivalencia resultantes son:

$$\{v_1\}; \quad \{v_2\}; \quad \{v_3\}; \quad \{v_4, v_5\}; \quad \{v_6\}.$$

Luego, se extrae el arco (v_1, v_6) , y como sus extremos tienen colores diferentes, el mismo es agregado al árbol, obteniendo las siguientes clases de equivalencia:

$$\{v_1, v_6\}; \quad \{v_2\}; \quad \{v_3\}; \quad \{v_4, v_5\}.$$

Luego, el arco (v_5, v_6) es extraído y agregado al árbol, porque dichos vértices tienen distinto color. Se obtienen las siguientes clases de equivalencia:

$$\{v_2\}; \quad \{v_3\}; \quad \{v_1, v_6, v_4, v_5\}.$$

A continuación, se extrae el arco (v_4, v_6) , cuyos extremos están ambos en la misma clase de equivalencia. Eso indica que agregar este arco formaría un ciclo, por lo tanto debe ser descartado. Luego se considera (v_2, v_3) , que es agregado al árbol y se obtienen las clases de equivalencia:

$$\{v_2, v_3\}; \quad \{v_1, v_6, v_4, v_5\}.$$

Finalmente, se extrae (v_3, v_6) , el cual es agregado al árbol recubridor mínimo, obteniendo la única clase de equivalencia:

$$\{v_2, v_3, v_1, v_6, v_4, v_5\},$$

con lo cual el algoritmo finaliza.

Mostramos a continuación cómo implementar eficientemente una familia de clases de equivalencias como la mostrada en el ejemplo, de manera de poder determinar en qué clase está un vértice dado (esto es, poder determinar el color actual de un vértice), y unir dos clases de equivalencia.

Union/Find

Definimos a continuación un tipo de datos abstracto (TDA) que permite representar una colección de subconjuntos disjuntos (las clases de equivalencia vistas anteriormente). Dada una familia S_1, S_2, \dots, S_k de conjuntos disjuntos que son una partición de un conjunto $S = \{x_1, \dots, x_n\}$ de n elementos, dicho TDA implementa las siguientes operaciones:

- **makeset(x)**: construye el conjunto $\{x\}$. Sólo puede ejecutarse una única vez por cada elemento $x \in S$.
- **find(x)**: retorna el subconjunto que contiene actualmente a x .
- **union(x, y)**: construye la unión de los subconjuntos disjuntos **find(x)** y **find(y)**, y agrega el nuevo conjunto a la colección (de la cual elimina los conjuntos **find(x)** y **find(y)**).

Un TDA que implementa dichas operaciones es conocido como Union/-Find. Para facilitar su implementación, asumiremos que $S = \{1, \dots, n\}$. Sin embargo, en general este tipo de estructura de datos se puede aplicar a cualquier conjunto de n elementos que pueda ser enumerado.

Ejemplo 14.4.2 Consideremos $S = \{1, 2, 3, 4, 5, 6\}$. Simularemos las operaciones ejecutadas por el algoritmo de Kruskal en el Ejemplo 14.4.1, pero esta vez usando el TDA Union/Find. Inicialmente, si aplicamos la operación `makeset` para cada $x \in S$ obtenemos las clases de equivalencia:

$\{1\}; \{2\}; \{3\}; \{4\}; \{5\}; \{6\}.$

Luego, el primer arco que se extrae es (v_4, v_5) . Dado que $\text{find}(4) \neq \text{find}(5)$, tenemos que v_4 y v_5 están en distintas clases de equivalencias, por lo que podemos aplicar `union(4, 5)` para obtener:

$\{1\}; \{2\}; \{3\}; \{4, 5\}; \{6\}.$

El proceso continúa de esta manera, hasta obtener una única clase de equivalencia.

A continuación, estudiamos dos maneras de implementar el TDA Union/Find.

Implementación Basada en Listas Esta implementación representa cada subconjunto S_i con una lista enlazada. El primer elemento de cada lista es el representante del subconjunto. Este concepto es importante, ya que será el elemento que “*le da nombre*” al subconjunto. Cada nodo de una lista contiene un elemento x , un puntero al siguiente nodo de la lista, y un puntero al representante (es decir, un puntero al primer nodo de la lista). Las listas tienen orden arbitrario (más adelante veremos que depende del orden en que se aplican las operaciones sobre el TDA). Además, se mantiene un puntero al último elemento de cada lista (llamamos `tail` a ese puntero).

La operación `makeset(x)` simplemente crea una nueva lista con un único nodo. El puntero al primer nodo de la lista es un puntero al mismo nodo (i.e., un loop). La operación `find(x)` devuelve el puntero al primer nodo de la lista. Si para dos elementos distintos x e y el puntero al primer nodo de sus listas es el mismo, significa que ambos están dentro de la misma clase de equivalencia. Si los punteros son distintos, significa que están en clases de equivalencia distintas. Ambas operaciones se implementan en tiempo $O(1)$, asumiendo que `malloc` (necesario para crear el nodo en la operación `makeset`) toma tiempo constante.

La operación `union(x, y)`, por otro lado, requiere algo más de trabajo. En esta implementación, se resuelve concatenando listas enlazadas. El tema es en qué orden deberían concatenarse las listas. Una manera simple es concatenar la lista del elemento x detrás de la lista del elemento y (o viceversa). Esto puede hacerse modificando el puntero del último nodo de la lista de y , para que apunte al primer nodo de la lista de x . Esto puede hacerse en tiempo $O(1)$ usando el puntero `tail`. El representante de la lista resultante es el de la lista de y . Por lo tanto,

deben actualizarse todos los punteros al representante de los elementos que estaban en la lista de x , apuntando ahora al primer nodo de la lista de y . Esto toma tiempo lineal en el tamaño de la lista de x , lo que es el proceso más costoso en la implementación de la operación. Dado que la decisión de concatenación que hemos asumido es tan rígida, podemos tener la siguiente secuencia de n operaciones para $S = \{x_1, \dots, x_n\}$:

- $\text{makeset}(x_1)$,
- \vdots
- $\text{makeset}(x_n)$,

seguido por las siguientes $n - 1$ operaciones:

- $\text{union}(x_1, x_2)$,
- $\text{union}(x_2, x_3)$,
- $\text{union}(x_3, x_4)$,
- \vdots
- $\text{union}(x_{n-1}, x_n)$.

En total, son $2n - 1$ operaciones. Note que la operación $\text{union}(x_1, x_2)$ requiere 1 cambio de puntero al primer nodo. Luego, la operación $\text{union}(x_2, x_3)$ requiere de 2 cambios de punteros. La operación $\text{union}(x_3, x_4)$, por su parte, requiere de 3 cambios de punteros. Y así siguiendo, hasta la operación $\text{union}(x_{n-1}, x_n)$ que requiere $n - 1$ cambios de punteros. El costo total de estas $n - 1$ operaciones es $\sum_{i=1}^{n-1} i \approx \frac{n^2}{2}$. Dado que el costo total de las n operaciones makeset es $\Theta(n)$, las $2n - 1$ operaciones toman tiempo $\Theta(n^2)$, lo que amortizado da tiempo lineal (en n) por operación. Eso es excesivo y haría que, por ejemplo, el algoritmo de Kruskal tome tiempo cuadrático.

Para solucionar este problema normalmente se usa la heurística *weighted-union*, en donde el largo de las listas es tenido en cuenta para concatenarlas. La idea de la heurística es concatenar la lista más corta detrás de la lista más larga. Para implementarla eficientemente sólo necesitamos almacenar la longitud de cada lista (además, por supuesto, de todos los datos antes mencionados). Esta idea tan simple permite minimizar la cantidad de cambios de punteros que se hacen por cada union . Pero, además, permite realizar de forma eficiente secuencias de operaciones sobre el TDA Union/Find, tal como lo demuestra el siguiente teorema.

Teorema 14.4.4 *Al emplear una representación de listas enlazadas para los conjuntos disjuntos y al usar la heurística *weighted-union*, una secuencia de m operaciones makeset , union , y find , n de las cuales corresponden a la operación makeset , toma tiempo $O(m + n \lg n)$.*

Demostración. Para cada elemento $x \in S$, calcularemos una cota superior para el número de veces en que su puntero al representante del conjunto ha sido actualizado. Dado que usamos la heurística *weighted-union*, el puntero de x se actualiza cuando estaba en el conjunto más pequeño a ser unido. Suponga que ejecutamos $\text{union}(x, y)$. La primera vez que se actualiza el puntero de x , significa que el conjunto del elemento y tenía al menos un elemento, resultando la unión en un conjunto de al menos 2 elementos. La segunda vez que se modifica el

puntero asociado a x (que, como dijimos, ya está en un conjunto de al menos 2 elementos), significa que el conjunto de y tiene al menos 2 elementos también, por lo tanto la unión produce un conjunto de al menos 4 elementos. En general, para cualquier $k \leq n$, después de que el puntero ha sido actualizado $\lceil \lg k \rceil$ veces, el conjunto resultante (y que contiene a x) tiene al menos k elementos. Como tenemos n conjuntos iniciales (creados con `makeset`), eso significa que una de las relaciones de equivalencia tendrá a lo más n elementos, por lo que el puntero al representante de cada elemento x puede haber sido actualizado a lo más $\lceil \lg n \rceil$ veces a lo largo de todo el proceso. El tiempo total para las n operaciones es $O(m + n \lg n)$, en donde $O(m)$ viene de las operaciones `makeset` y `find`, que cuestan tiempo $O(1)$ cada una. ■

Note que al usar esta implementación de Union/Find, el algoritmo de Kruskal toma tiempo $O(|E| \lg |E|)$.

Implementación Basada en Bosque En esta representación, cada clase de equivalencia es un árbol enraizado, por lo que toda la estructura conforma un bosque. Cada nodo de un árbol almacena un elemento $x \in S$. Cada nodo en un árbol apunta solamente a su padre. La raíz de cada árbol contiene al representante del subconjunto, cuyo padre es él mismo. Este loop permitirá saber que se alcanzó la raíz de un árbol cuando subamos por sus nodos.

La operación `makeset` crea un nuevo árbol con un único nodo, en tiempo constante. La operación `find(x)`, por otra parte, debe seguir punteros al padre, comenzando desde el nodo correspondiente a x y hasta llegar a la raíz del árbol correspondiente. A diferencia de la implementación de listas enlazadas, esta operación toma tiempo $O(n)$: piense en un caso en que uno de los árboles del bosque tiene altura $O(n)$. Para mejorar el tiempo de ejecución de esta operación, normalmente se usa la técnica conocida como *path compression*. La idea es que luego de ejecutar la operación `find`, se modifican los punteros al padre de todos los nodos visitados durante el recorrido, haciendo que ahora apunten al nodo raíz del árbol. Note que, de esta forma, la siguiente operación `find` sobre cualquiera de esos nodos tomará tiempo $O(1)$.

Finalmente, para implementar `union(x, y)`, la raíz de uno de los árboles debe apuntar a la raíz del otro. Similar a la heurística *weighted-union* de la implementación de listas enlazadas, existe la heurística *union by rank*, que busca reducir lo más posible la altura de los árboles resultantes de una unión de subconjuntos. La idea es hacer que la raíz del árbol con menor altura apunte a la raíz del árbol con mayor altura.

Se puede demostrar que el costo total de m operaciones al usar *union by rank* (pero no *path compression*) es $O(m \lg n)$. Además, se puede demostrar que si se usa la heurística *path compression* (pero no *union by rank*), n operaciones `makeset` (por ende a lo más $n - 1$ operaciones `union`) y f operaciones `find` toman tiempo $O(n + f(1 + \lg_{2+f/n}(n)))$. Finalmente, si se usan ambas heurísticas, el tiempo total para m operaciones es $O(m\alpha(n))$, en donde $\alpha(n)$ es la inversa de la función de Ackermann. Dicha función crece muy lentamente, como puede verse a

continuación:

$$\alpha(n) = \begin{cases} 0, & \text{si } 0 \leq n \leq 2, \\ 1, & \text{si } n = 3, \\ 2, & \text{si } 4 \leq n \leq 7, \\ 3, & \text{si } 8 \leq n \leq 2047, \\ 4, & \text{si } 2048 \leq n \leq 10^{80}. \end{cases}$$

La Figura 14.2 muestra una posible implementación (en C++) del TDA Union/Find, usando árboles. Dicha implementación usa las heurísticas path compression y union by rank. Queda como ejercicio entender cómo se implementan esas heurísticas.


```
1 class UnionFind {
2     private:
3         int* parent;
4         int* rank;
5     public:
6         UnionFind(int N) {
7             parent = new int[N];
8             rank = new int[N];
9             for (int i=0; i < N; i++)
10                 parent[i] = rank[i] = -1;
11         }
12
13         void makeset(int i) {
14             parent[i] = i;    // loop en el nodo raíz
15             rank[i] = 0;
16         }
17
18         int find(int i) {
19             if (parent[i] == i)
20                 return i;
21             else {
22                 // lo siguiente implementa path
compression
23                 parent[i] = find(parent[i]);
24                 return parent[i];
25             }
26         }
27
28         void union(int i, int j) {
29             int iSet = find(i);
30             int jSet = find(j);
31             if (iSet != jSet) {
32                 if (rank[iSet] > rank[jSet])
33                     parent[jSet] = iSet;
34                 else {
35                     parent[iSet] = jSet;
36                     if (rank[iSet] == rank[jSet])
37                         rank[jSet]++;
38                 }
39             }
40         }
41     };
```

Figura 14.2: Implementación del TDA Union/Find usando bosque.

Ejercicios

1. ¿Cuáles de los siguientes conjuntos de monedas (en pesos) permiten dar vuelto de manera óptima usando la estrategia greedy propuesta? Suponga que tiene una cantidad ilimitada de monedas de cada tipo. Si para un conjunto no puede resolver el problema de manera óptima, muestre la cantidad más pequeña V de pesos con el que falla.
 - a) $\{10, 7, 5, 4, 1\}$.
 - b) $\{64, 32, 16, 8, 4, 2, 1\}$.
 - c) $\{13, 11, 7, 5, 3, 2, 1\}$.
 - d) $\{7, 6, 5, 4, 3, 2, 1\}$.
 - e) $\{21, 17, 11, 10, 1\}$.
2. **El Problema de los Dragones y los Caballeros:** Suponga que hay n cabezas de dragón y m caballeros. Cada cabeza de dragón tiene un diámetro, y cada caballero tiene una altura. Una cabeza de dragón de diámetro d puede ser cortada por un caballero de altura a si y sólo si $d \leq a$. Un caballero puede cortar a lo más una cabeza de dragón. Dado un arreglo $D[1..n]$ con los diámetros de las cabezas de dragón, y un arreglo $C[1..m]$ con las alturas de los caballeros, ¿Es posible cortar todas las cabezas de dragón? Diseñe una estrategia greedy para responder la pregunta.
3. **El Problema de la Mochila Fraccional:** Dados n elementos e_1, \dots, e_n , con pesos p_1, \dots, p_n y valores v_1, \dots, v_n , y dada una mochila capaz de albergar hasta un máximo de peso W , se quiere encontrar la fracción x_1, \dots, x_n ($0 \leq x_i \leq 1$) de cada uno de los elementos que tenemos que introducir en la mochila, de manera que el valor total de los elementos que podemos transportar en la mochila sea el máximo posible. Demuestre si su solución permite resolver el problema de forma óptima. En otro caso dé un ejemplo en donde su algoritmo falla en encontrar la solución óptima.
4. **El Problema de la Mochila (0, 1):** En su versión clásica, el problema de la mochila no permite fraccionar los elementos a transportar. Sólo pueden llevarse completos, o no se llevan. Es decir, $x_i = 1$ o $x_i = 0$. La solución propuesta en el ejercicio anterior, ¿Es válida para esta variante?
5. **El Problema de Planificar Trabajos (jobs) en un Sistema Operativo:** En un sistema operativo de único procesador se quieren planificar n jobs de duraciones t_1, \dots, t_n . Los jobs pueden ser ejecutados en cualquier orden, uno por vez. Se quiere encontrar el orden de ejecución que minimice el tiempo de espera de todos los jobs en el sistema. El tiempo de espera de un job es la suma de los tiempos de ejecución de los jobs que estaban antes de él en el orden, más el tiempo de ejecución del job. Diseñe un algoritmo greedy que solucione este problema. ¿Es su algoritmo óptimo?
6. La noción de árbol recubridor mínimo es aplicable a grafos conexos con pesos en sus arcos.
 - a) ¿Es posible aplicar Prim en el caso en que el grafo sea no conexo, o es necesario chequear antes la conectividad del mismo?
 - b) Un bosque recubridor mínimo es un bosque cuyos árboles son recubridores mínimos de las componentes conexas del grafo. ¿Qué cambios hay que hacer (de ser necesarios) al

algoritmo de Kruskal para que pueda encontrar el bosque recubridor mínimo de un grafo arbitrario?

7. Indicar cuáles de las siguientes afirmaciones son verdaderas y cuáles falsas, justificando en cada caso.
 - a) Si e es una arista de peso mínimo en un grafo conexo, entonces debe estar entre las aristas de al menos un árbol recubridor mínimo del grafo.
 - b) Si e es una arista de peso mínimo en un grafo conexo, entonces debe estar entre las aristas de cada árbol recubridor mínimo del grafo.
 - c) Si los pesos de las aristas de un grafo son todos distintos, el grafo debe tener exactamente un árbol recubridor mínimo.
 - d) Si los pesos de las aristas de un grafo no son todos distintos, el grafo debe tener más de un árbol recubridor mínimo.
8. Implemente la función `changeValue(heap *H, int i, int v)`, la cual permite modificar el valor del i -ésimo nodo de un heap (en un recorrido por niveles), sumando v al valor actual del nodo. Analice su solución, dando una cota asintótica para el tiempo de ejecución.
9. Suponga que se usa `Union/Find` para mantener familias de conjuntos disjuntos del universo $\mathcal{S} = \{x_1, \dots, x_n\}$ de n elementos. Suponga la implementación que usa listas enlazadas. Dados dos subconjuntos disjuntos S_x y S_y que contienen a los elementos x e y , respectivamente, asuma que se implementa `union(x, y)` concatenando la lista de S_y detrás de la lista de S_x . Dada la siguiente secuencia de operaciones:

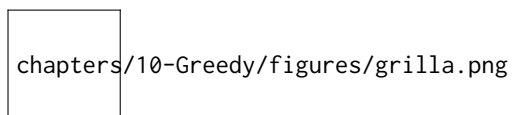
```

makeset( $x_1$ )
    :
makeset( $x_n$ )
union( $x_2, x_1$ )
union( $x_3, x_1$ )
    :
union( $x_n, x_1$ )

```

- a) ¿Cuál es el costo total de la secuencia de operaciones?
 - b) ¿Cuál es el costo amortizado por operación?
 - c) ¿Qué técnica se usa para mejorar el costo amortizado por operación? ¿Cuál es el costo amortizado que se logra con dicha técnica? Muestre un sketch de la demostración de dicho costo.
10. Agregue las siguientes funciones a la implementación de la estructura de datos `Union/Find` de la Figura 14.2:
 - int numDisjointSets():** retorna el número de conjuntos disjuntos que hay actualmente en la estructura.
 - int sizeofSet(int i):** retorna el tamaño del conjunto que actualmente contiene al elemento i .
11. Use los cambios agregados en la pregunta anterior para implementar una función que permita contar la cantidad de componentes conexas de un grafo no dirigido dado. Analice su algoritmo, indicando el tiempo de ejecución del mismo. Si el grafo tiene n nodos,

- m aristas, y k componentes conexas, ¿Cuántas veces se invoca a la función Find? ¿Cuántas veces a la función Union? Expresé sus respuestas en función de n , m , o k , según corresponda.
12. Ejecute el algoritmo propuesto en el punto anterior para el siguiente grafo $G = (V, E)$, en donde $V = \{a, b, c, d, e, f, g, h, i, j, k\}$ y las aristas del grafo son procesadas en el orden (d, i) , (f, k) , (g, i) , (b, g) , (a, h) , (i, j) , (d, k) , (b, j) , (d, f) , (g, j) , (a, e) . Muestre el resultado de la estructura paso a paso mientras ejecuta su algoritmo en los siguientes casos:
 - a) Usa listas enlazadas para implementar la estructura.
 - b) Usa una representación de bosque.
 13. Dado un grafo no dirigido arbitrario, se quieren responder en tiempo $O(1)$ (en el peor caso) consultas del tipo: dados dos vértices x e y , ¿existe un camino en el grafo que los conecte? Dado que las consultas a resolver son muchas, diseñe un algoritmo que permita procesar el grafo para que las consultas puedan resolverse en el tiempo requerido. Indique el tiempo de ejecución de su algoritmo. Para que el costo amortizado por consulta sea $O(1)$, ¿al menos cuántas consultas deben realizarse?
 14. ¿Es posible aplicar el mismo algoritmo del punto anterior sobre grafos dirigidos? Justifique su respuesta, sea cual sea.
 15. Sugiera un cambio a la función Union para la representación de listas enlazadas de manera que ya no sea necesario mantener el puntero al último nodo de cada lista. Aún cuando se use la heurística weighted-union, su cambio no debería cambiar el tiempo asintótico de ejecución de la función Union.
 16. Suponga que se quiere agregar la operación Print-Set(x) sobre la estructura de datos Union/Find, la cual imprime todos los elementos que están en el mismo conjunto que el elemento x dado, en cualquier orden (en definitiva, es un conjunto). Asumiendo una representación de bosque, muestre cómo agregar un único atributo a cada elemento (nodo) de la estructura de datos tal que la operación toma tiempo lineal en el número de elementos del conjunto que contiene a x . Explique cómo llevaría a cabo el proceso. Además, el tiempo asintótico de las demás operaciones no se debe modificar. Asuma que los elementos pueden ser impresos en tiempo $\Theta(1)$.
 17. ¹ Suponga un bitmap cuadrado de $n \times n$ píxeles, en donde cada píxel puede tener uno de dos colores: blanco o negro. Nos interesa mantener los grupos de píxeles negros en un bitmap. Se dice que dos píxeles negros están en el mismo grupo si son adyacentes. Por ejemplo, el siguiente bitmap posee 7 grupos de píxeles negros. El mayor grupo dentro de este bitmap tiene tamaño 9, y ha sido marcado con puntos interiores blancos.

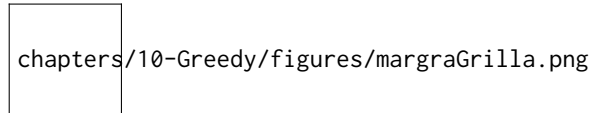


Se quiere implementar la función `marcar(i, j)`, la cual marca con negro el píxel con coordenadas (i, j) del bitmap y retorna el tamaño (en cantidad de píxeles) del grupo de píxeles negros más

¹ Ejercicio tomado del libro “Algorithms”, de Jeff Erickson, 2019.

grande del bitmap, después de que el nuevo píxel negro ha sido marcado. Se asume que se comienza con un bitmap que contiene únicamente píxeles blancos.

El siguiente ejemplo muestra (usando una cruz blanca) una secuencia de píxeles marcados de negro a partir de un bitmap inicial. Debajo de cada bitmap se muestra el tamaño del grupo más grande después de marcar el píxel (el grupo más grande en cada caso se marca con puntos blancos dentro de los píxeles).



- a) Proponga una estructura de datos que permita mantener de manera eficiente los grupos de píxeles negros, y obtener el tamaño del grupo más grande de manera eficiente. Muestre detalles y ejemplos de cómo usaría su estructura de datos para mantener los grupos de píxeles negros a medida que se marcan nuevos píxeles negros.
- b) Escriba el pseudocódigo de la operación `marcar(i, j)`, la cual usa la estructura de datos propuesta en el punto anterior para resolver el problema de manera eficiente.
- c) Si se comienza con un bitmap completamente blanco, ¿Cuál será el costo (total y por operación) si se realizan K ejecuciones de la operación `marcar`, para $K \leq n^2$?
- d) ¿Cuál sería el costo si se resolviera el problema de forma no eficiente?