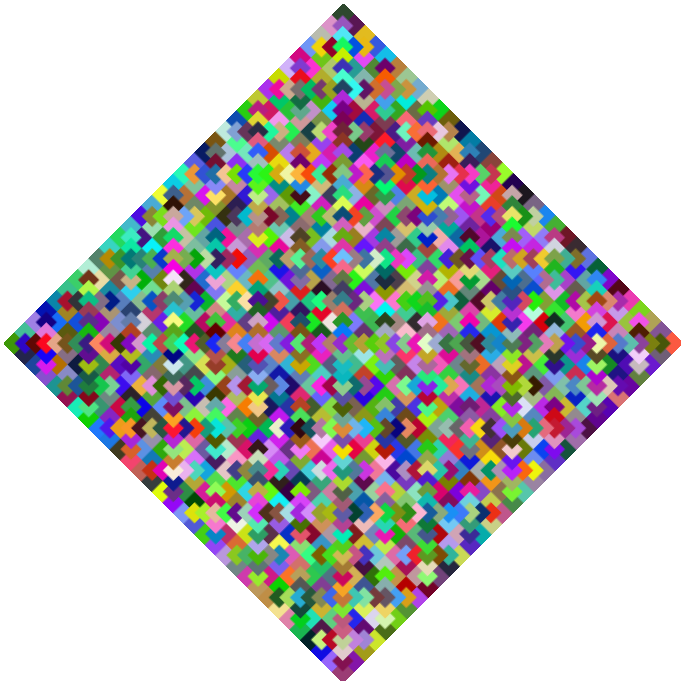


Diseño y Análisis de Algoritmos

Camilo Rocha

Email address: me@camilorocha.info

Para Laura.

© Derechos de autor 2019-2023 Camilo Rocha.
Última actualización 15 de abril de 2023.
Versión 0.0

Esta obra está bajo una licencia de Creative Commons
Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional.
Este trabajo puede ser copiado y distribuído libremente, como copia electrónica o en papel. No puede ser vendido por un valor mayor a su costo actual de reproducción, almacenamiento o transmisión.

Índice general

Capítulo 0

Preliminares

El propósito de un algoritmo es resolver mecánicamente un problema –o, equivalentemente, todas las instancias de un problema. Contar con una noción de algoritmo para este propósito requiere antes precisar qué se entiende por y cómo se define un problema algorítmico. La finalidad de este capútulo es presentar un marco general para la definición de problemas algorítmicos, e introducir convenciones para su posterior diseño y análisis.

0.1. Arreglos: notación y convenciones

Los problemas algorítmicos que se estudian en este texto, en su gran mayoría, son definidos y estudiados con base en arreglos. Un arreglo pueden contener números, caracteres, valores Booleanos, entre otros. También se pueden componer entre sí para formar matrices, hipermatrices como cubos, etc.

Definición 0.1.1

Un *arreglo* de *tamaño* N es una estructura de datos que almacena una colección de elementos (o valores), cada uno identificado por un *índice* correspondiente a un número natural en el rango $0 \,..\, N-1$.

No es de interés en este texto estudiar detalles de implementación de arreglos en lenguajes de programación en general o en un lenguaje de programación particular. Más bien, se supondrá que dado un arreglo, el acceso a cada uno de sus elementos se hace de manera eficiente, como sucede realmente en la práctica de la programación.

Nota 0.1.1

Se usa una notación similar a la de rangos en los números reales, con paréntesis circulares ( ) y llaves cuadradas [ ], para especificar arreglos y definir operaciones (de indexación y proyección) sobre ellos.

- La expresión $A[0..N)$ indica que A es un arreglo de N elementos con índices en el rango $0..N-1$.
- Con $A[0..N)$ y n un número natural, la *indexación* $A[n]$ denota el valor de A en la posición n siempre y cuando $0 \leq n < N$. Si $n \geq N$, entonces la expresión es un error.
  Por ejemplo, con $A = [1, 0, 25, -1, 8]$ se tiene $A[3] = -1$ y $A[6]$ es un error.
- Un arreglo sin índices se llama *vacío*.
- Con $A[0..N)$ y a, b números naturales, la expresión $A[a..b)$ denota la *sección* (o el *subarreglo*) de A, la cual es también un arreglo que:
  - cuando $0 \leq a < b \leq N$, re-indexa desde 0 y hasta $b - (a + 1)$, respetando su orden, los elementos de A entre a y $b - 1$ (i.e., b no se incluye), y
  - de lo contrario, el arreglo vacío.
  Por ejemplo, $A[1..3) = [0, 25]$ con $A = [1, 0, 25, -1, 8]$; $A[2..2)$ denota el arreglo vacío.
- El *tamaño* (o, equivalentemente, *la cantidad de índices*) de un arreglo o una sección de un arreglo se denota como $|\_|$.

Adoptando la notación introducida para arreglos en la Nota 0.1.1, una matriz A de tamaño $N \times M$ se especifica como $A[0..N)[0..M)$, y así sucesivamente. Una cadena de tamaño N puede representarse como un arreglo de tamaño N. Abusando un poco de la notación, las expresiones $A[0..N)$ y $A[0..N-1]$ denotan el mismo arreglo. Es importante tener en cuenta que cualquier operación de indexación en el arreglo vacío resulta en un error.

Los arreglos también se pueden concebir como objetos matemáticos. De esta forma es posible referirse a ellos de manera abstracta y formal (e.g., no atada a su implementación en un lenguaje de programación particular), posibilitando así un diseño y análisis algorítmico más enfocado en las ideas que una implementación en particular, cuando sea conveniente.

Nota 0.1.2

Matemáticamente, un arreglo $A[0..N)$ es una función A con dominio

$$\{0, 1, \ldots, N-1\}$$

y co-dominio

$$\{A[0], A[1], \ldots, A[N-1]\}.$$

El conjunto vacío es el dominio y co-dominio del arreglo vacío.

Note que no se menciona el tipo de elementos que almacena un arreglo. Esta información deberá entenderse del contexto dentro del cual se usan los arreglos en cada caso o se suministrará cuando sea necesario.

Ejercicios

1. Considere dos arreglos $A[0..M)$ y $B[0..N)$.
   a) Describa e implemente un algoritmo que permita determinar si los dos arreglos son iguales.
   b) Describa e implemente un algoritmo que permita determinar si B es sub-arreglo de A.

2. Investigue sobre árboles binarios casi llenos (e.g., aquellos que se utilizan en la implementación de *HeapSort* como implementación de colas de prioridad). Explique brévemente cómo representar un árbol binario casi lleno (e.g., de números) en un arreglo. ¿Qué relación hay entre los índices del arreglo y la profundidad de los elementos del árbol?

3. En el lenguaje de programación Python, los arreglos son en realidad listas indexadas. Investigue sobre cuál es la principal diferencia entre los arreglos de Python y los arreglos de C/C++/Java. ¿Qué ventajas tienen los arreglos de Python sobre los arreglos en estos otros lenguajes? ¿Tienen alguna desventaja?

4. Cuando están disponibles en un lenguaje de programación imperativo, los arreglos se pueden pasar como parámetros de funciones o procedimientos. Hay diferentes formas de pasar un arreglo como parámetro: por ejemplo, por valor o por referencia. Considere las siguientes preguntas para los lenguajes de programación Python, C, C++, Java:
   a) ¿Cuál es el mecanismo por defecto al pasar un arreglo como parámetro?
   b) ¿Se puede modificar el comportamiento por defecto? De ser posible, ¿cómo?
   c) ¿Cuáles son las principales ventajas y desventajas de cada uno de los mecanismos identificados anteriormente?

0.2. Problemas algorítmicos y su especificación

Un computador puede ser utilizado para asistir en la mecanización de un sinnúmero de tareas, de diferentes envergaduras y con propósitos muy diversos. Por ejemplo, un computador puede ser utilizado para encontrar la ruta más rápida (o corta) para conducir de un lugar a otro en una ciudad. También puede ser utilizado para almacenar y analizar grandes volúmenes de datos como los asociados al genoma humano. Hoy en día, los computadores son la herramienta de trabajo en muchas profesiones y se requiere para una gran cantidad de actividades.

Generalmente, es necesario diseñar piezas algorítmicas pequeñas y con propósitos muy específicos que, al componerse con otras piezas, permitan abordar tareas de mayor dificultad, como las mencionadas anteriormente. En algunos lenguajes de programación estas piezas reciben el nombre de funciones, procedimientos o módulos.

Nota 0.2.1

El propósito de este manuscrito es presentar técnicas de diseño y análisis de algoritmos para construir satisfactoriamente piezas algorítmicas fundamentales, que junto con otras piezas pueden conformar grandes sistemas de cómputo, información o procesamiento.

Conceptualmente, se puede aceptar que un problema es una pregunta formulada y a la cual se desea encontrar respuesta. De esta forma, se puede concebir un problema algorítmico como un problema cuya respuesta puede ser encontrada con la asistencia de un computador que ejecute macánicamente unas instrucciones.

Definición 0.2.1

La *especificación* de un problema algorítmico está conformada por dos partes: la descripción de la *entrada* y la descripción de la *salida*.

En un problema algorítmico, la descripción de la entrada identifica cuáles son los datos relevantes que hacen parte del planteamiento del problema. La descripción de la salida identifica la pregunta que se desea responder con base en los datos disponibles, previamente identificados en la descripción de la entrada.

Nota 0.2.2

Se usa el siguiente formato para presentar problemas algorítmicos:

Entrada: ...

Salida: ...

en donde a la derecha de "Entrada:" se describe la entrada del problema y a la derecha de "Salida:" la salida del problema.

El Ejemplo 0.2.1 presenta la especificación del problema de ordenamiento de un arreglo de números.

Ejemplo 0.2.1

Considere el problema de ordenar un arreglo de números. El problema algorítmico asociado puede definirse de la siguiente manera:

Entrada: Un arreglo $A[0..N)$, con $N \geq 0$, de números enteros.

Salida: El arreglo $A[0..N)$ ordenado ascendentemente.

Observe que en el proceso de especificar el problema algorítmico de ordenar ascendentemente un arreglo de números en el Ejemplo 0.2.1 se tomaron algunas decisiones. Primero, se indica concretamente la estructura de datos que representa dicha lista y cómo se llama: en este caso, la lista está dada como un arreglo llamado A y que contiene N números enteros, indexados desde 0. Segundo, se identifican qué casos de arreglos de números enteros son considerados: la entrada indica que se considera cualquier arreglo de números enteros, incluyendo el arreglo vacío (¿por qué?). Tercero, se establece unívocamente que el objetivo es ordenar el arreglo dado y no una copia de este, es decir, se plantea hacer un ordenamiento *in situ* del arreglo dado.

Un problema algorítmico puede contener "variables" a las cuales no han sido asignados valores concretos en su especificación. En el Ejemplo 0.2.1, este es el caso de A y N: indican que la entrada del problema puede ser *cualquier* arreglo (llamado A por conveniencia o por falta de imaginación) de *cualquier* tamaño. La intención de emplear variables en la especificación de un problema algorítmico, como A y N en el Ejemplo 0.2.1, es transmitir el mensaje de que su formulación es de carácter general y no particular.

Definición 0.2.2

Un *parámetro* es una variable que aparece en la especificación de un problema algorítmico y no tiene un valor concreto.

Dado que la especificación de un problema puede tener parámetros, un problema algorítmico en realidad representa una colección de problemas concretos: uno por cada asignación de valores concretos a sus parámetros y que cumple con las condiciones dadas. Es decir, un problema algorítmico con parámetros es una expresión simbólica que representa una colección de problemas concretos (o instancias), aquellos que específicamente se desea resolver en la práctica.

Definición 0.2.3

Una *instancia* de un problema algorítmico es una asignación de valores concretos a los parámetros de su especificación (y que cumplen con las condiciones dadas).

Un problema algorítmico, en general, tiene tantas instancias como asignaciones tengan sus parámetros. De esta forma, hay problemas algorítmicos que representan una cantidad finita de instancias y otros una cantidad infinita de ellas.

Ejemplo 0.2.2

Considere el problema algorítmico presentado en el Ejemplo 0.2.1. Las siguientes son algunas de sus instancias:

- $A = []$ y $N = 0$.
- $A = [1, 2, 3]$ y $N = 3$.
- $A = [2, 7, 1, 9, 8]$ y $N = 5$.

Ejemplo 0.2.3

La noción de instancia ha sido útil desde la formación básica en matemáticas. Por ejemplo, al establecer la ecuación $(a + b)^2 = a^2 + 2ab + b^2$, con a y b números reales, lo que realmente se está indicando es que para cualquier par de números reales (i.e., instancias de a y b) se puede calcular su suma al cuadrado calculando independientemente sus cuadrados y sumando el doble de su producto.

Aún no se ha definido claramente qué se entiende por responder una pregunta algorítmica. ¿Acaso se refiere a resolver una instancia de un problema algorítmico? ¿O a resolver algunas instancias? ¿O más bien se refiere a resolver todas las instancias? A continuación se propone una noción precisa de qué significa responder una pregunta algorítmica con base en las nociones presentadas anteriormente.

Definición 0.2.4

Una *solución de una instancia* de un problema algorítmico es la respuesta a la pregunta representada por dicha instancia. Una solución de un problema algorítmico es la respuesta a todas las instancias del problema.

En un problema algorítmico, puede haber una distancia abismal entre la dificultad que presenta resolver *una* de sus instancias específicas y resolver *todas* sus instancias. Existe el caso en el cual el tamaño de una instancia es pequeño y, como tal, podría ser fácil resolverla sin ingenio alguno. Por ejemplo, ordenar un arreglo con cinco números es fácil, mientras que ordenar otro con un millón puede ser más demandante en recursos. También está el caso en el cual ciertas instancias pueden estar sesgadas y "escondan", de alguna manera, la complejidad real de muchas otras instancias del problema. Por ejemplo, ordenar un arreglo que ya está ordenado –sin importar su tamaño– es trivial, mientras que resolver el problema de ordenamiento no es tan directo en el caso más general. En una situación más dramática, puede suceder que ciertas instancias de un problema algorítmico no tengan solución.

El estudio de algoritmos es interesante cuando su propósito es resolver una gran cantidad de instancias de un problema algorítmico, si no todas. Es decir, el reto de diseño algorítmico está en definir un algoritmo (e.g., unas instrucciones que un computador pueda ejecutar) que resuelva (todas las instancias de) un problema algorítmico dado.

Ejemplo 0.2.4

Para el problema de ordenamiento presentando en el Ejemplo 0.2.1, las siguientes son soluciones a las instancias en el Ejemplo 0.2.2:

- $A = []$.
- $A = [1, 2, 3]$.
- $A = [1, 2, 7, 8, 9]$.

En los dos primeros casos, el arreglo A estaba originalmente ordenado ascendentemente.

Vale la pena llamar la atención del lector acerca de un detalle que puede ser importante en la Definición 0.2.4. Allí se hace explícito el hecho de que pueden existir varias soluciones diferentes para una misma instancia de un problema algorítmico dado y, por ende, para un problema algorítmico dado. A continuación se presenta un problema algorítmico en el cual su única instancia puede tener más de una solución.

Ejemplo 0.2.5

Considere la siguiente especificación:

Entrada: Un tablero de ajedrez de 8×8 vacío.

Salida: Ocho celdas del tablero dado en las cuales se pueden ubicar reinas sin que se ataquen entre ellas.

Se propone como ejercicio al lector mostrar que la única instancia de este problema algorítmico, conocido como el "problema de las 8 reinas", tiene más de una solución.

Hay una diferencia en los niveles de abstracción empleados en las especificaciones de los problemas en los ejemplos 0.2.1 y 0.2.5. Para el ordenamiento, la entrada es un arreglo que puede ser considerado como un elemento primitivo en un lenguaje de programación. Sin embargo, la entrada en el problema de las 8 reinas es un tablero de ajedrez, que difícilmente es parte de los elementos primitivos en un lenguaje de programación. Esta diferencia no indica que alguno de los dos estilos de especificación esté mal. Lo que sugiere esta diferencia es, más bien, que en algunas ocasiones es conveniente especificar sin detallar tan finamente la forma de los parámetros porque se hacerlo así se considera algo sin importancia en ese momento o porque hay una razón de fondo para mantenerlo a ese nivel de abstracción alto. En el caso del problema de las 8 reinas, el tablero de entrada puede representarse fácilmente con una matriz de números enteros con dimensiones 8×8, llena de ceros indicando que el tablero está vacío. Sin embargo, la especificación dada asume que no hay necesidad de entrar en este tipo de detalles para entender la escencia del problema y pensar en una forma de resolverlo.

Especificar problemas algorítmicos requiere experiencia que se construye con la práctica. Puede ser retador adquirir esta disciplina, pero merece la pena: un problema bien especificado, es un problema que está más cerca de ser resuelto. En la Nota 0.2.3 se presentan algunas sugerencias para la especificación de problemas algorítmicos.

Nota 0.2.3: Regla de la mano derecha para especificar problemas

Un problema bien especificado formula cómo determinar si una salida dada es correcta para una entrada dada. Para ello, es recomendable:

- usar notación y términos claros y concisos, evitando argot especializado ajeno a la computación;

- si es necesario incluir argot especializado, este debe ser definido formalmente (y preferiblemente antes de ser usado);
- la especificación *no* debe mencionar
  - ni detalles de la fuente de datos de la cual proviene la entrada (e.g., base de datos, archivo de texto o un microservicio en la nube),
  - ni una estrategia de solución o detalles de su implementación,
- si la especificación *puede* ser simplificada, entonces *debe* ser simplificada.

Toda especificación debe ser precisa, corta y elegante.

El problema especificado en el Ejemplo 0.2.1 está bien especificado de acuerdo con la regla de la mano derecha en la Nota 0.2.3.

Esta sección concluye con ejemplos de buenas y malas especificaciones de problemas algorítmicos.

Ejemplo 0.2.6

A continuación se presentan buenas especificaciones de problemas algorítmicos.

Entrada: un grafo G dirigido.
Salida: un orden topológico de G, si existe; de lo contrario, una lista vacía.

Entrada: un grafo $G = (V, E)$ y un conjunto $C \subseteq V$.
Salida: ¿el subgrafo de G inducido por C es un clique (i.e., un grafo en donde cualquier par de vértices está conectado)?

Entrada: una colección P de coordendas en el plano Cartesiano de dos dimensiones.
Salida: mínima distancia entre cualquier par de puntos (distintos) en P.

Ejemplo 0.2.7

A continuación se presentan malas especificaciones de problemas algorítmicos; las razones por las cuales son consideradas malas especificaciones, en cada caso, se proponen como ejercicio para el lector.

Entrada: una secuencia de líneas de texto; en la primera un número x y en la segunda un arreglo A.
Salida: ¿está x en A?

Entrada: un grafo G con peso en los arcos.
Salida: el mejor camino para ir de u a v.

Entrada: una colección de cadenas.
Salida: la supercadena maximal de todas las cadenas de la entrada.

Ejercicios

1. Considere el problema de encontrar un número en un arreglo de números. Especifique los siguientes problemas asociados e identifique cuáles son sus parámetros:
   a) Determinar si el número está en el arreglo.
   b) Suponiendo que el número está en el arreglo, determinar un índice en donde está el número en el arreglo.
   c) Si el número está en el arreglo, determinar un índice en el arreglo en donde está el número; de lo contrario, identificar que no aparece.
   d) Suponiendo que el número está en el arreglo, determinar el índice más pequeño en donde está el número en el arreglo.
   e) Suponiendo que el número está en el arreglo, determinar el índice más grande en donde está el número en el arreglo.
2. Investigue y especifique el problema de buscar un valor numérico en un arreglo de números *ordenados* ascendentemente. ¿Qué diferencia fundamental hay entre este problema y los especificados en el Ejercicio 1?
3. Especifique el problema de calcular en $A[0..N)$ el valor de los primeros N números factoriales.
4. Especifique el problema de multiplicar dos matrices de números. ¿Qué restricciones deben cumplir las dimensiones de las matrices?
5. ¿Cuántas instancias tiene el problema algorítmico en el Ejemplo 0.2.1? Justifique su respuesta.
6. Investigue acerca de cada uno de los siguientes problemas y especifíquelos como problemas algorítmicos:
   a) El problema de primalidad de números.

b) El Teorema de Fermat.
c) La Conjetura de Goldbach.
d) El problema de satsifacibilidad proposicional.
e) El problema de la parada.

7. Clasifique los problemas indicados en el Ejercicio 1 de acuerdo a si tienen instancias con solución única o no. Ilustre su respuesta con ejemplos.
8. Clasifique los problemas indicados en el Ejercicio 6 de acuerdo a si tienen instancias con solución única o no. Ilustre su respuesta con ejemplos.
9. ¿Cuántas soluciones tiene el problema de las 8 reinas en el Ejemplo 0.2.5? Investigue y formule tres soluciones.
10. Considere la siguiente problema:

    Entrada: Un tablero de Sudoku parcialmente lleno.
    Salida: Una solución del tablero de Sudoku dado.

    a) ¿Cuántas instancias tiene el problema? Detalle los cálculos hechos.
    b) Especifique una instancia del problema que no tiene solución.
    c) ¿Es un problema para el cual cada instancia tiene a lo sumo una solución? Justifique su respuesta.
11. Investigue y especifique tres problemas algorítmicos que tengan una cantidad finita de instancias.
12. Investigue y especifique tres problemas algorítmicos que tengan una cantidad infinita de instancias.
13. ¿Existe un problema algorítmico con múltiples instancias, pero para el cual hay una única solución (i.e., independientemente de la entrada, la salida siempre es la misma)? Justifique su respuesta.
14. Explique brevemente por qué las especificaciones en el Ejemplo 0.2.7 no son consideradas como buenas especificaciones de problemas algorítmicos. En cada caso justifique su respuesta.

0.3. El concepto de algoritmo

De acuerdo con Donald Knuth, uno de los científicos más destacados y prolíficos de la informática, los algoritmos son los hilos que permiten relacionar y asociar diferentes disciplinas de las ciencias de la computación. Dada su importancia, definir qué se entiende por algoritmo ha sido una de las primeras tareas abordadas desde las matemáticas y desde las ciencias de la computación. Esta sección presenta una definición de algoritmo, los asocia a problemas algorítmicos e identifica algunas de sus propiedades principales.

Nota 0.3.1

La palabra *algoritmo* es de origen árabe y se asocia al nombre del matemático Mohamed ibn Musa, cuyo apodo era Al-Khwarismi y quien vivió entre los años 770 y 840 A.D. Se cree que, inicialmente, se usaban las palabras *alguarismo* y *guarismo*, y que estas palabras evolucionaron con influencia de la palabra griega *aritmo* (i.e., número) a algoritmo. Sin embargo, la Real Academia Española indica que esta palabra proviene del latín *algobarismus*, para abreviar en árabe la acción de calcular mediante cifras arábigas. De cualquier forma, Al-Khwarismi –además de desarrollar algoritmos para resolver ecuaciones de primer y segundo grado– explicó cómo usar ábacos en lugar de las manos para calcular a partir de una especificación clara y concisa. Esta es considerada una de las primeras expresiones de cálculo asistido mecánicamente en la humanidad. Así parece que surge la popularidad de la palabra 'algoritmo' como término para referirse a una secuencia de reglas para manipular números arábigos. Posteriormente, esta palabra fue generalizada para dar cuenta de cálculos con otros objetos matemáticos, no necesariamente numéricos. El primer caso de un algoritmo escrito para una máquina se encontró en las notas de Ada Lovelace hacia 1840, quien propuso cómo calcular los números de Bernulli en la Máquina Analítica de Charles Babbage. A pesar de que la Máquina Analítica nunca fue terminada y, en consecuencia, el algoritmo propuesto por Ada Lovelace no pudo ser ejecutado mecánicamente en dicha máquina, a ella se atribuye el título de primer programador de la humanidad.

El propósito intuitivo de un algoritmo es resolver un problema algorítmico a partir de una secuencia de pasos. Antes de que las ciencias de la computación existieran, los matemáticos se preocuparon por definir formalmente la noción de "computar", la cual está estrechamente ligada a la de algoritmo. Varias nociones fueron propuestas para finalmente llegar a un concenso.

Definición 0.3.1

Un *algoritmo* es un conjunto de instrucciones que pueden ser ejecutadas por una máquina de Turing.

Posiblemente, esta sea la definición más formal de lo que signifique computar, en cuanto a que en este contexto computar se refiere a las operaciones que puede realizar una máquina ideal, dada una secuencia de instrucciones (i.e., un algoritmo).

Nota 0.3.2

Es importante resaltar que la noción de algoritmo relativa a una *máquina de Turing* puede no solo ser la más formal, sino también la más general posible de acuerdo con la tesis de Church-Turing (en inglés, *Church-Turing thesis*). Este postulado afirma que la formalización de algoritmo en una máquina de Turing es *posiblemente* la noción más poderosa posible. Note que este postulado, como tal, es más una creencia que un hecho demostrado a causa de su formulación coloquial (i.e., no matemática). A pesar de ello, hay consenso en las comunidades académica y científica para aceptar esta afirmación.

Los computadores que se usan hoy en día están inspirados en el modelo de computación asociado a una máquina de Turing y, por ende, los cálculos que hacen están basados en lo que formalmente se entiende por algoritmo. Para el propósito de diseñar y analizar algoritmos, convenientemente se puede abstraer el concepto de "conjunto de instrucciones" de una máquina de Turing para obtener una definición de más alto nivel (pero sin perder de vista su definición formal) de lo que se entiende por algoritmo.

Definición 0.3.2

Un *algoritmo* es una colección finita y ordenada de pasos sin ambigüeadad que produce un resultado y termina en una cantidad finita de pasos (y de tiempo).

La descripción de un algoritmo debe ser finita por naturaleza, al igual que sucede con la descripción y la entrada de una máquina de Turing. El orden y la inexistencia de ambigüedad en los pasos que definen un algoritmo se refiere a que debe ser claro qué paso sigue en una "ejecución" de un algoritmo a partir de un estado parcial de su ejecución. Por ejemplo, si un paso es sumar dos números enteros, ambos números enteros deben estar definidos y la operación de adición estar identificada. No se puede confundir el operador de suma con otro operador, ni la acción de sumar con otro paso como almacenar un valor en una base de datos. El objetivo de un algoritmo es calcular algo como resultado; de lo contrario sería inoficioso seguir sus pasos. El resultado de un algoritmo se entiende como la salida de un proceso mecánico que obtiene un valor o modifica una cantidad. Finalmente, un algoritmo debe terminar después de una cantidad finita de tiempo o de pasos. De lo contrario, aquello definido *no* sería un algoritmo (esta restricción puede ser polémica en general, pero es conveniente y suficiente para el propósito de estudio en este texto).

Adicionalmente a las propiedades asociadas a un algoritmo y enunciadas en la Definición 0.3.2, un algoritmo puede tener cero, una o más entradas, dependiendo de lo que se desea calcular, y también producir uno o más resultados, cada uno de ellos asociados unívocamente con la entrada dada. Las operaciones usadas en los pasos de un algoritmo para producir los valores de retorno a partir de la entrada deben ser suficientemente básicas de tal forma que cada una de ellas se pueda procesar en tiempo finito.

La noción de algoritmo en la Definición 0.3.2 no está relacionada de ninguna manera con problemas algorítmicos, a pesar de que la finalidad de un algoritmo (al menos en este manuscrito) es resolverlos. De acuerdo con la Definición 0.2.4, una solución a un problema algorítmico necesariamente debe resolver todas sus instancias.

Definición 0.3.3

Sean P un problema algorítmico y A un algoritmo. Se dice que A *resuelve* (o *es una solución* de) P si y solo si A calcula una respuesta correcta para cada una de las instancias de P.

El concepto de solución algorítmica planteado en la Definición 0.3.3 indica que un algoritmo puede ser visto como solución a un problema algorítmico. Note que no hay excepción en cuanto a que algunas instancias del problema dado pueden quedar sin respuesta. En este sentido, considerar un algoritmo como solución a un problema algorítmico es una propiedad categórica de los problemas: debe resolver *todas* sus instancias. Note también que esta noción es agnóstica del lenguaje de especificación con que se describe el algoritmo y del lenguaje de programación con que se vaya a implementar. Como se verá a lo largo del texto, garantizar cuándo un algoritmo resuelve un problema algorítmico es una de las principales preocupaciones (si no la más importante) asociada a su diseño.

Ejemplo 0.3.1

Considere el problema de ordenamiento de un arreglo $A[0..N)$ de números enteros, especificado en el Ejemplo 0.2.1. A continuación se presenta un algoritmo especificado en lenguaje natural que "permite" resolver el problema.

1. Si $N \neq 0$, entonces repita, para $n = 0, 1, \ldots, N-1$, la siguiente secuencia de pasos:
   a) sea i el índice del mínimo valor en $A[n..N)$
   b) intercambie $A[n]$ y $A[i]$
2. Retorne $A[0..N)$.

Aparentemente, el algoritmo propuesto en el Ejemplo 0.3.1 ordena un arreglo de números. Sin embargo, dado que no hay una certeza matemática de ello, se usa la palabra permite en comillas: no es suficiente con creer, es necesario demostrar que en realidad funciona. Este será uno de los temas de énfasis en el texto.

Ejercicios

1. Elabore un pequeño resumen de las principales contribuciones de Donald Knuth a la informática y las matemáticas, identificando al menos cinco de sus principales manuscritos (e.g., libros y artículos).
2. Investigue sobre la vida de Mohamed ibn Musa y explique por qué (se cree) que lo apodaron Al-Khwarismi. Además, liste al menos cinco de sus principales aportes a la ciencia.
3. Investigue sobre el algoritmo diseñado por Ada Lovelace para Máquina Analítica de Charles Babbage y explique, a un alto nivel, su diseño.
4. Explique cuáles son las partes de una máquina de Turing y cómo se ejecutan algoritmos allí. ¿Puede proponer un corto algoritmo para una máquina de Turing? Justifique su respuesta.
5. Considere el algoritmo propuesto en el Ejemplo 0.3.1 para ordernar ascendentemente un arreglo de números. Identifique al menos una fuente de ambigüedad en su descripción y proponga cómo corregir este defecto.
6. Considere el problema de ordenamiento de un arreglo $A[0..N)$ de números enteros, especificado en el Ejemplo 0.2.1. A continuación se presenta un algoritmo especificado en lenguaje natural que "permite" resolver el problema.
   a) Si $N \geq 2$, entonces repita, para $n = 0, 1, \ldots, N-1$, la siguiente secuencia de pasos:
      1) sea i el índice del mínimo valor en $A[n..N)$
      2) intercambie $A[n]$ y $A[i]$
   b) Retorne $A[0..N)$.

   Aparentemente, este algoritmo es una alternativa al presentado en el Ejemplo 0.3.1 en cuanto ordena el arreglo $A[0..N)$. Proponga una forma de demostrar que las dos soluciones presentadas al problema algorítmico de ordenamiento son equivalentes, es decir, las dos solucionan correctamente todas las instancias del problema.
7. Considere el problema de ordenamiento de un arreglo $A[0..N)$ de números enteros, especificado en el Ejemplo 0.2.1. Suponga que hay un algoritmo que dado $A[0..N)$ ordena su contenido en un arreglo $B[0..N)$, el cual retorna como resultado. ¿Resuelve dicho algoritmo el problema formulado en el Ejemplo 0.2.1? Justifique su respuesta.

0.4. Especificación de algoritmos

En este texto se presentan técnicas de solución de problemas que resultan en algoritmos, recurrentes o iterativos. Independientemente de la forma que tome una solución, se debe contar con un lenguaje suficientemente expresivo para escribirla y conciso para entenderla. Para ello existen numerosas opciones, desde lenguaje natural (e.g., el que usamos para comunicarnos verbalmente) hasta lenguajes matemáticos o de programación con una semántica muy precisa y clara. Cada opción viene con beneficios e inconvenientes. Por ejemplo, una ventaja del lenguaje natural es su fácil descripción; el precio que se paga es, como tal, la posibilidad de involuntariamente llegar a una descripción ambigua dada la inherente ambiguedad del lenguaje natural. Una opción que contrarrestaría esa posible ambiguedad sería optar por un lenguaje de programación en lugar del lenguaje natural. Sin embargo, cada lenguaje de programación tiene su propia semántica (e.g., el comando `for` en Python y en Java puede ser interpretado de forma diferente) y esto obliga al lector a aprenderlo. Como estos, hay más ejemplos de beneficios e inconvenientes, animados por un sinfín de discusiones. En conclusión, en este texto se reconoce que no hay una solución perfecta, que convenga a cualquier tipo de lector, para el problema de cómo especificar algoritmos.

Por las razones expuestas anteriormente, se usarán varios *lenguajes* de especificación de algoritmos. Dependiendo del contexto, se podrá usar uno u otro. El compromiso del autor con los lectores es el siguiente: en ningún caso habrán ambigüedades ni especificaciones incompletas, sin importar el lenguaje que se use. Eso sí, el espectro de posibilidades se restringe a tres opciones como se anuncia en la Nota 0.4.1.

Nota 0.4.1

En orden de preferencia, en este texto se usaran los siguientes lenguajes para especificar un algoritmo:

- Enumeraciones estructuradas, como en el Ejemplo 0.3.1.
- Pseudo-código/código en el lenguaje de programación Python3, como en el Ejemplo 2.2.1
- Pseudocódigo con notación matemática, similar a lo que se emplea en otros textos de algoritmos, usualmente.

En las enumeraciones estructuradas se usa lenguaje natural (e.g., Castellano) en combinación con notación matemática sencilla, si es necesario. En estos casos,

los pasos de un algoritmo se describen con frases de palabras, que algunas veces contienen fórmulas. A pesar de ser una opción muy intuitiva, se ha de propender por ser preciso con las palabras porque de lo contrario describir un algoritmo podría fácilmente resultar en pasos ambigüos, como usualmente sucede en las recetas de cocina. Usar pseudo-código o código puede ayudar a eliminar fuentes de ambigüeadad gracias a la sintaxis técnica de los lenguajes de programación, como se explicó anteriormente. De esta forma, un algoritmo es o está muy cerca de ser un programa de computador. La relativa desventaja con esta opción es que la utilidad del algoritmo está supeditada a la semántica del lenguaje de programación elegido y se puede perder –entre tanto detalle técnico– la idea primordial detrás de la solución que encarna. Por ello, a lo largo de este texto se usará esta opción simpre y cuando la especificación del algoritmo no resulte en demasiado detalle técnico o sea extremadamente dependiente de la semántica de Pyhton. Finalmente, al usar notación matemática se elimina de raíz el problema de la ambigüedad, pero la especificación puede resultar extraña a los ojos poco entrenados o desconocedores de la notación elegida.

Ejercicios

1. Investigue sobre el formalismo conocido como *diagramas de flujo* y úselo para describir el algoritmo en el Ejemplo 0.3.1.

2. Investigue sobre el formalismo conocido como *lenguaje de comandos guardados* (en inglés, *guarded command language*) y úselo para describir el algoritmo en el Ejemplo 0.3.1.

3. Recuerde el problema de buscar un número en un arreglo de números ordenado ascendentemente (Ejercicio 0.2.2). Para este problema se conoce una solución llamada "búsqueda binaria". Investigue sobre esta solución y:
   a) Especifique el algoritmo de búsqueda binaria en lenguaje natural para resolver el problema dado.
   b) Especifique el algoritmo de búsqueda binaria en diagramas de flujo para resolver el problema dado (ver Ejercicio 1).
   c) Especifique el algoritmo de búsqueda binaria en el lenguaje de comandos guardados para resolver el problema dado (ver Ejercicio 2).
   d) Especifique el algoritmo de búsqueda binaria en el lenguaje de programación Python para resolver el problema dado.
   e) Especifique el algoritmo de búsqueda binaria en el lenguaje de programación Ada para resolver el problema dado.
   f) Especifique el algoritmo de búsqueda binaria en el lenguaje de programación Ruby para resolver el problema dado.

g) Especifique el algoritmo de búsqueda binaria en el lenguaje de programación Julia para resolver el problema dado.
h) Especifique el algoritmo de búsqueda binaria en el lenguaje de programación Scala para resolver el problema dado.
i) Especifique el algoritmo de búsqueda binaria en el lenguaje de programación Maude para resolver el problema dado.
j) Especifique el algoritmo de búsqueda binaria en el lenguaje de programación Brainfuck para resolver el problema dado.

4. Considere el problema de calcular la descomposición en factores primos de un número natural. Una aproximación para resolver el problema es la construcción de la Criba de Eratóstenes, para luego factorizar el número dado.
   a) Investigue sobre la Criba de Eratóstenes. ¿Cuáles son las operaciones básicas en su construcción? Explique su respuesta.
   b) Especifique el problema de construir la Criba de Eratóstenes en un arreglo $A[0..N)$ para los primeros N números naturales.
   c) Proponga una algoritmo para resolver el problema dado, utilizando el lenguaje de especificación de su preferencia.
   d) ¿Cómo puede garantizar que la solución propuesta es un algoritmo y funciona?

Notas del capítulo y referencias

En numerosos textos, los arreglos con considerados como conjuntos de variables elementales e indexadas consecutivamente. Esta concepción coincide con el tratamiento presentado en este capítulo. Autores como E. Dijkstra [**Dij76**] proponen un tratamiento alejado de la tradición y más cercano a la lógica para los arreglos, introduciendo el concepto de "variable arreglo" y adoptando una notación distinta para su indexación. Un tratamiento más extenso de la relación entre arreglos y funciones es propuesta por A. Kaldewaij [**Kal90**], incluyendo las nociones de sección e indexación, y notación para operar con ellos.

La especificación de problemas a partir de una relación entre la entrada dada y la salida esperada, y la concepción de un problema algorítmico como una colección de instancias, son usuales en la literatura; ver, e.g., [**CLRS22, Bha15**]. Autores como J. Kleinberg y É. Tardos [**KET06**] usan enumeraciones estructuradas para especificar algoritmos, mientras que autores como T. Cormen et al. [**CLRS22**] J. Erickson [**Eri19**] usan pseudo-código. Autores como H. Bhasin [**Bha15**] y S. Skiena [**Ski08**] usan complementariamente pseudo-código y un lenguaje de programación para especificar algoritmos. D. Gries [**Gri81**] usa el lenguaje de comandos

guardados (en inglés, *guarded command language*) para especificar algoritmos iterativos; este puede ser considerado un lenguaje matemático para la especificación de algoritmos.

Capítulo 1

Análisis asintótico

Considere por un momento que cuenta con dos algoritmos, digamos A_0 y A_1, que resuelven un problema algorítmico dado. Al ser soluciones del mismo problema, se está suponiendo que tanto A_0 como A_1 resuelven correctamente todas las instancias del problema. Sin más información y con poca reflexión, cualquiera de las dos soluciones podría adoptarse como "la" solución del problema algorítmico. Sin embargo, en la práctica es importante entender cómo se comportarían A_0 y A_1 para decir objetivamente cuál de las dos soluciones es "mejor" o "más conveniente". Por ello, entender cuáles son las características que hacen que un algoritmo sea mejor que el otro es clave para diseñar algoritmos.

El *análisis asintótico* es una herramienta matemática que permite comparar algoritmos con base en algunas de sus medidas. En el caso puntual de este manuscrito, el interés es las medidas que permiten identificar cuántas instrucciones, o cuántas unidades de tiempo o memoria –haciendo suposiciones mínimas sobre el lenguaje de programación en el cual se implemente y la arquitectura de la máquina en donde se despliegue dicha implementación– se requieren para que un algoritmo resuelva las instancias del problema para el cual fue diseñado. Al uso de estos recursos (i.e., tiempo y espacio) que hace un algorimto se le denomina *eficiencia algorítmica*: es mejor usar menos instrucciones, o menos unidades de tiempo o memoria cuando sea posible.

Un reto fundamental al cual se enfrenta el análisis asintótico tiene que ver con el hecho de que un problema algorítmico puede contar con múltiples parámetros e infinitas instancias. Es decir, el análisis asintótico debe ser capaz de clasificar la eficiencia algorítmica de manera tal que refleje lo que en realidad sucede con cada instancia, pero de manera suficientemente general como para no entrar a distinguir casos muy específicos para cada una de ellas.

Al final de este capítulo, el lector estará familiarizado con notación asintótica, complejidad temporal y espacial asociada a un algoritmo, y con nociones fundamentales para poder comparar algoritmos con base en esta notación.

1.1. Medidas para problemas y algoritmos

Como se aprenderá más adelante, la cantidad de problemas algorítmicos es inmensa. Por ello, es prácticamente imposible construir un marco matemático lo suficientemente general como para albergar un análisis asintótico de cualquier algoritmo. Por ejemplo, ¿cómo analizar soluciones a un problema algorítmico que tiene diez parámetros de la misma forma como se analiza para otro par de un problema algorítmico con un solo parámetro? Así como hay heterogeneidad en la cantidad de parámetros, también puede haber heterogeneidad en el tipo de operaciones que los algoritmos realizan. En algunos casos, por ejemplo, estas operaciones pueden tener naturaleza aritmética (e.g., multiplicar dos números enteros o calcular la raíz cuadrada de un número real no negativo), o –en otros casos– permitir manipular y construir cadenas de caracteres. En general, es necesario asumir algunas suposiciones acerca de la forma de los problemas algorítmicos y sobre el tipo de instrucciones que un algoritmo puede utilizar como instrucciones básicas.

1.1.1. El tamaño de la entrada. Para muchos problemas algorítmicos, es fácil identificar una medida razonable del tamaño de la entrada. En la especificación del Ejemplo 0.2.1, en la cual el problema consiste en ordenar ascendentemente un arreglo $A[0..N)$ dado, la cantidad N de elementos del arreglo parece ser una buena elección para identificar el tamaño de la entrada (tampoco hay muchas más opciones). En el caso del tablero de ajedrez en el Ejemplo 0.2.5, el tamaño de la entrada está dado por las dimensiones del tablero; en particular, hay 8×8 filas y columnas o, de manera equivalente, 64 casillas.

Nota 1.1.1

Dado un problema algorítmico P, se denomina *tamaño de la entrada de* P a alguna medida razonable sobre los parámetros de P.

La noción de *tamaño de la entrada* presentada en la Nota 1.1.1 está lejos de ser un concepto matemáticamente preciso, además que no indica las unidades con las cuales se medirá. Algunas veces se usará la cantidad de elementos de un arreglo como unidad de medida y otras veces la cantidad de bits que conforman un número; sin embargo, no hay una sola unidad preferida para medir el tamaño de las entradas de los problemas algorítmicos. Otra "mala" noticia es que, en la práctica, no existe una forma automática (i.e., algorítmica) de asignar el tamaño de la entrada para cualquier problema algorítmico. Por ello, es necesario desarrollar rápidamente la

destreza de especificar claramente los problemas algorítmicos, para luego identificar cuáles son los parámetros clave que permiten definir el tamaño de su entrada.

Ejemplo 1.1.1

Considere el problema de sumar los elementos de una matriz de números:

Entrada: Matriz $A[0..M)[0..N)$, con $M, N \geq 0$, de números enteros.

Salida: Suma de los elementos en $A[0..M)[0..N)$.

En este caso, el tamaño de la entrada es la cantidad de filas y columnas de la matriz dada, es decir, el tamaño del problema es una función de M y N.

Aún en los problemas sencillos especificados en los ejemplos 0.2.1 y 1.1.1, se está ignorando el tamaño de los números que se están ordenando o sumando. El tamaño de los números puede llegar a ser un factor importante a la hora de entender cómo se comporta un algoritmo: no cuesta lo mismo comparar un par de números de pocos dígitos o comparar otro par con miles de dígitos. Sin embargo, este nivel de detalle resulta innecesario en la mayoría de los casos dado que se opera con datos cuyos tamaños básicos no son exageradamente grandes.

Nota 1.1.2

Para evitar un detalle excesivo al definir el tamaño de la entrada de un problema, y en la medida de las posibilidades, se supondrá que el tamaño de los números que conforman la entrada es despreciable.

El acuerdo en la Nota 1.1.2 será utilizado en la mayoría de los problemas abordados en este manuscrito. Este acuerdo tiene una justificación en la práctica de implementar algoritmos. Por ejemplo, es común emplear procesadores de 64 bits para ejecutar algoritmos implementados en un lenguaje de programación. Esto último quiere decir que las operaciones aritméticas de comparación, suma o multiplicación entre números de 64 bits se pueden relizar directamente con operaciones del procesador, sin incurrir en "costos" adicionales para realizarlas externamente. En general, y aunque no siempre es posible, el acuerdo establecido en la Nota 1.1.2 eliminará dificultades técnicas y facilitará la tarea de analizar asintóticamente un algoritmo.

1.1.2. Asociando medidas a un algoritmo. Es indispensable modelar matemáticamente la cantidad de instrucciones, o de unidades de tiempo o memoria que un algoritmo emplea para resolver las instancias de un problema algorítmico.

La intuición es que dicha cantidad de operaciones o de unidades de tiempo/memoria pueden depender del tamaño de cada instancia. Matemáticamente, esto se puede modelar directamente con el concepto de función.

Nota 1.1.3

La cantidad de operaciones, tiempo o memoria requeridos por un algoritmo A con $k \in \mathbb{N}$ parámetros, se pueden representar por medio de una función

$$T_A : \mathbb{N}^k \to \mathbb{R}_{\geq 0}.$$

De esta forma, si A tiene un único parámetro, la expresión $T_A(n)$ representaría la cantidad de operaciones, tiempo o memoria empleadas por el algoritmo A al resolver una instancia de tamaño n.

Note que las funciones T_A presentadas en la Nota 1.1.3 tienen como dominio los números naturales y rango los números reales no negativos. Esto obedece a que el tamaño de una instancia se mide, generalmente, en unidades enteras (e.g., cantidad de elementos, cantidad de bits), y a que la cantidad de operaciones, tiempo o memoria nunca es negativa (por cuestiones técnicas, se prefiere que sea un número real y no uno natural).

Abstraer la cantidad de recursos empleados por un algoritmo con una función matemática tiene varios beneficios. Primero, la noción de función es un concepto básico no solo en matemáticas sino en cualquier ciencia. Segundo, hay herramientas disponibles desde las matemáticas para analizar funciones, i.e., para clasificar algoritmos en función de la cantidad de recursos que requieren. Tercero, esta brinda un nivel de abstracción conveniente para comparar algoritmos, dejando un poco de lado la velocidad del procesador o de la memoria de una máquina concreta en la cual se implanten.

Ejercicios

1. Investigue sobre el problema de determinar la ruta más corta de un vértice a otro en un grafo dirigido *sin* pesos en los arcos. Especifique este problema y determine el tamaño de la entrada.
2. Investigue sobre el problema de determinar la ruta más corta de un vértice a otro en un grafo dirigido *con* pesos en los arcos. Especifique este problema y determine el tamaño de la entrada.
3. Investigue sobre el problema del morral (en inglés, *knapsack*). Especifique este problema y determine el tamaño de la entrada. ¿Se puede expresar el tamaño de la entrada en función de solo una de sus variables?

1.2. Notación asintótica

Esta sección presenta la definición formal de la notación O (en inglés, *big Oh notation*), y algunas notaciones relacionadas. También se incluyen ejemplos de cómo utilizarlas para clasificar y comparar funciones (matemáticas).

Definición 1.2.1

Sea $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$. El *orden de* f, denotado $O(f)$, es el conjunto definido de la siguiente manera:

$$O(f) = \{g : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid (\exists n_0 \in \mathbb{N})(\exists c \in \mathbb{R}_{>0})(\forall n \in \mathbb{N}) n \geq n_0 \Rightarrow g(n) \leq cf(n)\}.$$

El orden $O(f)$ de una función f es el conjunto de todas las funciones que en el límite están acotadas (por encima) por un factor constante de f. En Castellano, se dice que una función g está en $O(f)$ (i.e., g es del orden de f) si eventualmente g está acotada por encima por un múltiplo constante de f. En este caso, se escribe $g \in O(f)$. Las variables n_0 y c cuantificadas existencialmente en la Definición 1.2.1 son usadas para denotar "eventualmente" y "factor constante", respectivamente.

La notación O puede también definirse sobre funciones con otro tipo. Por ejemplo, se pueden considerar funciones con dominio en los reales o de tuplas de números naturales. Para el propósito de este manuscrito, basta con enfocarse en aquellas funciones que tienen dominio en los naturales y codominio en los reales no negativos. La definición de O, tal y como se presenta con dominio en los naturales y rango en los reales no negativos, es suficiente para clasificar y comparar una gran cantidad de funciones asociadas a algoritmos.

Nota 1.2.1

Por conveniencia, en algunas ocasiones se abusa la notación de orden permitiendo funciones más generales. Por ejemplo, de vez en cuando, se indica que $g \in O(f)$ aún si $g(n)$ es negativo o indefinido para algunos valores $n < n_0$. También se permite que f sea negativa o indefinida para alguna cantidad finita de valores en su dominio. En estos casos, es necesario escoger un n_0 lo suficientemente grande para que estos comportamientos sean excluídos para cualquier $n \geq n_0$.

La Figura 1 ilustra cuándo una función g está en $O(f)$. La constante c indica cuál es el múltiplo de f con el que se acota a g y la constante n_0 indica a partir de qué punto cf permanece estable como acotamiento de g.

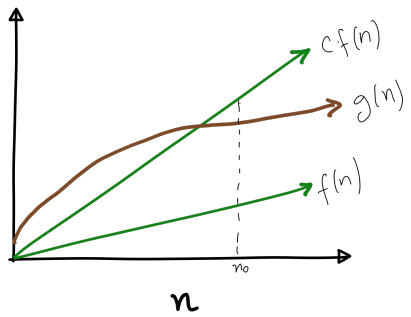


Figura 1. La función g está acotada por un factor constante positivo c de f en el límite (i.e., $g \in O(f)$).

La necesidad de introducir una definición formal de la notación O es para establecer observaciones precisas sobre el comportamiento asintótico de algoritmos. En estos casos, se requieren demostraciones rigurosas que permitan clasificarlos. Para establecer $g \in O(f)$ se debe demostrar la existencia de dos valores $n_0 \in \mathbb{N}$ y $c \in \mathbb{R}_{>0}$ de tal forma que $g(n) \leq cf(n)$ sea cierto para cualquier $n \geq n_0$. Una forma alternativa de ver este tipo de demostraciones es a modo de juego contra un oponente que siempre juega de forma perfecta (i.e., si hay forma de ganar, el oponente escoge una opción que le garantiza ganar).

Nota 1.2.2

El jugador con el primer turno es quien tiene el objetivo de establecer $g \in O(f)$. En el primer turno, dicho jugador escoge valores para n_0 y c. En el segundo turno, el oponente "todopoderoso" escoge un valor $n \geq n_0$. El juego lo gana el primer jugador si $g(n) \leq cf(n)$; de lo contrario, gana el oponente.

La justificación de ver una posible demostración de $g \in O(f)$ a modo de juego, como lo propone la Nota 1.2.2, es la siguiente. Si $g \in O(f)$, entonces existen constantes n_0 y c tal que $g(n) \leq cf(n)$ para $n \geq n_0$, y de esta forma el primer jugador tiene una estrategia ganadora. De lo contrario, sin importar cómo el primer jugador escoja n_0 y c, el oponente encontrará (dado que siempre juega de forma perfecta) un $n \geq n_0$ que falsifique la desigualdad y así ganará el juego. Es importante entender que en ocasiones se dice que las variables n_0 y c son constantes. En realidad, este abuso del lenguaje tiene el propósito de dejar claro que estos valores no dependen de n.

Ejemplo 1.2.1

Se desea demostrar que $5n + 10$ está en $O(n)$. Para ello, es necesario esgoger valores $n_0 \in \mathbb{N}$ y $c \in \mathbb{R}_{>0}$ tales que para cualquier $n \geq n_0$ se dé $5n + 10 < cn$.

El primer jugador puede tomar $n_0 = 3$ y $c = 10$, y se observa que $5n+10 < 10n$ siempre y cuando $2 < n$. Como $n_0 = 3$, la desigualdad $5n + 10 < 10n$ es cierta cuando $n \geq n_0$. Es decir, no importa qué valor escoja el oponente en su turno, aquel que siempre juega perfecto, pues cualquier escogencia hace que la desigualdad sea cierta. En conclusión,

$$(5n + 10) \in O(n).$$

Un ejercicio común en la comparación asintótica de funciones es el de establecer cuándo una función *no* está en el orden de otra. Para este propósito es indispensable entender muy bien la definición de O y, en particular, la alternancia de cuantificadores en ella. Intuitivamente, $g \notin O(f)$ cuando sin importar con qué factor constante se amplifique a f es imposible que dicho factor constante de f domine a g en el límite. Estas demostraciones tienden a ser posiblemente más retadoras que las de pertenencia dado que se debe demostrar que las constantes n_0 y c no existen. Dicho de otra forma, que sin importar cómo se escojan n_0 y c, al menos un valor de $n \geq n_0$ satisface $g(n) > cf(n)$. En estos casos, resulta conveniente usar la técnica de demostración por contradicción: suponer que dichos valores existen para luego llegar a un sinsentido o absurdo lógico.

Ejemplo 1.2.2

Se desea demostrar que n^2 no está en $O(10000n)$. Para ello, es necesario garantizar que es imposible escoger valores $n_0 \in \mathbb{N}$ y $c \in \mathbb{R}_{>0}$ tales que si $n \geq n_0$ se dé $n^2 \leq 10000cn$.

Hacia una contradicción, suponga que dichos valores existen. Sin pérdida de generalidad, también suponga que $n_0 \geq 1$. Entonces, para $n \geq n_0$ se tiene

$$n^2 < 10000cn \qquad \text{sii} \qquad \frac{n}{10000} < c.$$

Es decir, se debe dar que la constante c acote la función $\frac{n}{10000}$, lo cual es imposible (¿por qué?). En conclusión,

$$n^2 \notin O(10000n).$$

La notación asintótica es suficientemente robusta y de alto nivel como para obviar detalles que se desean ignorar y que dependen –por ejemplo– de la arquitectura de máquina o del lenguaje de programación en el cual se implementa un

algoritmo. También, es precisa como para poder establecer comparaciones entre diferentes alternativas de alto nivel que resuelvan un mismo problema algorítmico, especialmente cuando el tamaño de la entrada es grande y se requiere ingenio para plantear un algoritmo que sea práctico.

La notación O es útil para estimar una cota *superior* del uso de recursos que un algoritmo requiere para reolver un problema. De forma dual, también puede ser interesante estimar una cota *inferior* del uso de estos recursos. A continuación se introduce nueva notación para tal fin.

Definición 1.2.2

Sea $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$. El conjunto $\Omega(f)$ se define para cualquier $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ como:

$$g \in \Omega(f) \quad \text{sii} \quad f \in O(g).$$

El conjunto $\Omega(f)$ (leído, 'ómega de f') es el dual de $O(f)$, pues es el conjunto de todas aquellas funciones que están acotadas por debajo (en el límite) por un múltiplo positivo de f.

Nota 1.2.3

La definición de Ω en la Definición 1.2.2 no es la habitual. En este texto se opta por esta formulación dado que directamente depende de O. De este modo, basta con recordar solamente la definición de O, y una equivalencia sencilla para utilizar y razonar sobre Ω. El Ejercicio 14 de esta sección presenta la definición usual de Ω (para cualquier $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$)

$$\Omega(f) = \{g : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid (\exists n_0 \in \mathbb{N})(\exists c \in \mathbb{R}_{>0})(\forall n \in \mathbb{N}) n \geq n_0 \Rightarrow g(n) \geq cf(n)\}.$$

y plantea la equivalencia entre las dos definiciones.

En adición a O y Ω, existe otra noción asintótica para clasificar funciones que cuentan con una función que, simultánteamente en el límite, sirve como cota superior e inferior con factores que pueden ser distintos en cada caso.

Definición 1.2.3

Sea $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$. El conjunto $\Theta(f)$ se define como:

$$\Theta(f) = O(f) \cap \Omega(f).$$

El conjunto $\Theta(f)$ se denomina el *exacto orden de* f y, como se puede deducir de su definición, es más poderoso (en relación a que brinda más información) que $O(f)$ o $\Omega(f)$ individualmente. Al igual que con la definición de Ω, hay una definición distinta que se usa habitualmente (Ejercicio 15 de esta sección). Por los mismos motivos explicados anteriormente para Ω, se prefiere la caracterización de Θ en la 1.2.3.

Ejemplo 1.2.3

Se desea demostrar, para $n \in \mathbb{N}$:

$5n \in \Theta(n + \log n^2)$.

Considere el siguiente razonamiento:

	$5n \in \Theta(n + \log n^2)$	
sii	$5n \in \Theta(n + 2\log n)$	(aritmética)
sii	$5n \in O(n + 2\log n) \cap \Omega(n + 2\log n)$	(def. de Θ)
sii	$5n \in O(n + 2\log n) \wedge 5n \in \Omega(n + 2\log n)$	(def. de $\cap$)
sii	$5n \in O(n + 2\log n) \wedge (n + 2\log n) \in O(5n)$	($O(5n) = O(n + 2\log n)$)
sii	$(n + 2\log n) \in O(n + 2\log n) \wedge 5n \in O(5n)$	($f \in O(f)$)
sii	*true*.	

La demostración de la justificación dada en el quinto paso del razonamiento se propone como ejercicio al lector.

Ejercicios

1. Justificando su respuesta, clasifique ascendentemente por orden asintótico las siguientes funciones:

$$2^{2^n} \qquad \log\sqrt{n} \qquad n^2 \log n \qquad \log(n!) \qquad n! \qquad n^{2,5}$$

2. Justificando su respuesta, clasifique ascendentemente por orden asintótico las siguientes funciones:

$$\sqrt{2n} \qquad \left(\frac{3}{2}\right)^n \qquad (\log n)^2 \qquad 2^{\sqrt{2\log n}} \qquad n \log n \qquad \log(n^2)$$

3. En la Nota 1.2.1 se indica que en ocasiones se pueden considerar funciones negativas o indefinidas en algunos puntos. Explique por qué los abusos identificados en la Definición 1.2.1 no afectan el uso formal de la notación O.

4. Considere la demostración del Ejemplo 1.2.1. Escoja valores diferentes para n_0 y c de tal forma que se obtenga una nueva demostración.

5. En la demostración por contradicción en el Ejemplo 1.2.1 se indica que sin pérdida de generalidad se puede suponer $n_0 \geq 1$. Explique por qué esta afirmación es correcta y no limita la generalidad de la demostración.

6. Suponga que un algoritmo A realiza $T_A : \mathbb{N} \to \mathbb{R}_{>0}$ operaciones básicas para ordenar un arreglo (i.e., $T_A(n)$ es la cantidad de operaciones que realiza A para ordenar un arreglo de tamaño n). Como parte de una tarea, Pedrito demostró que $T_A \in O(n^2)$, Juanita que $T_A \in O(n^{3/2})$ y el profesor publicó una solución con una demostración de $T_A \in O(n \log n)$.
   a) ¿Es posible que las tres demostraciones sean correctas? Justifique su respuesta.
   b) Si las tres demostraciones fueran correctas, ¿cuál de las tres afirmaciones brinda "más" información sobre el comportamiento de A? Explique brevemente su respuesta.

7. En el Ejemplo 1.2.2 se usa el argumento de que la función $\frac{n}{100000}$ no puede ser acotada por la constante c. Demuestre formalmente que esta afirmación es cierta.

8. Para $n \in \mathbb{N}$, demuestre que $O(5n) = O(n + 2\log n)$.

9. Para $n \in \mathbb{N}$, demuestre que $2n^2 \in O(n^3)$ y que $n^3 \notin 2n^2$.

10. Para $n \in \mathbb{N}$, demuestre o refute: $2^{n+1} \in O(2^n)$.

11. Para $n \in \mathbb{N}$, demuestre o refute: $(n+1)! \in O(n!)$, en donde $\_!$ es la función factorial.

12. Para $n \in \mathbb{N}$ y $f : \mathbb{N} \to \mathbb{R}_{>0}$, demuestre o refute:
   a) Si $f \in O(n)$, entonces $f^2 \in O(n^2)$.
   b) Si $f \in O(n)$, entonces $2^f \in O(2^n)$.

13. Considere la siguiente definición para una función $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$:

$$O^*(f) = \{g : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid (\exists c \in \mathbb{R}_{>0})(\forall n \in \mathbb{N}) g(n) \leq cf(n)\}.$$

   Demuestre: si $f : \mathbb{N} \to \mathbb{R}_{>0}$, entonces $O(f) = O^*(f)$.

14. Sea $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$. El conjunto $\Omega^*(f)$ se define de la siguiente manera:

$$\Omega^*(f) = \{g : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid (\exists n_0 \in \mathbb{N})(\exists c \in \mathbb{R}_{>0})(\forall n \in \mathbb{N}) n \geq n_0 \Rightarrow g(n) \geq cf(n)\}.$$

   Demuestre que $\Omega(f) = \Omega^*(f)$.

15. Sea $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$. El conjunto $\Theta^*(f)$ se define de la siguiente manera:

$$\Theta^*(f) = \{g : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid (\exists n_0 \in \mathbb{N})(\exists c, d \in \mathbb{R}_{>0})(\forall n \in \mathbb{N}) \\ n \geq n_0 \Rightarrow cf(n) \leq g(n) \leq df(n)\}.$$

   Demuestre que $\Theta(f) = \Theta^*(f)$.

1.3. Propiedades de la notación asintótica

El razonamiento para clasificar y comparar funciones con las notaciones O, Ω, Θ se hace más práctico explotando sus propiedades matemáticas. Esta sección presenta algunas de estas propiedades, enfocándose principalmente en O, y las ilustra con ejemplos. Los ejercicios de la sección incluyen más propiedades de la notación asintótica.

Los factores constantes en una expresión que determina la cantidad de recursos requeridos para resolver un problema tienden a ser extremadamente dependientes del sistema en donde se desplieguen los algoritmos. Por ello, si no se desean cálculos que estén comprometidos con un lenguaje de programación específico o un entorno de ejecución particular, tiene todo el sentido ignorar los términos constantes al clasificar y comprar asintóticamente las funciones asociadas a los algoritmos.

Teorema 1.3.1: Regla de constantes

Sea $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$. Si $c \in \mathbb{R}_{>0}$, entonces $O(f) = O(cf)$.

Demostración

Se supone $c \in \mathbb{R}_{>0}$ y se procede por doble inclusión:

- El objetivo es demostrar $O(f) \subseteq O(cf)$. Si $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ es tal que $g \in O(f)$, basta con demostrar $g \in O(cf)$. Si $g \in O(f)$, entonces hay $n_0 \in \mathbb{N}$ y $c_0 \in \mathbb{R}_{>0}$ tales que $g(n) \leq c_0 f$ para $n \geq n_0$. Tome $n_1 = n_0$ y $c_1 = \frac{c_0}{c}$, y note que para $n \geq n_1$ se tiene:

  $$\begin{aligned} g(n) &\leq c_0 f(n) && \text{(por suposición)} \\ &= c_1 cf(n) && \text{(por definición de } c_1\text{)}. \end{aligned}$$

  Luego, $g \in O(cf)$ con testigos n_1 y c_1.

- El objetivo es demostrar $O(cf) \subseteq O(f)$. Si $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ es tal que $g \in O(cf)$, basta con demostrar $g \in O(f)$. Si $g \in O(cf)$, entonces hay $n_0 \in \mathbb{N}$ y $c_0 \in \mathbb{R}_{>0}$ tales que $g(n) \leq c_0 cf$ para $n \geq n_0$. Tome $n_1 = n_0$ y $c_1 = c_0 c$, y note que para $n \geq n_1$ se tiene:

  $$\begin{aligned} g(n) &\leq c_0 cf(n) && \text{(por suposición)} \\ &= c_1 f(n) && \text{(por definición de } c_1\text{)}. \end{aligned}$$

  Luego, $g \in O(f)$ con testigos n_1 y c_1.

Es importante aclarar que los factores constantes pueden ser determinantes en el diseño de un algoritmo. Puede pensarse que la notación O es una herramienta

que permite comparar algoritmos con enfoques fundamentalmente distintos que resuelven un problema dado y determinar, entre ellos, cuál tiene el mejor potencial para escalar a entradas de gran tamaño. Una vez el algoritmo de alto nivel esté claro, es importante esforzarse por reducir el factor constante asociado al algoritmo. De cualquier forma, si el éxito de la solución algorítmica depende de qué tan rápido pueda resolver un problema, claramente este debe funcionar tan rápido como sea posible.

Los términos que tienen orden bajo, en relación con los demás términos que definen una función, tienden a ser irrelevantes a medida que el tamaño del problema a resolver aumenta.

Teorema 1.3.2: Subsunción por suma

Sea $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$. Si $g \in O(f)$, entonces $O(f + g) = O(f)$, en donde $f + g$ denota la suma punto a punto de f y g (i.e., $(f + g)(n) = f(n) + g(n)$).

Demostración

Se supone $g \in O(f)$ y se procede por doble inclusión.

- El objetivo es demostrar $O(f + g) \subseteq O(f)$. Si $h : \mathbb{N} \to \mathbb{R}_{\geq 0}$ es tal que $h \in O(f + g)$, basta con demostrar $h \in O(f)$. Primero, por la suposición $g \in O(f)$, hay $n_0 \in \mathbb{N}$ y $c_0 \in \mathbb{R}_{>0}$ tales que $g(n) \leq c_0 f(n)$ para $n \geq n_0$. Segundo, por la suposición $h \in O(f + g)$, hay $n_1 \in \mathbb{N}$ y $c_1 \in \mathbb{R}_{>0}$ tales que $h(n) \leq c_1(f + g)(n)$ para $n \geq n_1$. Observe que:

$$\begin{aligned}
h(n) &\leq c_1(f + g)(n) && \text{(por suposición si } n \geq n_1) \\
&= c_1 f(n) + c_1 g(n) && \text{(por aritmética)} \\
&\leq c_1 f(n) + c_1 c_0 f(n) && \text{(por suposición si } n \geq n_0) \\
&\leq c_1(c_0 + 1) f(n) + c_1(c_0 + 1) f(n) && \text{(por aritmética)} \\
&= (c_1(c_0 + 1))2 f(n) && \text{(por aritmética).}
\end{aligned}$$

  Luego, con testigos $n_2 = n_0 \uparrow n_1$ y $c_2 = c_1(c_0 + 1)$, se concluye $h \in O(f)$.
- El caso $O(f) \subseteq O(f + g)$ se propone como ejercicio al lector.

El Ejemplo 1.3.1 presenta un caso en el cual se pueden emplear los teoremas 1.3.1 y 1.3.2 para simplificar la clasificación asintótica de funciones. En conjunto, estas dos propiedades son de gran utilidad práctica pues permiten eliminar factores constantes e ignorar términos irrelevantes de una función.

Ejemplo 1.3.1

Una cota superior para la cantidad de operaciónes que toma el algoritmo MergeSort es aproximadamente

$$6n \log n + 6n,$$

en donde n es la cantidad de elementos a ordenar. Entonces, la complejidad temporal de este algoritmo puede abstraerse asintóticamente así:

$$\begin{aligned} O(6n \log n + 6n) &= O(6n \log n) && \text{(por el Teorema 1.3.2)} \\ &= O(n \log n) && \text{(por el Teorema 1.3.1).} \end{aligned}$$

Los conjuntos construídos a partir de la notación O satisfacen una especie de transitividad.

Teorema 1.3.3: Transitividad

Sean $f, g, h : \mathbb{N} \to \mathbb{R}_{\geq 0}$. Si $h \in O(g)$ y $g \in O(f)$, entonces $h \in O(f)$.

Demostración

Se propone como ejercicio al lector.

También es posible usar la notación O para establecer una relación de orden parcial entre funciones y, consecuentemente, comparar indirectamente la eficiencia relativa de diferentes algoritmos para resolver un problema dado.

Teorema 1.3.4

Sean $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$:

1. $O(f) = O(g)$ sii $f \in O(g)$ y $g \in O(f)$.
2. $O(f) \subset O(g)$ sii $f \in O(g)$ y $g \notin O(f)$.

Demostración

Se proponen como ejercicio al lector.

Finalmente, se presentan propiedades de la notación O que son especialmente útiles para analizar la complejidad temporal de algoritmos secuenciales e iterativos.

Teorema 1.3.5: Reglas de suma y producto

Sean $f, f_1, g, g_1 : \mathbb{N} \to \mathbb{R}_{\geq 0}$:

1. $O(f+g) = O(f \uparrow g)$, en donde la suma y el máximo se interpretan punto a punto.
2. Si $f_1 \in O(f)$ y $g_1 \in O(g)$, entonces $f_1 g_1 \in O(fg)$, en donde el producto se interpreta punto a punto.

La Propiedad 1 en el Teorema 1.3.5 se conoce como la *regla de la suma*, mientras que la Propiedad 2 como la *regla del producto*.

Ejercicios

1. Sea $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$. Demuestre que, si $c \in \mathbb{R}_{>0}$, entonces:
   a) $\Omega(f) = \Omega(cf)$.
   b) $\Theta(f) = \Theta(cf)$.
2. Complete la demostracíon del Teorema 1.3.2 con el caso $O(f) \subseteq O(f+g)$.
3. Sean $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ tales que $g \in O(f)$. Demuestre o refute:
   a) $\Omega(f+g) = \Omega(f)$.
   b) $\Theta(f+g) = \Theta(f)$.
4. Demuestre el Teorema 1.3.3.
5. Sean $f, g, h : \mathbb{N} \to \mathbb{R}_{\geq 0}$ tales que $g \in O(f)$. Demuestre o refute:
   a) Si $h \in \Omega(g)$ y $g \in \Omega(f)$, entonces $h \in \Omega(f)$.
   b) Si $h \in \Theta(g)$ y $g \in \Theta(f)$, entonces $h \in \Theta(f)$.
6. Demuestre la Propiedad 1 del Teorema 1.3.4.
7. Demuestre la Propiedad 2 del Teorema 1.3.4.
8. Sean $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$. Demuestre o refute:
   a) $\Omega(f) = \Omega(g)$ sii $f \in \Omega(g)$ y $g \in \Omega(f)$.
   b) $\Omega(f) \subset \Omega(g)$ sii $f \in \Omega(g)$ y $g \notin \Omega(f)$.
   c) $\Theta(f) = \Theta(g)$ sii $f \in \Theta(g)$ y $g \in \Theta(f)$.
   d) $\Theta(f) \subset \Theta(g)$ sii $f \in \Theta(g)$ y $g \notin \Theta(f)$.
9. Demuestre la Propiedad 1 del Teorema 1.3.5.
10. Demuestre la Propiedad 2 del Teorema 1.3.5.

11. Formule duales para Ω y Θ de las propiedades de suma y producto en el 1.3.5. Justifique su respuesta con demostraciones.
12. Proponga funciones $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ tales que $f \notin O(g)$ y $g \notin O(f)$. Justifique su respuesta.
13. Sean $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$. Demuestre que las siguientes propiedades son equivalentes:
    a) $O(f) = O(g)$.
    b) $\Theta(f) = \Theta(g)$.
    c) $f \in O(g)$.
14. Considere la siguiente argumentación:

$$\begin{aligned} O(n^2) &= O(n^3 + (n^2 - n^3)) && \text{(por aritmética)} \\ &= O(n^3) && \text{(por Teorema 1.3.2 con } n^2 - n^3 \in O(n^3)). \end{aligned}$$

    Claramente $O(n^2) \neq O(n^3)$. Entonces, ¿cuál es el error en la argumentación?
15. La noción de límite es una herramienta versátil y útil para comparar funciones. Para $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$, demuestre:
    a) Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}_{>0}$, entonces $O(f) = O(g)$.
    b) Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, entonces $O(f) \subset O(g)$.
    c) Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}_{>0}$, entonces $f \in \Theta(g)$.
    d) Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, entonces $f \in O(g) \wedge f \notin \Theta(g)$.
16. Encuentre funciones $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ tales que $O(f) = O(g)$, pero para las cuales $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ no exista.

1.4. El Teorema Maestro

El Teorema Maestro es un resultado del análisis de algoritmos que permite estimar una cota asintótica para una clase de funciones recurrentes. Esta sección presenta una versión del Teorema Maestro que es útil a lo largo del texto, caracteriza el tipo de función recurrente que puede ser resuelto con su aplicación y lo ilustra con ejemplos. Como tal, la presentación del Teorema Maestro se hace sin acompañamiento de una demostración formal de su validez, pues esta está fuera del alcance del texto.

El Teorema Maestro es de gran utilidad para analizar la eficiencia de algoritmos recurrentes como los que resultan de aplicar técnicas como, e.g., dividir y conquistar. Básicamente, este teorema brinda un método que permite tomar como entrada una función definida recurrentemente (e.g., determinando la cantidad de operaciones o de recursos que un algoritmo necesita para resolver un problema) para calcular una cota superior (asintóticamente definida) para dicha función. Por ejemplo, si se tiene una función $T(\_)$ que define recurrentemente la cantidad de operaciones que efectúa

MergeSort para ordenar un arreglo en función de su tamaño, entonces usando el Teorema Maestro se puede obtener una función $f(\_)$ que depende de dicho tamaño y tal que $T \in O(f)$. Como se verá en el desarrollo de esta sección, este proceso requiere únicamente de pocos cálculos artiméticos.

Dado que el Teorema Maestro no puede ser usado con cualquier función recurrente, es importante caracterizar el tipo de función que puede ser analizado con el teorema.

Definición 1.4.1: Función simple

Se dice que la función $T : \mathbb{N} \to \mathbb{R}_{\geq 0}$ es *simple* si satisface las siguientes condiciones para $n \in \mathbb{N}$:

- Caso base: para n suficientemente pequeño,

  $$T(n) \in O(1).$$

- Caso recurrente: para los demás valores de n y constantes $a, b, k \in \mathbb{R}_{\geq 0}$,

  $$T(n) \leq a \cdot T(n/b) + O(n^k)$$

  tales que:
  - $a \geq 1$ representa la cantidad de llamados recurrentes hechos por el algortitmo asociado a T y
  - $b > 1$ representa el factor por el cual se reduce el espacio en cada llamado recurrente de T.

El caso base en una función simple indica que una vez el tamaño de la entrada es suficientemente pequeño como para no hacer llamados recurrentes, entonces el problema se puede resolver en tiempo/espacio constante (i.e., $O(1)$). El caso recurrente asume que se recurre sobre a instancias similares a la instancia dada, pero de tamaño reducido en una fración b, y que estos llamados requieren $O(n^k)$ tiempo/espacio para ser consolidados como respuesta de la instancia inicial.

Ejemplo 1.4.1

Sea $T(n)$ la función que determina al cantidad de operaciones realizadas por *MergeSort* para ordenar un arreglo de tamaño n. Como los casos base, que corresponden a ordenar un arreglo vacío o de un solo elemento, toman tiempo constante, se tiene:

$$T(n) \in O(1), \text{ para } n \leq 1.$$

En el algoritmo hay dos llamados recurrentes (i.e., $a = 2$), cada uno sobre una mitad del arreglo dado (i.e., $b = 2$), y se requiere un cálculo lineal en el

tamaño de la entrada (i.e., $k = 1$) para consolidar los resultados obtenidos de los llamados recurrentes. Es decir, para el caso recurrente, se tiene:

$$T(n) \leq 2 \cdot T(n/2) + O(n), \text{ para } n > 1.$$

Luego, $T(\_)$ es una función simple.

En la práctica se puede trabajar con aproximaciones de a, b, k, que son los parámetros asociados a una función simple. También es importante entender que cuando una función ha sido definida/declarada como simple, no es necesario fijarse en el valor exacto de su(s) caso(s) base (¿por qué?). Estas funciones, a su vez, cuentan con algunas restricciones. Por ejemplo, cada llamado recurrente debe hacerse sobre subproblemas del mismo tamaño. Aunque este no siempre es el caso, los algoritmos que se diseñan en este texto usando dividir y conquistar permiten ser asociados a funciones simples, similares a la asociada a *MergeSort*.

Se puede precisar ahora el enunciado del Teorema Maestro con base en la noción de función simple.

Teorema 1.4.1: Teorema Maestro

Si $T : \mathbb{N} \to \mathbb{R}_{\geq 0}$ es una función simple con parámetros $a \geq 1$, $b > 1$ y $k \geq 0$, entonces para $n \in \mathbb{N}$:

$$T(n) \in \begin{cases} O(n^k \log n) & \text{, si } a = b^k \\ O(n^k) & \text{, si } a < b^k \\ O(n^{\log_b a}) & \text{, si } a > b^k. \end{cases}$$

El Teorema Maestro identifica tres casos que provienen de la comparación de dos cantidades: a y b^k. Intuitivamente, a representa la tasa con la que se recurre sobre subproblemas de un problema dado, mientras que b^k representa la tasa con la que se reduce "la complejidad" para resolver estos subproblemas. En este contexto, hay menos operaciones/recursos requeridos por un algoritmo cuando $a < b^k$; lo contrario sucede cuando $a > b^k$ porque hay demasiada proliferación de subproblemas. Cuando $a = b^k$, la intuición indica que el algoritmo lleva a cabo la misma cantidad de trabajo en cada uno de los niveles de la recurrencia: $O(n^k)$ operaciones en cada uno de los $O(\log n)$ niveles de la recurrencia, es decir, $O(n^k \log n)$ operaciones en total. Cuando $a < b^k$, la cantidad de trabajo va decreciendo a medida que se profundiza la recurrencia: es decir, la cantidad de trabajo $O(n^k)$ al tope de la recurrencia domina la cantidad de operaciones totales. Cuando $a > b^k$, la cantidad de operaciones en cada nivel de la recurrencia aumenta con su profunidad: la cantidad total de operaciones es dominada por la cantidad de operaciones en el

último nivel de la recurrencia, es decir, está acotada por $O(n^{\log_b a})$ (sabiendo que $n^{\log_b a} = a^{\log_b n}$, lo cual se propone como ejercicio para el lector).

A continuación se presentan ejemplos del uso del Teorema Maestro.

Ejemplo 1.4.2: Complejidad temporal de MergeSort

La cantidad de operaciones básicas que lleva a cabo el algortimo *MergeSort* está definida por la función simple T cuyo caso recurrente corresponde a:

$$T(n) = 2 \cdot T(n/2) + O(n).$$

Luego, $a = 2$, $b = 2$ y $k = 1$. Note que:

$$a = 2 = 2^1 = b^k,$$

lo cual corresponde al primer caso del Teorema Maestro. En consecuencia,

$$T(n) \in O(n \log n).$$

Ejemplo 1.4.3: Complejidad temporal de búsqueda binaria

La cantidad de operaciones básicas que lleva a cabo el algoritmo de búsqueda binaria está definida por la función simple T cuyo caso recurrente corresponde a:

$$T(n) = T(n/2) + O(1).$$

Luego, $a = 1$, $b = 2$ y $k = 0$. Note que:

$$a = 1 = 2^0 = b^k,$$

lo cual corresponde al primer caso del Teorema Maestro. En consecuencia,

$$T(n) \in O(\log n).$$

Ejemplo 1.4.4

Considere una función simple cuyo caso recurrente está dado por:

$$T(n) = 2 \cdot T(n/3) + O(n).$$

Luego, $a = 2$, $b = 3$ y $k = 1$. Note que:

$$a = 2 < 3 = 3^1 = b^k,$$

lo cual corresponde al segundo caso del Teorema Maestro. En consecuencia,

$$T(n) \in O(n).$$

Ejemplo 1.4.5: Complejidad temporal de Strassen

El algoritmo de Strassen permite multiplicar dos matrices cuadradas de tamaño n. De una forma ingeniosa, este algoritmo hace 7 llamados recurrentes sobre subproblemas reduciendo en mitades los tamaños de los suproblemas. Las respuestas obtenidas de los llamados recurrentes se consolidan con $O(n^2)$ operaciones:

$$T(n) = 7 \cdot T(n/2) + O(n^2).$$

Luego, $a = 7$, $b = 2$ y $k = 2$. Note que:

$$a = 7 > 4 = 2^2 = b^k$$

lo cual corresponde al tercer caso del Teorema Maestro. En consecuencia, como $\log_2 7 = 2{,}81$, se tiene

$$T(n) \in O(2^{2,81}).$$

Existen herramientas que permiten analizar el comportamiento de funciones recurrentes más generales que las caracterizadas en esta sección como funciones simples. Este es el caso de los árboles de recurrencia, los cuales permiten analizar visualmente funciones recurrentes muy generales. Por ejemplo, un árbol de recurrencia se puede utilizar para analizar una función que resulta de dividir y conquistar en la cual los llamados recurrentes no se hacen sobre subproblemas del mismo tamaño. Algunos de los ejercicios de esta sección están propuestos para familiarizarse con esta técnica y requieren consultar material no contenido en este manuscrito; referencias apropiadas se indican como de consulta complementaria.

Ejercicios

1. Use el Teorema Maestro para calcular $T(n)$ en cada uno de los siguientes casos:
   a) $T(n) = T(n/2) + 1$
   b) $T(n) = 3T(n/3) + 8n$
   c) $T(n) = 2T(4n/5) + n^3$
2. Use el Teorema Maestro para calcular $T(n)$ en cada uno de los siguientes casos:
   a) $T(n) = 2T(n/3) + n^3$
   b) $T(n) = 3T(n/3) + 5n$
   c) $T(n) = 6T(4n/5) + 4n^2$
3. Justifique por qué, para analizar con ayuda del Teorema Maestro el orden de la cantidad de recursos que emplea un algoritmo, no es necesario contar con los valores exactos de los casos base de la función simple asociada.
4. Sean $a, b \in \mathbb{R}$ tales que $a \geq 1$ y $b > 1$. Demuestre, para $n \in \mathbb{N}$, que $n^{\log_b a} = a^{\log_b n}$.
5. Investigue acerca del Algoritmo de Strassen para la multiplicación de matrices. Explique brevemente en qué consiste, cómo reduce la cantidad de llamados recurrentes de 8 a 7 e ilustre su funcionamiento con un ejemplo.
6. Existe una versión más general del Teorema Maestro en la cual se pueden analizar funciones recurrentes $T : \mathbb{N} \to \mathbb{R}_{\geq 0}$ cuyos casos bases son $O(1)$ y cuyos casos recurrentes son de la forma

   $$T(n) \leq a \cdot T(n/b) + \Theta(n^k),$$

   para $a, b, k \in \mathbb{R}_{\geq 0}$ tales que $a \geq 1$ y $b > 1$. Investigue acerca de esta versión del Teorema Maestro y enúncielo.
7. Los árboles de recurrencia son una herramienta visual para resolver recurrencias más generales que las admitidas para funciones simples. Investigue acerca de los árboles de recurrencia, elabore una breve descripción y presente un ejemplo de cómo usarlos para analizar funciones recurrentes más generales que las simples.
8. Use árboles de recurrencia (ver Ejercicio 7) para resolver cada una de las siguientes recurrencias:
   a) $T(n) = 2T(n/4) + \sqrt{n}$
   b) $T(n) = 2T(n/4) + n$
   c) $T(n) = 2T(n/4) + n^2$
9. Use árboles de recurrencia (ver Ejercicio 7) para resolver cada una de las siguientes recurrencias:
   a) $T(n) = T(n/2) + T(n/3) + T(n/6) + n$
   b) $T(n) = T(n/2) + 2T(n/3) + 3T(n/4) + n^2$
   c) $T(n) = 2T(n/2) + O(n \log n)$

d) $T(n) = 2T(n/2) + O(n/\log n)$
e) $T(n) = \sqrt{n}T(\sqrt{n}) + n$

10. Use árboles de recurrencia (ver Ejercicio 7) para resolver cada una de las siguientes recurrencias:
    a) $T(n) = T(n-1) + n$
    b) $T(n) = 2T(n-1) + n\log n$
    c) $T(n) = 2T(n-1) + n^2$

Notas del capítulo y referencias

Asociar funciones que miden diferentes aspectos de un algoritmo es una práctica común en el análisis de algoritmos. Las notaciones O, Ω y Θ son también muy comunes. Sin embargo, la presentación en este texto de Ω y Θ no es la habitual. Presentaciones convencionales de estas nociones se encuentran en, e.g., [**CLRS22, KET06**]. Los ejercicios de la Sección 1.2 que establecen las respectivas equivalencias entre las definiciones habituales de Ω y Θ con las presentadas en este texto, están inspirados por la presentación de la notación asintótica de G. Brassard y P. Bratley en [**BB88**]. Cormen et al. en [**CLRS22**] hacen una presentación sucinta de los análogos o, ω y θ de las notaciones presentadas en la Sección 1.2.

Las propiedades para el análisis asintótico en la Sección 1.3 son bien conocidas y se pueden consultar en varios de los textos de referencia citados aquí. Algunos ejercicios de esta sección están inspirados o aparacen en [**BB88**], especialmente los relacionados con límites que fueron transcritos literalmente en algunos casos. Los textos [**CLRS22, KET06, BB88, Eri19**] son una fuente extensa de más propiedades de O, Ω y Θ, y ejemplos de su uso.

El tratamiento del Teorema Maestro en la Sección 1.4 sigue la propuesta de T. Roughgarden [**Rou17**]; en este texto se encuentra una demostración del teorema, al igual que en [**CLRS22**]. Los ejercicios de la sección relacionados con árboles de recurrencia están inspirados en el tratamiento del tema y algunos ejercicios en [**Eri19**]. Cormen et al. en [**CLRS22**] incluyen técnicas para resolver recurrencias, complementarias al Teorema Maestro y a los árboles de recurrencia.

El texto [**SF13**] de R. Sedgewick y P. Flajolet es una fuente de profundización y temas especializados en el área de análisis de algoritmos; se recomienda como lectura obligatoria (y selectiva) al lector decididamente interesado en el tema.

Capítulo 2

Dividir, conquistar y combinar

Dividir y conquistar (acá llamada, *dividir, conquistar y combinar*) es la técnica por excelencia para resolver problemas algorítmicos. La idea está basada en un principio que la humanidad conoce y usa desde hace muchos años para construir demostraciones: el principio de *inducción matemática*. Como su nombre lo indica, la técnica permite resolver un problema tratando de dividirlo en subcasos y siendo oportunista para resolverlos directamente cuando no es necesario/posible reducirlos más. Técnicamente, una instancia de un problema se divide en subcasos que a su vez son instancias del mismo problema, pero más pequeñas (i.e., dividir), y estas soluciones se combinan para resolver la instancia dada (i.e., combinar); si la subdivisión no es necesaria, entonces la instancia del problema se resuelve directamente (i.e., conquistar).

Este capítulo hace una presentación de esta técnica de diseño algorítmico, la relaciona con el principio de inducción matemática y presenta varios ejemplos de cómo se utiliza en algoritmos sobre arreglos que han sido ampliamente estudiados. Algunos problemas y demostraciones se proponen como ejercicios para el lector.

2.1. Teselación de tableros de ajedrez

Esta sección tiene dos propósitos. Uno es mostrar cómo la técnica de dividir, conquistar y combinar permite resolver el problema de la teselación de tableros de ajedrez. El otro es establecer (de manera informal) la relación que existe entre esta técnica algorítmica y la demostración por inducción matemática.

Nota 2.1.1

A continuación se indican las tres partes que forman parte de una demostración por inducción matemática:

- **Casos base:** son aquellos casos que no dependen de otros casos y que se pueden resolver directamente.
- **Hipótesis inductiva:** son aquellos casos que se pueden suponer resueltos (con base en un ordenamiento de las instancias del problema).
- **Casos inductivos:** son aquellos casos que se apoyan en la hipótesis inductiva para poder llegar al objetivo de la demostración.

Esta es una sobresimplificación (un poco abusiva) del principio de inducción matemática. Sin embargo, es útil para el propósito de explicar la técnica de dividir, conquistar y combinar.

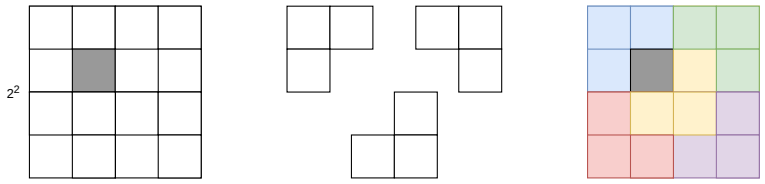
2.1.1. Especificación del problema. A continuación se especifica el problema de teselación de los tableros de ajedrez generalizados.

Problema 2.1.1: Teselación de un tablero generalizado

Entrada: un tablero cuadrado de lado 2^N, $N \geq 1$, con una celda faltante.

Salida: ¿es posible teselar el tablero con fichas en forma de L que comprenden exactamente tres celdas del tablero?

A continuación se muestran: (a) un tablero generalizado de lado 4; (b) fichas con las cuales se teselan los tableros (de las cuales siempre hay suficientes); y (c) una posible teselación de un tablero de lado 4.

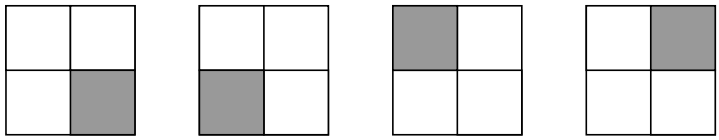


El problema se plantea de manera general, i.e., para cualquier tablero que cumpla con las indicaciones de la entrada. Como N no está acotado, esto significa que la cantidad de tableros es infinita y que no es posible tratar de construir teselaciones enumerando explícitamente todos los tableros. Si se formulase el problema directamente a modo de una fórmula matemática (e.g., $\forall N.N \geq 1 \cdots$), sería claro que una opción es proceder por inducción matemática.

2.1.2. Diseño de una solución. Se propone un algoritmo recurrente que está basado en análisis de casos, siguiendo las ideas de una demostración por inducción. Los tableros que pueden ser teselados directamente se tratan como casos base, mientras que los tableros que no son lo suficientemente pequeños como para ser teselados directamente se teselan al componer las teselaciones de tableros generalizados más pequeños.

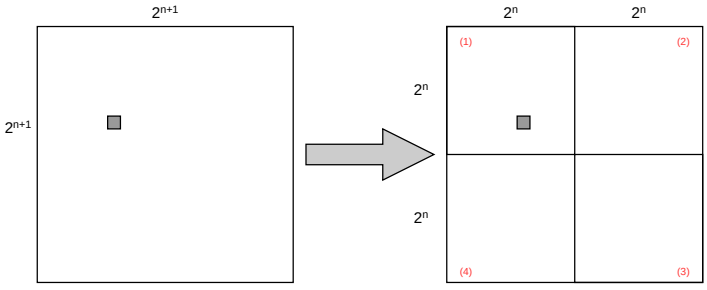
Algoritmo 2.1.1

- Los tableros más pequeños se pueden enumerar; estos corresponden a $N = 1$ y son 4:



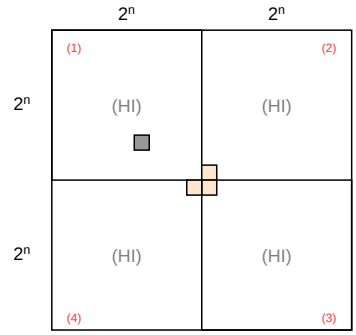
Cada uno de estos tableros se puede teselar con una ficha en forma de L. Esto concluye el *caso base* o, más bien, los *casos base*.

- Los demás tableros tienen lado 2^N, con $N \geq 2$. Se supondrá, a modo de *hipótesis inductiva*, que todo tablero de lado 2^n, con $n \geq 1$, se puede teselar (tal cual como se procede por inducción matemática sobre N). Tenga en cuenta que cada uno de estos tableros tiene exactamente un hueco. La idea es entonces encontrar una forma de teselar un tablero de lado 2^{n+1} que tiene exactamente un hueco. Note también que si un tablero de lado 2^{n+1} se divide en 4 partes iguales, resultan 4 tableros de lado 2^n (enumerados de 1 a 4 en la gráfica a continuación) y se acerca a algo parecido a la hipótesis inductiva. Estos dos hechos se representan gráficamente a continuación:



Como el subtablero 1 tiene lado 2^n y un hueco, se puede teselar por la hipótesis inductiva. Sin embargo, los otros 3 subtableros no tiene hueco y por ello no se puede usar la hipótesis inductiva directamente. Pero, el hecho de que no tengan hueco no quiere decir que no se pueda suponer

convenientemente que tienen un hueco: uno que resulta de ubicar una de las fichas en forma de L en sus tres esquinas concéntricas:



Ahora, los subtableros 2-4 se pueden tratar como si tuvieran un hueco en una de sus esquinas y consequentemente cumplen con la condición de la hipótesis inductiva (HI). Es decir, cada uno de ellos se puede teselar al tener lado 2^n y exactamente un hueco. Por consiguiente, todo el tablero de lado 2^{n+1} se puede teselar.

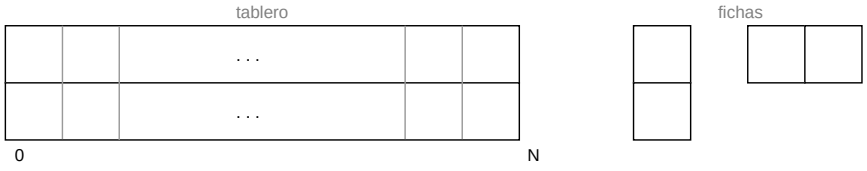
La solución dada en el problema anterior muestra cómo un tablero de ajedrez generalizado y con un hueco puede ser teselado. En realidad, brinda más que eso, pues demuestra constructivamente la posibilidad de encontrar una teselación con ayuda de un algoritmo. En los casos base, el problema se resuelve directametne teselando con una ficha (i.e., conquistar). Mientras que en los casos inductivos: (i) se "recurre" sobre el subtablero en donde está el hueco dado que es una subinstancia del problema inicial (i.e., dividir); y (ii) en los otros tres subtableros se marcan las esquinas concéntricas como si tuvieran un hueco, se recurre en cada subtablero (i.e., dividir) y finalmente se ubica una ficha en forma de L sobre las tres esquinas concéntricas de los subtableros teselados para completar la teselación (i.e., combinar). Este algoritmo puede ser usado en cualquier tablero por el estilo: está garantizada su correctitud por una demostración por inducción matemática.

La siguiente tabla hace explícita la relación que hay entre el principio de inducción matemática, la técnica dividir, conquistar y combinar, y un algoritmo recurrente.

Inducción mat.	**DCC**	**Alg. recurrente**
Caso base	Conquistar	Solución directa
Hipótesis inductiva	Dividir	Recurrencia
Caso inductivo	Combinar	Consolidación de soluciones a subproblemas

Ejercicios

1. Con base en la demostración que prueba la existencia de una teselación de un tablero de ajedrez generalizado:
   a) Diseñe un algoritmo en el lenguaje de programación Python3 que tesele un tablero dado.
   b) Invesigue sobre el lenguaje de programación `ditaa`. Intente traducir el algoritmo diseñado en Python para el numeral (a) a `ditta`.
2. Un tablero de ajedrez *doblemente* generalizado es un tablero cuadrado de lado 2^N, $N \geq 1$, al cual le hacen falta dos celdas de colores opuestos (asumiendo que está coloreado como un tablero de ajedrez convencional; por ejemplo, una celda blanca y una negra).
   a) ¿Es posible teselar dicho tablero con (suficientes) fichas de dominó, en donde cada ficha cubre exactamente dos celdas adyacentes del tablero? Si su respuesta es negativa, suministre un contra-ejemeplo. De lo contrario, construya una demostración siguiendo la estructura propuesta en esta sección.
   b) Considere la misma pregunta del numeral anterior para tableros cuadrados de lado $N \geq 1$ (a los cuales le faltan dos celdas de colores opuestos). Tenga en cuenta que esta es una generalización del problema inicial. ¿Es posible teselar cualquiera de estos tableros?
3. Considere un tablero de ajedrez de 8×8 celdas que ha sido "mutilado" removiendo exactamente dos celdas de esquinas opuestas.
   a) Especifique el problema de determinar si dichos tableros se pueden teselar con (suficientes) fichas de dominó, en donde cada ficha de dominó cubre exactamente dos celdas adyacentes.
   b) Resuelva el problema especificado en el numeral anterior.
4. Considere tableros de $2 \times N$ celdas y fichas de dominó que cubren exactamente dos celdas adyacentes.



   Proponga una función recurrente que determine la *cantidad* de formas de teselar el tablero con (suficientes) fichas de dominó. Justifique su respuesta con una demostración.
5. De acuerdo con Wikipedia,

> Las *Torres de Hanói* es un rompecabezas o juego matemático inventado en 1883 por el matemático francés Édouard Lucas. Este juego de mesa individual consiste en un número de discos perforados de radio creciente que se apilan intertándose en uno de los tres postes fijados a un tablero (...) Para realizar este objetivo, es necesario seguir tres simples reglas:
>
> - Solo se puede mover un disco cada vez y para mover otro los demás tienen que estar en postes.
> - Un disco de mayor tamaño no puede estar sobre uno más pequeño que él mismo.
> - Solo se puede desplazar el disco que se encuentre arriba en cada poste.

En este ejercicio tiene dos objetivos: demostrar que el juego tiene solución y que hay una algoritmo que lo resuelve con una cantidad de movimientos dado.

a) Especifique el problema de las Torres de Hanói para $N \geq 1$ discos perforados.

b) Demuestre que el problema especificado es soluble para cualquier N.

c) Proponga un algoritmo que lleve a cabo exactamente $2^N - 1$ movimientos de discos para resolver el problema. Demuestre que su algoritmo resuelve el problema dado.

6. Sean k y l números naturales. Se dice que k es *divisible* por l si y solo si existe un $p \in \mathbb{N}$ tal que $k = p \cdot l$ (en este caso, p se llama el *testigo* de la divisibilidad de k entre l). Por ejemplo, 15 es divisible por 3 con testigo 5 porque $15 = 5 \cdot 3$. Use el principio de inducción matemática para demostrar que $11^n - 4^n$ es divisible por 7 para todo $n \in \mathbb{N}$.

7. Demuestre que $x^n - 1$ es divisible por $x - 1$ para todos $n, x \in \mathbb{N}$.

8. La función $F : \mathbb{N} \to \mathbb{N}$ de Fibonacci se define inductivamente de la siguiente manera:

$$F(0) = 0 \qquad F(1) = 1 \qquad F(n) = F(n-2) + F(n-1), \text{ para } n \geq 2.$$

Demuestre que la función F de Fibonacci satisface las siguientes igualdades:

a) $F(1) + F(3) + \cdots + F(2(n-1)+1) = F(2n)$, para $n \geq 1$.

b) $F(0) + F(2) + \cdots + F(2n) = F(2n+1) - 1$, para $n \geq 0$.

9. Demuestre que la función F de Fibonacci satisface, para $n \in \mathbb{N}$, la siguiente igualdad:

$$F(0)^2 + F(1)^2 + \cdots + F(n)^2 = F(n)F(n+1).$$

10. Demuestre que la función F de Fibonacci satisface, para $n \in \mathbb{N}$, la siguiente igualdad:

$$F(n)^2 - F(n+1)F(n-1) = (-1)^{n+1}.$$

11. Demuestre que la función F de Fibonacci, para $n \geq 1$, satisface

$$F(n) \geq \left(\frac{3}{2}\right)^{n-2}.$$

12. El producto entre dos matrices de dimensión 2×2 se define como:

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \times \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} = \begin{pmatrix} a_{00}b_{00} + a_{01}b_{10} & a_{00}b_{01} + a_{01}b_{11} \\ a_{10}b_{00} + a_{11}b_{10} & a_{10}b_{01} + a_{11}b_{11} \end{pmatrix}.$$

Dada una matriz A de 2×2, se define A^n para $n \geq 1$ de la siguiente manera:

$$A^1 = A$$
$$A^{n+1} = A \times A^{n-1} \quad , n \geq 2.$$

Usando el principio de inducción matemática demuestre, para $n \geq 1$, la siguiente igualdad relacionada con la función F de Fibonacci:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{pmatrix}.$$

2.2. Ordenamiento de arreglos

Ordenar un arreglo de, por ejemplo, números es un problema fundamental en computación.

Problema 2.2.1: Ordenamiento de un arreglo de números

Entrada: Un arreglo $A[0..N)$, con $N \geq 0$, de números.

Salida: $A[0..N)$ ordenado ascendentemente.

A pesar de la simplicidad de su planteamiento y de contar con soluciones muy intuitivas, es un problema que ha sido estudiado extensivamente. La principal razón, pareciera, es la dificultad para resolverlo eficientemente por medio de un algoritmo cuando no se cuenta con información adicional sobre el arreglo dado.

El objetivo ahora es derivar un algoritmo que ordene un arreglo de números a partir de la técnica dividir, conquistar y combinar. Para ello, es deseable identificar rápidamente los subcasos que generalmente surgen al tratar de dividir el problema en suproblemas similares. En este sentido, es importante fijarse en qué casos se pueden resolver directamente sin necesidad de un ordenamiento explícito y los demás casos que parece no se puede resolver directamente. Los primeros casos corresponderán, muy seguramente, a los casos base de una demostración por inducción y al *conquistar* de la técnica algorítmica. Los segundos, serán los casos inductivos

(i.e., *combinar*), que a su vez nos permitirán identificar cuál debería ser la hipótesis inductiva (i.e., *dividir*). Por supuesto, dependiendo de cómo se haga este análisis, diferentes algoritmos resultarán.

Algoritmo 2.2.1

1. Si $n = N$, entonces no se hace nada.
2. Si $n \neq N$, entonces hay al menos un elemento en $A[n..N)$:
   a) identificar el índice $n \leq m < N$ con el mínimo valor en $A[n..N)$,
   b) intercambiar $A[n]$ y $A[m]$, y
   c) recurrir sobre $A[n+1..N)$.

Note que el caso $n = N$, identifica la situación en la cual $A[n..N)$ es vacío (y está ordenado). En el caso $n \neq N$ es posible que el arreglo no esté ordenado, por lo cual se ubica en $A[n]$ el elemento más pequeño en $A[n..N)$; de esta forma y se recurre ordenando $A[n+1..N)$. La intuición es que en este último caso, dado que $A[n]$ es el mínimo valor en $A[n..N)$ y $A[n+1..N)$ está ordenado ascendentemente, necesariamente $A[n..N)$ queda ordenado ascendentemente.

Implementación 2.2.1: Ordenamiento basado en el Algoritmo 2.2.1

```
def sort(A, n, N):
  if n==N: pass
  else:
    m = n
    for i in range(n+1, N):
      if A[i] < A[m]: m = i
    A[n],A[m] = A[m],A[n]
    sort(A, n+1, N)
```

Ejemplo 2.2.1

La función `sort` se usa a continuación con algunos ejemplos:

```
A = [8, -10, -2, 3, -50, 2, 10, 8, 4 ]
sort(A, 0, len(A))
print(A)

A = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
sort(A, 0, len(A))
```

```
print(A)

A = [ ]
sort(A, 0, len(A))
print(A)

A = [ 5 ]
sort(A, 0, len(A))
print(A)
```

El resultado de esta ejecución es el siguiente:

```
[-50, -10, -2, 2, 3, 4, 8, 8, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[]
[5]
```

Se han mostrado ejemplos que sugieren indicar que funciona bien para algunas instancias del problema, mas no una demostración de que el algoritmo funciona correctamente (i.e., resuelve adecuadamente todas las instancias del problema). Para ello es necesario demostrar que en realidad el código hace lo que debe hacer, es decir, ordenar un arreglo ascendentemente. Dicho de otra forma: se desea demostrar que `sort` ordena ascendentemente cualquier arreglo de números.

Teorema 2.2.1: Correctitud de `sort`

Para $N = len(A)$ y $0 \leq n \leq N$, el llamado `sort(A, n, N)` ordena ascendentemente el arreglo $A[n..N)$. En particular, `sort(A, 0, N)` ordena ascendentemente a $A[0..N)$.

Demostración

Se procede por inducción sobre n:

- Caso base ($n = N$): en este caso, el algoritmo no hace ningúna operación. Note que $A[n..N)$ es el arreglo vacío, el cual por definición está ordenado (pues no tiene un par de elementos en desorden).
- Caso inductivo ($n \neq N$): en este caso, n es un índice dentro del arreglo y $A[n]$ representa el valor en esa posición. El ciclo **`for`** y la asignación posterior al ciclo ubican en $A[n]$ el mínimo valor en $A[n..N)$. Por la hipótesis

inductiva, $A[n+1..N)$ está ordenado ascendentemente (este es el llamado recurrente de `sort`). Como ningún elemento en $A[n+1..N)$ es menor que $A[n]$ y $A[n+1..N)$ está ordenado ascendentemente, necesariamente $A[n..N)$ está ordenado ascendentemente.

Finalmente, como el resultado es cierto para cualquier n que satisface $0 \leq n \leq N$, en particular vale para $n = 0$ y de esa forma se tiene que `sort(A, 0, N)` ordena ascendentemente a $A[0..N)$.

La complejidad temporal del algoritmo `sort` se establece asintóticamente a continuación.

Teorema 2.2.2: Complejidad temporal de `sort`

Para $N = len(A)$ y $0 \leq n \leq N$, el llamado `sort(A, n, N)` toma tiempo $O((N-n)^2)$. En particular, el llamado `sort(A, 0, N)` toma tiempo $O(N^2)$.

Demostración

Sin pérdida de generalidad, se supondrá que las operaciones aritméticas, de asignación y de control toman exactamente una unidad de tiempo. Además, se define $k = N - n$ para simplificar la manipulación aritmética en los cálculos. Note que en cada llamado recurrente se realizan $N - n = k$ iteraciones del ciclo **`for`** y cada una de ellas toma tiempo constante. Entonces, la cantidad de operaciones está dada por:

$$\begin{aligned}
&(N-n) + (N-(n+1)) + \cdots + (N-(N-1)) + (N-N) \\
=&k + (k-1) + \cdots + 1 + 0 \\
=&\frac{k(k+1)}{2} \\
=&\frac{(N-n)(N-n+1)}{2} \in O((N-n)^2).
\end{aligned}$$

En particular, para $n = 0$, se tiene que la cantidad de operaciones está dada por $O(N^2)$.

La complejidad espacial se debe calcular con cuidado, en especial bajo la ausencia de información acerca de detalles de implementación de la recurrencia. A continuación, se presenta un estimado que supone que la creación del ambiente de ejecución de cada uno de los llamados recurrentes es de orden constante.

Teorema 2.2.3: Complejidad espacial de `sort`

Para $N = len(A)$ y $0 \leq n \leq N$, el llamado `sort(A, n, N)` usa espacio $O(N - n)$. En particular, el llamado `sort(A, 0, N)` usa espacio $O(N)$.

Demostración

En cada llamado recurrente de `sort` se crean a lo sumo dos variables, llamadas `m` e `i`. Como hay a lo sumo $N - n$ llamados recurrentes en donde se crean estas dos variables (únicamete se evita su creación en el caso base), la complejidad espacial de `sort(A, n, N)` es $O(N - n)$. Luego, la complejidad espacial de `sort(A, 0, N)` es $O(N)$.

Es importante aclarar que en el Teorema 2.2.3 se está suponiendo que las variables que se crean en cada llamado recurrente son independientes entre los llamados. Sin embargo, si se pudieran reutilizar de tal forma que fuera una cantidad constante para todos los llamados recurrentes, la complejidad espacial del algoritmo sería de orden constante.

Ejercicios

1. El algoritmo de ordenamiento diseñado e implementado en esta sección es muy conocido y tiene un nombre. ¿Cuál es?
2. Demuestre la siguiente versión, más general, del teorema de corrección de la función `sort`

   Teorema. Para $N = len(A)$ y $0 \leq n \leq m \leq N$, el llamado `sort(A, n, m)` ordena ascendentemente el arreglo $A[n..m)$.
3. Se propone el siguiente algoritmo para ordenar un arreglo $A[0..N)$ de números: mientras que $A[0..N)$ no esté ordenado, permutarlo.
   a) ¿Puede estimar la complejidad temporal de este algoritmo?
   b) ¿Puede demostrar que funciona?

   Si es necesario contar con algunos supuestos, indique cuáles son. Justifique sus respuestas.
4. Considere la siguiente variación del Algoritmo 2.2.1:
   - Si $n = N$, entonces no se hace nada.
   - Si $n = N - 1$, entonces no se hace nada.
   - Si $n \neq N$, entonces hay al menos un elemento en $A[n..N)$:
     - identificar el índice $n \leq m < N$ con el mínimo valor en $A[n..N)$,

- intercambiar $A[n]$ y $A[m]$, y
- recurrir sobre $A[n+1..N)$.

Con base en este algoritmo:

a) Modifique la función `sort` en la Implementación 2.2.1 para que considere el segundo caso base planteado anteriormente.

b) Formule el teorema de corrección y demuestre que es correcto para esta nueva versión del algoritmo.

c) ¿Cuál es la complejidad temporal y espacial del algoritmo? Justifique su respuesta.

5. En el caso recurrente del Algoritmo 2.2.1, para ordenar el arreglo $A[n..N)$ se usa la estrategia de identificar el índice $n \leq m < N$ con el mínimo valor e intercambiarlo con $A[n]$ antes de recurrir sobre $A[n+1..N)$.

   Otra estrategia para resolver el Problema 2.2.1 resulta de identificar valores *máximos* y no mínimos.

   a) Diseñe un algoritmo recurrente que, usando dividir, conquistar y combinar, aplique esta estrategia.

   b) Implemente su diseño en una función del lenguaje de programación Python.

   c) Enuncie y demuestre el teorema de corrección de su algoritmo.

   d) ¿Cuál es la complejidad temporal del algoritmo?

6. Considere el problema de ordernar *descendentemente* un arreglo de números.

   a) Especifique el problema dado.

   b) Usando dividir, conquistar y combinar, diseñe un algoritmo recurrente que resuelva el problema especificado anteriormente.

   c) Demuestre que el algoritmo es correcto con respecto a la especificación dada.

   d) Calcule la complejidad temporal del algoritmo diseñado.

7. Considere la siguiente colección de problemas algorítmicos sobre un arreglo de números:

   a) Calcular la suma de los elementos del arreglo.

   b) Determinar el mínimo valor en el arreglo.

   c) Determinar el máximo valor en el arreglo.

   d) Calcular la cantidad de apariciones de un número dado en el arreglo.

   Para cada uno de estos problemas:

   - Especifique el problema dado.
   - Usando dividir, conquistar y combinar, diseñe un algoritmo recurrente que resuelva el problema especificado.
   - Demuestre que el algoritmo es correcto con respecto a la especificación.
   - Calcule la complejidad temporal del algoritmo diseñado.
   - Implemente el algoritmo en el lenguaje de programación Python.

2.3. Una versión iterativa del ordenamiento

En la Sección 2.2 se diseñó un algoritmo recurrente que ordena ascendentemente un arreglo de números. Su diseño se basó en la técnica dividir, conquistar y combinar, con la cual se obtuvo la implementación de la función `sort` en el lenguaje de programación Python. También es posible obtener una versión iterativa del Algoritmo 2.2.1 de ordenamiento. Para ello es necesario usar la noción de *invariante* como guía para diseñar un ciclo iterativo. Esta sección presenta la noción de invariante asociada a un ciclo iterativo y la ilustra con el ejemplo de una versión iterativa del algoritmo de ordenamiento ascendente de un arreglo de números.

Nota 2.3.1

Un *invariante* (de ciclo) es una fórmula lógica asociada a un ciclo iterativo que cumple las siguientes dos condiciones:

- ser cierta antes de que inicien las iteraciones del ciclo
- ser cierta después de cualquier iteración del ciclo.

El diseño de algoritmos con invariantes requiere de disciplina, práctica y un entendimiento claro de la estrategia de solución que materializa el ciclo al cual está asociado. Cuando los invariantes están bien planteados y son suficientemente completos, es relativamente fácil obtener un ciclo iterativo correcto. La estrategia en estas notas es diseñar ciclos iterativos a partir de invariantes.

Se utilizarán los siguientes invariantes para representar el avance del algoritmo hacia el ordenamiento del arreglo dado. Para evitar confusiones porque el ordenamiento se hará en el mismo arreglo (i.e., es un ordenamiento in-situ) se identificará con $\overline{A}$ al estado inicial del arreglo, antes de que el algoritmo a ser diseñado lo manipule para lograr su ordenamiento. Es decir, el estado inicial del arreglo A se llama $\overline{A}$.

P_0 : $A[0..N)$ es una permutación de $\overline{A}[0..N)$.

P_1 : $A[0..n)$ tiene los n elementos de $\overline{A}[0..N)$ más pequeños y ordenados ascendentemente.

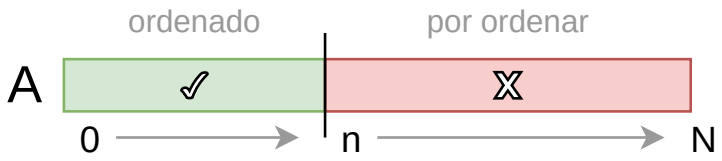
P_2 : $0 \leq n \leq N$.

Hay tres invariantes, llamados P_0, P_1, P_2. El invariante P_0 indica que los elementos de A son los mismos que los de $\overline{A}$, posiblemente en diferente orden. El invariante P_1 indica no solo que $A[0..N)$ está parcialmente ordenado en las posiciones $0..n-1$, sino que los elementos allí corresponden a los más pequeños en $\overline{A}$. El invariante P_2 indica que el valor de n varía entre 0 y N, inclusive.

Antes de continuar con el diseño de los casos del algoritmo, es clave pensar en cuál sería la situación ideal que –con base en los invariantes dados– garantizará que el arreglo A está ordenado. En este sentido, el invariante P_1 es clave, pues cuando n sea N (lo cual es permitido por el invariante P_2), se tendría la siguiente situación:

> $A[0..N)$ tiene los n elementos de $\overline{A}[0..N)$ más pequeños y ordenados ascendentemente.

Es decir, $A[0..N)$ tendría sus elementos ordenados ascendentemente; no faltaría ni sobraría elemento alguno gracias al invariante P_0. De esta forma, se sugiere que la estrategia es ir avanzando n, que puede iniciar desde 0, hasta que su valor sea N, mientras se mantienen los invariantes formulados. Un representación gráfica de los invariantes P_1 y P_2, y de esta idea de solución algorítmica, se muestra a continuación:



Con base en los invariantes y en la idea gráfica de diseño, se propone el Algoritmo 2.3.1 para ordenar iterativa y ascendentemente un arreglo de números.

Algoritmo 2.3.1

Para $n = 0, 1, \ldots, N-1$:

1. identificar el índice $n \leq m < N$ con el mínimo valor en $A[n..N)$ e
2. intercambiar $A[n]$ y $A[m]$.

La propuesta del Algoritmo 2.3.1 es similar a la del Algoritmo 2.2.1 recurrente, diseñado en la Sección 2.2. La gran diferencia es que después del paso (2) no se recurre sino que se continúa con las iteraciones del ciclo. Este diseño resulta en la siguiente implementación.

Implementación 2.3.1: Ordenamiento basado en el Algoritmo 2.3.1

```
1 def sortiter(A):
2   N = len(A)
3   # P_0 ∧ P_1 ∧ P_2
4   for n in range(0, N):
5     m = n
```

```
6     for i in range(n+1, N):
7       if A[i] < A[m]: m = i
8     A[n],A[m] = A[m],A[n]
```

Note que el código que selecciona el valor mínimo en $A[n..N)$ es el mismo de la versión recurrente del algoritmo.

La demostración de que un algoritmo iterativo de correcto está directamente ligada a la noción de invariante.

Nota 2.3.2

Para demostrar que un ciclo iterativo C con condición de terminación B es correcto con respecto a un invariante P es necesario demostrar que:

1. [Iniciación] P es cierto antes de la primera iteración de C y
2. [Estabilidad] Si P es cierto antes de una iteración de C, entonces P es cierto después de una iteración de C (i.e., que $P \wedge B$ implica P después de la terminación de una iteración de C).

Adicionalmente, una vez terminen las iteraciones de C, el invariante P debe suministrar información importante acerca del objetivo de C (i.e., $P \wedge \neg B$ deben proveer información sobre la finalidad de C).

A manera de ejemplo de cómo establecer la correctitud del ciclo principal de Algoritmo 2.3.1 con respecto sus invariantes, se presentan demostraciones de iniciación y estabilidad. Además, se muestra cómo los invariantes ayudan a determinar qué logra la ejecución del algoritmo. Dado que hay un ciclo anidado dentro del ciclo principal, se supondrá que dicho ciclo cumple su objetivo: calcular en la variable `m` el índice del valor mínimo en $A[n..N)$.

Teorema 2.3.1

Los invariantes P_0, P_1, P_2 son ciertos antes de la ejecución del ciclo (principal) de la Implementación 2.3.1.

Demostración

Antes de la primera iteración del ciclo, las variables n y N tienen los valores 0 y $len(A)$, respectivamente. Note que:

- Dado que el arreglo A no ha sido modificado, claramente P_0 es cierto.

- Como $n = 0$, el arreglo $A[0..n)$ es vacío y por tanto no tiene elementos. Esto concuerda con el invariante P_1.
- Con $n = 0$ y $N = len(A)$ se establecen trivialmente las desigualdades $0 \leq n \leq N$.

Teorema 2.3.2

Si los invariantes P_0, P_1, P_2 son ciertos y se itera una vez el ciclo (principal) de la Implementación 2.3.1, entonces P_0, P_1, P_2 siguen siendo ciertos.

Demostración

Lo que cambia en una iteración del ciclo son la variable n y (posiblemente) el arreglo $A[n..N)$. Para proceder, se supone que P_0, P_1, P_2 son ciertos antes de que estos camibios surtan efecto. Además, como es necesario que la condición $0 \leq n < N$ (esta es la guarda del ciclo) se cumpla para que dicha iteración se dé, se supondrá también que es cierta. Observe:

- Como $A[0..N)$ es una permutación de $\overline{A}[0..N)$ (suposición de que P_0 es cierto antes de la iteración) y como lo único que afecta a A en una iteración es el intercambio $A[n], A[m] = A[m], A[n]$, el invariante P_0 es cierto después de una iteración del ciclo.
- Por P_1, se tiene que $A[0..n)$ está ordenado ascendentemente con los n elementos más pequeños de $\overline{A}[0..N)$. En una iteración, $A[n]$ es actualizado con el elemento más pequeño de $A[n..N)$. Luego, $A[0..n+1)$ está ordenado ascendentemente con los $n+1$ elementos más pequeños de $\overline{A}[0..N)$ (este es el invariante P_1 después de una iteración del ciclo).
- Como $0 \leq n < N$ y n aumenta en uno con una iteración del ciclo, entonces claramente $0 \leq n + 1 \leq N$, lo cual corresponde al invariante P_2 después de incrementar n.

Finalmente, note que el ciclo termina cuando $n = N$. Como el ciclo es correcto con respecto a los invariantes P_0, P_1, P_2, se tiene que $A[0..N)$ tiene los N elementos de $\overline{A}[0..N)$ más pequeños y ordenados. Es decir, `sortiter(A)` ordena ascendentemente el arreglo `A`. El análisis de las complejidades temporal y espacial, al igual que la corrección del ciclo interno de Implementación 2.3.1 se proponen como ejercicio al lector.

Ejercicios

1. Demuestre que el ciclo interno en Implementación 2.3.1 termina.
2. Proponga invariantes para el ciclo interno en Implementación 2.3.1 y demuestre que el diseño del ciclo es correcto.
3. Considere el problema del ordenamiento de un arreglo de números. Suponga que únicamente se condieran arreglos cuyos valores están entre 0 y 10000.
   a) Especifique el problema.
   b) Diseñe un algoritmo que resuelva el problema dado, y cuyas complejidades temporal y espacial sean $O(N)$, en donde N es el tamaño del arreglo.
4. Considere la siguiente colección de problemas algorítmicos sobre un arreglo de números:
   a) Calcular la suma de los elementos del arreglo.
   b) Determinar el mínimo valor en el arreglo.
   c) Determinar el máximo valor en el arreglo.

   Para cada uno de estos problemas:
   - Especifique el problema dado.
   - Diseñe un algoritmo iterativo, incluyendo los invariantes correspondientes, que resuelva el problema especificado.
   - Demuestre que el algoritmo es correcto con respecto a la especificación.
   - Calcule la complejidad temporal del algoritmo diseñado.
   - Implemente el algoritmo en el lenguaje de programación Python.
5. Considere la siguiente función `sortitergen`:

```
def sortitergen(A, low, hi):
  for n in range(low, hi):
    m = n
    for i in range(n+1, hi):
      if A[i] < A[m]: m = i
    A[n],A[m] = A[m],A[n]
```

   Esta es una versión generalizada de la función `sortiter` en la Implementación 2.3.1 que –dados $A[0..N), low, hi$, con $0 \leq low \leq hi \leq N$– ordena ascendentemente el subarreglo $A[low..hi)$.

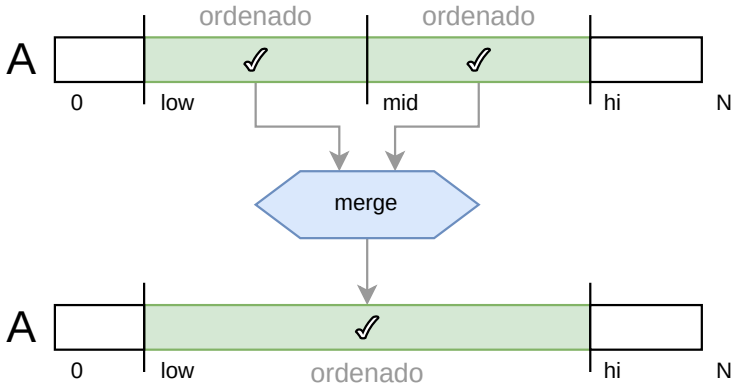
   Proponga invariantes para el ciclo externo y demuestre la correctitud de la función (suponiendo que el ciclo interno es correcto).
6. La siguiente es una enumeración de algoritmos de ordenamiento iterativos ampliamente conocidos:
   a) *BubbleSort*
   b) *InsertionSort*
   c) *ShellSort*

Para cada uno de estos algoritmos en el caso de ordenamiento de un arreglo de números:

a) Plasme el diseño del algoritmo al estilo del Algoritmo 2.3.1.
b) Proponga invariantes para los ciclos que materializan los algoritmos iterativos.
c) Demuestre que son correctos con respecto a la especificación propuesta.
d) Determine la complejidad temporal y espacial del algoritmo.

2.4. Mergesort: un ordenamiento de arreglos más eficiente

La idea principal detrás del algoritmo *MergeSort* no es muy distinta a la de los algoritmos de ordenamiento presentado en las secciones secciones 2.2 y 2.3, pues su diseño se puede también explicar con base en dividir, conquistar y combinar. La principal característica de *MergeSort* es que usa la hipótesis inductiva dos veces en cada paso del "combinar": el subarreglo se divide en dos "mitades" que, siendo problemas más pequeños que el original, se puede suponer que están ordenadas ascendentemente (i.e., se hace un llamado recurrente para cada una de ellas), para posteriormente combinarlas a modo de cremallera, manteniendo el orden de los elementos al ser consolidados en el arreglo original.



A cada momento, el algoritmo mantiene un "cerco" para el ordenamiento de un subarreglo de A con base en las varibles *low* y *hi*: el objetivo es ordenar $A[low..hi)$. Si esto es posible para cualesquiera *low* y *hi*, en particular se tiene resuelto el ordenamiento de todo el arreglo $A[0..N)$.

Antes de continuar explicando la imagen anterior que resume la idea de *MergeSort*, se introducen las siguientes condiciones:

C_0 : $A[0..N)$ es una permutación de $\overline{A}[0..N)$.

C_1 : $0 \leq low \leq hi \leq N$.

Note que la idea de ordenar el arreglo dado /in-situ/ se captura con la condición C_0. La condición C_1 brinda pistas de cómo se puede avanzar con *low* y *hi* ordenando por partes:

- Si $A[low..hi)$ tiene a lo sumo un elemento, entonces no hay mucho que hacer pues el arreglo vacío y cualquier arreglo unitario están ordenados por definición. Estos casos suceden cuando $low + 1 \geq hi$.

- Si $A[low..hi)$ tiene al menos dos elementos, entonces se puede "partir" en dos mitades (aproximadas) $A[low..mid)$ y $A[mid..hi)$ ordenando cada una de ellas recurrentemente y posteriormente combinándolas. Estos casos suceden cuando $low + 1 < hi$.

La idea principal detrás de la función `mergesort` en el Algoritmo 2.4.1 debería estar clara en este punto. Por ende, el foco estará en entender cómo la función `merge` combina las dos partes del arreglo que están ordenadas garantizando el ordenamiento general del arreglo $A[low..hi)$.

Implementación 2.4.1: MergeSort en Python

```
def mergesort(A, low, hi):
  if low+1<hi:
    mid = low+((hi-low)>>1)  # mid = (low+hi)//2
    mergesort(A, low, mid)   # induction hypothesis on the first half
    mergesort(A, mid, hi)    # induction hypothesis on the second half
    merge(A, low, mid, hi)   # combine the two halves preserving the order

def merge(A, low, mid, hi):
  global tmp                 # a global array at least the size of A
  for i in range(low, hi): tmp[i] = A[i]   # copy A[low..hi) to tmp[low..hi)
  l,r = low,mid
  for n in range(low, hi):
    if l==mid:  A[n],r = tmp[r],r+1   # only process the right half
    elif r==hi: A[n],l = tmp[l],l+1   # only process the left half
    else:
      # the first pending element of each half needs to be compared
      if tmp[l]<=tmp[r]: A[n],l = tmp[l],l+1 # choose the one on the left
      else: A[n],r = tmp[r],r+1              # choose the one on the right
```

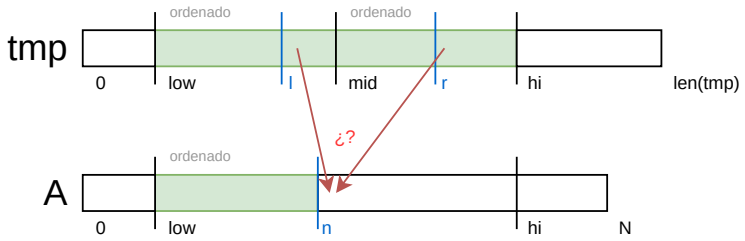
Para entender cómo encajan las piezas del algoritmo, se proponen los siguientes invariantes para el segundo ciclo de la función `merge`:

P_0 : $A[low..n)$ es un ordenamiento de $tmp[low..l)$ y $tmp[mid..r)$.

P_1 : $low \leq l \leq mid \leq r \leq hi$.

P_2 : $low \leq n \leq hi$.

El invariante P_0 indica que en $A[low..n)$ se van almacenando, en orden, los elementos de $tmp[low..l)$ y $tmp[mid..r)$ que han sido procesados. Los invariantes P_1 y P_2 indican entre qué rangos pueden tomar valores las variables l, r y n. Intuitivamente, las variables l y r pueden entenderse como dos (dedos) índices que señalan el siguiente elemento de la mitad izquierda y de la derecha, respectivamente, que debe ser procesado. Cuando $l = mid$ se tiene que la mitad izquierda ha sido agotada; de la misma forma, cuando $r = hi$ se tiene que la mitad derecha ha sido agotada. En cualquiera de estos dos casos no hay que comparar nada y basta con copiar a A los elementos pendientes en la mitad correspondiente. Si hay al menos un elemento en cada una de las mitades, entonces es necesario compararlos para determinar cuál de los dos se copiará a $A[n]$: el menor de los dos o, en caso de igualdad, el de la mitad izquierda. El siguiente diagrama resume visualmente el rol de cada una de las variables y de los arreglos involucrados en `merge`:



Antes de proceder a demostrar que `merge` y `mergesort` son correctos con respecto a su especificación, se presentan ejemplos de cómo el algoritmo ordena algunos arreglos de números.

Ejemplo 2.4.1

La función `mergesort` se usa a continuación con algunos ejemplos:

```
1 tmp = [ None for _ in range(10) ]
2 A = [8, -10, -2, 3, -50, 2, 10, 8, 4 ]
3 mergesort(A, 0, len(A))
4 print(A)
5
6 A = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
```

```
mergesort(A, 0, len(A))
print(A)

A = [ ]
mergesort(A, 0, len(A))
print(A)

A = [ 5 ]
mergesort(A, 0, len(A))
print(A)
```

El resultado de esta ejecución es el siguiente:

```
[-50, -10, -2, 2, 3, 4, 8, 8, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[]
[5]
```

A continuación se presenta y demuestra el teorema de correctitud de `mergesort`.

Teorema 2.4.1

Si $A[low..mid)$ y $A[mid..hi)$ están ordenados ascendentemente, entonces el llamado `merge(A, low, mid, hi)` garantiza el ordenamiento ascendente de $tmp[low..hi)$ en $A[low..hi)$.

Demostración

Se procede por inducción sobre n.

- Caso base ($n = hi$): en este caso no debe hacerse nada pues las dos mitades *tmp* han sido procesadas. Para observar esto con mayor claridad, a partir del código se puede deducir la siguiente igualdad (esto hay que demostrarlo y se propone como ejercicio al lector):

  $$n - low = (l - low) + (r - mid)$$

  Con $n = hi$ y con base en el invariante P_1, se tiene que las igualdades $l = mid$ y $r = hi$ son ciertas (esto también se propone como ejercicio al lector); es decir, no hacen falta elementos en $tmp[low..hi)$ por ser procesados.
- Caso inductivo ($n \neq hi$): esto indica que hay al menos un valor por ser procesado, y que al menos uno de los arreglos $tmp[l..mid)$ y $tmp[r..hi)$ no es vacío. Se procede por casos:

- Si $l = mid$, entonces la mitad izquierda no tiene elementos pendientes por ser procesados y el siguiente a ser almacenado en $A[n]$ es $tmp[r]$; se "recurre" ordenando $tmp[r+1..hi)$ en $A[n+1..hi)$.
- Si $r = hi$, el caso es similar al anterior, pero se procesa el siguiente valor en la mitad izquierda pues la derecha no tiene elementos por ser procesados; se "recurre" ordenando $tmp[l+1..mid)$ en $A[n+1..hi)$.
- De lo contrario, $l \neq mid$ y $r \neq hi$ y se debe decidir entre $tmp[l]$ y $tmp[r]$ para actualizar $A[n]$. Si $tmp[l] \leq tmp[r]$ entonces se actualiza $A[n]$ con $tmp[l]$, y se "recurre" ordenando $tmp[l+1..mid)$ y $tmp[r..hi)$ en $A[n+1..hi)$. De lo contrario (i.e., $tmp[l] > tmp[r]$) se actualiza $A[n]$ con $tmp[r]$, y se "recurre" ordenando $tmp[l..mid)$ y $tmp[r+1..hi)$ en $A[n+1..hi)$.

Por la hipótesis inductiva, dado que hay menos elementos que ordenar en los casos recurrentes, se tiene que $A[low..hi)$ es un ordenamiento ascendente de $tmp[low..hi)$.

En consecuencia, como $tmp[low..hi)$ es una copia de la versión inicial de $A[low..hi)$, se concluye que `merge(A, low, mid, hi)` ordena ascendemente a $A[low..hi)$.

Con base en la correctitud (o corrección) de `merge`, se puede establecer la correctitud de `mergesort`.

Teorema 2.4.2

Para $N = len(A)$ y $0 \leq low \leq hi \leq N$, el llamado `mergesort(A, low, hi)` ordena ascendentemente $A[low..hi)$.

Demostración

Se procede por inducción sobre el tamaño de $A[low..hi)$:

- Caso base ($low + 1 \geq hi$): en este caso, `mergesort` no modifica a A. Note que cuando $low + 1 \geq hi$, $A[low..hi)$ es vacío o tiene exactamente un elemento, por lo cual está ordenado ascendentemente.
- Caso inductivo ($low + 1 < hi$): en este caso, el problema se divide en dos subproblemas: ordenar $A[low..mid)$ y $A[mid..hi)$. Bajo la condición dada, el código garantiza que la desigualdad $low < mid < hi$ (esto se debe demostrar y se propone como ejercicio para el lector). Esto garantiza que cada uno de los sub-arreglos tiene al menos un elemento y ambos tienen menor tamaño que $A[low..mid)$: como son más pequeños,

por hipótesis inductiva, se puede suponer que después de los llamados recurrentes `mergesort(A, low, mid)` y `mergesort(A, mid, hi)` ordenan $A[low..mid)$ y $A[mid..hi)$, respectivamente. Dado que `merge` es correcto (Teorema 2.4.1), el llamado `merge(A, low, mid, hi)` ordena ascendentemente a $A[low..hi)$.

En cualquiera de los dos casos, $A[low..hi)$ resulta ordenado ascendentemente por `mergesort(A, low, hi)`.

La ventaja que tiene el algoritmo `mergesort` con respecto al primer algoritmo de ordenamiento estudiado, es que su comportamiento asintótico en función de la cantidad de operaciones que lleva a cabo para ordenar un arreglo es mejor (i.e., crece menos rápido).

Teorema 2.4.3

Para $N = len(A)$ y $0 \leq low \leq hi \leq N$, el llamado `mergesort(A, low, hi)` toma tiempo $O((hi - low)\log(hi - low))$. En consecuencia, `mergesort(A, 0, N)` toma tiempo $O(N \log N)$.

Demostración

Sea $n = hi - low$ (note que $n \geq 0$). A modo de ejercicio para el lector, se puede establecer que `merge(A, low, mid, hi)` toma tiempo $O(hi - low) = O(n)$ (esto es lo que cuesta combinar las soluciones a los subproblemas de `mergesort`). Entonces, la ecuación de recurrencia que define la cantidad de operaciones básicas está dada por

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

pues se recurre una vez en cada una de las dos mitades y cuesta $O(n)$ combinar estas soluciones. De acuerdo con el Teorema Maestro, se tiene que

$$T(n) \in O(n \log n).$$

Note que con $low = 0$ y $hi = N$, que corresponden al llamado `mergesort(A, 0, N)`, se tiene $n = N$ y consecuentemente $T(N) \in O(N \log N)$.

Finalmente, se analiza la complejidad espacial de `mergesort`.

Teorema 2.4.4

Para $N = len(A)$ y suponiendo que la variable `tmp` es suministrada para `merge`, el llamado `mergesort(A, 0, N)` usa $O(\lceil \log N \rceil)$ espacio, en donde $\lceil \_ \rceil$ es la función techo.

Demostración

Se propone como ejercicio para el lector.

Ejercicios

1. En la demostración del Teorema 2.4.1 se usa la siguiente igualdad:

   $$n - low = (l - low) + (r - mid)$$

   Demuestre que esta igualdad es cierta. Concluya que, cuando $n = hi$ y con base en el invariante P_1, $l = mid$ y $r = hi$.

2. Demuestre que la desigualdad $low < mid < hi$ es cierta en `mergesort`.

3. Demuestre el Teorema 2.4.3.

4. Demuestre el Teorema 2.4.4.

5. Considere la función `merge` que es llamada desde la función `mergesort` en la Implementación 2.4.1. Allí se usa la variable global `tmp` para almacenar una copia de partes de `A` en cada llamado. Suponiendo que dicha variable no es global y que en cada llamado recurrente se crea un arreglo de tamaño $hi - low$ para la copia temporal, demuestre que `mergesort(A, 0, N)` usa espacio $O(N \log N)$. Justifique su respuesta.

6. La cantidad de inversiones de un arreglo es un indicador de qué tan desordenado está: si este valor es 0, entonces el arreglo está completamente ordenado. Si el arreglo está ordenado descendentemente, entonces este valor es el máximo. Formalmente, una *inversión* en en un arreglo $A[0..N)$, $N \geq 0$, es una pareja de índices $0 \leq i < j < N$ tales que $A[i] > A[j]$. Por ejemplo, en el arreglo $[3, 1, 2]$ hay dos inversiones: $(3, 1)$ y $(3, 2)$.
   a) Diseñe un algoritmo que calcule la cantidad de inversiones en $A[0..N)$ en tiempo $O(N \log N)$.
   b) Demuestre que el algoritmo propuesto es correcto con respecto a la especificación.

2.5. Búsqueda binaria

El problema de búsqueda binaria consiste en encontrar un valor en un arreglo ordenado.

Problema 2.5.1: Búsqueda binaria

Entrada: Un arreglo $A[0..N)$ de números ordenado ascendentemente, $N \geq 0$, y un número x.

Salida: ¿Está x en $A[0..N)$?

Esta versión del problema corresponde al problema de decisión, es decir, el que requiere responder *sí* o *no*. Hay otras versiones en las cuales se desea averiguar más información como, por ejemplo, un índice en el arreglo en el cual aparece el valor dado (si aparece) o la cantidad de ocurrencias de este. También hay variantes en las cuales el arreglo dado no está ordenado ascendentemente sino descendentemente. Cualquiera de estas variantes se puede resolver con modificaciones (en su mayoría sencillas) del algoritmo que se presenta en esta sección.

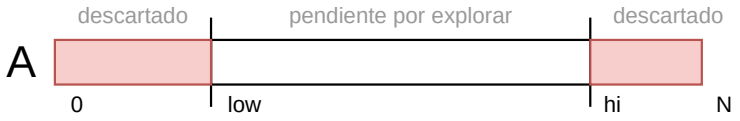
La idea general es explotar la información que se tiene sobre el arreglo $A[0..N)$. En particular, el hecho de que esté ordenado (ascendentemente, aunque –como se explica anteriormente– puede estar ordenado de otra forma) brinda la posibilidad de evitar hacer comparaciones y operaciones innecesarias. La idea general es la siguiente: suponga que se compara el valor dado x con el valor $A[n]$ del arreglo ($0 \leq n < N$). Allí hay tres opciones:

- Si $A[n] = x$, entonces x está en $A[0..N)$ y se responde afirmativamente.
- Si $A[n] < x$, entonces x necesariamente NO está en $A[0..n)$ dado que el arreglo está ordenado; en este caso, solo tiene sentido buscar a partir del índice $n + 1$. Es decir, se puede recurrir con un problema de tamaño menor.
- Si $A[n] > x$, entonces x necesariamente NO está en $A[n..N)$ dado que el arreglo está ordenado; en este caso, solo tiene sentido buscar antes del índice n. Es decir, se puede recurrir con un problema de tamaño menor.

Cualquier caso, bajo las suposiciones hechas en la especificación del problema, cae en una de las tres opciones enumeradas anteriormente. Así, se puede explicar de manera más precisa la idea central detrás del algoritmo de búsqueda: descartar rápidamente porciones del arreglo de entrada durante el proceso de búsqueda teniendo la certeza de que la porción del espacio de búsqueda que se preserva tiene suficiente información para responder la pregunta. La razón por la cual el algoritmo tiene la palabra "binaria" en su nombre es por la forma en cual el espacio restante

para la búsqueda se divide en dos "mitades", de las cuales una se mantiene y la otra se descarta.

El diseño general del algoritmo se presenta a continuación:



El algoritmo mantiene dos *centinelas* low y hi que definen las fronteras del espacio de búsqueda activo $A[low..hi)$ para el valor x. Las otras dos porciones $A[0..low)$ y $A[hi..N)$ han sido descartadas y se tiene certeza de que no es necesario seguir buscando allí. Este diseño usa las siguientes condiciones:

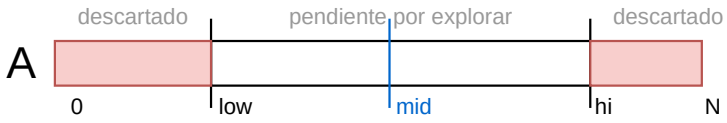
$$C_0: \quad (\exists i \mid 0 \leq i < N : A[i] = x) \equiv (\exists i \mid low \leq i < hi : A[i] = x).$$
$$C_1: \quad 0 \leq low \leq hi \leq N.$$

La lectura de la condición C_0 se puede facilitar si la equivalencia lógica se interpreta como una doble implicación: si el valor está en el arreglo, entonces está en el espacio de búsqueda pendiente por explorar (implicación de izquierda a derecha); si el valor está en el espacio pendiente por explorar, entonces (claramente) está en el arreglo (derecha a izquierda). De manera alternativa y apelando a la contraposición proposicional, esta segunda implicación se puede interpretar de la siguiente forma: si el valor NO está en el arreglo, entonces NO está en el espacio de búsqueda pendiente por explorar. La condición C_1 es auxiliar y limita los valores que las variables low y hi pueden tomar. Es importante hacer dos observaciones sobre estas variables. Primero, low nunca debe sobrepasar a hi. Segundo, bajo estas restricciones necesariamente low es un índice de $A[0..N)$ cuando el arreglo no es vacío (i.e., cuando $N \neq 0$), pero este no es el caso con hi (¿por qué?). Estas propiedades son importantes pues la corrección y terminación del algoritmo dependen, en gran medida, de que sean ciertas.

La idea del algoritmo de búsqueda binaria es estrechar incrementalmente el espacio pendiente por explorar. Es decir, la idea es que las variables low y hi se acerquen de tal manera que la cantidad de valores pendientes por explorar $A[low..hi)$) sea cada vez más pequeña. ¿En qué casos indican estas variables que se ha estrachado suficientemente el cerco? De acuerdo con la condición C_1, estos casos están dados por $low = hi$ y $low+1 = hi$: o no hay espacio por explorar o hay exactamente una posición de A por ser explorada. En el primer caso, claramente la respuesta es negativa pues un arreglo vacío no contiene valor alguno. Como se verá en los algoritmos que se diseñarán, este caso base solo es posible cuando el arreglo $A[0..N)$ es vacío. En el segundo caso, esto quiere decir que la máxima reducción que se puede hacer del espacio de búsqueda, de acuerdo con los invariantes propuestos, es a $A[low..low+1)$ que corresponde a $A[low]$. Si se combina esta observación con la

condición C_0, entonces se habría logrado reducir la búsqueda en todo el arreglo a solo un punto: para determinar si x está en $A[0..N)$, basta con determinar si x es el valor en $A[low]$.

La siguiente pregunta a responder es la siguiente: ¿en qué casos (y cómo) se debe estrechar el cerco establecido por *low* y *hi* sobre $A[0..N)$? Estos son en realidad los casos inductivos y están caracterizados por la desigualdad $low+1 < hi$. Note que bajo esta suposición, existe un número entero *mid* que satisface $low < mid < hi$. Esto en combinación con la condición C_1, cuando $N \neq 0$, garantiza que *mid* es un índice del arreglo $A[0..N)$ y por ende, se puede consultar el valor almacenado allí: es precisamente contra $A[mid]$ que se compara x para decidir cuál de las dos partes $A[low..mid)$ o $A[mid..hi)$ descartar directamente. El cómo estrechar el cerco se responde de manera oportunista: como se ha de descartar una de dos partes, es conveniente que (sin saber a priori cuál) las dos partes sean lo suficientemente grandes como para estrechar el cerco lo más ajustado posible. Es de esta forma como se llega a la conclusión de que es conveniente que *mid* sea un punto medio (puede haber más de uno, ¿por qué?) entre *low* y *hi*.



Algoritmo 2.5.1

1. si $low = hi$, entonces responder negativamente;
2. si $low + 1 = hi$, entonces responder afirmativamente si $A[low] = x$ y negativamente de lo contrario; y
3. si $low + 1 < hi$, entonces sea *mid* un punto medio entre *low* y *hi* y:
   a) si $A[mid] = x$, entonces responder afirmativamente;
   b) si $A[mid] < x$, entonces recurrir sobre $A[mid..hi)$; y
   c) si $A[mid] > x$, entonces recurrir sobre $A[low..mid)$.

Con base en este diseño se propone la siguiente implementación:

Implementación 2.5.1: Búsqueda binaria basada en el Algoritmo 2.5.1

```
1 def binsearch(A, x, low, hi):
2   ans = None
3   if low==hi: ans = False
4   elif: low+1==hi: ans = A[low]==x
5   else:
```

```
    mid = low+((hi-low)>>1)      # mid = (low+hi)//2
    if A[mid]==x: ans = True
    elif A[mid]<x: ans = binsearch(A, x, mid, hi)
    else: ans = binsearch(A, x, low, mid)
  return ans
```

La función `binsearch` identifica tres casos. Si el espacio de búsqueda es vacío, entonces responde negativamente. Si el espacio de búsqueda ha sido reducido a una sola posición, entonces se responde con base en lo que hay en esa posición: si x está allí, entonces se responde afirmativamente; de lo contrario, negativamente. Si el espacio de búsqueda tiene al menos dos elementos, entonces se calcula la "mitad" *mid* del subarreglo y se recurre con base en la relación existente entre x y el elemento $A[mid]$, cuando el elemento no se encuentra allí. Esta es una traducción literal Algoritmo 2.5.1 al lenguaje de programación Python.

A continuación se presenta la demostración de que la función `binsearch` es correcta con respecto a su especificación.

Teorema 2.5.1

Para $N = len(A)$ y $0 \leq low \leq hi \leq N$, el llamado `binsearch(A, x, low, hi)` determina si x está en $A[low..hi)$. En consecuencia, `binsearch(A, x, 0, N)` determina si x está en $A[0..N)$.

Demostración

Se procede por inducción sobre la distancia entre *low* y *hi*:

- Caso base ($low = hi$): el espacio de búsqueda es vacío, luego se responde negativamente
- Caso base ($low + 1 = hi$): el espacio de búsqueda tiene una sola posición $A[low]$, entonces se determina directamente si x corresponde a es valor.
- Caso inductivo ($low + 1 < hi$): el espacio de búsqueda tiene al menos dos posiciones. Por la forma en que se calcula *mid*, se puede demostrar que: (i) $low < mid < hi$ y (ii) *mid* es un índice del arreglo A (estas demostraciones se proponen como ejercicio para el lector). Con base en estas observaciones, se tiene que la búsqueda de x en $A[low..mid)$ y $A[mid..hi)$, respectivamente, son subproblemas de la búsqueda en $A[low..hi)$. Note que si $A[mid] < x$, es imposible que x esté en $A[low..mid)$ (¿por qué?). De manera similar, si $A[mid] > x$, es imposible que x esté en $A[mid..hi)$ (¿por

qué?). En cada uno de los casos se recurre sobre la "mitad" que tiene el potencial de contener a x.

En consecuencia, el llamado `binsearch(A, x, 0, N)` resuelve el problema de búsqueda de `x` en el arreglo `A` ordenado ascendentemente.

La demostración de que la función `binsearch` termina se propone como ejercicio para el lector.

Teorema 2.5.2

Para $N = len(A)$ y $0 \leq low \leq hi \leq N$, el llamado `binsearch(A, x, low, hi)` termina.

Demostración

Se propone como ejercicio para el lector.

Teorema 2.5.3

Para $N = len(A)$, el llamado `binsearch(A, x, 0, N)` toma $O(\lceil \log N \rceil)$ tiempo, en donde $\lceil \_ \rceil$ es la función techo.

Demostración

Ver el Ejemplo 1.4.3.

Una versión iterativa del algoritmo de búsqueda binaria se presenta a continuación. Inicialmente identifica dos casos. Si el arreglo dado es vacío, la respuesta necesariamente es negativa. De lo contrario, iterativamente se estrecha el cerco de la búsqueda tratando de acercar `low` y `hi` hasta que sean números consecutivos. En esta segunda parte, en la cual en realidad se lleva a cabo la búsqueda, note que la inicialización `low,hi = 0,N` hace que las condiciones C_0 y C_1, vistas como invariantes del ciclo iterativo, sean ciertas trivialmente: inicialmente el espacio de búsqueda es todo el arreglo.

Implementación 2.5.2: Búsqueda binaria iterativa

```python
def binsearch(A, x):
  N,ans = len(A),False
  if N!=0:
   low,hi = 0,N
   # C_0 ∧ C_1
   while low+1!=hi:
     mid = low+((hi-low)>>1)    # mid = (low+hi)//2
     if A[mid]<=x: low = mid
     else: hi = mid
   ans = A[low]==x
  return ans
```

La correctitud y el análisis asintótico de la versión iterativa de `binsearch` se proponen como ejercicio para el lector.

Ejercicios

1. Simule la ejecución del llamado `binsearch(A, 0, len(A))` para
   `A = [ -4, 2, 2, 7, 11, 14, 18, 23, 100]` y `x = 23`.
   Explique gráficamente qué sucede en cada llamado recurrente en relación con las variables `low`, `hi` y `mid`.
2. En la demostración del Teorema 2.5.1, caso inductivo, se hacen las afirmaciones (i) y (ii). Proponga demostraciones para cada una de ellas.
3. Demuestre el Teorema 2.5.2.
4. Demuestre el Teorema 2.5.3.
5. Proponga y demuestre el teorema de corrección del algoritmo de búsqueda binaria iterativa en la Implementación 2.5.2. Demuestre que el algoritmo termina.
6. Especifique el problema de búsqueda binaria en un arreglo ordenado *descendentemente*.
7. La siguiente lista de problemas tienen como entrada un arreglo de números $A[0..N)$, $N \geq 0$, y un número x.
   a) Identificar el primer índice en el que aparece x en $A[0..N)$.
   b) Identificar el último índice en el que aparece x en $A[0..N)$.
   c) Determinar la cantidad de veces que aparece x en $A[0..N)$.
   En cada uno de los problemas anteriores, la respuesta debe ser, e.g., -1 cuando la búsqueda falla. Con base en esta información y para cada uno de los problemas:
   a) Diseñe un algoritmo que resuelva el problema en tiempo $O(\log N)$.
   b) Demuestre que el algoritmo es correcto.

c) Justifique por qué el algoritmo propuesto termina.

8. En un arreglo $A[0..N)$ de números, un número x es *mayoritario* si aparece, al menos, $N/2$ veces. Suponiendo que $A[0..N)$ está ordenado ascendentemente:
   a) Diseñe un algoritmo que determine si x es mayoritario en tiempo $O(\log N)$.
   b) Demuestre que el algoritmo es correcto.
   c) Demuestre que el algoritmo termina.
   d) ¿Cómo puede resolver el problema si el arreglo no necesariamente está ordenado?

9. Un *punto fijo* en un arreglo es un índice que almacena su mismo valor. Por ejemplo, 3 es un punto fijo en $[-2, 5, 1, 3, 0, 0]$.
   a) Especifique el problema de encontrar un punto fijo en un arreglo $A[0..N)$ ordenado ascendentemente.
   b) Diseñe un algoritmo que resuelva el problema dado en tiempo $O(\log N)$.
   c) Demuestre que el algoritmo es correcto y termina.

10. En el ámbito de la estadística, la mediana representa el valor de la variable de posición central en un conjunto de datos ordenados. Por ejemplo, en el conjunto $\{7, 8, 9, 10, 11, 12\}$ la mediana es 9,5. Dado un arreglo de números ordenado ascendentemente, la mediana se puede calcular en tiempo $O(1)$. Un problema interesante, más bien, es determinar la mediana de un conjunto de datos cuando estos están divididos en dos grupos de igual tamaño.

    Entrada: Arreglos $A[0..N)$ y $B[0..N)$, $N \geq 0$, de números y ordenados ascendentemente.

    Salida: Mediana del conjunto de datos representado por los elementos de $A[0..N)$ y $B[0..N)$.

    Diseñe y analice un algoritmo que en tiempo $O(\log N)$ resuelva el problema dado. Justifique su respuesta.

11. Un arreglo de números no repetidos es *bitónico* si está conformado por una secuencia ascendente con al menos dos elementos seguida de una secuencia descendente con al menos dos elementos, o si puede ser rotado circularmente para que cumpla esta condición. Por ejemplo, $[1, 2, 3, 0, -1]$ y $[5, 4, 3, 2]$ son bitónicos, mientras que $[0, 2]$, $[2, 4, 3, 5]$ ni $[1, 2, 1]$ lo son. Suponiendo que los números son distintos, diseñe y analice un algoritmo que determine si una arreglo dado $A[0..N)$ es bitónico en tiempo $O(\log N)$.

12. Considere un arreglo $A[0..N)$ con la propiedad especial de que $A[0] \geq A[1]$ y $A[N-2] \leq A[N]$. Se dice que un índice n de $A[0..N)$ es un *mínimo local* cuando $A[n-1] \geq A[n] \leq A[n+1]$ (necesariamente $0 < n < N-1$). Con las condiciones dadas, el arreglo debe tener al menos un mínimo local. Por ejemplo, en $[7, 2, 2, 4, -1, 5, 6]$ los mínimos locales son 1, 2 y 4. Diseñe y analice un algoritmo que en tiempo $O(\log N)$ encuentre un mínimo local en $A[0..N)$.

13. Considere un arreglo $A[0..N)$ que ha sido rotado circularmente k pasos, con $1 \leq k < N$. Por ejemplo, el arreglo $[3, 7, 10, 1, 2]$ está rotado 3 pasos.
    a) Diseñe y analice un algoritmo que en tiempo $O(\log N)$ calcule la cantidad de pasos que ha sido rotado $A[0..N)$.
    b) Diseñe y analice un algoritmo que en tiempo $O(\log N)$ determine si un número x está en $A[0..N)$.

Notas del capítulo y referencias

El planteamiento en este capítulo de dividir, conquistar y combinar es similar al propuesto por T. Cormen et al. [**CLRS22**]. J. Kleinberg y É. Tardos [**KET06**], al igual que en este texto, introducen la técnica con *MergeSort*, y diseñan algoritmos para resolver problemas como los de encontrar pares de puntos más cercanos y la transformada rápida de Fourier. Estas dos referencias son fuentes variadas de ejercicios y problemas, sugeridas para profundizar en el tema.

Otros autores como H. Bhasin [**Bha15**] y A. Levitin [**Lev12**] distinguen diferentes versiones de dividir y conquistar. Por ejemplo, incluyen *decrementar* y conquistar, y *transformar* y conquistar. Por decrementar se entiende un caso de dividir y conquistar en el cual cada subproblema resulta de reducir el problema inicial en una cantidad constante y no en un factor constante (e.g., como el algoritmo de ordenamiento en la Sección 2.2 en donde cada subproblema tiene una posición menos que ordenar). Por transformar se entiende que las instancias iniciales son pre-procesadas antes de aplicar dividir y conquistar. Por ejemplo, si se desea determinar si un elemento en un arreglo es único, se puede ordenar el arreglo para luego determinar la unicidad del elemento. Estos textos también son una buena fuente de ejemplos y problemas algorítmicos.

El diseño de ciclos iterativos basado en invariantes es común en la literatura; ver, e.g., [**CLRS22**]. La disciplina de plantear invariantes a modo de predicados formales proviene de enfoques como los de E. Dijkstra [**Dij76**], D. Gries [**Gri81**], E. Cohen [**Coh90**] y A. Kaldewaij [**Kal90**]. En Castellano, el uso de predicados para la derivación de algoritmos iterativos y recurrentes es propuesto por J. Bohórquez [**Boh06**].

La inmensa mayoría de problemas propuestos como ejercicios en esta sección hacen parte del folclor de computación y algoritmos, y como tal se pueden encontrar en varias fuentes como libros, artículos científicos y divulgativos, e internet. El Ejercicio 2*b* de la Sección 2.1 es un resultado conocido como el Teorema de Gomory y cuya demostración usa ciclos Hamiltonianos. Los ejercicios 11, 12 y 13 de la Sección 2.5 están inspirados por sendos problemas en [**Eri19**].

Capítulo 3

Programación dinámica

La programación dinámica es una técnica de solución de problemas algorítmicos y de programación. Dada una función recurrente (matemáticamente hablando) que resuelve un problema algorítmico (e.g., decisión, conteo u optimización), la programación dinámica permite diseñar un algoritmo para computar dicha función evitando recalcular subinstancias iguales que aparecen en diferentes subproblemas. Esta situación es común en soluciones que resultan al aplicar, e.g., dividir, conquistar y combinar cuando en varias recurrencias resulta una misma instancia.

Al usar programación dinámica el objetivo es garantizar que no se repita el cálculo de ningún subproblema. La apuesta que hace la programación dinámica es entonces permitir que el cálculo de la función de interés use memoria a modo de "registro", de forma tal que los resultados intermedios (o los potencialmente intermedios) de la función se escriban una vez allí y se puedan consultar tantas veces como sea necesario.

Nota 3.0.1

Richard E. Bellman (1920–1984) fue un matemático aplicado que acuñó por primera vez, circa 1950, el término *programación dinámica*. El siguiente es un apartado traducido de su autobiografía (ver notas y referencias al final de capítulo) en el cual relata la 'curiosa' historia detrás de la intención de este nombre.

> Una pregunta interesante es, "¿de dónde viene el nombre 'programación dinámica'?" Los años 1950s no eran buenos para la investigación en matemáticas. Tuvimos un personaje muy interesante en Washington, llamado Wilson. Él era el Secretario de Defensa, y realmente

> tenía miedo y odio patológicos de la palabra 'investigación'. No estoy usando estos adjetivos ligeramente; los estoy usando precisamente. Su cara se descomponía, se ponía roja, y se tornaba violento si alguien usaba el término 'investigación' en su presencia. Se podrán imaginar cómo se sentía, entonces, acerca del término 'matemático'. La Corporación RAND estaba empleada por la Fuerza Aérea y la Fuerza Aérea tenía, escencialmente, a Wilson como jefe. Por esto, sentí la necesidad de hacer algo para proteger de Wilson y de la Fuerza Aérea el hecho de que en realidad estaba haciendo matemáticas al interior de la Corporación RAND. ¿Qué título, qué nombre, podía escoger? En primer lugar, estaba interesado en planeación (en inglés, *planning*), en toma de decisiones, en pensar. Pero la palabra 'planeación' no era una buena elección por varias razones. Por ello decidí usar el término 'programación'. Quería lograr la idea de que esto era dinámico, con múltiples escenarios, que dependía del tiempo; entonces se me ocurrió matar dos pájaros con una sola piedra. Usemos una palabra con un significado preciso y absoluto, es decir 'dinámica', en el sentido físico clásico. Esta palabra también tiene propiedades interesantes como adjetivo dado que es imposible usarla en un sentido peyorativo. Trate de pensar en una combinación que le haga tomar un significado peyorativo. Es imposible. Luego, pensé que 'programación dinámica' era un buen nombre. Era algo que ni siquiera un congresista podría objetar. Y ese fue el término que usé como sombrilla para mis actividades.

Este capítulo presenta las ideas principales que respaldan la programación dinámica, incluyendo técnicas de implementación, una metodología para su uso y varios ejemplos.

3.1. Ideas clave

La programación dinámica es una técnica que puede ser útil para resolver problemas algorítmicos. Sin embargo, no en todos los problemas algorítmicos tiene sentido usarla. Aquellos problemas que son susceptibles de abordar usando la técnica exhiben ciertas propiedades en común que relacionan la solución de instancias con subinstancias.

Nota 3.1.1

A continuación se enumeran dos propiedades de un problema algorítmico necesarias para resolverlo con programación dinámica:

- *Propiedad de subestructura*: debe ser fácil relacionar la solución del problema dado con las soluciones de subproblemas relacionados.
- *Propiedad de solapamiento*: al plantear una solución recurrente, repetitivamente resultan instancias del mismo subproblema.

Un ejemplo conocido de la propiedad de subestructura es el de ordenamiento de un arreglo de números (Problema 2.2.1): ordenar un arreglo $A[0..N)$ se puede relacionar fácilmente con ordenar el subarreglo $A[1..N)$. Sin embargo, este problema no cuenta con la propiedad de solapamiento: e.g., ordenar $A[0..n)$ y $A[n..N)$, con $0 < n < N$, son subproblemas de ordenar $A[0..N)$ que no dependen el uno del otro. Es importante aclarar que la noción de 'subproblema' debe estar asociada a una medida concreta y bien definida como, e.g., el tamaño del arreglo o subarreglo a ser ordenado; esto dependerá del problema mismo.

La propiedad de subestructura se puede especializar dependiendo de la naturaleza del problema que se desea resolver (e.g., decisión, conteo u optimización). En el caso de un problema de optimización, la propiedad de subestructura es denominada 'propiedad de subestructura óptima' (o 'subestructura óptima'). El principio de subestructura óptima indica que la soluciones óptimas de un problema incorporan soluciones óptimas de sus subproblemas relacionados. El término original se puede especializar para problemas de decisión o conteo, resultando en principios particulares.

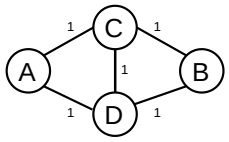
Ejemplo 3.1.1

Considere el problema de determinar la ruta más corta de un punto A a un punto B en una ciudad. Note que si una ruta óptima (porque puede haber más de una) de A a B pasa por un punto C, entonces necesariamente en esa ruta la subruta de A a C también debe ser óptima. De lo contrario, la ruta de A a C no sería óptima: se podría llegar a C "más rápido" de otra forma y, en consecuencia, a B más rápido usando otra ruta. Pero esto riñe con la suposición de que la ruta de A a B que pasa por C es óptima; i.e., no puede haber una ruta mejor. Para este problema, ¿pueden existir solapamientos entre subproblemas?

A simple vista pareciera que la propiedad de subestructura es innocua por su sencillez y obviedad; sin embargo, no todos los problemas cuentan con esta propiedad.

Ejemplo 3.1.2

Considere el problema de determinar la ruta *simple* más larga de un punto A a un punto B en una ciudad. Simple se refiere a que no se visita un sitio más de una vez (i.e., no hay ciclos, porque de lo contrario el problema estaría indefinido). En este problema, el principio de optimalidad *no* aplica. Considere la siguiente instancia en donde hay 4 sitios en la ciudad llamados A, B, C y D, relacionados de la siguiente manera:



El camino simple más largo de A a B cuesta/toma/demora 3 unidades. Si el principio de optimalidad fuera cierto en este problema, el tamaño del camino más largo de A a B se podría obtener de los caminos más largos de A a puntos intermedios y de estos puntos intermedios a B. Pero esto falla, por ejemplo, cuando se toma D como punto "pivote" pues los caminos simples más largos no se pueden componer para obtener un camino simple más largo: el camino simple más largo de A a C es $A \to D \to B \to C$ y el camino simple más largo de C a B es $C \to A \to D \to B$. Sin embargo, la composición de estos dos caminos

$$A \to D \to B \to C \to A \to D \to B$$

no es simple.

Ejercicios

1. Justificando su respuesta, responda la pregunta formulada al final del Ejemplo 3.1.1.
2. Considere el problema de determinar si dada una colección de números $A[0..N)$ y un número x es posible escoger algunos elementos de A cuya suma es x. Determine si este problema exhibe las propiedades de subestructura y solapamiento. Justifique e ilustre su respuesta con un ejemplo.
3. Justifique por qué el problema de buscar un valor en un arreglo dado no cumple las propiedades de subestructura y solapamiento.
4. Suponga que se cuenta con el registro de precios de una acción a lo largo de N días, un precio por día. Se desea determinar la longitud de la mejor "corrida" de la acción durante los N días: es decir, la mayor cantidad de días (no necesariamente consecutivos) en los cuales la acción aumentó de precio. ¿Exhibe este problema alguna de las propiedades de subestructura o solapamiento? Justifique e ilustre su respuesta con un ejemplo.
5. Un empresario desea hacer una gira por varias ciudades conduciendo su vehículo (debe iniciar y terminar en la ciudad de origen, visitando cada una de las demás ciudades exactamente una vez). Para cada trayecto cuenta con la cantidad de dinero que cuesta el desplazamiento. Suponiendo que existe un camino directo entre cualquier par de ciudades, indique si determinar el menor costo posible de la gira exhibe alguna de las propiedades de subestructura o solapamiento. Justifique e ilustre su respuesta con un ejemplo.

3.2. Cálculo de los números de Fibonacci

Se deja de lado, por un momento, el proceso de plantear una función recurrente que resuelva un problema con las propiedades del subproblema y solapamiento. El objetivo de esta sección es presentar las dos principales formas, *memorización* y *tabulación*, de instrumentar la programación dinámica. Para ilustrarlas, se parte de una recurrencia bien conocida: la función que calcula los números de Fibonacci.

Definición 3.2.1

Para $n \in \mathbb{N}$, la definición de la función *fib* es la siguiente:

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

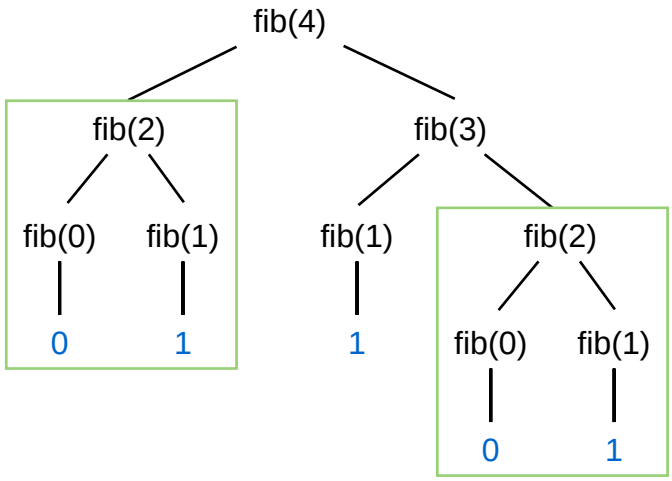
El problema que se desea resolver es el siguiente.

Problema 3.2.1: Cálculo de la función de Fibonacci

Entrada: Un número entero N, $N \geq 0$.

Salida: $\mathit{fib}(N)$.

Calcular *fib* es en el fondo un problema de conteo. Hay dos casos bases: para $N = 0$ o $N = 1$, se tiene que $\mathit{fib}(N) = N$. El caso inductivo se plantea para cualquier otro valor de N (recuerde que es un número natural) recurriendo sobre los dos valores inmediatamente anteriores (por eso son necesarios los dos casos base). Note que el Problema 3.2.1 cuenta con las propiedades de subestructura y solapamiento. Por un lado, el cálculo recurrente de la función depende de sí misma, con valores más pequeños (en el orden usual de los números). Por otro lado, dos llamados recurrentes distintos dependen de un mismo suproblema. Por ejemplo, en el cálculo de $\mathit{fib}(4)$ se recalcula $\mathit{fib}(2)$:



Una implementación directa de esta función en el lenguaje de programación Python 3 se presenta a continuación.

Implementación 3.2.1: Cálculo de la función fib

```
def fib(n):
  ans = None
  if n<=1: ans = n
  else: ans = fib(n-2)+fib(n-1)
  return ans
```

La función `fib` se usa a continuación con algunos ejemplos:

```
print('fib(0):', fib(0))
print('fib(1):', fib(1))
print('fib(2):', fib(2))
print('fib(10):', fib(10))
print('fib(15):', fib(15))
```

El resultado de esta ejecución es el siguiente:

```
fib(0): 0
fib(1): 1
fib(2): 1
fib(10): 55
fib(15): 610
```

Si la implementación de la función *fib* es tan sencilla y básicamente resulta de transcribir la definición matemática como código, ¿por qué hay que preocuparse por usar programación dinámica? Para responder esa pregunta directamente (pero no necesariamente rápidamente) vale la pena tratar de calcular `fib(100)`. Posiblemente en un par de días aún no habrá terminado el cálculo, pero se entenderá por qué es necesario usar programación dinámica: los llamados recurrentes se solapan y, en consecuencia, muchos llamados de `fib` se repiten; hay valores que se calculan una y otra vez. En realidad, la complejidad temporal de la función `fib(n)` es $O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$, lo cual resulta de resolver la ecuación de diferencia $X^2 - X - 1 = 0$ con variable X (los detalles de una demostración que soporte esta afirmación se proponen como ejercicio para el lector). Es decir, la complejidad temporal de `fib` es exponencial, lo cual no es una buena noticia si el objetivo es contar con un algoritmo eficiente para calcular *fib*.

La forma en que resulta útil la programación dinámica es precisamente almacenando el resultado de los cálculos, ahorrando tiempo de cómputo cuando sea necesario visitar una y otra vez dichas instancias en cada uno de los subproblemas.

Ejemplo 3.2.1

Hoy dos formas usualmente empleadas para diseñar un algoritmo con programación dinámica y que permiten implementar eficientemente la función recurrente asociada a su solución:

Memorización: se calcula la función objetivo *por demanda* de modo tal que los valores intermedios, resultado de los llamados recurrentes, se van calculando en la medida que sea necesario. Generalmente, resultan algoritmos recurrentes muy parecidos a la función objetivo. Hay una memoria compartida que se usa: para una instancia específica del problema se consulta si ha sido resuelta antes (i.e., el valor está en la memoria compartida), entonces se usa el valor registrado en la memoria; de lo contrario, se calcula el valor correspondiente a dicha instancia, ya bien sea directa o recurrentemente, y se almacena en la memoria compartida para que pueda ser usado posteriormente de ser necesario.

Tabulación: se calcula la función objetivo *exhaustivamente* considerando todos los casos posibles para implementar la función dada con base en los parámetros de interés. La memoria compartida se va "llenando" incrementalmente con todos estos casos hasta obtener el valor deseado. Generalmente, resultan algoritmos iterativos.

En lo que resta de la sección se ilustra cómo usar memorización y tabulación para obtener una implementación eficiente de la función *fib*.

3.2.1. Memorización. La programación dinámica con memorización es conocida en inglés como programación dinámica *top-down*. Básicamente, se extiende la función a implementar con una memoria compartida que puede ser leída y modificada en los llamados recurrentes. Esto, específicamente, puede hacerse con una variable global o aumentando la lista de parámetros de la función con una variable que es pasada por referencia. Una de las ventajas de las implementaciones con memorización es que la estructura del código resultante es básicamente la misma que la de una transcripción de la función objetivo.

La idea general es la siguiente:

- Si el valor que se desea calcular está en la memoria compartida (i.e., ha sido calculado previamente), este valor se usa.
- De lo contrario, se calcula directa o recurrentemente el valor deseado y se almacena en la memoria compartida. En las invocaciones recurrentes para los subproblemas también se hace uso de la memoria compartida de modo tal que los cálculos intermedios queden almacenados en ella.

La implementación de *fib* con memorización se presenta en el Algoritmo 3.2.2 con la función `fib_memo`; un diccionario representa la memoria compartida y es identificado como `mem`.

Implementación 3.2.2: Cálculo de fib con memorización

```
def fib_memo(n, mem):
  ans = None
  if n in mem: ans = mem[n]   # if the value is available, use it!
  else:                       # otherwise, it needs to be computed
    if n<=1: ans = n
    else: ans = fib_memo(n-2, mem)+fib_memo(n-1, mem)
    mem[n] = ans              # store the value in the shared memory
  return ans
```

Cuando el valor que se quiere calcular está disponible en `mem`, el algoritmo lo retorna. De lo contrario, el valor se calcula (incluyendo los casos base) y, una vez se cuenta con este valor, se almacena antes de retornarlo. Note que los llamos recurrentes también hacen uso de la memoria compartida: dado que `mem` se pasa por referencia, todos los cálculos que se hagan en los llamados recurrentes quedan almacenados en `mem`.

Es importante observar que la función `fib_meo` se comporta correctamente siempre y cuando se respeten algunas condiciones sobre sus parámetros. En particular, se debe garantizar que `mem` no contiene "basura" en el sentido de que los valores almacenados allí son correctos. De lo contrario, no habría garantía acerca de los cálculos realizados ni del valor retornado. Por ello, es importante contar con condiciones de representación que apoyen el diseño del algoritmo.

$$C_0: \quad (\forall k \mid k \in \mathbb{N} : k \in \texttt{mem} \Rightarrow \texttt{mem}[k] = \mathit{fib}(k)).$$
$$C_1: \quad n \geq 0.$$

La condición C_0 indica que si k es una de las llaves del diccionario, necesariamente dicha llave está asociada al valor $\mathit{fib}(k)$; es decir, lo que está almacenado en el diccionario (suponiendo que solamente hay números naturales como llaves) son valores correctos de Fibonacci. La condición C_1 es auxiliar e indinca que el valor de n siempre es un número natural.

Teorema 3.2.1

Sea $n \in \mathbb{N}$ y `mem` un diccionario. Si n y `mem` satisfacen las condiciones C_0 y C_1, entonces el llamado `fib_memo(n, mem)` es tal que:

1. al terminar su ejecución C_0 y C_1 son ciertos; y
2. calcula $fib(n)$.

Demostración

Se procede por análisis de casos sobre `fib_memo` (usando inducción):

- Caso $n \in$ `mem`: no se modifica `mem`, por lo cual C_0 es cierto dada la suposición. Como C_0 se supone y $n \in$ `mem`, entonces `mem`$(n) = fib(n)$. Luego, `fib_memo(n, mem)` calcula $fib(n)$.
- Caso $n \notin$ `mem`: si $n = 0$ o $n = 1$, entonces `ans` tiene el valor de $fib(n)$. Si $n \geq 2$, por las hipótesis inductivas se tiene que `ans` tiene el valor $fib(n)$. Por la misma razón se puede suponer que la variable `mem` satisface la condición C_0 después de los dos llamados recurrentes. Luego, aumentar `mem` con `mem[n] = ans` garantiza que C_0 sigue siendo cierto una vez termine la ejecución de `fib_memo(n, mem)`.

  Se propone como ejercicio al lector demostrar que la condición C_1 es cierta bajo las suposiciones del enunciado del teorema.

Resta por hacer explícito cómo `fib_memo` resuelve el problema planteado inicialmente.

Teorema 3.2.2

El llamado `fib_memo(N, {})`, en donde `{}` representa el diccionario vacío, calcula $fib(N)$.

Demostración

La demostración es un corolario del Teorema 3.2.1 y se propone como ejercicio al lector.

A continuación se muestra el resultado de ejecución de `fib_memo`, incluyendo el cálculo para $n = 100$, y el contenido de la memoria compartida después de este llamado. Note que todos los valores hasta 100 han sido calculados y almacenados en la memoria compartida. Una ventaja práctica de la memorización es que la memoria compartida puede ser calculada incrementalmente: por ejemplo, hacer el llamado para calular `fib_memo(15, mem)` y posteriormente hacer el llamado para calcular

`fib_memo(100, mem)` de tal forma que en el segundo llamado se cuente con algunos valores previamente (y correctamente) calculados en `mem`.

```
print('fib(0):', fib_memo(0, dict()))
print('fib(1):', fib_memo(1, dict()))
print('fib(2):', fib_memo(2, dict()))
print('fib(10):', fib_memo(10, dict()))
print('fib(15):', fib_memo(15, dict()))

mem = dict()
print('fib(100):', fib_memo(100, mem))
print(mem)
```

El resultado de esta ejecución es el siguiente:

```
fib(0): 0
fib(1): 1
fib(2): 1
fib(10): 55
fib(15): 610
fib(100): 354224848179261915075
{0: 0, 1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8, 7: 13, 8: 21, 9: 34, 10: 55,
 11: 89, 12: 144, 13: 233, 14: 377, 15: 610, 16: 987, 17: 1597,
 18: 2584, 19: 4181, 20: 6765, 21: 10946, 22: 17711, 23: 28657,
 24: 46368, 25: 75025, 26: 121393, 27: 196418, 28: 317811, 29: 514229,
 30: 832040, 31: 1346269, 32: 2178309, 33: 3524578, 34: 5702887,
 35: 9227465, 36: 14930352, 37: 24157817, 38: 39088169, 39: 63245986,
 40: 102334155, 41: 165580141, 42: 267914296, 43: 433494437,
 44: 701408733, 45: 1134903170, 46: 1836311903, 47: 2971215073,
 48: 4807526976, 49: 7778742049, 50: 12586269025, 51: 20365011074,
 52: 32951280099, 53: 53316291173, 54: 86267571272, 55: 139583862445,
 56: 225851433717, 57: 365435296162, 58: 591286729879, 59: 956722026041,
 60: 1548008755920, 61: 2504730781961, 62: 4052739537881,
 63: 6557470319842, 64: 10610209857723, 65: 17167680177565,
 66: 27777890035288, 67: 44945570212853, 68: 72723460248141,
 69: 117669030460994, 70: 190392490709135, 71: 308061521170129,
 72: 498454011879264, 73: 806515533049393, 74: 1304969544928657,
 75: 2111485077978050, 76: 3416454622906707, 77: 5527939700884757,
 78: 8944394323791464, 79: 14472334024676221, 80: 23416728348467685,
 81: 37889062373143906, 82: 61305790721611591, 83: 99194853094755497,
 84: 160500643816367088, 85: 259695496911122585, 86: 420196140727489673,
 87: 679891637638612258, 88: 1100087778366101931,
 89: 1779979416004714189, 90: 2880067194370816120, 91: 4660046610375530309,
```

```
29 92: 7540113804746346429, 93: 12200160415121876738,
30 94: 19740274219868223167, 95: 31940434634990099905,
31 96: 51680708854858323072, 97: 83621143489848422977,
32 98: 135301852344706746049, 99: 218922995834555169026,
33 100: 354224848179261915075}
```

Finalmente, se establece el orden de la complejidad temporal de `fib_memo`.

Teorema 3.2.3

Sea $n \in \mathbb{N}$. Si el acceso y modificación de `mem` con `mem[n]` toma tiempo $O(\psi(n))$, entonces la complejidad temporal del llamado `fib_memo(n, mem)` toma tiempo $O(n\psi(n))$.

Demostración

Se propone como ejercicio al lector.

Si el acceso y modificación de diccionarios es de orden constante, entonces la complejidad de `fib_memo` es de orden lineal en n. Es importante resaltar que se está suponiendo algo que en la práctica no es necesariamente cierto: operar números grandes es de orden constante. En realidad, como la función *fib* crece rápidamente, es fácil encontrar valores de n no muy grandes para los cuales $fib(n)$ sobrepasa la cota de 2^{63} para números enteros en un procesador de 64 bits. En el caso particular de `fib_memo`, Python "esconde" del usuario esta complicación dado que sus enteros básicos son de precisión arbitraria. Este no necesariamente es el caso en otros lenguajes de programación, y deben usarse librerías espciales o implementar las operaciones de enteros grandes partiendo de cero. Con cualquiera de estas opciones, las operaciones aritméticas dejan de ser de orden constante en el peor de los casos.

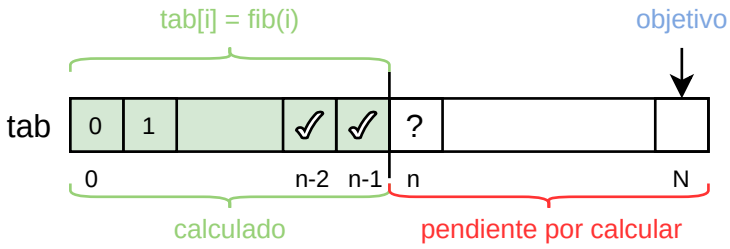
3.2.2. Tabulación. Al igual que la memorización, la tabulación es una alternativa para diseñar algoritmos con programación dinámica. Esta técnica es conocida en inglés como *bottom-up*. Aunque sirven el mismo propósito, los algoritmos con tabulación son intrínsecamente distintos a los que resultan con memorización. Primero, los algoritmos con tabulación son iterativos por naturaleza. Segundo, con memorización los valores de la función objetivo se calculan por demanda, mientras que con tabulación se pueden calcular más valores intermedios de los que son necesarios para calcular la función para el objetivo final. Tercero, en algunos casos las tabulaciones pueden ser optimizadas para ahorrar espacio. Esta última

característica hace que la tabulación sea preferida, en muchos situaciones, sobre la memorización.

A modo de ejemplo, se diseñará un algoritmo por tabulación que permita calcular la función *fib*. Esto se hará con la ayuda de una variable que almacene el cálculo parcial de los valores de la función hasta obtener el valor deseado.

$$P_0: \quad (\forall i \mid 0 \leq i < n : tab[i] = fib(i)).$$
$$P_1: \quad 0 \leq n \leq N + 1.$$

La variable de almacenamiento es el arreglo $tab[0..N+1)$ que permite guardar valores de *fib*. En particular, $tab[0..n)$ almacena los primeros n valores de *fib*. La variable n indica qué valores se han calculado; note que cuando $n = N+1$, el valor deseado está en $tab[N]$. Visualmente, estos invariantes se pueden representar como se muestra en el siguiente *diagrama de necesidades*:



En una tabulación, el diagrama de necesidades indica cómo se inicializa la memoria adicional (en este caso $tab[0] = 0$ y $tab[1] = 1$) y cómo los valores que se calculan recurrentemente (i.e., $fib[n]$ para $n \geq 2$) se pueden determinar a partir de valores previamente calculados en la memoria (en este caso, $fib[n-2]$ y $fib[n-1]$). Note que de acuerdo con el invariante P_0, los valores de *fib* están calculados correctamente en $tab[0..n)$, y n varía entre 0 y $N+1$, inclusive.

Con base en estos invariantes y en el diagrama de necesidades, se propone la función `fib_iter` en el Algoritmo 3.2.3 para calcular iterativamente el valor de la función *fib*.

Implementación 3.2.3: Cálculo de fib con tabulación

```
1 def fib_iter(N):
2   ans = None
3   if N<=1: ans = N
4   else:
5     tab,n = [ None for _ in range(N+1) ],2
6     tab[0],tab[1] = 0,1
7     # P_0 ∧ P_1
```

```
8     while n!=N+1:
9       tab[n],n = tab[n-2]+tab[n-1],n+1
10    ans = tab[N]
11  return ans
```

La función `fib_iter` determina el valor de retorno distinguiendo dos casos. Si $N \leq 1$, que corresponde a los casos base, el valor se calcula directamente. De lo contrario, se crea la memoria auxiliar en `tab[0..N]` inicializada con valores indefinidos; excepto por `tab[0]` y `tab[1]` que son 0 y 1, respectivamente, dado que $fib(0) = 0$ y $fib(1) = 1$. Para cualquier valor a partir de 2, el valor `tab[n]` se calcula sumando los dos valores inmediatamente anteriores en `tab`: por P_0, estos valores coinciden con $fib(n-2)$ y $fib(n-1)$, luego se esperaría que `tab[n]` corresponda a $fib(n)$ cuando se actualice la tabulación.

Para demostrar que el algoritmo iterativo de tabulación `fib_iter` funciona, es necesario demostrar que el ciclo satisface los invariantes P_0 y P_1.

Teorema 3.2.4

El ciclo en `fib_iter` satisface los invariantes P_0 y P_1. Es decir,

1. P_0 y P_1 son ciertos antes de la (posible) primera iteración del ciclo.
2. P_0 y P_1 son ciertos antes de una iteración del ciclo, entonces son ciertos después de una iteración del ciclo.

Demostración

Las propiedades se demostrarán para P_0; las demostraciones para P_1 se proponen como ejercicios para el lector.

1. Note que antes de la primera iteración del ciclo, se tiene $n = 2$, `tab`$[0] = 0$ y `tab`$[1] = 1$. Por su parte, P_0 indica que en $tab[0..n)$ están almacenados los primeros n valores de *fib*. Luego, P_0 es cierto antes de la primera iteración del ciclo.
2. Suponga que P_0 es cierto; se desea demostrar que P_0 es cierto después de una iteración del ciclo (es decir, se supone que P_0 es cierto para n y se demuestra que es cierto para $n+1$). Note que al suponer que P_0 es cierto antes de una iteración del ciclo, se está suponiendo que $tab[0..n)$ tiene los primeros n valores de *fib*. En el cuerpo del ciclo se modifican $tab[n]$ y n con los valores $tab[n-2] + tabn - 1$ y $n+1$, respectivamente. Por la suposición, se tiene que $tab[n] = fib(n-2) + fib(n-1)$: es decir,

$tab[n] = fib(n)$. Como n incrementa en una unidad, es cierto que:

$(\forall i \mid 0 \leq i < n + 1 : tab[i] = fib(i))$.

Con base en el Teorema 3.2.4 es fácil demostrar que la función `fib_iter` es correcta con respecto a la especificación del Problema 3.2.1.

Teorema 3.2.5

El llamado `fib_iter(N)` calcula $fib(N)$.

Demostración

Recuerde que $N \in \mathbb{N}$, como se indica en la formulación del problema. La demostración de terminación del algoritmo se propone como ejercicio al lector. Si $N = 0$ o $N = 1$, entonces el algoritmo responde 0 o 1, respectivamente. De lo contrario (i.e., $N \geq 2$), suponga que la ejecución del ciclo termina para cualquier $N \geq 2$. Note que esto sucede cuando $n = N + 1$. En este caso, el invariante P_0 es equivalente a:

$(\forall i \mid 0 \leq i < N + 1 : tab[i] = fib(i))$.

En particular, $tab[N] = fib(N)$. Es decir, en el caso $N \geq 2$, `fib_iter(N)` responde con el valor de $fib(N)$.

En cualquiera de los casos `fib_iter(N)` calcula el valor de $fib(N)$.

Los cálculos de las complejidades temporal y espacial de `fib_iter` se proponen como ejercicios al lector (¿cuáles son estas complejidades?).

Hay una versión iterativa para calcular *fib* con tabulación que toma tiempo $O(N)$ y espacio $O(1)$. Diseñar este algoritmo se propone como ejercicio al lector (note que en el caso recurrente solo depende de dos valores).

Ejercicios

1. Al estudiar la función `fib`, se afirmó que su complejidad temporal es

$$O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right),$$

en donde n es valor que se desea calcular. También se indicó que esta expresión resulta de resolver la ecuación de diferencia $X^2 - X - 1 = 0$ con variable X.

   a) Investigue sobre ecuaciones de recurrencia y explique de dónde resulta la ecuación dada.
   b) Investigue sobre técnicas para resolver ecuaciones de recurrencia y aplique una de ellas para demostrar que la complejidad temporal de `fib(n)` es $O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$.

2. Complete la demostración del Teorema 3.2.1 con el análisis de la condición C_1.
3. Demuestre el Teorema 3.2.2.
4. Demuestre el Teorema 3.2.3.
5. En el Teorema 3.2.2 se concluye que el llamado `fib_memo(N, {})`, en donde `{}` representa el diccionario vacío, calcula $fib(N)$. Proponga (justificando su respuesta) otra forma de invocar `fib_memo` para que calcule $fib(N)$.
6. Demuestre que el ciclo en `fib_iter` satisface el invariante P_1.
7. Demuestre que `fib_iter` termina.
8. Para $N \in \mathbb{N}$, demuestre:
   a) La complejidad temporal de `fib_iter(N)` es $O(N)$.
   b) La complejidad espacial de `fib_iter(N)` es $O(N)$.
9. Diseñe un algoritmo iterativo que use la técnica de programación dinámica con tabulación que permita calcular la función fib en tiempo $O(N)$ y espacio $O(1)$. Formule invariantes y demuestre que el algoritmo es correcto. ¿Cuál es el diagrama de necesidades?
10. Un *coeficiente binomial* es un número entero positivo que corresponde a la expansión polinomial de la potencia de un binomio. Para $n, k \in \mathbb{N}$,

    $$\binom{n}{k}$$

    es el coeficiente del término x^k en la expansión de $(1+x)^n$. El valor numérico de la expresión $\binom{n}{k}$, leída "de n se toman k", corresponde a la cantidad de subconjuntos de k elementos de un conjunto de n elementos (si $k > n$, este valor es 0). Los coeficientes binomiales satisfacen la siguiente recurrencia:

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

    Con base en esta definición recurrente:
    a) Explique por qué el problema de calcular coeficientes binomiales requiere programación dinámica.
    b) Diseñe un algoritmo recurrente que calcule $\binom{n}{k}$ usando memorización.
    c) Diseñe un algoritmo iterativo que calcule $\binom{n}{k}$ usando tabulación.

3.3. Una metodología

Se propone una metodología para aplicar la técnica de programación dinámica a partir de la especificación de un problema. Esta sección detalla los pasos de la metodología que, en las secciones posteriores, se usa para resolver cada uno de los problemas formulados a modo de casos de estudio.

Nota 3.3.1

Dada la especificación de un problema que cuenta con las propiedades de subestructura y solapamiento, se propone la siguiente metodología para diseñar un algoritmo con programación dinámica que resuelva el problema.

Función objetivo: se formula una función objetivo que generalice la salida del problema dado o que, en su defecto, permita resolver el problema dado. Es importante que dicha función, en este punto, cuente con una descripción clara mas *no* con una definición formal. Comúnmente esta función se obtiene al generalizar los parámetros del problema que hacen parte de la salida de la especificación; puede involucrar uno o más parámetros, dependiendo del caso.

Reformulación de la especificación: se debe validar que la especificación del problema se puede reformular con ayuda, directa o indirecta, de la función objetivo. Debe ser posible reescribir la salida del problema en términos de la función que generaliza el problema. Si esto no es posible, será necesario reformular la función planteada en el paso anterior.

Planteamiento recurrente: se presenta una definición recurrente de la función objetivo. Esta definición será la base de la programación dinámica y, como tal, debe ser completa, cubriendo todas las instancias del problema, y no ser ambigüa.

¿Memorización o tabulación?: partiendo de la definición recurrente de la función objetivo, se decide si se usa memorización o tabulación. Dependiendo de la elección:

- Si se prefiere memorización, es clave determinar qué estructura de datos se usará para representar la memoria compartida (e.g., en `fib_memo` fue un diccionario). Se debe tener cuidado con la profundidad de la recurrencia; en algunos casos, puede generar errores de ejecución si hay llamados recurrentes muy profundos.
- Si se prefiere tabulación, es deseable diseñar un *diagrama de necesidades* que muestre cómo se da la recurrencia en términos de dependencias con el objetivo de determinar cómo llevar a cabo el cálculo de la tabulación.

Reducción del espacio (opcional): si se escogió tabular, explorar la posibilidad de reducir el espacio de la memoria de tabulación.

Implementación: implementar la solución diseñada.

Estos pasos se ilustrarán con cada uno de los ejemplos que se presentan en el resto del capítulo.

Ejercicios

1. El problema de multiplicación de una secuencia de matrices consiste en minimizar la cantidad de operaciones básicas (sumas y multiplicaciones escalares, básicamente) para multiplicar $N \geq 0$ matrices. Este algoritmo es uno de los ejemplos clásicos de programación dinámica.
   a) Investigue sobre este problema y formule su especificación.
   b) Investigue sobre su solución con programación dinámica y, paso a paso, identifique el uso en ella de la metodología propuesta en esta sección. Indique claramente en qué consiste cada uno de los pasos y, en caso tal de que alguno de los pasos no esté presente, justifique esta situación.
2. El problema del corte de un lingote de oro (en inglés conocido como *rod cutting*) consiste en maximizar la ganancia que se puede obtener de cortar verticalmente el lingote en varios pedazos (para venderlos individualmente).

   Entrada: La longitud $N \geq 0$ de un lingote y una tabla $T[0..N]$ especificando la ganancia a obtener por cada tamaño vendido.

   Salida: Máxima ganancia que se puede obtener vendiendo el lingote de tamaño N, posiblemente en varios pedazos, en relación con la tabla $T[0..N]$.

   Considere la siguiente función objetivo, para $0 \leq n \leq N$:

   $\phi(n)$: "Máxima ganancia que se puede obtener vendiendo el lingote de tamaño n, posiblemente en varios pedazos, en relación con la tabla $T[0..N]$".

   Como definición de ϕ, se propone la siguiente función recurrente para $0 \leq n \leq N$:

   $$\phi(n) = \begin{cases} 0 & \text{, si } n = 0, \\ (\uparrow i \mid 1 \leq i \leq n : T[i] + \phi(n-i)) & \text{, si } n \neq 0. \end{cases}$$

   El caso base indica que no hay ganancia alguna cuando no hay nada que vender. El caso inductivo indica que se escoge aquella opción que resulta de cortar el

lingote en un pedazo de tamaño i y lo mejor que se pueda hacer con el tamaño restante $n - i$ (¿por qué esto cubre todos los casos?).

Con base en este planteamiento, siga la metodología propuesta en esta sección para diseñar:

a) Un ejemplo que muestre cómo ϕ tiene las propiedades de subestructura óptima y solapamiento.
b) Un algoritmo con memorización para ϕ.
c) Un algoritmo con tabulación para ϕ (¿se puede hacer reducción de la tabulación?)

En cada caso debe resultar código en Python ejecutable.

3.4. Suma máxima de un subarreglo

El problema del subarreglo de suma máxima consiste en calcular la máxima suma posible de un subarreglo en un arreglo de números dado.

Problema 3.4.1: Subarreglo de suma máxima

Entrada: Un arreglo $A[0..N)$, con $N \geq 0$, de números.
Salida: Suma máxima de un subarreglo de $A[0..N)$.

Este ha sido un problema estudiado exhaustivamente desde su formulación en 1977 por Ulf Grenander (inicialmente se planteó para una matriz y no para un arreglo unidimensional). En el procesamiento de imágenes, por ejemplo, este problema tiene aplicación en la identificación de áreas brillantes en mapas de bits. También tiene aplicaciones en biología computacional y en finanzas.

Ejemplo 3.4.1

Considere el arreglo $A = [-1, 3, -2, 1, 4, -2, 1, 0]$. El subarreglo de suma máxima es $A[1..5) = [3, -2, 1, 4]$ con suma 6.

Este problema se puede resolver trivialmente para algunas instancias. Por ejemplo, si el arreglo dado únicamente tiene números no negativos, la respuesta es la suma de sus elementos. También, si todos sus elementos son números negativos, la respuesta es 0, pues el arreglo vacío es subarreglo de cualquier arreglo y tiene suma 0. El caso general se puede resolver exhaustiva pero ineficientemente, como se presenta en el Algoritmo 3.4.1.

Algoritmo 3.4.1

1. Para cada pareja (i, j) que satisface $0 \leq i \leq j \leq N$, determinar la suma de $A[i..j)$.
2. Retornar aquella suma que sea máxima.

La complejidad temporal del Algoritmo 3.4.1 es $O(N^3)$ pues calcular todas las parejas (i, j) toma tiempo $O(N^2)$ y para cada una de ellas la suma de $A[i..j)$ toma tiempo $O(N)$. Esta complejidad se puede reducir a $O(N^2)$ con un pre-proceso que toma tiempo y espacio $O(N)$. En particular, se pueden calcular las sumas de los prefijos (o sufijos) de $A[0..N)$ de tal forma que la suma de cada subarreglo $A[i..j)$ se obtiene en tiempo $O(1)$, reduciendo de $O(N^3)$ a $O(N^2)$ el tiempo que toma la solución global.

Aplicando la técnica dividir, conquistar y combinar se puede obtener una solución más eficiente (o menos ineficiente).

Algoritmo 3.4.2

Suponga que *low* y *hi* son tales que $0 \leq low \leq hi \leq N$. El objetivo de este algoritmo es calcular la suma máxima de un subarreglo en $A[low..hi)$.

1. Si $low = hi$, entonces retornar 0.
2. Si $low + 1 = hi$, entonces retornar $A[low]$.
3. De lo contrario:
   a) sea *mid* un punto medio entre *low* y *hi*,
   b) recurrir sobre $A[low..mid)$ para determinar la máxima suma de un subarreglo de $A[low..mid)$,
   c) recurrir sobre $A[mid..hi)$ para determinar la máxima suma de un subarreglo de $A[mid..hi)$,
   d) determinar la suma máxima de un subarreglo de $A[low..hi)$ que incluye el valor en *mid* (i.e., que cruza de izquierda a derecha), y
   e) retornar el máximo valor obtenido en los pasos $(3b)$,$(3c)$,$(3d)$.

Si se incluye el subarreglo vacío como opción, basta con maximizar el resultado del los pasos (2) y (3) con 0.

Suponiendo que el Algoritmo 3.4.2 es correcto (es un ejercicio para el lector), su llamado con $low = 0$ y $hi = N$ resuelve el problema del subarreglo de suma máxima en $A[0..N)$. Además, si el paso $(3d)$ se calcula en tiempo lineal en función de $hi - low$, con peor caso $O(N)$, entonces la cantidad de operaciones básicas que el algoritmo realiza para resolver el problema está dado por la función simple $T(N)$

definida en su caso recurrente por

$$T(N) = 2T(N/2) + O(N).$$

Es decir, bajo la suposición hecha, el Algoritmo 3.4.2 toma tiempo $O(N \log N)$ en resolver el problema de suma máxima de un subarreglo.

Si se cuenta con un algoritmo relativamente eficiente que resuelve el problema sin usar programación dinámica, ¿qué hace este problema en este capítulo? Pues bien, lo que sucede es que se puede diseñar una solución que toma tiempo $O(N)$ si se usa programación dinámica. Ese es el objetivo de lo que resta de esta sección: derivar dicha solución.

Nota 3.4.1

Se rumora que hacia 1984, en un seminario en Carnegie Melon (una universidad en EE.UU., prestigiosa en computación) se presentó la versión unidimensional del problema del subarreglo de suma máxima. En ese seminario estaba Jay Kadane quien, en un par de minutos, diseñó una solución lineal para el problema. Se cree que esta es, desde el punto de vista asintótico, la mejor solución posible. Por ello, la solución lineal al problema del subarreglo de suma máxima recibe el nombre de *Algoritmo de Kadane* (es el diseño al cual se llega al final de esta sección).

El caso de la suma máxima de un subarreglo en un arreglo vacío es trivial, como se ha explicado anteriormente. Igualmente, considerar el arreglo vacío como uno de los posibles subarreglos (aún si el arreglo dado no es vacío) resulta en restringir la salida de la solución a números no negativos. Los casos interesantes se presentan, en realidad, al fijar la atención en aquellos subarreglos que tienen al menos un elemento. El desarrollo que se presenta a continuación se enfoca, entonces, en determinar la suma máxima de un subarreglo no vacío, dando por descontado el caso en el cual $A[0..N)$ tiene tamaño $N = 0$. Posteriormente, se presentará la solución para el caso más general en el cual el arreglo puede ser vacío.

Un arreglo de suma máxima (no vacío) termina en alguno de los índices de $A[0..N)$. Es decir, para algún índice $0 \leq n < N$, un subarreglo de suma máxima debe terminar en n (i.e., incluyendo a $A[n]$ como último elemento de la suma de elementos contíguos). Si se contara con la información de las mejores sumas que terminan en cada uno de los índices $0 \leq n < N$, entonces el problema de determinar la mejor suma de un subarreglo (no vacío) en $A[0..N)$ estaría resuelto: bastaría con maximizar sobre todos estos valores. Esta es la estrategia de diseño que se adopta para resolver el problema en cuestión. Se sigue la metodología propuesta en la Nota 3.3.1 para diseñar una solución de programación dinámica con tabulación.

Función objetivo. Para $0 \leq n < N$, se define:

$$\phi(n) \ : \ \text{“suma máxima de un subarreglo de } A[0..N) \text{ que termina en } n.\text{”}$$

La función ϕ tiene como dominio los índices de A y co-dominio a todos los números. Una forma alternativa (y equivalente) de interpretar ϕ es la siguiente: $\phi(n)$ es la suma máxima de un sufijo de $A[0..n)$.

Reformulación de la especificación. Note que, con base en ϕ, se puede especificar la salida del problema (restringido) como la siguiente maximatoria:

$$(\uparrow n \mid 0 \leq n < N : \phi(n)).$$

Si por convención se adopta que esta maximatoria es 0 cuando $N = 0$, entonces esta expresión también abarca el caso en el cual el arreglo dado es vacío. Es más, si también se desea considerar el subarreglo vacío como opción entre los subarreglos de A, basta con extender la maximatoria de la siguiente manera (¿por qué?):

$$(\uparrow n \mid 0 \leq n < N : \phi(n)) \uparrow 0.$$

En cualquier caso, es evidente que la función objetivo permite reformular el problema de interés.

Planteamiento recurrente. El objetivo es obtener una definición recurrente para ϕ. Siguiendo los principios de dividir, conquistar y combinar, el primer paso es identificar aquellos casos que se pueden resolver directamente sin necesidad de recurrir (i.e., conquistar). Dada la defición de la función objetivo, ese caso corresponde a $\phi(0)$: ¿cuál es la suma máxima de un subarreglo de $A[0..N)$ que termina en 0? Solo $A[0..0]$ satisface esta condición. Es decir, $\phi(0) = A[0]$. ¿Qué sucede cuando $n \neq 0$? Para responder esta pregunta, suponga que $A[k..n]$ es un arreglo de suma máxima que inicia en algún $0 \leq k \leq n$ y termina en n. Es decir, el objetivo es que $\phi(n)$ tenga como valor la suma de los elementos de este subarreglo. Si $k = n$, entonces esta suma máxima es $A[n]$. Si $k \neq n$ (i.e., $k < n$), entonces esta suma máxima tiene la forma $A[k] + \cdots + A[n-1] + A[n]$ o, de manera equivalente, $\phi(n-1) + A[n]$ (¿por qué? Ayuda: pensar en la propiedad de la subestructura óptima).

Con base en el análisis anterior, se propone la siguiente definición para ϕ, con $0 \leq n < N$ y $N \neq 0$:

$$\phi(n) = \begin{cases} A[0] & \text{, si } n = 0, \\ A[n] \uparrow (A[n] + \phi(n-1)) & \text{, si } n \neq 0. \end{cases}$$

El planteamiento recurrente se hace con base en observaciones que, aunque intuitivas y “obvias”, deben ser respaldadas por un razonamiento formal.

Teorema 3.4.1

Sea $0 \leq n < N$. Si $N \neq 0$, entonces $\phi(n)$ es la suma máxima de un subarreglo de $A[0..N)$ que termina en n.

Demostración

Se procede por inducción sobre n:

- Caso base $n = 0$: note que solo un subarreglo de $A[0..N)$ termina en 0 y tiene suma $A[0]$. Luego, $\phi(0) = A[0]$ es la definición correcta en este caso.
- Caso inductivo $n \neq 0$: sea $A[k..n]$ de suma máxima (inicia en k y termina en n). Se distinguen dos subcasos:
  - Caso $k = n$: entonces $A[n]$ es de suma máxima entre todos los subarreglos que terminan en n. En particular, $A[n]$ no es menor que la suma del arreglo que resulta de extender el de suma máxima que termina en $n-1$ con $A[n]$, i.e.,

    $A[n] \geq \phi(n-1) + A[n]$.

    Note que esto es cierto si y solo si $\phi(n-1) \leq 0$. Entonces se tiene:

    $$\begin{aligned}
    \phi(n) &= A[n] && (A[0] \text{ es de suma máxima}) \\
    &= A[n] + 0 && (\text{aritmética}) \\
    &= A[n] + (0 \uparrow \phi(n-1)) && (\phi(n-1) \leq 0) \\
    &= (A[n] + 0) \uparrow (A[n] + \phi(n-1)) && (\text{distr. de suma sobre máximo}) \\
    &= A[n] \uparrow (A[n] + \phi(n-1)) && (\text{arimética}).
    \end{aligned}$$

  - Caso $k \neq n$: entonces $k < n$ y la suma de los elementos de $A[k..n]$ no es menor que $A[n]$ (i.e., $A[k] + \cdots + A[n] \geq A[n]$). Como se observó anteriormente:

    $A[k] + \cdots + A[n-1] + A[n] = \phi(n-1) + A[n]$.

    Luego, $\phi(n-1) \geq 0$. Con base en estas observaciones, se tiene:

    $$\begin{aligned}
    \phi(n) &= A[k] + \cdots + A[n-1] + A[n] && (A[k..n] \text{ es de suma máxima}) \\
    &= \phi(n-1) + A[n] && (\text{observación anterior}) \\
    &= (\phi(n-1) \uparrow 0) + A[n] && (\phi(n-1) \geq 0) \\
    &= (A[n] + \phi(n-1)) \uparrow (A[n] + 0) && (\text{distr. de suma sobre máximo}) \\
    &= A[n] \uparrow (A[n] + \phi(n-1)) && (\text{arimética}).
    \end{aligned}$$

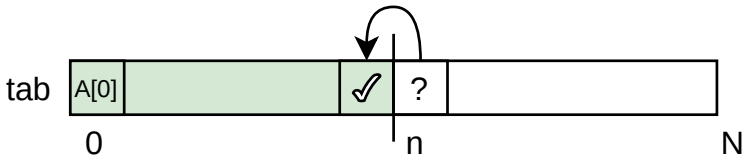
  En cualquiera de los dos casos, $\phi(n) = A[n] \uparrow (A[n] + \phi(n-1))$.

En conclusión, $\phi(n)$ es la suma máxima de un subarreglo de $A[0..N)$ que termina en n.

Con base en el planteamiento recurrente de la función objetivo, es fácil ver con ejemplos que existen solapamientos entre diferentes instancias de la solución. Se propone al lector, como ejercicio, encontrar instancias para las cuales esto pasa y explicar en qué consisten los solapamientos con esos casos en particular.

¿Memorización o tabulación? Se prefiere, dado que uno de los objetivos del diseño de la solución es ahorrar espacio, implementar la técnica de programación dinámica para la función objetivo con tabulación.

De acuerdo con la definición recurrente de ϕ, se propone el siguiente diagrama de necesidades.



En el diagrama se usa la memoria de tabulación $tab[0..N)$ para almacenar los valores de la función y una variable n para indicar el índice de la tabla que se debe calcular en un momento dado. Esta tabla en la posición 0 es inicializada con $A[0]$, como lo indica la definición de ϕ. Para calcular el valor para $n \geq 1$ (i.e., en el caso recurrente) en la tabla se depende directamente del valor inmediatamente anterior (esto sucede porque $\phi(n)$ depende de $\phi(n-1)$). Esto quiere decir que el valor en $tab[n-1]$ debe calcularse antes de intentar calcular $tab[n]$. Estas observaciones son importantes para diseñar el algoritmo que llena la tabla con los valores de ϕ.

El diseño del algoritmo iterativo para llenar la memoria de tabulación se basa en los siguientes invariantes:

$$P_0: \quad (\forall i \mid 0 \leq i < n : tab[n] = \phi(n)).$$
$$P_1: \quad 0 \leq n \leq N.$$

El invariante P_0 indica que el subarreglo $tab[0..n)$ tiene, en orden, los primeros n valores de ϕ: el valor en la i-ésima posición de tab almacena el valor $\phi(i)$. El segundo invariante indica en qué rango toma valores la variable n. Note que se pudo restringir a 1 como límite inferior para n, pero no es necesario. Con estos invariantes la estrategia está clara: llenar tab de izquierda a derecha. Con esta tabla completamente llena es posible, entonces, resolver el problema de determinar la suma máxima de los subarreglos de $A[0..N)$ al calcular el valor máximo en tab. Esto último se hará con una segunda pasada sobre tab, usando el siguiente invariante adicional.

$$P_2: \quad ans = (\uparrow i \mid 0 \leq i \leq n : \phi(i)).$$

Con base en este diseño, se propone el Algoritmo 3.4.3.

Algoritmo 3.4.3

```
def mss(A):
  ans,N = 0,len(A)
  if N!=0:
    tab = [ None for _ in range(N) ]
    tab[0],ans = A[0],max(ans, A[0])
    # P_0 ∧ P_1
    for n in range(1, N):
      tab[n] = max(A[n], tab[n-1]+A[n])
    # P_0 ∧ P_1 ∧ P_2
    for n in range(N):
      ans = max(ans, tab[n])
  return ans
```

El caso en el cual el arreglo dado es vacío, se trata de manera especial con ayuda de una instrucción condicional. Es necesario ahora demostrar que `mss` es correcta con respecto a la especificación dada, y determinar sus complejidades temporal y espacial.

Teorema 3.4.2

El llamado `mss(A)`:

1. Calcula la suma máxima de un subarreglo de $A[0..N)$.
2. Toma tiempo $O(N)$.
3. Toma espacio $O(N)$.

Demostración

Se propone como ejercicio al lector.

Reducción del espacio. En el diagrama de necesidades se puede observar que para el cálculo de $tab[n]$ solo es necesario contar con el valor precalculado en $tab[n-1]$. Es decir, de manera más general, para los casos recurrentes solo es necesario contar con un valor pre-calculado y no con un prefijo completo de tab. Con base en esta observación, se propone el invariante P_3 que introduce la variable entera $prev$ para reemplazar el arreglo tab:

$$P_3: \quad prev = \phi(n-1).$$

La variable *prev* representa la mejor suma de $A[0..N)$ que termina en la posición inmediatamente anterior a n. Por ello, la idea es usar este valor en el cuerpo del ciclo iterativo e ignorar todos los valores para índices inferiores a $n-1$. Además, es necesario incluir el cálculo del valor máximo de ϕ dentro del cuerpo del ciclo principal puesto que sería imposible recuperarlo de otra forma habiendo reducido el espacio de la tabulación (no hay arreglo que recorrer para calcular el máximo). Esta estrategia resulta en la función `mss_opt` que se presenta en el Algoritmo 3.4.4.

Algoritmo 3.4.4

```
1 def mss_opt(A):
2   ans,N = 0,len(A)
3   if N!=0:
4     prev,ans = A[0],max(ans, A[0])
5     # P_1 ∧ P_2 ∧ P_3
6     for n in range(1, N):
7       prev = max(A[n], prev+A[n])
8       ans = max(ans, prev)
9   return ans
```

Si se demuestra correcto, el algoritmo optimizado resolvería el problema de interés en tiempo $O(N)$ y en espacio $O(1)$.

Teorema 3.4.3

El llamado `mss_opt(A)`:

1. Calcula la suma máxima de un subarreglo de $A[0..N)$.
2. Toma tiempo $O(N)$.
3. Toma espacio $O(1)$.

Demostración

Se propone como ejercicio al lector.

Finalmente, el Algoritmo de Kadane se puede obtener del Algoritmo 3.4.4 bajo ciertos supuestos o acuerdos. La observación clave es notar que las asignaciones a las varibles `prev` y `ans` en las líneas 4 y 7-8 son básicamente las mismas: `prev` se actualiza con el siguiente "mejor" y `ans` lleva registro del mejor encontrado entre los primeros n valores de la función. Si se supone que el máximo de una colección vacía

de números es 0 (en este problema tiene todo el sentido pues se opera sobre números naturales y el 0 es el elemento neutro del máximo sobre este conjunto), entonces no hay ningún problema si se decide iniciar las iteraciones del ciclo con $n = 0$. La transformación del Algoritmo 3.4.4 bajo las observaciones anteriores resulta en el Algoritmo de Kadane, el cual se presenta en el Algoritmo 3.4.5.

Algoritmo 3.4.5: Algoritmo de Kadane

```
def kadane(A):
  N,curr,ans = len(A),0,0
  for n in range(N):
    curr = max(A[n], curr+A[n])
    ans = max(ans, curr)
  return ans
```

Al igual que `mss_opt`, la función `kadane` resuelve el problema de suma máxima de un subarreglo en un arreglo usando tiempo lineal y espacio constante. El diseño de los invariantes para el ciclo en `kadane` y su corrección se proponen como ejercicio para el lector.

Ejercicios

1. Diseñe una función `mss_bf` en Python que implemente el diseño del Algoritmo 3.4.1. Corra la implementación con arreglos de números generados aleatoriamente de tamaños 10, 50, 100 y 250. ¿Cuál es la diferencia en tiempo entre las ejecuciones?

2. Después de presentar el Algoritmo 3.4.1, se propone una mejora para disminuir su complejidad de $O(N^3)$ a $O(N^2)$. Diseñe una función `mss_bf_opt` en Python que implemente este diseño optimizado. Corra la implementación con arreglos de números generados aleatoriamente de tamaños 10, 100, 1000 y 5000. ¿Cuál es la diferencia en tiempo entre las ejecuciones?

3. Diseñe una función `mss_dcc` en Python que implemente el diseño del Algoritmo 3.4.2. Corra la implementación con arreglos de números generados aleatoriamente de tamaños 10, 100, 1000, 10000 y 100000. ¿Cuál es la diferencia en tiempo entre las ejecuciones?

4. En la reformulación de la especificación con base en ϕ, se afirma que la expresión

$$(\uparrow n \mid 0 \leq n < N : \phi(n)) \uparrow 0.$$

permite considerar el subarreglo vacío como opción entre los subarreglos de A. Proponga un ejemplo de un arreglo $A[0..N)$ en el cual el resultado de la maximatoria sobre n sea negativo.

5. En el planteamiento recurrente de ϕ, se supone que $A[k..n]$ es un arreglo de suma máxima que inicia en algún $0 \leq k \leq n$ y termina en n. En el caso $k < n$ se indica que la suma $A[k] + \cdots + A[n-1] + A[n]$ es igual a $\phi(n-1) + A[n]$. Ilustre esta afirmación con un ejemplo y elabore una justificación de por qué la igualdad es cierta (no es necesaria una demostración).
6. Elabore un ejemplo en el cual se exhiba la propiedad de solapamiento para ϕ. Explique brevemente su respuesta.
7. Diseñe un algoritmo con memorización para la función ϕ, incluyendo los predicados correspondientes a las condiciones que debe satisfacer el algoritmo. Demuestre que el algoritmo es correcto con respecto a la especificación dada, y estime las complejidades temporal y espacial.
8. Demuestre el Teorema 3.4.2.
9. Demuestre el Teorema 3.4.3.
10. Proponga invariantes para el Algoritmo de Kadane (Algoritmo 3.4.5) y demuestre que es correcto. También determine las complejidades temporal y espacial del algoritmo.
11. Modifique el Algoritmo de Kadane (Algoritmo 3.4.5) para que no se tenga en cuenta el arreglo vacío entre los subarreglos de A al calcular la suma máxima. Explique brevemente por qué funciona el algoritmo propuesto.
12. Modifique cualquiera de los algoritmos propuestos en esta sección que toman tiempo lineal para que, además de calcular la suma máxima de un subarreglo de $A[0..N)$, retorne k, n tales que el subarreglo $A[k..n)$ es de suma máxima (la complejidad del algoritmo resultante también debe ser lineal en función del tamaño del arreglo dado).

3.5. El problema del morral

El problema del morral es un problema de optimización combinatoria.

Nota 3.5.1

El nombre del problema del morral (en inglés, *knapsack* o *KS*) se deriva de una analogía en la cual se desea escoger de una colección de elementos para cargar en un morral que tiene una restricción de peso (o volumen). El objetivo

es escoger dicha combinación de elementos maximizando su valor total sin exceder la capacidad del morral.

Este es un problema que surge naturalmente en situaciones de asignación de recursos en las cuales quienes toman las decisiones deben escoger proyectos o tareas indivisibles bajo una restricción presupuestal o de tiempo.

De forma general, dada una colección de elementos, cada uno con un peso y un valor asociados, el problema consiste en determinar cuál es la suma máxima de valores que se puede obtener al tomar algunos de los elementos sin que su suma exceda una restricción de peso global.

Problema 3.5.1: Problema del morral

Entrada: Arreglos $V[0..N)$ y $W[0..N)$, con $N \geq 0$, de valores y pesos (números no negativos), y un número $X \geq 0$.

Salida: Suma máxima de elementos con valores en $V[0..N)$ cuyos pesos en $W[0..N)$ no exceden X.

La condición de salida del problema supone que el valor y el peso del i-ésimo elemento están dados por $V[i]$ y $W[i]$, respectivamente. La naturaleza combinatoria del problema del morral se revela al notar que se requiere una exploración sobre los subconjuntos de índices de A. Para cualquier $S \subseteq \{0, \ldots, N-1\}$, se definen las funciones *value* y *weight*:

$$\mathit{value}(S) = (+i \mid i \in S : V[i]) \qquad \text{y} \qquad \mathit{weight}(S) = (+i \mid i \in S : W[i]).$$

La expresión $\mathit{value}(S)$ representa el valor total de los elementos indexados por S, mientras que $\mathit{weight}(S)$ el peso total de los elementos indexados por S. Así, la salida del problema del morarral corresponde a la expresión:

$$(\uparrow S \mid S \subseteq \{0, \ldots, N-1\} \wedge \mathit{weight}(S) \leq X : \mathit{value}(S)).$$

Como hay 2^N subconjuntos S de índices de $A[0..N)$ (¿por qué?), se dice que el problema del morral es de naturaleza combinatoria, pues debe considerar todos aquellos subconjuntos S para determinar la mejor combinación posible bajo la restricción dada. Debe ser claro, entonces, que un algoritmo de fuerza bruta para resolver el problema del morral tomará tiempo exponencial en la cantidad de elementos entre los cuales se elige.

Ejemplo 3.5.1

Considere 4 elementos con valores y pesos dados por, respectivamente:

$$V = [4, 6, 5, 1] \qquad \text{y} \qquad W = [3, 5, 5, 2].$$

- Con $X = 15$, lo mejor opción es llevar todos los elementos, obteniendo como valor total 16 y peso total 15.
- Con $X = 11$, el valor máximo es 11. Esto resulta de dos configuraciones cuyo peso total es 10: llevar los elementos $0, 1, 3$ o los elementos $1, 2$.
- Con $X = 5$, la mejor opción es llevar el elemento 1, con valor total 6 y peso total 5.
- Con $X = 1$, la mejor (y única) opción es no echar nada al morral.

El objetivo es diseñar un algoritmo de programación dinámica que resuelva el problema del morral, pues exhibe las propiedades de subestructura óptima y solapamiento (¿por qué?). Para ello se seguirá la metodología presentada en la Sección 3.3.

Función objetivo. Hay dos variables que permiten generalizar la salida del problema: N y X. Por una parte, la cantidad de elementos disponibles se puede hacer variar para considerar diferentes prefijos de $A[0..N)$ (también se pueden hacer variar los sufijos, resultando en un plateamiento similar). Por otra parte, el peso disponible en el morral cambiaría cada vez que se decide escoger un elemento. Con base en estas observaciones, se propone la siguiente función objetivo para $0 \leq n \leq N$ y $0 \leq x \leq X$:

$$\phi(n, x) \ : \ \text{“suma máxima de elementos con valores en } V[0..n) \text{ cuyos pesos en } W[0..n) \text{ no exceden } x\text{.”}$$

Reformulación de la especificación. El objetivo del problema es calcular $\phi(N, X)$.

Planteamiento recurrente. Se escogerá n como variable “pivote”. Esto quiere decir que para razonar por casos, estos primero se estructuran con base en condiciones sobre n y, de ser necesario, luego se dividien en subcasos con base en condiciones sobre la segunda variable x. Es importante tener en cuenta que $\phi(n, x)$ debe ser definida para valores de n y x que satisfacen $0 \leq n \leq N$ y $0 \leq x \leq X$.

Se procede por casos sobre n y x:

- No hay elementos para escoger (i.e., $n = 0$). Luego, sin importar la capacidad x del morral, lo máximo (y único) que se puede obtener es 0.
- Hay al menos un elemento para escoger (i.e., $n > 0$). Entonces están las opciones de que el elemento $n-1$ (este es el n-ésimo elemento) pese más que la capacidad x del morral o que no:

- si $W[n-1] > x$, no hay más opción que ignorar ese elemento, y escoger lo mejor posible con los elementos que restan, sin modificar la capacidad del morral; es decir, se recurre con $n-1$ y x.
- si $W[n-1] \leq x$, entonces estan las opciones de adicionar el elemento al morral o no. Como el objetivo es tomar la mejor decisión, se apuesta a los dos caballos. No escogerlo resulta en la misma situación del caso anterior (i.e., se recurre con $n-1$ y x). Escogerlo tiene el efecto de disminuir la capacidad del morral a $x - W[n-1]$ y de aumentar el beneficio en $V[n-1]$ unidades; es decir, se recurre con $n-1$ y $x-W[n-1]$, y se acumula $V[n-1]$ beneficio.

En cada uno de los casos recurrentes se respeta las condiciones $0 \leq n \leq N$ y $0 \leq x \leq X$ (¿por qué?), lo cual es clave para contar con una buena definición de ϕ. La naturaleza exponencial de la búsqueda de "la" mejor elección de elementos se ve en el segundo caso recurrente: allí se apuesta a llevar y a no llevar el elemento, es decir, se consideran todos los subconjuntos de índices pendientes sin acumular el n-ésimo elemento y todos los subconjuntos de índices pendientes acumulando el n-ésimo elemento.

La definición formal de ϕ se presenta para $0 \leq n \leq N$ y $0 \leq x \leq X$:

$$\phi(n,x) = \begin{cases} 0 & \text{, si } n = 0, \\ \phi(n-1,x) & \text{, si } n \neq 0 \wedge W[n-1] > x, \\ \phi(n-1,x) \uparrow & \\ \quad (\phi(n-1, x - W[n-1]) + V[n-1]) & \text{, si } n \neq 0 \wedge W[n-1] \leq x. \end{cases}$$

Es necesario demostrar que la definición de ϕ permite resolver el problema del morral.

Teorema 3.5.1

Si $0 \leq n \leq N$ y $0 \leq x \leq X$, entonces $\phi(n,x)$ es la suma máxima de elementos con valores en $V[0..n)$ cuyos pesos en $W[0..n)$ no exceden x.

Demostración

Sea $S \subseteq \{0, \ldots, n-1\}$ una solución óptima para el problema del morral con los primeros n elementos y capacidad x. Se procede por casos sobre S.

Si $S = \emptyset$, entonces no hay elementos para escoger (i.e., $n = 0$) o todos los elementos disponibles tienen un peso mayor a x (i.e., $W[i] > x$ para $0 \leq i < n$). En cualquiera de los dos casos $\phi(n,x) = 0$, lo cual coincide con el hecho de que $S = \emptyset$ (¿por qué?).

Si $S \neq \emptyset$, suponga que $S' \subseteq \{0, \ldots, n-1\}$ es escogencia "hecha" por $\phi(n, x)$ (i.e., $\phi(n, x) = \mathit{value}(S')$). Hacia una contradicción suponga que

$$\mathit{value}(S) > \mathit{value}(S').$$

Sin pérdida de generalidad también suponga que k (con $0 \leq k < n$) es el máximo índice de $A[0..n)$ en el cual S y S' no coinciden, y x' (con $0 \leq x' \leq x$) es la capacidad del morral cuando ϕ decide si el elemento k se incluye o no. Por la forma en que se escogió k, se cumple

$$\mathit{value}(S \setminus \{k+1, \ldots, n-1\}) > \mathit{value}(S' \setminus \{k+1, \ldots, n-1\}).$$

Entonces, para simplificar la escritura de la demostración y sin pérdida de generalidad, basta con suponer que $k = n-1$ y $x' = x$. En particular,

$$S = S \setminus \{k+1, \ldots, n-1\} \quad \text{y} \quad S' = S' \setminus \{k+1, \ldots, n-1\}.$$

Se consideran los siguientes casos:

- si $(n-1) \in S'$, entonces $(n-1) \notin S$ y

  $$\phi(n-1, x) \leq \phi(n-1, x - W[n-1]) + V[n-1].$$

  Luego

  $$\mathit{value}(S') = \phi(n, x) = \phi(n-1, x - W[n-1]) + V[n-1].$$

  Por la hipótesis inductiva, $\phi(n-1, x)$ es la suma máxima de elementos con valores en $V[0..n-1)$ cuyos pesos en $W[0..n-1)$ no exceden x. En particular,

  $$\mathit{value}(S) \leq \phi(n-1, x)$$

  dado que $\phi(n-1, x)$ es máximo entre todas las opciones al considerar los primeros $n-1$ elementos y con capacidad x. Observe que los elementos de S necesariamente están entre los primeros $n-1$ elementos, en este caso. Entonces se tiene:

  $$\begin{aligned}
  \mathit{value}(S) &\leq \phi(n-1, x) && \text{(optimalidad de } \phi\text{)} \\
  &\leq \phi(n-1, x - W[n-1]) + V[n-1] && ((n-1) \in S') \\
  &= \phi(n, x) && \text{(suposición)} \\
  &= \mathit{value}(S') && \text{(definición de } S'\text{)} \\
  &< \mathit{value}(S) && \text{(suposición inicial).}
  \end{aligned}$$

  Esto es una contradicción.

- si $(n-1) \notin S'$, entonces $(n-1) \in S$ y

  $$\phi(n-1, x) \geq \phi(n-1, x - W[n-1]) + V[n-1].$$

Luego,

$$value(S') = \phi(n, x) = \phi(n-1, x).$$

Como $(n-1) \in S$, note que

$$value(S) = value(S \setminus \{n-1\}) + V[n-1].$$

Por la misma observación del caso anterior, la optimalidad de $\phi(n-1, x - W[n-1])$ implica que

$$value(S \setminus \{n-1\}) \leq \phi(n-1, x - W[n-1]).$$

Entonces se tiene:

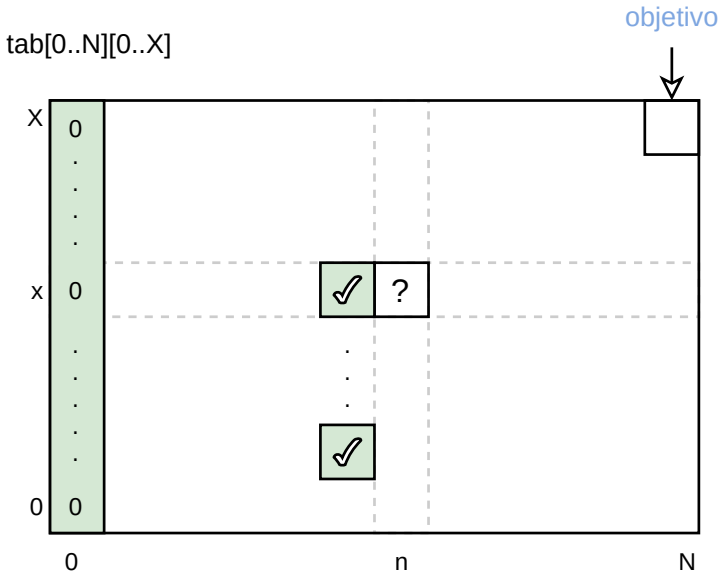
$$\begin{aligned}
value(S) &= value(S \setminus \{n-1\}) + V[n-1] && ((n-1) \in S) \\
&\leq \phi(n-1, x - W[n-1]) + V[n-1] && \text{(optimalidad de } \phi) \\
&\leq \phi(n-1, x) && \text{(suposición)} \\
&= \phi(n, x) && \text{(suposición)} \\
&= value(S') && \text{(definición de } S') \\
&< value(S) && \text{(suposición inicial).}
\end{aligned}$$

Esto es una contradicción.

En cualquiera de los dos casos se llega a una contradicción. Luego, $value(S) \leq value(S')$. Como S es óptimo (i.e., de suma máxima bajo la restricción de capacidad), se concluye que $value(S) = value(S')$. Es decir, $\phi(n, x)$ calcula el valor óptimo.

La demostración de la correctitud de ϕ puede parecer difícil. Sin embargo, la estrategia es y será la misma una y otra vez: garantizar que los casos base son correctos y, posteriormente, demostrar que los casos recurrentes también lo son con base en la hipótesis inductiva. En la demostración del Teorema 3.5.1 se usa el hecho de que una solución óptima existe y se concluye que la definición recurrente de la función objetivo permite calcular ese valor óptimo (no necesariamente con la misma configuración, pues puede haber más de una solución óptima). Note que en este caso se usó una estrategia similar a la usada en la demostración del Teorema 3.4.1.

¿Memorización o tabulación? Se diseñará una tabulación para implementar eficientemente ϕ con base en el siguiente diagrama de necesidades.

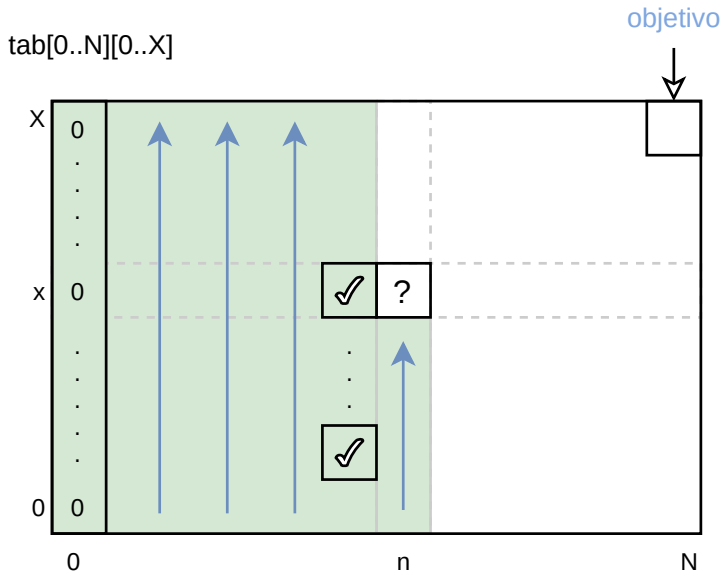


La tabulación se implementa con una matriz $tab[0..N][0..X]$ con $N+1$ columnas y $X+1$ filas. El objetivo es calcular el valor correspodiente a $tab[N][X]$ dado que el problema se resuelve con $\phi(N, X)$. Cuando $n = 0$, sin importar el valor de x, el valor $\phi(n, x)$ es 0. Por ello, la primera columna de la matriz tiene únicamente el valor 0. Para calcular $\phi(n, x)$, en el primer caso recurrente se depende de $\phi(n-1, x)$ y en el segundo caso recurrente además de $\phi(n-1, x-W[n-1])$. Por ello, en la matriz se identifican dos celdas de la columna $n-1$: cada una corresponde a cada uno de los llamados recurrentes, para los dos posibles valores de la capacidad restante en el morral. Estas observaciones indican que se puede llenar la tabulación por columnas, comenzando por la columna $n = 1$ (esta no es la única forma de hacerlo, ¿cierto?). Esta decisión se ve reflejada en los siguientes invariantes:

$$
\begin{aligned}
P_0 &: \quad (\forall i, j \mid 0 \leq i < n \wedge 0 \leq j \leq X : tab[i][j] = \phi(i, j)).\\
P_1 &: \quad (\forall j \mid 0 \leq j < x : tab[n][j] = \phi(n, j)).\\
P_2 &: \quad 0 \leq n \leq N + 1.\\
P_3 &: \quad 0 \leq x \leq X + 1.
\end{aligned}
$$

El invariante P_0 indica que todas las columnas a la izquierda de la columna n han sido llenadas con los valores correspondientes de ϕ. El invariante P_1 especifica que los valores en la columna n, por debajo de la fila x, corresponden a los valores de ϕ. Los invariantes P_2 y P_3 son de apoyo: el primero indica que los valores de n varían entre 0 y $N+1$, inclusive; el segundo, que los valores de x varían entre 0 y $X+1$, inclusive. Los dos invariantes de apoyo sugieren una forma de llenar la tabla: por

columnas, de izquierda a derecha, hasta llenar la columna N. Visualmente, esta decisión se puede incluir en el diagrama de necesidades.



Para implementar el algoritmo de tabulación, basta con seguir los invariantes y tener clara la imagen mental de ellos en el diagrama de necesidades. La idea es entonces, una vez creada la tabla e inicializada su primera columna, iniciar a llenarla desde la columna $n = 1$. Esto se hará como con las máquinas de escribir: se llena una columna, se procesa la siguiente. Intencionalmente se permite que la variable x llegue a $X + 1$, aunque esta fila no haga parte de la tabla: con $x = X + 1$ se sabe que se ha llenado la columna actual y se procede a iniciar la siguiente. Algo similar sucede con n: se habrá llenado la tabla por completo cuando se tenga $n = N + 1$.

Algoritmo 3.5.1

```
def ks_tab(V, W, X):
  N = len(V)
  tab = [ [ None for _ in range(X+1) ] for _ in range(N+1) ]
  for x in range(X+1): tab[0][x] = 0
  n,x = 1,0
  # P_0 ∧ P_1 ∧ P_2 ∧ P_3
  while n!=N+1:
    if x==X+1: n,x = n+1,0
    else:
      if x<W[n-1]: tab[n][x] = tab[n-1][x]
```

```
11        else: tab[n][x] = max(tab[n-1][x], tab[n-1][x-W[n-1]]+V[n-1])
12        x += 1
13    return tab[N][X]
```

La demostración de la corrección del algoritmo es rutina con base en los invariantes dados y la especificación del problema. La complejidad espacial está dominada por el tamaño de la tabla, y la temporal por su creación y actualización. Antes de introducir formalmente estos resultados, se presenta un ejemplo de la ejecución del algoritmo.

```
1  >>> V = [4, 6, 5, 1]
2  >>> W = [3, 5, 5, 2]
3  >>> print(ks_tab(V, W, 15))
4  16
5  >>> print(ks_tab(V, W, 11))
6  11
7  >>> print(ks_tab(V, W, 5))
8  6
9  >>> print(ks_tab(V, W, 1))
10 0
```

Teorema 3.5.2

El llamado `ks_tab(V, W, X)`, con $len(V) = len(W) = N$:

1. Calcula la suma máxima de elementos con valores en $V[0..N)$ cuyos pesos en $W[0..N)$ no exceden X.
2. Toma tiempo $O(NX)$.
3. Usa espacio $O(NX)$.

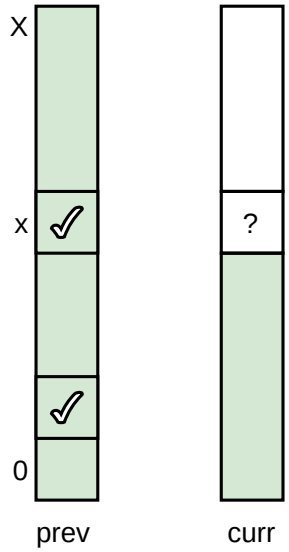
Demostración

Se propone como ejercicio al lector.

La complejidad temporal del algoritmo propuesto es polinomial en función de N y X. Sin embargo, es importante tener en cuenta que la cota X *no* es polinomial en función de N; a esto se le llama tiempo pseudo-polinomial. Por ejemplo, si $X = 2^{32}$, basta con 32 bits para representar X. Sin embargo, el algoritmo propuesto usa 2^{32} filas para construir y llenar la tabla. Por ello, es usual indicar que la complejidad temporal (y espacial) de la función `ks_tab` es $O(N \cdot 2^{\text{bits en } X})$.

3.5.1. Primera optimización. El diagrama de necesidades sugiere que el diseño de la tabulación se puede optimizar. En realidad, el cálculo de la columna $n \geq 1$ solo depende de valores en la columna $n-1$. Además, al iniciar con la columna $n = 1$, la columna $n = 0$ está completamente inicializada. Esta observación permite hacer una optimización importante en el espacio usado por la tabulación.

El siguiente diagrama de necesidades reduce la tabulación a dos columnas: una para el presente (que corresponde a la columna n) y otra para el pasado inmediatamente anterior (que corresponde a la columna $n-1$).



La noción de "presente" se representa con el arreglo *curr* y la de "pasado inmediato" con *prev*. Un reto surge ante la pregunta de qué hacer cuando *curr* haya sido llenado. Note que en el caso en el cual exista la necesidad de seguir procesando la tabulación (i.e., $n \neq N+1$), este arreglo pasará a ser el pasado inmediato y no habrá uso para los valores almacenados en *prev*. Por ello, y para evitar la creación de memoria adicional y copias innecesarias, se puede decidir reutilizar *prev* para el rol de nuevo presente. Para facilitar esta decisión en el código final, se prefiere que la tabulación sea una matriz de dos columnas, cada una de ellas de tamaño $X+1$, y usar dos variables (abusando la notación, *prev* y *curr*) que alternen opuestamente con los

valores 0 y 1: cuando *prev* sea 0, necesariamente *curr* es 1, y viceversa.

$$
\begin{aligned}
Q_0: &\quad (\forall j \mid 0 \leq j \leq X : tab[prev][j] = \phi(n-1, j)).\\
Q_1: &\quad (\forall j \mid 0 \leq j < x : tab[curr][j] = \phi(n, j)).\\
Q_2: &\quad 0 \leq prev \leq 1.\\
Q_3: &\quad 0 \leq curr \leq 1.\\
Q_4: &\quad prev + curr = 1.\\
P_2: &\quad 0 \leq n \leq N+1.\\
P_3: &\quad 0 \leq x \leq X+1.
\end{aligned}
$$

La matriz para la tabulación es $tab[0..2][0..X]$. El invariante Q_0 indica que $tab[prev]$ contiene todos los valores del pasado inmediatamente anterior, mientras que el arreglo $tab[curr][0..x)$ los del presente (aún faltan por procesar valores del presente en $tab[curr][x..X]$), de acuerdo con Q_1. Los invariantes Q_2 y Q_3 especifican que los valores de *prev* y *curr* solo pueden ser 0 o 1, y el invariante Q_4 que los valores de estas variables no debe coincidir. Los invariantes P_2 y P_3 corresponden a los de la especificación del algoritmo sin optimizar. Con base en este diseño se presenta la función `ks_tab_opt1` en el Algoritmo 3.5.2.

Algoritmo 3.5.2

```
def ks_tab_opt1(V, W, X):
  N = len(V)
  tab = [ [ 0 for _ in range(X+1) ] for _ in range(2) ]
  n,x,prev,curr = 1,0,0,1
  # Q0 ∧ Q1 ∧ Q2 ∧ Q3 ∧ Q4 ∧ P2 ∧ P3
  while n!=N+1:
    if x==X+1: n,x,prev,curr = n+1,0,1-prev,1-curr
    else:
      tab[curr][x] = tab[prev][x]
      if W[n-1]<=x:
        tab[curr][x] = max(tab[curr][x], tab[prev][x-W[n-1]]+V[n-1])
      x += 1
  return tab[prev][X]
```

Además de los cambios derivados de la nueva forma de hacer la tabulación, la función `ks_tab_opt1` incluye una actualización en la inicialización de la tabla: en el momento de su creación, todos los valores en ella son 0. Esto evita tener que procesar la primera columna después de su creación (es decir, se está ahorrando una pasada por una columna completa). Inicialmente, *prev* es asignada el índice 0 y *curr* el índice 1. Es decir, el pasado está registrado en la columna 0 de *tab* y

el presente en la columna 1. El cuerpo del ciclo tiene dos partes: una de cambio de columna (línea 7) y otra de cambio de fila (líneas 8-12). El cambio de columna se hace de manera similar al algoritmo sin optimización, incrementando n en una unidad y asignando 0 a x. Las variables *prev* y *curr* invierten sus valores (¿por qué la resta funciona?). Se pudo optar por intercambiar directamente los valores de estas dos variables apelando a la sustitución "simulatánea" ofrecida por Python. Sin embargo, esto no es posible en lenguajes de programación imperativos como C, C++ o Java. Por esto, se prefiere hacer el "complemento" con 1, dado que esta operación aritmética se puede implementar con una asignación individual sin la necesidad de una variable intermedia/temporal. El cuerpo del ciclo se encarga de encontrar el mejor valor posible para $\phi(n, x)$ de acuerdo con su definición recurrente. Observe el valor que se retorna está al final de la columna *prev* y no en la columna *curr* (¿por qué?).

Se ilustra el uso de la función `ks_tab_opt1` con algunas instancias del problema:

```
1 >>> V = [4, 6, 5, 1]
2 >>> W = [3, 5, 5, 2]
3 >>> print(ks_tab_opt1(V, W, 15))
4 16
5 >>> print(ks_tab_opt1(V, W, 11))
6 11
7 >>> print(ks_tab_opt1(V, W, 5))
8 6
9 >>> print(ks_tab_opt1(V, W, 1))
10 0
```

El diseño de la tabulación para `ks_tab_opt1` permite reducir el espacio de la tabulación a $O(2X) = O(X)$, mientras que el tiempo de ejecución se mantiene en $O(NX)$.

Teorema 3.5.3

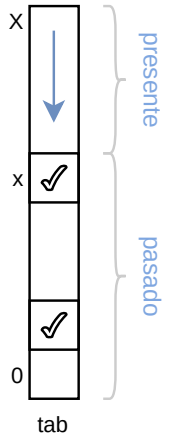
El llamado `ks_tab_opt1(V, W, X)`, con $len(V) = len(W) = N$:

1. Calcula la suma máxima de elementos con valores en $V[0..N)$ cuyos pesos en $W[0..N)$ no exceden X.
2. Toma tiempo $O(NX)$.
3. Usa espacio $O(X)$.

Demostración

Se propone como ejercicio al lector.

3.5.2. Segunda optimización. El espacio de la tabulación puede ser reducido aún más. Note que los valores que se utilizan para actualizar la columna n (i.e., la del presente) están en un espacio acotado dentro de la columna $n-1$ (i.e., la del pasado inmediatamente anterior). Si de alguna forma se pudiera consolidar la información de los dos momentos en el tiempo en una sola columna, esto permitiría reducir el espacio de $2X$ a X posiciones de memoria. La observación clave es que para tabular $\phi(n,x)$, solo se requieren valores de $\phi(n-1,y)$, con $0 \leq y \leq x$. Es decir, en la tabulación deseada se podría "partir" esa columna de tal forma que la parte inferior al índice x tuviera solo valores del pasado inmediato y en la parte superior de x solo valores del presente. El diagrama de necesidades a continuación resume visualmente estas observaciones.



Como la parte inferior de $tab[0..X]$ debe tener disponibles los valores del pasado, necesariamente la forma de llenarla debe iniciar por su parte superior. Los siguientes invariantes hacen explícita esta y las observaciones anteriores.

$$
\begin{aligned}
R_0: &\quad (\forall j \mid 0 \leq j \leq x : tab[j] = \phi(n-1, j)).\\
R_1: &\quad (\forall j \mid x < j \leq X : tab[j] = \phi(n, j)).\\
P_2: &\quad 0 \leq n \leq N+1.\\
P_3: &\quad -1 \leq x \leq X.
\end{aligned}
$$

El invariante R_0 especifica que el subarreglo $tab[0..x]$ tiene los valores del pasado inmediato (i.e., desde $\phi(n-1,0)$ hasta $\phi(n-1,x)$). El invariante R_1, a su vez, especifica que el subarreglo $tab[x+1..X]$ tiene los valores del presente (i.e., desde $\phi(n,x+1)$ hasta $\phi(n,X)$). Los invariantes P_2 y P_3 son los mismos de las dos versiones anteriores de los algoritmos de tabulación de ϕ.

Algoritmo 3.5.3

```
def ks_tab_opt2(V, W, X):
  N = len(V)
  tab = [ 0 for _ in range(X+1) ]
  n,x = 1,X
  # R_0 ∧ R_1 ∧ P_2 ∧ P_3
  while n!=N+1:
    if x==-1: n,x = n+1,X
    else:
      if W[n-1]<=x:
        tab[x] = max(tab[x], tab[x-W[n-1]]+V[n-1])
      x += 1
  return tab[X]
```

Inicialmente, $tab[0..X]$ es creado con valores 0. A partir de la columna $n = 1$ y hasta la columna $n = N$, se hace una de dos cosas. O bien se detecta que se ha procesado una nueva columna (i.e., $x = -1$) y se avanza a la (posible) siguiente columna para comenzar el proceso desde la fila X. De lo contrario, el valor en $tab[x]$ es $\phi(n-1, x)$. Si se tiene capacidad disponible en el morral (i.e., $W[n-1] \leq x$) y hay mayor beneficio en llevar el elemento a no llevarlo, entonces se actualiza $tab[x]$ con $\phi(n-1, x-W[n-1]) + V[n-1]$.

Teorema 3.5.4

El llamado `ks_tab_opt2(V, W, X)`, con $len(V) = len(W) = N$:

1. Calcula la suma máxima de elementos con valores en $V[0..N)$ cuyos pesos en $W[0..N)$ no exceden X.
2. Toma tiempo $O(NX)$.
3. Usa espacio $O(X)$.

Demostración

Se propone como ejercicio al lector.

Asintóticamente, los algoritmos 3.5.2 y 3.5.3 usan la misma cantidad de espacio. Sin embargo, el espacio se reduce exactamente a la mitad en `ks_tab_opt2` y, además, el código es más sucinto y fácil de seguir. Por estas razones debería preferirse esta función a `ks_tab_opt1`.

Ejercicios

1. Demuestre que la cantidad de subconjuntos de un conjunto de N elementos es 2^N.
2. Considere los 4 elementos con pesos y valores dados en el Ejemplo 3.5.1. Enumere los 16 conjuntos de índices correspondientes a esta instancia del problema del morral, y a cada uno de ellos asocie el peso y valor total. Verifique que la conclusión es correcta para cada X en dicho ejemplo.
3. Justifique con un ejemplo que el problema del morral exhibe las propiedades de subestructura óptima y solapamiento.
4. Demuestre que la definición recurrente de $\phi(n, x)$, para $0 \leq n \leq N$ y $0 \leq x \leq X$, en cualquier llamado recurrente mantiene los valores de n y x en estos rangos.
5. En la demostración del Teorema 3.5.1 se afirma que si $S = \emptyset$, entonces $\phi(n, x) = 0$ observando que no hay elementos para escoger (i.e., $n = 0$) o que todos los elementos disponibles tienen un peso mayor a x (i.e., $W[i] > x$ para $0 \leq i < n$). Demuestre en detalle que esta afirmación acerca de ϕ es correcta.
6. En la tabulación propuesta para ϕ en `ks_opt` se opta por llenar $tab[0..N][0..X]$ por columnas. Diseñe una tabulación para ϕ que resulte de llenar tab por filas. Bajo esta decisión de llenar la matriz, ¿es posible optimizar el espacio de la tabulación? Justifique su respuesta.
7. Demuestre el Teorema 3.5.2.
8. Diseñe un algoritmo con memorización para ϕ. Demuestre que es correcto con respecto a la especificación hecha. ¿Cuáles son sus complejidades temporal y espacial? Justifique su respuesta.
9. Investigue acerca de tiempo pseudo-polinimial y explique su relación con algoritmos numéricos. En particular, explique su relación con el algoritmo común para determinar si un número es primo.
10. Demuestre el Teorema 3.5.3.
11. Suponga que a es un número entero que satisface $0 \leq a \leq 1$. Demuestre que $0 \leq 1 - a \leq 1$.
12. En la función `ks_tab_opt1` se retorna el valor `tab[prev][X]`. Explique por qué sería incorrecto retornar el valor `tab[curr][X]`.
13. Demuestre el Teorema 3.5.4.
14. En el planteamiento de la función objetivo, la cantidad de elementos disponibles se hizo variar para considerar diferentes prefijos de los índices de $A[0..N)$. Plantee una definición alternativa a ϕ en la cual se hagan variar los diferentes prefijos de $A[0..N)$.

15. Hay una variante del problema del morral en la cual de cada elemento hay una cantidad ilimitada de copias (y se permite llevar cuantas copias sean deseadas de cada elemento).
    a) Especifique esta variante del problema del morral.
    b) Diseñe una solución con tabulación para el problema especificado usando la metodología propuesta en la Sección 3.3. Si es posible reducir el espacio de la tabulación, redúzcalo al máximo.
16. La siguiente especificación corresponde al problema, comúnmente denominado, suma exacta de un subconjunto (en inglés, *Subset Sum*):

    Entrada: Arreglo $A[0..N)$, con $N \geq 0$, de números y un número X.

    Salida: ¿Hay un subarreglo de $A[0..N)$ cuyos elementos sumen X?

    a) En la literatura se indica que el problema de la suma exacta de un subconjunto es una instancia particular del problema del morral. Justifique por qué esta afirmación es cierta e ilústre su respuesta con ejemplos.
    b) Diseñe un algoritmo con tabulación que permita resolver el problema de suma exacta de un subconjunto.

3.6. El problema del agente viajero

El del agente viajero, es un problema de optimización combinatoria en grafos. Los grafos son un formalismo de las matemáticas utilizado ampliamente como herramienta para abstraer y modelar relaciones entre objetos.

Nota 3.6.1

Un *grafo* G es una pareja (V, E) de *vértices* V y de *arcos* $E \subseteq V \times V$ sobre los vértices. Si $(u, v) \in E$ se dice que hay un arco entre u y v. El grafo puede ser *dirigido* o *no-dirigido*. En un grafo dirigido los arcos tienen dirección, es decir, un origen y un destino. En un grafo no-dirigido los arcos no tienen dirección, únicamente conectan dos vértices. En este texto, mientras no se haga la aclaración, los grafos se suponen no-dirigidos. Se dice que un grafo es *completo* cuando hay un arco entre cualquier par de vértices (distintos). Un *circuito* en un grafo es un camino que visita exactamente cada uno de sus vértices y al finalizar regresa al vértice de origen

Las relaciones de amistad o de seguimiento en redes sociales se pueden modelar como grafos, al igual que sistemas biológicos como redes de co-expresión genética o redes de interacción entre proteinas.

Nota 3.6.2

El problema del agente viajero (en inglés, *travelling salesman problem* o *TSP*) se ocupa de responder la siguiente pregunta: dada una colección de ciudades y las distancias entre cada par de ellas, ¿cuál es la ruta más corta posible que visita cada ciudad exactamente una vez y al finalizar regresa a la ciudad origen?

Este problema fue formulado por primera vez a finales de los años 1920s y ha sido estudiado exhaustivamente en las áreas de optimización y teoria de la complejidad.

De forma general, el sistema de ciudades y rutas en el problema del agente viajero se puede abstraer en un grafo en el cual las ciudades se modelan como vértices y los caminos entre ciudades como arcos. En este problema se menciona una noción de medida para las rutas. Para incluir este elemento en el modelado del problema, se puede optar por tener una función de distancia sobre los arcos delg rafo. Note que en el problema del agente viajero se desea encontrar un circuito de costo mínimo en un grafo completo con respecto a una función de distancia sobre sus arcos.

Problema 3.6.1: El problema del agente viajero

Entrada: Un grafo $G = (V, E)$ completo, con función de peso sobre los arcos $w : E \to \mathbb{R}_{\geq 0}$.

Salida: Un circuito de costo mínimo (con respecto a w) en G.

La salida el Problema 3.6.1 se puede precisar matemáticamente y exhibir así, entre otras cosas, su estructura combinatoria. La primera observación importante es que en un circuito *no* es importante identificar cuál es el vértice de "inicio", pues cualquiera de los vértices en dicho circuito puede ser considerado de inicio (¿por qué?). Entonces, para el modelado del problema es natural volcar el interés en los arcos del grafo. Suponga que se cuenta con un predicado *circuit* que indica, para cualquier $C \subseteq E$, si C es un circuito en G. Es decir, $\mathit{circuit}(G, C)$ (o, simplemente, $\mathit{circuit}(C)$ cuando G se entiende del contexto) es cierto únicamente cuando C es un circuito en G. Además, se define la función *cost* para cualquier subconjunto $C \subseteq E$ y función de peso $w : E \to \mathbb{R}_{\geq 0}$ de la siguiente manera:

$$\mathit{cost}(C) = (+e \mid e \in C : w(e)).$$

Es decir, $cost(C)$ es la suma de los pesos de los arcos en C. Con base en estas definiciones, se formula la salida del problema del agente viajero de la siguiente manera:

$$(\downarrow C \mid C \subseteq E \wedge circuit(C) : cost(C)).$$

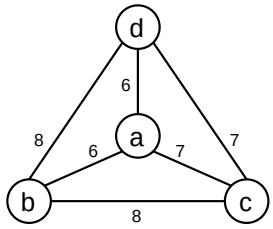
Esta fórmula expresa que se desea calcular el mínimo costo entre todos los circuitos en G. Note que la cantidad de posibles circuitos está acotada por $2^{|E|}$. Más precisamente, por aquellos subconjuntos de $|E|$ de tamaño $|V|$; es decir, por $2^{\binom{|E|}{|V|}}$, en donde

$$\binom{|E|}{|V|}$$

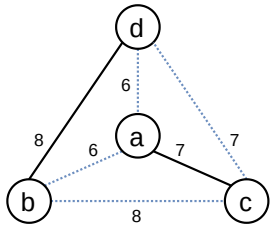
denota la cantidad de subconjuntos de tamaño $|V|$ en un conjunto de tamaño $|E|$. En esta observación yace la naturaleza combinatoria del problema del agente viajero.

Ejemplo 3.6.1

A continuación se presenta un grafo completo con 4 vértices a, b, c, d.



El circuito $a \to b \to c \to d \to a$ es óptimo y tiene costo 27:



3.6.1. Diseño de una función objetivo. Necesariamente hay un circuito de suma mínima cuando G no es vacío (i.e., $V \neq \emptyset$); suponga que $C \subseteq E$ es un circuito óptimo de G en este sentido. Sin pérdida de generalidad, también suponga que

$$C = \{(u_0, u_1), (u_1, u_2), \ldots, (u_{N-1}, u_0)\},$$

en donde $N = |V|$ y $V = \{u_0, u_1, u_2, \ldots, u_{N-1}\}$. Observe que el circuito

$$u_0, u_1, u_2, \ldots, u_{N-1}, u_0$$

es óptimo, si y solo si el camino de u_0 a u_{N-1} en este circuito es óptimo, entre todos los caminos posibles de u_0 a u_{N-1} que incluyen los demás vértices de G (de lo contrario, no sería de costo mínimo –¿por qué?). Esta es la propiedad de la subestructura óptima que se "explotará" para formular una función objetivo que ayude a resolver el problema del agente viajero usando programación dinámica.

Función objetivo. Dado que en un circuito no importa desde dónde se inicie el recorrido, se fija arbitrariamente un vértice $s \in V$ como punto de partida. Se usa el hecho de que el grafo G no es vacío (i.e., $V \neq \emptyset$); de lo contrario el problema es trivial ya que dicho circuito no existe.

Para $X \subseteq V$ y $u \in X$:

$\phi(u, X)$: "mínimo costo de un camino (simple) que inicia en s, visita cada uno de los vértices en X y termina en u."

Basada en la propiedad de la subestructura óptima, la observación importante para aceptar la formulación de ϕ como (potencialmente) útil es que todo circuito óptimo se puede construir con base en rutas óptimas que visiten todos los vértices del grafo dado.

Reformulación de la especificación. El objetivo es entonces calcular el valor de la siguiente expresión:

$$(\downarrow u \mid u \in V \setminus \{s\} : \phi(u, V \setminus \{s\}) + w(u, s)).$$

Entre todos los posibles caminos que inician en s, y que incluyen todos los demás vértices del grafo y terminan en u, se desea calcular aquel camino que sumando la ruta de regreso directa de u a s sea de costo mínimo. Note que al tener que $w(e) \geq 0$ para cualquier arco $e \in E$, es natural que el valor asociado a la minimatoria anterior sea 0 cuando V tiene exactamente un vértice (i.e., $V = \{s\}$).

Definición recurrente. Una ruta de costo mínimo de s a cualquier otro vértice debe estar conformada por subrutas óptimas. Es decir, cada ruta que inicie en s en dicho camino, también debe ser de costo mínimo.

La definición formal de ϕ se presenta para $X \subseteq V \setminus \{s\}$ y $u \in V \setminus \{s\}$:

$$\phi(u, X) = \begin{cases} +\infty & , \text{si } x \notin X, \\ w(s, u) & , \text{si } X = \{u\}, \\ (\downarrow v \mid v \in X \setminus \{u\} : \phi(v, X \setminus \{u\}) + w(v, u)) & , \text{si } u \in X \wedge |X| \geq 2. \end{cases}$$

Se consideran tres casos para definir ϕ. Hay dos opciones entre u y X: que u sea elemento de X o no y, cuando lo es, que sea el único o no. Es imposible que haya un camino que visite todos los elementos de X y que termine en u cuando $u \notin X$; la forma de expresar que esto es imposible es con el valor $+\infty$ dado que es la identidad del mínimo en los reales. Si u es el único elemento de X, solo hay un camino de s

a u: el camino directo entre s y u, el cual tiene costo $w(s, u)$. Cuando en X hay otros elementos en adición a u, se apuesta por todos lo caballos: se prefiere aquél vértice $v \in X \setminus \{u\}$ que en dicho camino conecte directamente con u y para el cual el camino desde s permita construir un camino hasta u de costo mínimo.

Teorema 3.6.1

Sean $s \in V$ y $X \subseteq V \setminus \{s\}$. Si $u \in X$, entonces $\phi(u, X)$ es el costo mínimo de un camino (simple) que inicia en s, visita cada uno de los vértices en X y termina en u.

Demostración

Se propone como ejercicio al lector.

La función ϕ es distinta a las que se han usado para resolver los problemas anteriores. En particular, tiene como parámetro un conjunto. Como tal, esta definición se puede implementar directamente en un lenguaje de programación (usando conjuntos disponibles en sus librerías o implementando la estructura de datos correspondiente). No resulta fácil pensar cómo tabular cuando en una de las dos dimensiones hay un conjunto. Tampoco resulta fácil pensar cómo implementar una memorización eficiente cuando el acceso a los datos depende de una colección y que esta colección puede "mutar" entre los distintos llamados recurrentes. En realidad, estas son noticias *parcialmente* malas. La noticia parcialmente buena es que bajo ciertas suposiciones acerca de la cantidad de vértices en el grafo, los conjuntos se pueden representar con números naturales y las operaciones básicas sobre ellos (al menos las que se requieren para calcular la función objetivo) se pueden realizar en tiempo constante. Entonces, antes de proponer una solución con programación dinámica, se estudia cómo representar conjuntos y algunas de sus operaciones con números naturales.

3.6.2. Especificación de conjuntos con máscaras de bits. El propósito de esta sección es mostrar cómo, bajo ciertas suposiciones, se pueden usar máscaras y operaciones de bits para representar conjuntos y operaciones sobre ellos. En particular, para un conjunto de N elementos, es deseable contar con las siguientes operaciones:

- Representación del conjunto completo o *universal*, junto con todos sus subconjuntos.
- Sustraer un elemento de un conjunto.
- Consultar si un elemento hace parte de un conjunto.

- Determinar si un conjunto es el conjunto unitario determinado por un elemento dado.

Estando disponible y de manera eficiente, la funcionalidad enumerada sería suficiente para implementar un algoritmo con programación dinámica que permita calcular la función ϕ eficientemente. Al menos, esta es la intención.

Las máscaras de bits (i.e., secuencias de bits) ofrecen una posibilidad si la cantidad de elementos N está acotada: un conjunto universal con N elementos se puede representar con N bits. Es más, cualquier conjunto con a lo sumo N elementos se puede también representar con una secuencia de bits de tamaño N. Sin pérdida de generalidad, se puede suponer que hay un orden entre los elementos del conjunto universal que se quiere representar. Bajo esta suposición, se puede asumir que el conjunto a codificar es $U = \{0, 1, \ldots, N-1\}$ de tal forma que 0 identifica el primer elemento de dicho conjunto, 1 el segundo y así, con $N-1$ representando el último elemento en este orden arbitrario. Una secuencia de bits de tamaño N se puede, entonces, usar como función característica para representar exactamente un subconjunto de U: el n-ésimo bit, yendo del menos significativo al más significativo, indica si el n-ésimo elemento de U hace parte del subconjunto, para $0 \leq n < N$.

Ejemplo 3.6.2

Considere el conjunto $\{a, b, c, d\}$ de etiquetas. Suponga que las etiquetas tienen el siguiente orden:

$$a < b < c < d.$$

Entonces:

- La secuencia 1111 representa el conjunto $\{a, b, c, d\}$.
- La secuencia 1010 representa el conjunto $\{b, d\}$.
- La secuencia 0100 representa el conjunto unitario $\{c\}$.
- La secuencia 0000 representa el conjunto vacio $\emptyset$.

Entonces, en el problema del agente viajero se puede suponer que el conjunto de vértices $V = \{v_0, v_1, \ldots, v_{N-1}\}$ y sus subconjuntos, se pueden representar con máscaras de bits de tamaño N (fijando algún orden sobre V).

En un lenguaje de programación, como Python, se puede construir rápidamente un conjunto universo para cualquier $N \in \mathbb{N}$ haciendo corrimiento de bits y una resta (ojo, no es la única manera de hacerlo). A continuación se presenta la función `universe` que dado dicho N, retorna el número entero cuya representación en base 2 es una secuencia de exactamente N unos:

```
def universe(N): return (1<<N)-1
```

Por ejemplo, `universe(3)` es 7 dado que su representación en base 2 es 111. La expresión `1<<N` calcula el número natural 2^N. Al restarle 1, se "apaga" el único bit existente y todos los 0 a su derecha de "encienden", resultando en una secuencia de N unos.

El siguiente reto es determinar si un bit específico está encendido o apagado en una máscara de bits. Suponiendo que se cuenta con una máscara de N bits, para cualquier $0 \leq n < N$, se puede identificar si el n-ésimo bit está encendido con una disyunción bit a bit.

```
def is_elt(n, X): return (X|(1<<n))==X
```

El llamado `is_elt(n, X)` es cierto cuando el n-ésimo bit del código binario del número natural X es 1, suponiendo que n es una posición en la máscara de bits correspondiente a X.

De una manera muy similar, se puede apagar el n-ésimo bit en una máscara de bits.

```
def remove_elt(n, X): return X-(1<<n) if is_elt(n, X) else X
```

Si ese bit está encendido, entonces se apaga; de lo contrario, la máscara no se modifica.

Finalmente, determinar si una máscara representa un conjunto unitario dado es fácil de calcular: en la máscara correspondiente al conjunto unitario $\{n\}$ únicamente hay un bit encendido, exactamente el n-ésimo.

```
def singleton(n, X): return X==(1<<n)
```

Estas cuatro operaciones, basadas en comparación y aritmética de bits, son eficientes en la práctica cuando N no supera el tamaño de la palabra en un procesador. Por ejemplo, en procesadores de 32 bits estas operaciones se pueden implementar directa y eficientemente siempre y cuando $N \leq 32$. Lo mismo sucede con procesadores de 64 bits, etc. De lo contrario, se deberá incurrir en procedimientos adicionales que difícilmente serán de orden constante.

3.6.3. Diseño de un algoritmo con memorización. Bajo la suposición de que se pueden representar los subconjuntos de V con máscaras de bits, se obtiene casi directamente un algoritmo para implementar ϕ con memorización. Concretamente, se supondrá que G tiene N vértices $V = \{0, 1, \ldots, N-1\}$ y que su representación es una matriz de ayacencia $w[0..N)[0..N)$ tal que para cualquier $u, v \in V$, se tiene $w[u][v] = w[v][u] = w(u, v)$ y $w[u][u] = 0$. Es decir, los vértices se identifican con los primeros N números naturales y la función de peso entre vértices se especifica con una matriz cuadrada y simétrica en donde cada entrada indica el peso del arco entre los vértices correspondientes; esta matriz es 0 en su diagonal.

El Algoritmo 3.6.1 presenta las funciones `phi_memo` y `tsp` que resuelven el problema del agente viajero para un grafo de N vértices y función w de peso en los arcos. La función principal es `tsp`, la cual recibe estos dos parámetros (se pudo también optar por tener estos dos parámetros como variables globales, simplificando –pero oscureciendo en este caso en particular– el código). La función `phi_memo` es la solución por memorización que implementa la función ϕ. Note que el vértice 0 juega el papel del vértice s en la especificación de la función ϕ.

Algoritmo 3.6.1

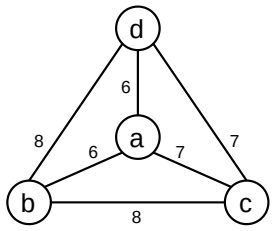
```
INF = float('inf')

def phi_memo(N, w, u, X, mem):
  ans,key = None,(u,X)
  if key in mem: ans = mem[key]
  else:
    if not(is_elt(u, X)): ans = INF
    elif singleton(u, X): ans = w[0][u]
    else:
      ans,Y = INF,remove_elt(u, X)
      for v in range(1, N):
        if is_elt(v, Y):
          ans = min(ans, phi_memo(N, w, v, Y, mem)+w[v][u])
    mem[key] = ans
  return ans

def tsp(N, w):
  ans = INF
  X = remove_elt(0, universe(N))
  mem = dict()
  for u in range(1, N):
    ans = min(ans, phi_memo(N, w, u, X, mem)+w[u][0])
  return ans
```

Un llamado `tsp(N, w)` inicialmente construye en X el conjunto de todos los vértices excepto el vétice de partida, es decir, el conjunto $\{1, \ldots, N-1\}$. Iniciando con la memoria vacía, se itera sobre cada uno de los vértices en este conjunto tratando de encontrar aquella combinación que permita construir un circuito de costo mínimo que inicie en 0 y termine en 0, agotando todos los demás vértices en X. Esto corresponde, tal cual, con la reformulación de la especificación en la Sección 3.6.1.

El llamado `phi_memo(N, w, u, X, mem)` calcula la función $\phi(u, X)$ para el grafo de N vértices con función de peso $w[0..N)[0..N)$, usando la memorización en el diccionario `mem`. Las llaves del diccionario son parejas en donde el primer argumento corresponde a un identificador de vértice y el segundo a una máscara de bits (representando un conjunto de vértices). Si el valor asociado a la llave (u, X) está presente en el diccionario, se usa directamente como valor de retorno. De lo contario, se calcula el valor correspondiente a esta llave y se almacena en el diccionario antes de retornarlo. Note que se calcula este valor con base en cada uno de los tres casos que hacen parte de la definición recurrente de ϕ.

Ejemplo 3.6.3

A continuación se presenta el grafo completo con 4 vértices a, b, c, d del Ejemplo 3.6.1.



Este es un grafo con $N = 4$ vértices y función de peso w dada por la siguiente matriz de adyacencia:

```
w = [ [ 0, 6, 7, 6 ],
      [ 6, 0, 8, 8 ],
      [ 7, 8, 0, 7 ],
      [ 6, 8, 7, 0 ] ]
```

El valor calculado por la función `tsp` en este caso es 27:

```
>>> tsp(4, w)
27
```

La correctitud de `tsp` depende directamente de la correctitud de `phi_memo`, bajo las siguientes condiciones:

$$C_0: \quad (\forall u, X \mid (u, X) \in mem : mem[u][X] = \phi(u, X)).$$
$$C_1: \quad (\forall u, X \mid (u, X) \in mem : 1 \leq u < N \wedge 0 \leq X < 2^N).$$

Las condiciones C_0 y C_1 son condiciones de representación, respectivamente, de la memorización y de conjuntos con máscaras de bits. La condición C_0 indica que los valores almacenados en la memoria compartida son correctos en relación con ϕ. La condición C_1 indica que los valores utilizados como índice de la memoria son los

esperados: un vértice distinto al de inicio y una máscara de bits para el conjunto de N vértices. La correctitud de `phi_memo` consiste, básicamente, en garatizar que estas dos condiciones se mantienen a lo largo de todos los llamados recurrentes.

Teorema 3.6.2: Correctitud de `phi_memo`

Para $V = \{0, 1, \ldots, N-1\}$ y función de peso $w[0..N)[0..N)$, el llamado `phi_memo(N, w, u, X, memo)` con $u \in V \setminus \{0\}$ y $0 \leq X < 2^N$:

1. Preserva las condiciones C_0 y C_1, y
2. Calcula $\phi(u, X)$.

Demostración

Se propone como ejercicio para el lector.

La complejidad temporal de `phi_memo`, suponiendo que el acceso al diccionario y las operaciones sobre la máscara de bits son de orden constante, principalmente, es $O(N \cdot 2^N)$ que resulta principalmente de las líneas de código 11-13. Con base en esta observación (y suposiciones), es fácil concluir que la complejidad temporal del llamado `tsp(N, w)` es $O(N^2 \cdot 2^N)$. Los detalles de estas afirmaciones se proponen como ejercicio para el lector.

Teorema 3.6.3: Correctitud de `phi_memo`

El llamado `tsp(N, w)` toma tiempo $O(N^2 \cdot 2^N)$ y espacio $O(N \cdot 2^N)$.

Demostración

Se propone como ejercicio para el lector.

Ejercicios

1. Encuentre 5 ejemplos cotidianos de grafos, y explique cuáles son los vértices y arcos en cada uno de ellos. En cada caso, explique si el grafo es dirigido o no-dirigido, y si es completo. Si hay una noción de peso, distancia o importancia, ¿cuál es?
2. Suponga que se cuenta con un circuito en un grafo $G = (V, E)$. Explique por qué cada uno de los vértices en dicho circuito puede ser considerado como vértice inicial del circuito.
3. Proponga una definición formal para el predicado *circuit*(C) relativa a un grafo $G = (V, E)$ tal que $C \subseteq E$. Si prefiere, puede tratar *circuit* como una función Booleana *circuit* : $2^E \to \mathbb{B}$, en donde 2^E denota la colección de subconjuntos de E.
4. Explique por qué la cantidad máxima de circuitos en un grafo completo $G = (V, E)$ está acotada por $2^{\binom{|E|}{|V|}}$. ¿Son circuitos todos los subconjuntos de E de tamaño $|V|$? ¿Por qué?
5. Enumere todos los circuitos del grafo en el Ejemplo 3.6.1 sin distinguir entre rotaciones (¿cuántos hay?). Justifique que el circuito de costo mínimo allí es 27.
6. En la definición recurrente de ϕ se incluye como caso base $\phi(u, X) = +\infty$ cuando $u \notin X$. En realidad, este caso se incluye por completitud de ϕ dado que $X \subseteq V \setminus \{s\}$ y $u \in V \setminus \{s\}$. Explique por qué este caso no es necesario para calcular la salida del problema. En particular, justifique cómo ϕ se puede definir, sin introducir inconsistencias matemáticas, únicamente para aquellos casos en los cuales $u \in X$.
7. Es posible obtener una versión distinta de la función `remove_elt` usando exclusivamente operaciones sobre bits (sin operaciones aritméticas). Investigue acerca de esta operación y proponga una versión alternativa de `remove_elt`.
8. ¿Cómo se puede iterar sobre todos los subconjuntos de un conjunto de tamaño N con máscaras de bits? Diseñe un algoritmo que itere sobre todas las posibles máscaras de 2 bits y calcule la tabla de verdad para la disyunción y conjunción.
9. Demuestre el Teorema 3.6.2.
10. Demuestre el Teorema 3.6.3.
11. Extienda el Algoritmo 3.6.1 para que, además de calcular el costo de un circuito óptimo, también retorne un circuito de costo óptimo.
12. Diseñe un algoritmo de programación dinámica con tabulación para ϕ y un algoritmo iterativo para resolver el problema del agente viajero. Estime su complejidad temporal y espacial.

Notas del capítulo y referencias

La noción de programación dinámica cobra un aire más general cuando se considera más allá de la computación y del estudio de algoritmos eficientes para implementar cierto tipo de funciones recurrentes. Para el estudio de fenómenos y problemas económicos, como originalmente nace, es un método recursivo para resolver problemas de decisión secuenciales. Se le conoce también como *inducción reversa* (en inglés, *backward induction*), y es empleada para encontrar reglas de decisión óptimas en juegos, equilibrios perfectos de juegos multi-agente y equilibrios competitivos en modelos económicos dinámicos. El texto autobiográfico de R. Bellman [**Bel84**] incluye apartes de los orígenes de la programación dinámica en este contexto más general, al igual que algunos trabajos más recientes en Economía como el de J. Rust [**Rus08**]. El aparte textual que se presenta en la Nota 3.0.1 es tomado de [**Bel84**, p. 159]; la traducción al Castallano es responsabilidad del autor.

Las propiedades de la subestructura y solapamiento son planteadas comúnmente para estudiar problemas cuyas soluciones son candidatas a ser encontradas con programación dinámica. Lo mismo sucede con las técnicas de implementación por memorización (en inglés, *top-down*) y tabulación (en inglés, *bottom-up*). Estos pares de temas se encuentran explicados, de manera similar a la seguida en este capítulo, en [**CLRS22**] y [**KET06**]. La metodología propuesta en la Sección 3.3 es original, pero de ninguna forma alejada a otras propuestas como la encontrada en [**Eri19**]. En el caso de este texto, el enfoque se basa en la definición de una función objetivo que posteriormente se define formalmente de manera recurrente.

Los problemas escogidos para ilustrar el uso de la programación dinámica también son extensamente conocidos en la comunidad de algoritmos. Posiblemente, el enfoque de anotar los diseños de memorización con condiciones y de tabulación con invariantes explícitos no sea tan popular. En este capítulo se ha hecho el esfuerzo de exponer las principales propiedades de los problemas y los algoritmos usando una notación matemática sencilla. El uso de invariantes à la Dijkstra & Scholten también es planteado por J. Bohórquez [**Boh06**]. El algoritmo que implementa la reducción a una sola columna en la Sección 3.5 está inspirado en observaciones hechas en [**Eri19**].

Los textos de T. Cormen et al. [**CLRS22**], J. Kleinberg y É. Tardos [**KET06**], J. Erickson [**Eri19**], H. Bhasin [**Bha15**] y A. Levitin [**Lev12**] son fuentes extensas de ejemplos de soluciones con programación dinámica y ejercicios para ser resueltos con esta técnica. Textos de entrenamiento diseñados para programación competitiva como los de S. Skiena [**Ski08**] y S. Halim et al. [**HHE20**] contienen ejemplos complementarios, además de 'trucos' de implementación, al igual que páginas de internet asociados a programación competitiva y algoritmos.

Además de las técnicas de reducción de espacio vistas para tabulaciones, existen técnicas para la reducción del tiempo de cómputo para algunos problemas que admiten programación dinámica como solución. Entre ellas se encuentran las basadas en monotonías y desigualdades propuestas por D. Knuth en [**Knu88**]. Existen otras basadas en envolvimientos convexos, y en dividir y conquistar; se sugiere al lector [**Bei13**] para un recuento más detallado y ejemplos.

Capítulo 4

Algoritmos voraces

Los algoritmos voraces (en inglés, *greedy algorithms*) sirven comúnmente para resolver problemas de optimización. Están basados en la premisa de que la elección reiterativa de óptimos locales garantiza, al final del proceso de elección, un óptimo global. Es decir, para resolver un problema de optimización, un algoritmo voraz una y otra vez toma una mejor opción local (sin fijarse en sus consecuencias) de tal forma que la colección de elecciones hecha finalmente constituye una solución óptima global.

Sin embargo, en la práctica, la optimización local rara vez conduce a un óptimo global. Entonces, ¿por qué estudiar este tipo de algoritmos? La razón principal es que cuando los algoritmos voraces funcionan, son muy eficientes, sencillos de programar y elegantes. Hay una frase anónima que resume muy bien esta situación que parece paradójica:

> Los algoritmos voraces no funcionan, pero cuando sí, lo hacen muy bien.
>
> *Autor desconocido.*

El diseño de un algoritmo voraz debe estar *siempre* acompañado de una demostración de su corrección, i.e., de que logra resolver el problema de optimización planteado. Generalmente, estas demostraciones siguen un mismo patrón, el cual se identifica en el desarrollo de este capítulo y se ilustrará con ejemplos de diferente índole. Es importante estar familiarizado con este tipo de demostraciones porque, a veces, no es fácil encontrar un argumento formal que justifique la correctitud de los algoritmos voraces.

4.1. Agendamiento de actividades

Considere el problema de agendar procedimientos quirúrgicos en la sala de cirugías de un hospital o clínica. Cada cirugía tiene un horario, con horas de inicio y finalización. Suponiendo que entre más cirugías se hagan, mayor será el beneficio para los pacientes (acá nada tiene que ver el dinero, ¿cierto?), surge la siguiente pregunta: ¿cómo maximizar la cantidad de cirugías a realizar sin que haya conflicto de horario entre ellas?

De una forma más general, esta pregunta es formulada en el problema del agendamiento de actividades.

Problema 4.1.1: Agendamiento de actividades

Entrada: Un arreglo $A[0..N)$, $N \geq 0$, de parejas de números (s_n, e_n) tales que $0 \leq s_n < e_n$ indican el tiempo de inicio y finalización de la actividad $0 \leq n < N$, respectivamente.

Salida: Máxima cantidad de actividades en $A[0..N)$ que se pueden agendar sin conflicto.

Como convención para cualquier n, con $0 \leq n < N$, las expresiones $A[n][0]$ y $A[n][1]$ denotan, respectivamente, el tiempo de inicio y finalización de la actividad n: es decir, $s_n = A[n][0]$ y $e_n = A[n][1]$. Note que todos los tiempos se expresan con cantidades no negativas. Es necesario precisar qué significa actividades *compatibles* y en *conflicto* para entender completamente la especificación del problema de agendamiento de actividades.

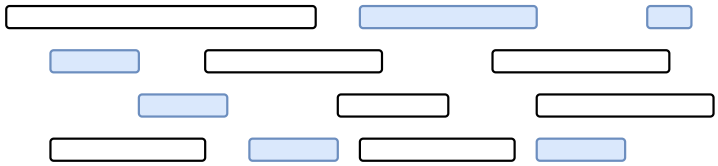
Nota 4.1.1

Por convención, se entenderá que cada actividad (s, e) representa el intervalo cerrado-abierto de tiempo $[s..e)$. De esta forma, dos actividades (s_i, e_i) y (s_j, e_j) son *compatibles* sii $[s_i..e_i) \cap [s_j..e_j) = \emptyset$; de lo contrario, se dice que están en *conflicto*.

Entonces, la salida del problema indica que se desea determinar el tamaño de un conjunto maximal de actividades en $A[0..N)$ que sean compatibles mútuamente (o, de forma equivalente, en el cual ningún par de ellas esté en conflicto).

Ejemplo 4.1.1

Se ilustra cada cirugía como un rectángulo cuya coordenada izquierda en el eje horizontal denota el tiempo de inicio y la derecha el tiempo de finalización.



Las actividades que tienen su fondo resaltado hacen parte de un conjunto maximal de cirugías compatibles, sin conflictos entre ellas. Dada esta entrada, la salida es 6 para el problema de agendamiento.

El problema de agendamiento se puede resolver exhaustivamente enumerando todos los subconjuntos de actividades y, entre aquellos que no tienen conflictos, determinar el tamaño máximo posible. Es más, este problema exhibe las propiedades de la subestructura óptima y solapamiento (¿por qué?), y por ello es buen candidato a ser resuelto con programación dinámica (ver Ejercicio 14).

4.1.1. Abstracción y estrategia voraz. El enfoque voraz es distinto al de una exploracíon exhaustiva o la de utilizar programación dinámica, pues evita apostarle a 'todos los caballos' y en cambio apuesta solo a uno de ellos, a uno que ganará (o que no perderá).

La solución voraz que se presenta a continuación está basada en la siguiente observación/intuición: entre más extenso sea el rango para agendar cirugías, más cirugías se podrán agendar. Suponga que se cuenta con un conjunto X de cirugías para agendar. La pregunta clave es, ¿cuál agendar primero? Bajo la intuición anterior, se puede elegir una cirugía $a \in X$ que termine lo más pronto posible dado que esto permite que, para una posterior elección, haya más tiempo para agendar otras cirugías bajo la misma estrategia. Esto es, habiendo escogido a, el proceso de agendamiento de las cirugías en X se repetiría con aquellas actividades que no estén en conflicto con a, seleccionando aquellas que terminen primero y que no generen conflicto alguno.

Esta será la apuesta por una estrategia voraz en la cual se optimizan localmente las decisiones, esperando que al final resulte un agendamiento con la mayor cantidad de cirugías posibles. Para ello, se demuestra formalmente que dicha intuición en realidad funciona, con la ayuda de dos teoremas: uno que justifica la optimización local y otro que garantiza que una secuencia de optmizaciones locales resulta en una selección óptima globalmente.

Teorema 4.1.1: Optimización local

Sea X un conjunto finito de actividades. Si $a \in X$ es tal que su tiempo de finalización es mínimo en X, entonces a hace parte de un conjunto maximal y sin conflicto de actividades de X.

Demostración

Suponga que $Y \subseteq X$ es un conjunto maximal y sin conflicto. Como $X \neq \emptyset$, necesariamente $Y \neq \emptyset$ (¿por qué?). Se procede por casos:

- Si $a \in Y$, entonces la conclusión del teorema es correcta.
- Si $a \notin Y$, entonces sea b aquella actividad en Y que tiene tiempo de finalización mínimo. Necesariamente $b \neq a$, porque de lo contrario $a \in Y$ (una contradicción). Considere el conjunto $Z = (Y \setminus \{b\}) \cup \{a\}$. Como $b \in Y$ y $a \notin Y$, se tiene que $|Z| = |Y|$. Además note que Z no tiene conflictos porque el tiempo de finalización de b no puede ser menor que el de a (a tiene tiempo de finalización mínimo entre todas las actividades de X, luego también de Y): al remover b de Y, las actividades que están en $Y \setminus \{b\}$ no tienen conflicto y como a no termina después de b, Z no tiene conflictos (por suposición, en Y no hay conflictos). Luego, Z es un conjunto maximal y sin conflictos de actividades de X que contiene a a, como se desea.

El efecto práctico del Teorema 4.1.1 es el siguiente: en un conjunto de actividades, cualquiera de aquellas que termine lo más pronto posible puede ser parte de un agendamiento óptimo. Luego, la estrategia es aplicar reiterativamente este principio para obtener, por construcción, un agendamiento óptimo.

Algoritmo 4.1.1

Para un conjunto (finito) de actividades X:

1. Si $X = \emptyset$, entonces no hacer nada.
2. De lo contrario, sea a una actividad en X con tiempo de finalización mínimo:
   a) seleccionar a como parte del agendamiento y
   b) recurrir con las actividades en X que no estén en conflicto con a.

El Algoritmo 4.1.1 considera dos casos sobre un conjunto (finito) de actividades X. Si X es vacío, entonces no hay nada que agendar y el algoritmo no hace nada. Si X no es vacío, entonces se identifica una de la actividades en X que termina lo más pronto posible. Esta actividad es seleccionada como parte de la respuesta, la cual se sigue construyendo al recurrir con aquellas actividades en X que son compatibles con a. La corrección de este algoritmo corresponde a la demostración del teorema de optimización global: el algoritmo basado en la elección localmente óptima logra un agendamiento óptimo (globalmente).

Teorema 4.1.2: Optimización global

Sea X un conjunto finito de actividades. El Algoritmo 4.1.1 construye un agendamiento óptimo para X.

Demostración

El Algoritmo 4.1.1 construye un agendamiento sin conflictos (¿por qué?); luego, basta demostrar que la colección de actividades seleccionadas por este algoritmo es de tamaño máximo. Se procede por inducción (y por casos):

- Si $X = \emptyset$, el agendamiento óptimo es vacío; el algoritmo responde correctamente en este caso pues no hace ningún agendamiento.
- Si $X \neq \emptyset$, suponga que el algoritmo selecciona $n \geq 1$ actividades en X. Por ejemplo, la siguiente colección ordenada ascendentemente por el tiempo de *inicio*:

  $$a_0, \ldots, a_{n-1}.$$

  Suponga también que

  $$b_0, \ldots, b_{m-1}$$

  es un agendamiento óptimo de actividades en X, también ordenado ascendentemente por el tiempo de inicio de las actividades. El objetivo es demostrar que $n = m$. Si los dos agendamientos son iguales, entonces la propiedad es cierta. De lo contrario, sin pérdida de generalidad, se supone que los dos agendamientos son iguales en las primeras k actividades, con $0 \leq k < n$: su primera diferencia es $a_k \neq b_k$ (note que esto descarta la posibilidad de que $a_0, \ldots, a_{n-1}$ sea un prefijo de $b_0, \ldots, b_{m-1}$; ¿por qué esta suposición es correcta?). Es decir, el agendamiento óptimo es de la forma

  $$a_0, \ldots, a_{k-1}, b_k, b_{k+1}, \ldots, b_{m-1}.$$

Note que a_k y b_k son compatibles con las actividades en $a_0, \ldots, a_{k-1}$ (¿por qué?). Como a_k es una actividad que finaliza lo más temprano posible entre aquellas en X que no tienen conflicto con $a_0, \ldots, a_{k-1}$, necesariamente a_k no puede terminar después de b_k (dado que b_k también está en X y no tiene conflicto con $a_0, \ldots, a_{k-1}$). En consecuencia, a_k no entra en conflicto con ninguna de las actividades en $b_{k+1}, \ldots, b_{m-1}$. Luego, el siguiente agendamiento no tiene conflictos:

$$a_0, \ldots, a_{k-1}, a_k, b_{k+1}, \ldots, b_{m-1}.$$

Sea $Y \subsetneq X$ el conjunto de actividades que no tiene conflictos con ninguna actividad en $\{a_0, \ldots, a_k\}$. Observe que $\{a_{k+1}, \ldots, a_{n-1}\} \subseteq Y$ y $\{b_{k+1}, \ldots, b_{m-1}\} \subseteq Y$. Por la hipótesis inductiva, con $|Y| < |X|$, el agendamiento

$$a_{k+1}, \ldots, a_{n-1}$$

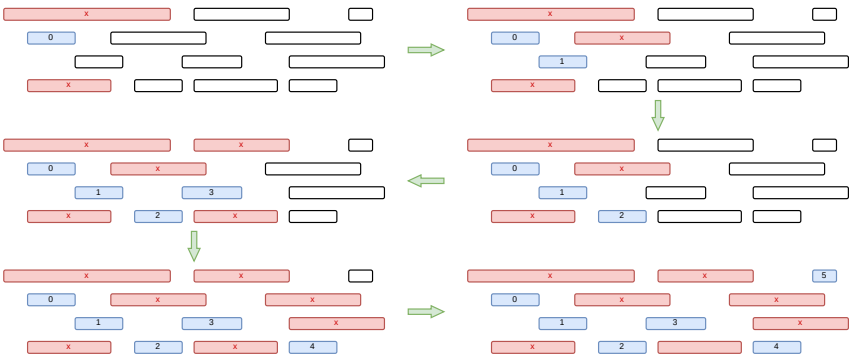
es maximal para las actividades en Y. Luego, $n - k \geq m - k$; es decir, $n \geq m$. Como m es la longitud de una agendamiento óptimo para las actividades en X, necesariamente $n \leq m$. En conclusión, $n = m$.

En cualquiera de los dos casos, el agendamiento hecho por el Algoritmo 4.1.1 es óptimo.

Es importante resaltar que puede haber más de un agendamiento óptimo; esto ha sido indicado antes, incluyendo el cuerpo de la demostración del Teorema 4.1.2. El Ejemplo 4.1.2 muestra cuál es el agendamiento hecho por el Algoritmo 4.1.1 para el conjunto de actividades presentado en el Ejemplo 4.1.1, al inicio de esta sección.

Ejemplo 4.1.2

El siguiente diagrama muestra, paso a paso, cómo el Algoritmo 4.1.2 selecciona un conjunto máximo de actividades compatibles a partir del conjunto de actividades en el Ejemplo 4.1.1.



Cada elección hecha por el algoritmo se indica con un número consecutivo y aquellas actividades que van siendo descartadas porque tiene conflictos con al menos una de las actividades seleccionadas se marcan con una 'x'.

4.1.2. Una implementación eficiente. Hasta este punto, el diseño y análisis del algoritmo voraz se ha hecho con base en una abstracción del problema original, suponiendo que la colección de actividades es un conjunto. De acuerdo con la especificación del problema, la colección de actividades es un arreglo $A[0..N)$ de parejas. Por ello, aún hace falta algo más de trabajo para llegar a tener un algoritmo que opere sobre arreglos, con la estrategia voraz diseñada.

El reto fundamental es evitar hacer un cálculo costoso, en el caso recurrente, de aquellas actividades que son compatibles con la actividad seleccionada. Esto se puede lograr de varias formas. Por ejemplo, bajo el supuesto de que $A[0..N)$ está ordenado ascendentemente por el tiempo de finalización de las actividades. Note que en este caso, si se ha hecho el mejor agendamiento posible para $A[0..n)$, con $0 \leq n \leq N$, las actividades candidatas para extender este mejor agendamiento parcial están en $A[n..N)$. Esto es porque el tiempo de finalización de estas actividades no es menor que el tiempo de finalización de la actividad $n-1$ (y, por ende, de ninguna actividad antes de ella). Es decir, si se tiene registro de qué actividades han sido exploradas (e.g., $A[0..n)$) y el tiempo de finalización de la última tarea seleccionada allí, entonces se puede recurrir sobre las demás actividades en $A[n..N)$, en orden: se descarta $A[n]$ cuando tiene conflicto con alguna de las ya seleccionadas en $A[0..n)$ (esto se hace con base en el tiempo de finalizción que la última actividad seleccionada) o se selecciona porque termina más temprano entre todas aquellas actividades en $A[n..N)$ que no generan conflicto con la selección hecha hasta ese punto.

La siguiente función objetivo, para $0 \leq n \leq N$ y $t \in \mathbb{R}_{\geq 0}$, ayuda a plasmar matemáticamente la idea elaborada en el párrafo anterior:

$$\phi(n,t) \ : \ \text{"máxima cantidad de actividades compatibles en } A[n..N) \text{ que no tienen conflicto con el intervalo } (0..t)\text{."}$$

Para resolver el problema de agendamiento de actividades, el objetivo es calcular $\phi(0,0)$ suponiendo que ninguna actividad tiene tiempos expresados con números negativos, como está estipulado en la especificación del problema.

La definición recurrente de ϕ se establece tomando a n como pivote, y seleccionando o descartando actividades en relación con el intervalo $(0..t)$. Formalmente, para $0 \leq n \leq N$ y $t \in \mathbb{R}_{\geq 0}$, se propone la siguiente definición recurrente de la

función objetivo:

$$\phi(n,t) = \begin{cases} 0 & \text{, si } n = N, \\ \phi(n+1,t) & \text{, si } n \neq N \wedge A[n][0] < t, \\ 1 + \phi(n+1, A[n][1]) & \text{, si } n \neq N \wedge A[n][0] \geq t. \end{cases}$$

El caso base corresponde a la situación en la cual no hay actividades para agendar; el agendamiento óptimo tiene tamaño 0. El caso inductivo se divide en dos, siempre suponiendo que hay al menos una actividad que potencialmente puede ser agendada (i.e., $n \neq N$). Si hay conflicto entre la actividad n y el intervalo "protegido" (i.e., $A[n][0] < t$), entonces esta se ignora y se recurre con el resto de las actividades. De lo contrario, si no hay conflicto entre la actividad n y el intervalo protegido (i.e., $A[n][0] \geq t$), entonces se selecciona la actividad y se recurre con las demás que están pendientes por explorar, actualizando a $(0..A[n][1])$ el intervalo protegido (i.e., con el cual se quiere evitar un conflicto).

Teorema 4.1.3

Suponga que las actividades en $A[0..N)$ están ordenadas ascendentemente por tiempo de finalización (i.e., $A[i][1] \leq A[j][1]$, para $0 \leq i < j < N$). Entonces:

1. Para $0 \leq n \leq N$ y $t \in \mathbb{R}_{\geq 0}$, el valor de $\phi(n,t)$ es la máxima cantidad de actividades compatibles en $A[n..N)$ que no tienen conflicto con el intervalo $(0..t)$.
2. El valor de $\phi(0,0)$ es la máxima cantidad de actividades en $A[0..N)$ que se pueden agendar sin conflicto.

Demostración

Se propone como ejercicio al lector.

Con base en la definición de ϕ, es claro cómo diseñar un algoritmo recurrente para su implementación. A continuación se presenta el Algoritmo 4.1.2, como la implementación de la función `act` que permite calcular ϕ.

Algoritmo 4.1.2

```
1 def act(A, N, n, t):
2   ans = None
3   if n==N: ans = 0
4   else:
```

```
    if A[n][0]<t: ans = act(A, N, n+1, t)
    else: ans = 1 + act(A, N, n+1, A[n][1])
  return ans
```

A continuación se presenta un ejemplo con un llamado a la función `act` para un conjunto de actividades que, a escala, representan a aquellas en el Ejemplo 4.1.1.

Ejemplo 4.1.3

Considere el siguiente arreglo de parejas que especifica las actividades del Ejemplo 4.1.1:

```
A = [ (0, 14), (16, 24), (29, 31),
      (2, 6), (9, 17), (22, 30),
      (6, 10), (15, 20), (24, 32),
      (2, 9), (11, 15), (16, 23), (24, 28) ]
```

Después de ordenar el arreglo ascendentemente por la segunda componente de las parejas, se obtiene:

```
>>> A.sort(key = lambda x: x[1])
>>> A
[ (2, 6), (2, 9), (6, 10), (0, 14), (11, 15),
  (9, 17), (15, 20), (16, 23), (16, 24), (24, 28),
  (22, 30), (29, 31), (24, 32) ]
```

Con base en la versión ordenada del arreglo, se tiene:

```
>>> act(A, len(A), 0, 0)
6
```

El Teorema 4.1.4 presenta los resultados de correctitud de la función `act`, junto con sus complejidades temporal y espacial.

Teorema 4.1.4

Si las actividades en $A[0..N)$ está ordenadas ascendentemente por tiempo de finalización, entonces el llamado `act(A, N, 0, 0)` calcula $\phi(0, 0)$, y toma tiempo y espacio $O(N)$. Si es necesario ordenar el arreglo, entonces la complejidad temporal es $O(N \log N)$.

Demostración

Note que la función `act` es una transcripción de la función ϕ, agregando los parámetros `A` y `N` que nunca son modificados. El tiempo de ejecución de `act` es linealmente proporcional a la cantidad de elementos en $A[0..N)$, i.e., es $O(N)$. El espacio es $O(N)$ si no hay optimizaciones que se apliquen a los llamados recurrentes.

Ejercicios

1. Diseñe un algoritmo de búsqueda exhaustiva que enumere todos los subconjuntos de actividades en $A[0..N)$ para resolver el problema de agendamiento.
2. Enumere todo los agendamientos óptimos del conjunto de actividades presentado en el Ejemplo 4.1.1.
3. Explique por qué el problema de agendamiento de actividades exhibe las propiedades de subestructura óptima y solapamiento. Ilustre la explicación con un ejemplo.
4. Demuestre que cada una de las siguientes afirmaciones hechas en la demostración del Teorema 4.1.1 es cierta:
   a) Si $X \neq \emptyset$, entonces $Y \neq \emptyset$.
   b) Por suposición, Y no tiene conflictos.
   c) El conjunto Z no tiene conflictos.
5. En el caso inductivo de la demostración del Teorema 4.1.2 se afirma que $a_0, \ldots, a_{n-1}$ no puede ser un prefijo de $b_0, \ldots, b_{m-1}$. Demuestre que esta afirmación es correcta.
6. En el caso inductivo de la demostración del Teorema 4.1.2 se afirma que a_k y b_k son compatibles con las actividades en $a_0, \ldots, a_{k-1}$. También se indica que el tiempo de finalización de a_k no puede ser mayor que el tiempo de finalización de b_k. Justifique por qué estas afirmaciones son ciertas.
7. La argumentación final en la demostración del Teorema 4.1.2 se basa, principalmente, en el hecho de que $|Y| < |X|$. ¿Por qué es necesario que esta desigualdad sea estricta? Justifique claramente su respuesta.
8. Teniendo como marco la demostración del Teorema 4.1.2, demuestre o refute: el agendamiento $b_{k+1}, \ldots, b_{m-1}$ es óptimo para Y. *Ayuda:* suponga lo contrario y obtenga una contradicción en relación con la optimalidad del agendamiento $b_0, \ldots, b_{m-1}$ para X.
9. En el planteamiento recurrente de ϕ, el caso en el cual $n \neq N \wedge A[n][0] \geq t$ indica que la actividad n se considera como parte del agendamiento óptimo que se está construyendo. ¿Por qué no hay conflicto si $A[n][0]$ puede ser igual a t?

10. Justifique con un ejemplo, por qué la suposición de ordenamiento de $A[0..N)$ es necesaria para que el Teorema 4.1.3 sea cierto.

11. Considere la siguiente función objetivo para el problema de agendamiento de actividades, con $0 \leq n \leq N$ y $t \in \mathbb{R}_{\geq 0} \cup \{+\infty\}$:

$$\overline{\phi}(n,t) \ : \ \text{“máxima cantidad de actividades compatibles en } A[0..n) \text{ que no tienen conflicto con el intervalo } [t.. + \infty)\text{.”}$$

    Para resolver el problema de agendamiento de actividades, el objetivo es calcular $\overline{\phi}(N, +\infty)$. Proponga un planteamiento recurrente para $\overline{\phi}$, y diseñe un algoritmo voraz que la calcule en tiempo $O(N)$ y espacio $O(1)$.

12. Diseñe un algoritmo voraz que resuelva en tiempo $O(N)$ el problema de agendamiento bajo la suposición de que el arreglo $A[0..N)$ está ordenado:
    a) Ascendentemente por tiempo inicio de las actividades.
    b) Descendentemente por tiempo de inicio de las actividades.

13. La estrategia voraz diseñada en esta sección está basada en la idea de escoger aquella actividad que termine lo más pronto posible entre aquellas que no generan conflicto, para posteriormente recurrir. Para cada una de las siguientes estrategias voraces, o bien demuestre que sirve como regla de optimización local para resolver el problema de agendamiento de actividades (si hay empates, se debería poder escoger cualquiera de las posibilidades) ó describa un pequeño contraejemplo que iluestre por qué no sirve:
    a) Escoja aquella actividad a que *termina* lo más tarde posible, descarte las actividades que están en conflicto con a y recurra.
    b) Escoja aquella actividad a que *comienza* lo más pronto posible, descarte las actividades que están en conflicto con a y recurra.
    c) Escoja aquella actividad a que *comienza* lo más tarde posible, descarte las actividades que están en conflicto con a y recurra.
    d) Escoja aquella actividad a que tenga la *menor cantidad de conflictos*, descarte las actividades que están en conflicto con a y recurra.
    e) Si no hay conflicto, escoja todas las actividades; de lo contrario, descarte la actividad con la *mayor duración* y recurra.
    f) Si no hay conflicto, escoja todas las actividades; de lo contrario, descarte la actividad con la *mayor cantidad de conflictos* y recurra.

14. Diseñe un algoritmo con programación dinámica para calcular ϕ, sin suponer el principio de optimalidad local, y que resuelva el problema de agendamiento de actividades, suponiendo que los tiempos de las actividades en $A[0..N)$ son números naturales.

15. Diseñe un algoritmo iterativo que, en tiempo $O(N)$ y espacio $O(1)$, permita calcular la función ϕ.

4.2. Árboles de cubrimiento mínimo

Identificar un árbol de cubrimiento mínimo es un problema de optimización sobre grafos. Dado un grafo no dirigido y conexo, con peso en sus arcos, el problema consiste en encontrar un subgrafo que cumpla con dos propiedades específicas: una estructural de ser conexo y sin ciclos, y otra de optimalidad en cuanto a la menor suma posible de los pesos de sus arcos. Este problema, conocido en inglés como *minimum spanning tree* (MST), ha sido ampliamente estudiado y se conocen soluciones algorítmicas *voraces* eficientes, además relativamente fáciles de implementar. En esta sección se establecen los principios fundamentales sobre los cuales están diseñados varios de algoritmos voraces que resuelven el problema de calcular árboles de cubrimiento mínimo.

Antes de especificar el problema algorítmico es indispensable introducir formalmente algunos conceptos y términos útiles para su definición. En particular, se precisan las nociones de árbol de cubrimiento y de minimalidad en este contexto a partir de nociones de grafos que comúnmente se conocen.

Definición 4.2.1

Un *árbol* es un grafo no dirigido que es conexo y acíclico. Dado un grafo no dirigido $G = (V, E)$ y una función de peso $w : E \to \mathbb{R}_{>0}$, se dice que $T = (V_T, E_T)$ es un *árbol de cubrimiento* de G (con respecto a w) sii:

$$T \text{ es un árbol,} \qquad V_T = V \qquad \text{y} \qquad E_T \subseteq E.$$

Además, se dice que T es un *árbol de cubrimiento mínimo* de G (con respecto a w) sii ningún otro árbol de cubrimiento de G tiene una suma de pesos de arcos (con respecto a w) menor que E_T.

Intuitivamente, un árbol de cubrimiento mínimo de un grafo es un subgrafo que conecta todos los vértices, sin ciclos y lo menos pesado posible. Un escenario común en el cual se requieren árboles de cubrimiento mínimo es en el diseño de redes de comunicaciones. Considere diferentes ubicaciones que deben ser conectadas y que el costo de conectar dos ubicaciones es proporcional a su distancia. El objetivo es conectar toda las ubicaciones usando el presupuesto más modesto posible cuando. La solución deberá ser un árbol de cubrimiento mínimo.

Problema 4.2.1: Árbol de cubrimiento mínimo

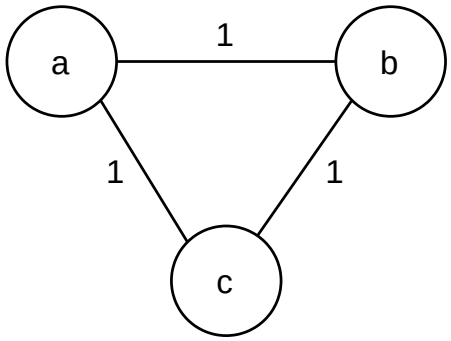
Entrada: Un grafo $G = (V, E)$ conexo con función de peso $w : E \to \mathbb{R}_{>0}$.

Salida: Un árbol $T = (V, E_T)$ de cubrimiento mínimo de G (con respecto a w).

Una aproximación de fuerza bruta para resolver el problema consiste en enumerar todos los suconjuntos de E y determinar, entre aquellos que representen un árbol de cubrimiento de G, uno de suma mínima. Como hay un total de $\Theta(2^{|E|})$ subconjuntos de arcos de E, necesariamente este enfoque no es viable en la práctica. Aún observando que un árbol con vértices V tiene $|V| - 1$ arcos (¿por qué?), la enumeración de los subconjuntos de $|E|$ de tamaño $|V| - 1$ sigue siendo exponencial en tamaño. Es decir, los enfoques de fuerza bruta no son una opción si se desea considerar grafos con, al menos, centenares de arcos.

Ejemplo 4.2.1

El MST de un grafo dado no es necesariamente único. Por ejemplo, considere el grafo de tres vértices que se dibuja a continuación:



Este grafo tiene tres árboles de cubrimiento mínimo, dados por los siguientes conjuntos de arcos, cada uno con suma de pesos 2: $\{(a, b), (b, c)\}$, $\{(a, b), (a, c)\}$ y $\{(a, c), (b, c)\}$.

La idea general para construir un MST de $G = (V, E)$ con función de peso $w : E \to \mathbb{R}_{>0}$ usando un principio de optimización local es la siguiente: mantener un conjunto de arcos $A \subseteq E$ que cumpla con la siguiente condición:

$$C_0 : \quad A \text{ únicamente contiene arcos que hacen parte de un MST de } G.$$

El reto consiste, entonces, en construir incrementalmente a A agregando nuevos arcos, uno a la vez, mientras se respeta la condición C_0. De esta forma, cuando no sea posible agregar nuevos arcos, A incluirá únicamente los arcos de un MST de G.

El diseño genérico del algoritmo descrito verbalmente hasta ahora se presenta en el Algoritmo 4.2.1.

Algoritmo 4.2.1: Algoritmo genérico para construir un MST

1. $A = \emptyset$.
2. Mientras A no sea un MST de G:
   a) Encontrar un arco *seguro* en E para extender A (e.g., e)
   b) extender A con e.
3. Retornar A.

El algoritmo genérico está basado en el concepto de arco seguro. En este contexto, un arco $e \in E$ es *seguro* para A sii $A \cup \{e\}$ es un subconjunto de arcos de un MST de G. Es decir, lo que propone el Algoritmo 4.2.1 es aumentar un conjunto de arcos seguros con nuevos arcos seguros, mientras sea posible, con el propósito que ese conjunto maximal de arcos seguros en realidad constituya un MST de G.

Nota 4.2.1

Se presentan algunas definiciones que serán útiles en los resultados que se presentan en el resto de la sección:

- Un *corte* de V es una pareja (S_0, S_1) tal que $\{S_0, S_1\}$ es una partición de V (se permite que alguno entre S_0 y S_1 sea vacío).
- Un arco $e \in E$ *cruza* un corte (S_0, S_1) de V sii uno de los vértices en e está en S_0 y el otro está en S_1.
- Un corte (S_0, S_1) de V *respeta* un subconjunto de arcos $E' \subseteq E$ sii ningún arco en E' cruza (S_0, S_1).
- Un arco $e \in E$ es *ligero* para un corte (S_0, S_1) de V sii $w(e)$ es mínimo entre todos los arcos que cruzan (S_0, S_1) (note que puede haber más de un arco ligero para un corte (S_0, S_1) de V.)

El resultado que se presenta a continuación es la base matemática sobre la cual funcionan los algoritmos voraces comúnmente utilizados para calcular el MST de un grafo. Este no es más que el principio de optimización local que también se identificó para el problema del agendamiento de actividades en la Sección 4.1.

Teorema 4.2.1: Optimización local

Sea $A \subseteq E$ tal que cada uno de sus arcos hace parte de un MST de G y $(S, V \setminus S)$ un corte de V que respeta a A. Si $(u, v) \in E$ es un arco ligero que cruza $(S, V \setminus S)$, entonces (u, v) es seguro para A.

Demostración

El grafo G tiene un MST dado que es conexo (y no dirigido); sea $T = (V, E_T)$ un MST de G tal que $A \subseteq E_T$. Note que T existe con esta característica por la suposición sobre A. Se procede por casos:

- Si $(u, v) \in E_T$, entonces $A \cup \{(e, v)\} \subseteq E_T$. Por ende, (u, v) es seguro para A.
- Si $(u, v) \notin E_T$, entonces note que $(V, E_T \cup \{(u, v)\})$ no es un árbol (¿por qué?). Como T es un MST de G, hay un camino único (¿por qué?) entre u y v en T. Dado que u y v están en "lados" opuestos de $(S, V \setminus S)$, al menos un arco en el camino de u a v cruza este corte. Sea (x, y) dicho arco: observe que (x, y) no está en A porque $(S, V \setminus S)$ respeta a A. Como el camino simple entre u y v es único en T, remover (x, y) de E_T desconecta T en dos componentes. Estos dos componentes se pueden volver a unir agregando el arco (u, v). Es decir, $T' = (V, (E_T \setminus \{(x, y)\}) \cup \{(u, v)\})$ es un árbol de cubrimiento de G.

  El objetivo ahora es demostrar que el árbol de cubrimiento T' de G es de costo mínimo. Para ello, note lo siguiente:

$$\begin{aligned} w(T') &= (+e \mid e \in (E_T \setminus \{(x, y)\}) \cup \{(u, v)\} : w(e)) \\ &= (+e \mid e \in E_T : w(e)) + w(u, v) - w(x, y) \\ &\leq (+e \mid e \in E_T : w(e)) \\ &= w(T). \end{aligned}$$

  El paso de la desigualdad es posible porque (u, v) es ligero con respecto a $(S, V \setminus S)$. Por otra parte, se sabe que T es un MST de G. Luego $w(T) \leq w(T')$. De las dos desigualdades se concluye

$$w(T) = w(T').$$

  Es decir, T' es un MST de G. Como $A \cup \{(u, v)\}$ está contenido en los arcos de T', la propiedad deseada es cierta. Es decir, los arcos en $A \cup \{(u, v)\}$ hacen parte de un MST de G.

Finalmente, con ayuda del Teorema 4.2.1 se puede obtener la correctitud del Algoritmo 4.2.1. Esto quiere decir que, al concretar las nociones de corte y arco

ligero resulta un algoritmo para calcular el MST del grafo G con función de peso w.

Teorema 4.2.2: Optimización global

El Algoritmo 4.2.1 calcula un MST de G.

Demostración

Se propone como ejercicio al lector (ayuda: suponer que no es así y llegar a una contradicción con base en el Teorema 4.2.1).

Como el Algoritmo 4.2.1 no es lo suficientemente concreto, no es útil aún tratar de determinar sus complejidades temporal y espacial. Lo que debe ser claro es, de forma general, que la complejidad es polinomial si la operación para determinar si un arco es seguro toma tiempo polinimial.

Ejercicios

1. Sea $G = (V, E)$ un grafo no dirigido y conexo, y $w : E \to \mathbb{R}_{>0}$ una función de peso para los arcos. Demuestre que si $T = (V, E_T)$, con $E_T \subseteq E$, es un subgrafo conexo de G de costo mínimo (con respecto a w), entonces T es un árbol (de cubrimiento mínimo de G).
2. Sea T un árbol con N vértices. Demuestre que T tiene $N - 1$ arcos.
3. Demuestre que la noción de árbol en la Definición 4.2.1 es equivalente a la siguiente definición:

   Un *árbol* es un grafo en el cual existe exactamente un camino simple (i.e., sin repeticiones) entre cualquier par de vértices.
4. En la demostración del Teorema 4.2.1 se indica que se puede escoger un MST $T = (V, E_T)$ de G tal que $A \subseteq E_T$. Demuestre que dicha afirmación es cierta, i.e., que tal T árbol existe.
5. El el caso cuando $(u, v) \notin E_T$, en la demostración del Teorema 4.2.1, se hacen las siguientes afirmaciones:
   a) El grafo $(V, A \cup \{(e, v)\})$ no es un árbol.
   b) En T hay un camino único entre u y v.
6. Demuestre el Teorema 4.2.2.

Capítulo 5

Problemas y algoritmos de decisión

Los computadores son máquinas fascinantes: posiblemente sean uno de los inventos más disruptivos de la humanidad después del lenguaje y la escritura. Desde su invención, han evolucionado tremendamente en relación con su poder de cómputo, y de la diversidad de tareas que pueden hacer o en las cuales pueden asistir. A pesar de esta notoria evolución, el potencial de lo que pueden hacer (o no hacer) no ha cambiado: hay un límite que no podrán rebasar aún teniendo a disposición más memoria, procesadores más veloces y almacenamiento casi ilimitado. Este capítulo resume una historia acerca del significado de la palabra "computabilidad", de lo que significa computar y de los límites del poder de los computadores. Esta historia se remonta a las etapas más tempranas de las ciencias de la computación (circa 1900) en donde se plantearon preguntas de impacto profundo acerca de la existencia de *algoritmos* o *procedimientos computacionales efectivos* para resolver cierto tipo de problemas.

Este capítulo concluye que, aún en el contexto restringido de los problemas de decisión, hay tareas que no pueden ser resueltas por un algoritmo implantado en un computador, por potente que este último sea: a este tipo de problemas se les denomina *indecidibles*. Un problema de decisión es un problema computacional que, ante una entrada dada, responde afirmativa o negativamente. En el proceso de exhibir la existencia de problemas indecidibles, se usan los lenguajes formales como marco inicial para caracterizar cualquier problema de decisión, y se usan funciones matemáticas (parciales y totales) para definir las nociones de aceptación y decisión asociadas a estos lenguajes. Este preámbulo permite abordar el significado de la

computabilidad desde una perspectiva (casi) agnóstica de un modelo de computación específico. Un algoritmo es una función computable, tal y como se postula en la tesis de Church y Turing. La existencia de problemas indecidibles se basa en una demostración no constructiva que concluye que los conjuntos de problemas de decisión y de algoritmos de decisión es distinto, siendo menor el segundo. Este esfuerzo usa nociones y propiedades de cardinales infinitos.

5.1. Problemas de decisión

Un problema de decisión es un problema algorítmico para el cual el conjunto de posibles respuestas es 'sí' o 'no'. Generalmente, siguen un patrón de especificación.

Problema 5.1.1: Especificación típica de un problema de decisión

Entrada: Un conjunto $X \subseteq U$ y un elemento $x \in U$.

Salida: ¿Está x en X (i.e., $x \in X$)?

Se usa U para identificar un conjunto universo que sirve de referencia de X y que se supone puede ser inferido del contexto del problema. La pregunta de decisión consiste en determinar si un elemento de interés (en este caso x) hace parte del conjunto dado (en este caso X).

A modo de ejemplo, se pueden enunciar algunos problemas de decisión conocidos usando el formato de especificación típica del problema de decisión presentado en el Problema 5.1.1.

Ejemplo 5.1.1

Considere los problemas que se enuncian a continuación, identificando versiones concretas de X, U y x.

Clique: determinar si un grafo (no dirigido) tiene un subgrafo completo de un tamaño $k \in \mathbb{N}$ dado. En este caso, X representa el conjunto de grafos con un subgrafo completo de tamaño k, U el conjunto de grafos (finitos) y x el grafo dado.

Primalidad: determinar si un número es primo. En este caso, X representa el conjunto de números primos, U el conjunto de números naturales y x el número sobre el cual se desea averiguar primalidad.

SAT: determinar si una proposición es satisfacible. En este caso, X corresponde al conjunto de proposiciones satisfacibles, U al conjunto de proposiciones y x a una proposición.

2Partition: determinar si un conjunto de números naturales puede ser divido exactamente en dos subconjuntos tal que la suma de sus elementos coincida. En este caso, X corresponde al conjunto de conjuntos finitos de números naturales que pueden ser particionados en dos subconjuntos con igual suma, U el conjunto de todos los conjuntos finitos de números naturales y x el conjunto dado.

La importancia de los problemas de decisión radica en dos fenómenos fundamentales. Primero, los problemas de decisión son, a la vez, suficientemente fáciles de entender, y suficientemente complejos para plasmar la naturaleza de la mayoría de los problemas computacionales y abordar el estudio de la intratabilidad. Segundo, problemas de optimización y de conteo pueden ser reformulados y resueltos a partir de un problema de decisión y una de sus soluciones. La investigación en el área de teoría de complejidad se enfoca típicamente en problemas de decisión, la cual ha sido ampliamente desarrollada en las últimas décadas, permeando diferentes campos del conocimiento dentro y fuera de la ingeniería y las ciencias.

Ejemplo 5.1.2

Considere el Problema 3.6.1 del agente viajero: dado un grafo $G = (V, E)$ completo, con función $w : E \to \mathbb{R}_{\geq 0}$ de peso sobre los arcos, determinar el costo mínimo (con respecto a w) de un circuito en G.

Este problema de optimización cuenta con el siguiente problema de decisión asociado:

Entrada: un grafo $G = (V, E)$ completo, con función $w : E \to \mathbb{R}_{\geq 0}$ de peso sobre los arcos y un número real x.

Salida: ¿Existe en G un circuito cuyo costo sea a lo sumo x?

En la práctica, el problema de optimización original se puede resolver usando como caja negra (e.g., como una función que responde sí o no) un algoritmo que resuelva el problema de decisión asociado aplicando bisección. Note que para un error de precisión razonable (e.g., 10^{-9}), la cantidad de llamados que se hacen a la caja negra está acotada por un polinomio. Como conesecuencia, la dificultad computacional intrínseca de resolver el problema de optimización con base en una solución al problema de decisión no es mayor asintóticamente

a una que se pueda dar directamente al problema de optimización, aún si esta última es polinomial.

Ejercicios

1. Considere una función $\phi : [0..10^9] \to \{0, 1\}$ no decreciente y suponga que para cualquier $x \in [0..10^9]$, calcular $\phi(x)$ cuesta tiempo polinomial. Demuestre que un algoritmo de bisección sobre $[0..10^9]$ para determinar el mínimo valor x tal que $\phi(x) = 1$ toma tiempo polinimial.
2. Investigue acerca de la lista de 21 problemas NP-completos publicada por Richard Karp en el artículo "Reducibility Among Combinatorial Problems" de 1972. Escoja 3 de ellos, preferiblemente entre aquellos que no conozca, y especifíquelos como problemas de decisión siguiendo el formato propuesto en esta sección.
3. Considere el Problema 3.5.1 del morral. Siguiendo el desarrollo del Ejemplo 5.1.2, reformule el problema de optimización original con base en un problema de decisión asociado. Explique claramente cómo una solución de la versión reformulada sirve para resolver el problema de optimización original.
4. El problema de satsifacibilidad de una proposición en forma normal conjuntiva (o CNFSAT) se puede formular como un problema de optimización en función de la cantidad de cláusulas. Explique cómo hacer esta formulación.
5. Suponga que se desea determinar la longitud máxima de un ciclo simple (i.e., sin repetir vértices) en un grafo dirigido. Defina un problema de decisión asociado y resuelva el problema original suponiendo que hay una solución para el problema de decisión. Calcule la complejidad temporal de la solución planteada como función de la complejidad temporal del algoritmo de decisión.
6. Investigue acerca de las científicos de la computación Stephen Arthur Cook y Leonid Levin, y explique brevemente cuál fue su principal aporte al estudio de problemas intratables.

5.2. Un marco universal basado en lenguajes

Los conceptos de clases de lenguajes que se trabajan en este capítulo se definen en términos de *lenguajes formales* y *máquinas de Turing*. Dado que el requisito de tiempo polinomial es suficientemente general, el tratamiento en este capítulo no ahonda en los detalles de las máquinas de Turing. En realidad, cualquier algoritmo ejecutable por una máquina de acceso aleatorio (en inglés, *random access machine*)

puede ser simulado por una máquina de Turing de una sola cinta y una sola cabeza con un factor polinimial de costo de cómputo. Por ello, esta sección deja a un lado los detalles de los modelos de computación y se centra en los lenguajes formales.

Definición 5.2.1

Un *alfabeto* es un conjunto finito de símbolos. Un *lenguaje (formal)* L sobre un alfabeto Σ es un conjunto de cadenas formadas con los símbolos de Σ. Se usan los símbolos λ para denotar la *cadena vacía*, $\emptyset$ el *lenguaje vacío* y Σ^* la *colección de todas las cadenas sobre* Σ.

De acuerdo con las convenciones introducidas en la Definición 5.2.1, cualquier lenguaje L sobre un alfabeto Σ es en realidad un subconjunto de Σ^*, i.e., $L \subseteq \Sigma^*$. Una ventaja de los lenguajes es que sobre ellos se pueden definir fácilmente operaciones de interés.

Definición 5.2.2

Sea Σ un alfabeto:

- El *complemento* $\overline{L}$ de un lenguaje $L \subseteq \Sigma^*$ es el conjunto
$$\overline{L} = \Sigma^* \setminus L.$$
- La *concatenación* L_1L_2 de dos lenguajes $L_1, L_2 \subseteq \Sigma^*$ es el conjunto
$$L_1L_2 = \{s_1s_2 \mid s_1 \in L_1 \wedge s_2 \in L_2\}.$$
- La *concatenación generalizada* L^k, con $k \geq 0$, de un lenguaje $L \subseteq \Sigma^*$ es el conjunto definido inductivamente, para cualquier $n \in \mathbb{N}$, de la siguiente manera:
$$L^0 = \{\lambda\}$$
$$L^{n+1} = L^nL.$$

Las operaciones de *unión* (i.e., $\_ \cup \_$), *intersección* (i.e., $\_ \cap \_$) y *potencia* (i.e., $\mathcal{P}(\_)$) son las usuales.

El complemento $\overline{L}$ de un lenguaje L es la colección de las cadenas que no están en L. La concatenación L_1L_2 es el conjunto de cadenas que pueden ser partidas en exactamente dos partes: la primera parte está en L_1 y la segunda en L_2. La concatenación generalizada L^k es el conjunto de cadenas que se construyen a partir de la concatenación de k cadenas en L. De acuerdo con la definición de conjunto potencia heredada de la teoría de conjuntos, las expresiones $L \subseteq \Sigma^*$ y $L \in \mathcal{P}(\Sigma^*)$ son equivalentes.

Ejemplo 5.2.1

Considere cada una de las siguientes situaciones.

- Si $\Sigma = \{0, 1, \ldots 9\}$ (i.e., el conjunto de los dígitos en base 10), entonces

  $$L_1 = \{w \mid w \in \Sigma^* \wedge w \neq \lambda\}$$

  es el cojunto de los números naturales (algunos repetidos) en representación decimal y

  $$L_2 = \{w \mid w \in L_1 \wedge \text{“}w \text{ termina en } 0\text{”}\}$$

  es el conjunto de números naturales múltiplos de 10 en representación decimal.
- Si Σ es el alfabeto, entonces

  $$L_1 = \{w \mid w \in \Sigma^* \wedge w \neq \lambda \wedge \text{“}w \text{ está en la base de datos de la RAE”}\}$$

  es la colección de palabras del Castellano y

  $$L_2 = \overline{L_1}$$

  la colección de palabras que no son castizas.

Nota 5.2.1

La inclinación por usar Σ^* para denotar el conjunto de todas las cadenas sobre Σ no es caprichosa. Obedece al hecho de que este conjunto corresponde a la clausura de Kleene de Σ, escrita Σ^*, y definida como:

$$\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n,$$

es decir, el conjunto de todas las cadenas finitas sobre el alfabeto Σ.

El alfabeto de interés en el desarrollo de esta sección y este capítulo, sin pérdida de generalidad, es el binario (i.e., $\Sigma = \{0, 1\}$). Con ello se supone la existencia de una función inyectiva que identifica cualquier elemento o tupla de elementos (de un universo contable) con una secuencia binaria en $\{0, 1\}^*$.

Nota 5.2.2

Se usa la notación $\langle \_ \rangle$ para denotar una *función de codificación* con rango en Σ^*. Por ejemplo, $\langle 5 \rangle$ identifica el código binario que representa a 5, mientras que $\langle$"abracadabra"$\rangle$ el código binario que representa la cadena "abracadabra". Esta notación se puede usar sobre otros tipos de estructuras como grafos, matrices, etc.

Es importante anotar que existen varias funciones de codificación. Sin embargo, hacer distinciones no es relevante para el tratamiento que sigue. Además, note que elegir una función de codificación en particular es coherente con la situación que se tiene en los computadores que procesan y calculan volúmenes brutales de información representada en cadenas de bits: cada sistema de archivos puede usar un formato distinto. Tampoco es indispensable que los códigos sean binarios; por ejemplo, pueden ser ternarios o usar cualquier otro alfabeto que garantice que cualquier conversión de códigos binarios al alfabeto de elección esté acotada polinomialmente. A las codificaciones que respetan esta restricción se les denominan *codificaciones razonables*. Por ejemplo, el alfabeto $\{1\}$ (i.e., el unario) no es razonable; ¿por qué?

Finalmente, una bondad de usar lenguajes formales como marco general de representación de problemas computacionales tiene que ver con la facilidad para formular problemas de decisión. Note que, de acuerdo con la especificación típica de un problema de decisión en la Sección 5.1, un problema de decisión es entonces una pregunta sobre la pertentencia o no de una cadena binaria a un lenguaje.

Problema 5.2.1: Especificación de un problema de decisión con lenguajes

Entrada: Un conjunto $L \subseteq \{0,1\}^*$ y un elemento $x \in \{0,1\}^*$.

Salida: ¿$x \in L$?

Gracias al uso de lenguajes formales, y a la transparencia de contar con las máquinas de Turing como modelo de computación abstracto, en las próximas secciones es posible abordar las definiciones y el estudio de las clases P y NP, los conceptos de decisión, aceptación y completitud, y la técnica de reducción polinomial con algo más que un poco de teoría de conjuntos básica.

Ejercicios

1. Investigue acerca de máquinas de acceso aleatorio (en inglés, *random access machine*) y explique su relación con los computadores modernos.
2. Investigue acerca de máquinas de Turing. Elabore una justificación de la siguiente afirmación: cualquier algoritmo ejecutable por una máquina de acceso aleatorio puede ser simulado por una máquina de Turing de una sola cinta y una sola cabeza con un factor polinimial de costo de cómputo.
3. Explique con ejemplos y justifique por qué el alfabeto unario no es razonable.

5.3. Aceptación y decisión

El objetivo de esta sección es presentar y estudiar las nociones de aceptación y decisión asociadas a los lenguajes formales. Esto se hace con base en funciones parciales y totales, respectivamente, con dominio en las secuencias binarias $\{0,1\}^*$ y rango en el alfabeto binario $\{0,1\}$. En esta sección se fija el alfabeto binario $\Sigma = \{0,1\}$ y, mientras no se precise, una función puede ser total o no (i.e., parcial; ¿por qué?).

Haber adoptado el marco de los lenguajes formales permite expresar concisa y consistentemente la relación entre problemas de decisión y la forma en que estos se pueden resolver.

Definición 5.3.1: Aceptación y rechazo

Sea $A : \Sigma^* \to \Sigma$ una función. Para $x \in \Sigma^*$, se dice que:

- A *acepta* x sii $A(x) = 1$ y
- A *rechaza* x sii $A(x) = 0$.

El *lenguaje aceptado* por A, denotado $L(A)$, es el conjunto:

$$L(A) = \{x \mid x \in \Sigma^* \wedge A(x) = 1\}.$$

En el marco de los lenguajes formales, la inclinación es por representar una solución a un problema de decisión como una función que, dada una palabra binaria, calcula un valor binario. Por convención, aceptar una cadena se indica con el valor 'Booleano' 1 y rechazarla con el 0, similar a lo que se tiene en la lógica Booleana implementada en los computadores de hoy en día. Dada una función A, se asocia a esta el lenguaje $L(A)$ que corresponde exactamente a las cadenas aceptadas por A. Evitar que las funciones sean necesariamente totales es clave, como se verá más

adelante, para caracterizar las nociones de aceptación y decisión asociadas a los problemas de decisión (i.e., a lenguajes binarios).

Ejemplo 5.3.1

Considere las siguientes afirmaciones para una función $A : \Sigma^* \to \{0, 1\}$:

- Si $A(x) = 1$ para cualquier $x \in \Sigma^*$, entonces $L(A) = \Sigma^*$. Hacia una contradicción, suponga lo contrario: es decir, que $L(A) \neq \Sigma^*$. Entonces, necesariamente $L(A) \subsetneq \Sigma^*$. En consecuencia, hay una cadena $s \in \Sigma^* \setminus L(A)$. Por definición, $A(s) \neq 1$ dado que s no está en $L(A)$ (¿por qué no se puede afirmar que $A(s) = 0$?). Pero esto es una contradicción dado que se ha supuesto que $A(x) = 1$ para cualquier $x \in \Sigma^*$; en particular, para s. Luego, necesariamente $L(A) = \Sigma^*$.
- Si $A(x) = 0$ para cualquier $x \in \Sigma^*$, entonces $L(A) = \emptyset$. La justificación se propone como ejercicio al lector.

En la justificación de la primera parte del Ejemplo 5.3.1, se concluye que $A(s) \neq 1$ cuando $s \notin L(A)$, pero no que $A(s) = 0$. Este es un aspecto aparentemente irrelevante, pero es en realidad extremadamente importante porque no toda función parcial es total. Note que, en general, es posible que $A(s)$ esté indefinido, es decir, que s sea uno de los puntos de indeterminación de A (i.e., para el cual A no tiene asociado ni 0 ni 1). Por eso, sin información que garantice la totalidad de la función A, puede ser un error deducir $A(s) = 0$ cuando $A(s) \neq 1$.

Es usual diseñar un algoritmo con base en otro existente. Por ejemplo, construir nuevos procedimientos en un lenguaje de programación a partir de otros procedimientos existentes en una librería. Uno de estos casos corresponde al de funciones que invierten la salida de otra función.

Definición 5.3.2

Sea $A : \Sigma^* \to \{0, 1\}$ una función. La función *complementaria* de A, denotada $\overline{A}$, se define para cualquier $x \in \Sigma^*$ como:

- $\overline{A}(x) = 1$ si $A(x) = 0$,
- $\overline{A}(x) = 0$ si $A(x) = 1$ y
- $\overline{A}(x)$ es indeterminada si $A(x)$ es indeterminada.

Cuando A en la Definición 5.3.2 es total, coinciden el complemento del lenguaje determinado por A y el lenguaje determinado por su función complementaria $\overline{A}$.

Teorema 5.3.1

Si $A : \Sigma^* \to \{0,1\}$ es una función total, entonces:

$$L(A) = \overline{L(\overline{A})}.$$

Demostración

Sea $x \in \Sigma^*$. El objetivo es demostrar:

$$x \in L(A) \iff x \in \overline{L(\overline{A})}.$$

Se procede por doble implicación.

($\Longrightarrow$): Se supone $x \in L(A)$ con el objetivo de demostrar que $x \in \overline{L(\overline{A})}$ o, equivalentemente, que $x \notin L(\overline{A})$. Como $x \in L(A)$, se tiene que $A(x) = 1$. Por definición de función complementaria, se tiene que $\overline{A}(x) = 0$. Es decir, $x \notin L(\overline{A})$.

($\Longleftarrow$): Se supone $x \in \overline{L(\overline{A})}$ con el objetivo de demostrar que $x \in L(A)$. Como $x \in \overline{L(\overline{A})}$, entonces $x \notin L(\overline{A})$. O bien $\overline{A}(x) = 0$ o $\overline{A}(x)$ es indeterminado. Por la definición de función complementaria y la suposición de que A es total, no se puede dar que $\overline{A}(x)$ sea indeterminado. Es decir, necesariamente $\overline{A}(x) = 0$. Por definición de $\overline{A}$, se tiene que $A(x) = 1$, como se deseaba.

En conclusión, $L(A) = \overline{L(\overline{A})}$.

El hecho de que una función A determine el lenguaje $L(A)$, no permite concluir que necesariamente A rechaza cualquier cadena $x \notin L(A)$. En la práctica, esto puede suceder cuando un ciclo no termina o cuando hay un error en la ejecución de un programa. Con este preámbulo, y lo desarrollado anteriormente, se motiva la introducción de las nociones de aceptación y decisión asociadas a los lenguajes.

Definición 5.3.3: Aceptación y decisión de un lenguaje

Sea L un lenguaje y $A : \Sigma^* \to \{0,1\}$ una función. Se dice que:

- A *acepta* L sii para cualquier $x \in L$ se tiene $A(x) = 1$.
- A *decide* L sii para cualquier $x \in L$ se tiene $A(x) = 1$ y para cualquier $x \notin L$ se tiene $A(x) = 0$.

La Definición 5.3.3 invita a varias observaciones importantes. Primero, toda función de decisión es una de aceptación, pero no necesariamente al contrario. Segundo, si una función A acepta L, independientemente de si A es solo de aceptación o si es de decisión, $L = L(A)$. Tercero, un lenguaje puede ser aceptado por más de una función pero decidido solo por una única función (¿por qué?). Finalmente, cualquier función total necesariamente es una función de decisión; el converso también es cierto, es decir, cualquier función de decisión es total. Esta última observación, en el marco de los lenguajes formales, es especialmente importante porque asocia unívocamente la noción de decisión de un lenguaje a la de totalidad de cierto tipo de funciones sobre lenguajes binarios.

Teorema 5.3.2

Si $A : \Sigma^* \to \{0, 1\}$ una función, entonces:

$$A \text{ es total} \iff A \text{ decide } L(A).$$

Demostración

Se propone como ejercicio al lector.

Ejercicios

1. Proponga condiciones suficientes para que la siguiente igualdad sea cierta para $A : \Sigma^* \to \{0, 1\}$:

   $$\overline{L(A)} = \{x \mid x \in \Sigma^* \wedge L(A) = 0\}.$$

2. Justifique formalmente la segunda afirmación en el Ejemplo 5.3.1.
3. Sea $A : \Sigma^* \to \{0, 1\}$. Demuestre o refute:
   a) $L(A) \cup \overline{L(A)} = \Sigma^*$.
   b) $L(A) \cup L(\overline{A}) = \Sigma^*$.
4. Presente los detalles de la siguiente afirmación, en el marco de la demostración del Teorema 5.3.1:

   > Por la definición de función complementaria y la suposición de que A es total, no se puede dar que $\overline{A}(x)$ sea indeterminado.

   Sugerencia: procer por contradicción.
5. Formule un problema aceptado exactamente por una única función de aceptación. Es necesario demostrar que dicha función es única.

6. Justifique brevemente por qué toda función de decisión es de aceptación, pero no viceversa.
7. Justifique brevemente por qué si una función A acepta L, independientemente de si A es de decisión o no, $L = L(A)$.
8. Defina un lenguaje L, y dos funciones distintas A_1 y A_2 tales que $L = L(A_1) = L(A_2)$.
9. Demuestre el Teorema 5.3.2.

5.4. Computabilidad

En la práctica, un problema computacional L puede ser resuelto cuando existe un programa o procedimiento que macánicamente calcule los valores correspondientes a una función A tal que $L = L(A)$. A este tipo de funciones se les denomina *efectivamente computables* o *computables*. La noción de computabilidad no se refiere a eficiencia (en tiempo o espacio), sino más bien a una posibilidad de que exista cierto programa de computador (eficiente o no, en tiempo o espacio) que calcule los valores de una función dada: es una propiedad acerca de las funciones de Σ^* en $\Sigma = \{0, 1\}$. Esta sección presenta una definición de función computable que corresponde a (y se usará como) la noción de algoritmo. Estudia también algunas de sus propiedades, y enuncia la tesis de Church y Turing acerca de lo que significa computar en el marco de los lenguajes formales.

Definición 5.4.1: Funciones computables

Sea $A : \Sigma^* \to \Sigma$. Se dice que A es *computable* si y solo si existe un programa de computador que cumple las siguientes dos propiedades, para cualquier $x \in \Sigma^*$:

- Si $A(x)$ está definido, entonces el programa termina cuando se invoca con x y retorna el valor de $A(x)$.
- Si $A(x)$ está indefinido, entonces el programa no termina (nunca) cuando se invoca con x y no retorna ningún valor.

Las funciones computables son el análogo formal de la noción intuitiva de algoritmo: una función es computable si existe un algoritmo que permita calcular sus valores. Además, las funciones computables son los objetos de estudio básico en el área denominada teoría de la computabilidad, especialmente porque permiten elaborar en la noción de computabilidad sin necesidad de referirse directamente a un modelo de computación. En realidad, en todos los modelos de computación aceptados actualmente, las correspondientes definiciones de computabilidad coinciden en identificar las mismas colecciones de funciones.

Por su parte, la noción de función computable puede ser más general en cuanto a considerar más parámetros y otro tipo de dominios como, por ejemplo, los números. Sin embargo, para el propósito que anima esta sección, y el capítulo en general, es suficiente enfocarse exclusivamente en funciones con dominio y rango en Σ^* y Σ, respectivamente, sin que los resultados asociados pierdan generalidad.

Como observación final de la noción de computabilidad, note que en la Definición 5.4.1 no se menciona explícitamente lenguaje de programación alguno. Se entiende que este tipo de detalle aporta poco o nada a la noción de computabilidad. Lo que sí es relevante (y clave) es el hecho de aceptar como acuerdo que cualquier algoritmo se define a partir de una cantidad finita de símbolos (en un lenguaje dado) que describen una secuencia de pasos que, por ejemplo, una persona con suficiente tiempo y recursos (como papel y lápiz) puede seguir.

Con ayuda de las funciones computables, la noción de computabilidad puede ser extendida a problemas de decisión.

Definición 5.4.2: Lenguajes computablemente enumerables y computables

Sea $L \subseteq \Sigma^*$. Se dice que:

- L es *computablemente enumerable* si y solo si existe una función computable $A : \Sigma^* \to \Sigma$ tal que $L(A) = L$.
- L es *computable* si y solo si existe una función total computable $A : \Sigma^* \to \Sigma$ tal que $L(A) = L$.

Las lenguajes computablemente enumerables y computables coinciden con las nociones de aceptación y decisión en la Definición 5.3.3, respectivamente, cuando las funciones parciales y totales que los caracterizan son computables. De esta forma, un problema de decisión L es computablemente enumerable cuando existe un algoritmo A que retorna 1 al ser invocado con cualquier instancia $x \in L$. Además, si dicho algoritmo retorna 0 al ser invocado con cualquier instancia $x \notin L$, entonces L es computable. Note que, por la definición, cualquier lenguaje computable es computablemente enumerable. Sin embargo, el converso no es necesariamente cierto.

Ejemplo 5.4.1

Considere los siguientes ejemplos:

- Determinar si un número entero hace parte de un arreglo de números enteros. Considere el lenguaje

$$\textsc{ArrFind} = \{\langle A[0..N), x\rangle \mid \text{“}A \text{ es un arreglo y } x \text{ está en } A\text{”}\}.$$

El siguiente algoritmo permite afirmar que ArrFind es computable. Corresponde a explorar todas las posiciones del arreglo dado para determinar si el número dado está presente o no. Esto se admite porque la igualdad de dos números enteros es computable (¿por qué?).

- Determinar si una fórmula proposicional es una tautología. Considere el lenguaje

  $$\textsc{Taut} = \{\langle\varphi\rangle \mid \text{“}\varphi \text{ es una proposición tautológica”}\}.$$

  El siguiente algoritmo permite afirmar que Taut es computable. Corresponde a generar la tabla de verdad de φ y si todos sus renglones resultan en verdadero, entonces la proposición es una tautología; de lo contrario, no lo es.
- Determinar si una fórmula de primer orden es un teorema. Considere el lenguaje

  $$\text{FOL}\textsc{Thm} = \{\langle\varphi\rangle \mid \text{“}\varphi \text{ es un teorema en lógica de primer orden}\}.$$

  El siguiente algoritmo permite afirmar que FOLThm es computablemente enumerable. Generar incrementalmente, en un orden arbitrario, las demostraciones en lógica de primer orden. Si aparece la fórmula dada como conclusión de una demostración, entonces dicha fórmula es un teorema de esta lógica. Note que este algoritmo no se detiene si dicha fórmula no es la conclusión de ninguna demostración.
- El problema de corresponcia de Post (ver Ejercicio 8) no es computablemente enumerable. En consecuencia, tampoco es computable.

Nota 5.4.1

Más allá del contexto de los lenguajes formales, los problemas computablemente enumerables o computables reciben los nombres de *semidecidibles* o *decidibles*, respectivamente. Por ejemplo, es un problema semidecidible determinar si una fórmula de primer orden es teorema, mientras que es un problema decidible determinar si una fórmula proposicional es tautología.

Nota 5.4.2

Como lo evidencia el problema de correspondencia de Post en el Ejemplo 5.4.1, hay problemas de decisión que no son ni siquiera semidecidibles (i.e., computablemente enumerables). A este tipo de problemas se les denomina *indecidibles*. La Sección 5.5 presenta un argumento (no constructivo) de la existencia de problemas indecidibles.

Volviendo al dominio de las funciones computables, se enuncian algunas de las operaciones bajo las cuales son cerradas (i.e., bajo las cuales la computabilidad se mantiene).

Teorema 5.4.1

Las siguientes afirmaciones son ciertas acerca de las funciones computables:

1. Son cerradas bajo suma binaria (punto a punto módulo 2).
2. Son cerradas bajo multiplicación (punto a punto).
3. Son cerradas bajo composición (cuando el dominio de, e.g. f, está restringido a Σ en $f \circ g$).
4. Son cerradas bajo mínimo y máximo (punto a punto).

Demostración

Se proponen como ejercicio al lector.

Finalmente, se enuncia la tesis Alonzo Church y Alan Turing acerca de lo que significa computar.

Nota 5.4.3: La tesis de Church y Turing

La tesis de Church y Turing afirma que las funciones computables son exactamente aquellas funciones que pueden ser calculadas mecánicamente con cantidades ilimitadas de tiempo y espacio.

Esta tesis es uno de varios intentos por formalizar la noción de computabilidad. Afirma que las funciones computables son aquellas para las cuales existe un algoritmo (y viceversa). Su caracter no es completamente formal o matemático dado que la noción de computabilidad asociado a las funciones es informal. Por esto, no es un postulado que se pueda demostrar.

Ejercicios

1. Defina brevemente en qué consiste cada uno de los siguientes modelos de computación y enuncie la definición formal de computación correspondiente:
   a) Máquinas de Turing.
   b) Cálculo λ (de Alonzo Church).
   c) Funciones μ-recursivas (de Kurg Gödel).
   d) Máquinas abstractas (de Emil Post).
   e) Algoritmos de Markov.

2. Demuestre que si $A : \Sigma^* \to \Sigma$ tiene dominio finito, entonces A es computable. Concluya que cualquier $L \subseteq \Sigma^*$ cuyo complemento es finito también es computable.

3. Demuestre, exhibiendo un algoritmo como testigo, que:
   a) Una función $A : \Sigma^* \to \Sigma$ constante es computable.
   b) El conjunto vacío es computable.
   c) El conjunto de números naturales es computable.
   d) El conjunto de números primos es computable.

4. En el Ejemplo 5.4.1 se afirma que el problema ArrFind es computable. Explique por qué este problema cuando se extiende a números reales deja de ser computable. Ayuda: fijarse en la computabilidad del problema de igualdad de dos números reales.

5. Demuestre el Teorema 5.4.1.

6. Sean $L_1, L_2 \subseteq \Sigma^*$ computables. Demuestre que:
   a) $L_1 \cup L_2$ es computable.
   b) $L_1 \cap L_2$ es computable.

7. Sea $A : \Sigma^* \to \Sigma$. Demuestre que A es computable si y solo si A y su complemento son computablemente enumerables.

8. Considere el problema de correspondencia de Post mencionado en el Ejemplo 5.4.1.
   a) Defina claramente en qué consiste este problema e ilustre la definición con un ejemplo.
   b) Investigue acerca de una demostración de indecidibilidad de este problema y explíque sus principales pasos.

9. Investigue acerca del décimo problema de Hilbert. Explique en qué consiste y cuál es su estado actual.

5.5. Indecidibilidad

El principal resultado de la sección es negativo en el sentido en el cual se demuestra que existen funciones que no son computables. Muestra que, en el contexto de lo que significa computar hoy en día, hay funciones que no se pueden mecanizar en un computador. El principal argumento en la demostración es la existencia de una cantidad tan grande de problemas de decisión, en el sentido de la Sección 5.2, que no hay suficientes programas de computador que los puedan resolver mecánicamente. Esto es equivalente a decir que no toda función con dominio en el conjunto de cadenas binarias Σ^* y rango en el conjunto binario Σ puede ser mecanizada en un computador (ver Teorema 5.3.2).

Una advertencia amigable acerca del desarrollo de la sección antes de entrar en materia. Se usarán conceptos y resultados de la teoría de conjuntos como, por ejemplo, cardinales y aritmética cardinal que generalmente no son estudiados en currículos de pregrado de ingeniería y computación. Por ello, la limitante potencial en esta sección es que algunos de los resultados empleados no se demuestran porque las definiciones o técnicas necesarias están fuera del alcance del texto; en consecuencia, esta sección no es autocontenida. Al final del capítulo se presentan referencias para profundizar en estos temas y conocer las demostraciones ausentes, en caso tal de que el lector quiera profundizar en ellos.

5.5.1. Conceptos básicos de cardinales. El plan general para lograr el objetivo de la sección es simple: comparar los tamaños del conjunto de lenguajes y del conjunto de funciones computables para determinar que el primero es mucho más grande que el segundo.

Para razonar sobre el tamaño de conjuntos infinitos es necesario desligar la noción de tamaño de la acción de contar uno a uno, tal y como sucede generalmente cuando los conjuntos son finitos. Por ejemplo, ¿se podrían contar, uno a uno, los números naturales? La respuesta es no: este conjunto no tiene una cota superior y, en consecuencia, cualquier intento de listarlos explícitamente nunca terminaría. En contraste, si se toma cualquier subconjunto finito de los números naturales, por grande que este sea, un proceso de conteo de uno tras otro seguro que terminará (posiblemente en varias generaciones de una familia si el subconjunto es muy grande, pero terminará).

Para referirse al tamaño de conjuntos infinitos, es necesario introducir la noción de cardinal y algunas propiedades asociadas a ella.

Definición 5.5.1: Número cardinal

El *cardinal* de un conjunto X se denota como $\text{card}(X)$. Sean X_1 y X_2 dos conjuntos. Entonces:

- $\text{card}(X_1) = \text{card}(X_2)$ si y solo si existe un función biyectiva entre X_1 y X_2. En este caso, se dice que X_1 y X_2 son *equipotentes*, escrito $X_1 \sim X_2$.
- $\text{card}(X_1) \leq \text{card}(X_2)$ si y solo si existe una función inyectiva de X_1 a X_2.

Además, si X_1 y X_2 son disyuntos, se tiene

$$\text{card}(X_1 \cup X_2) = \text{card}(X_1) + \text{card}(X_2).$$

La función $\text{card}(\_)$ asocia cualquier conjunto a un *ordinal*; para efectos prácticos, un *cardinal* es un tipo especial de ordinal que permite representar unívocamente el tamaño de un conjunto infinito. Intuitivamente, los ordinales son conjuntos que generalizan la noción de número y que coinciden con los números naturales cuando los objetos de estudio son los conjuntos finitos. De acuerdo con la Definición 5.5.1, para determinar la iguadad de los cardinales (o, equivalentemente, la equipotencia de dos conjuntos) es necesario exhibir una biyección entre los conjuntos en cuestión. Además, se extiende el orden total $\leq$ sobre los números naturales a los cardinales infinitos. Al igual que con conjuntos finitos, el tamaño de una unión disyunta de dos conjuntos es igual a la suma de los tamaños de las dos partes.

Ejemplo 5.5.1

El conjunto de los números naturales y el de los números naturales pares

$$2\mathbb{N} = \{2m \mid m \in \mathbb{N}\}$$

tienen el mismo tamaño (i.e., $\mathbb{N} \sim 2\mathbb{N}$). Esto es cierto porque la función determinada por el mapa

$$n \mapsto 2n$$

es biyectiva. Por una parte, es claramente inyectiva: si $2n_1 = 2n_2$, necesariamente $n_1 = n_2$. Para observar que es sobreyectiva, note que cualquier número en $2\mathbb{N}$ tiene la forma $2n$, cuya preimagen es $n \in \mathbb{N}$. En conclusión, los dos conjuntos tienen el mismo cardinal y son equipotentes.

Note que dado que hay una biyección entre $\mathbb{N}$ y $2\mathbb{N}$, también se tiene que $\text{card}(\mathbb{N}) \leq \text{card}(2\mathbb{N})$ y $\text{card}(2\mathbb{N}) \leq \text{card}(\mathbb{N})$.

El Ejemplo 5.5.1 exhibe una propiedad posiblemente sorprendente de los conjuntos infinitos: aún cuando $2\mathbb{N}$ es un conjunto propio de $\mathbb{N}$, los dos conjuntos tienen la misma cantidad de elementos. Con conjuntos finitos esta situación es imposible porque cualquier conjunto propio de un conjunto finito necesariamente tiene menos elementos. Por ejemplo, el axioma "el todo es mayor que cualquiera de sus partes" de Euclides es inválido para conjuntos infinitos.

Nota 5.5.1

El matemático Richard Dedekind usó la siguiente propiedad para caracterizar conjuntos finitos e infinitos:

> Si no hay un conjunto propio de X equipotente a X, entonces X es *finito*. Si hay un conjunto propio de X equipotente a X, entonces X es *inifito*.

Nota 5.5.2

Usualmente, los cardinales infinitos se identifican con los ordinales:

$$\aleph_0, \aleph_1, \aleph_2, \ldots$$

El símbolo $\aleph$ fue introducido por el matemático Georg Cantor para representar el tamaño de conjuntos infinitos y proviene del alfabeto hebreo; este símbolo se lee "alef" en Castellano.

Los conjuntos infinitos usualmente se clasifican en dos tipos: aquellos que tienen el mismo tamaño de los números naturales y los demás. A los primeros se les denomina contables, mientras que a los segundos no contables. Históricamente, la distinción entre conjuntos contables y no contables ha permitido entender diferencias profundas entre conjuntos de interés como lo son, por ejemplo, los números racionales y los reales que hacen parte de categorías distintas en relación con el tamaño de los números naturales. El argumento que se desarrolla a lo largo de esta sección es un ejemplo más del uso de esta caracterización, como se terminará de ver en los próximos apartes.

Definición 5.5.2

Sea X un conjunto. Se dice que:

- X es *contable* si y solo si X es finito o $X \sim \mathbb{N}$;
- X es *no contable* de lo contrario.

Se adopta $\text{card}(\mathbb{N}) = \aleph_0$ por convención.

Como consecuencia de la Definición 5.5.2, $\mathbb{N}$ es contable, al igual que cualquiera de sus subconjuntos. Por eso, abusando de la convención, se dice que $\aleph_0$ es el único cardinal infinito contable, además de ser el primer cardinal infinito.

5.5.2. Aritmética cardinal. Los cardinales, como números que son, se pueden operar entre sí. El Teorema 5.5.1 introduce una propiedad de la suma de cardinales que será útil posteriormente.

Teorema 5.5.1: Suma de cardinales infinitos

Sean X_1 y X_2 conjuntos, con X_1 infinito. Si $\text{card}(X_2) \leq \text{card}(X_1)$, entonces $\text{card}(X_1) + \text{card}(X_2) = \text{card}(X_1)$.

La suma de cardinales infinitos puede ser intuitiva si es interpretada, por ejemplo, con ayuda de la unión de conjuntos. Si un conjunto inifito se duplica, el cardinal del conjunto resultante es el mismo. De manera más general, si un conjunto infinito X se extiende con un conjunto que no sobrepasa su cantidad de elementos, el tamaño del conjunto resultante es el mismo de X, como se afirma en el Teorema 5.5.1. A nivel formal, la demostración del Teorema 5.5.1 está fuera del alcance de este texto (se propone como ejercicio al lector interesado en profundizar en el tema o con conocimiento previo acerca de cardinales).

Hay una forma mecánica de construir un conjunto infinito de mayor tamaño que cualquier conjunto infinito dado. Esta corresponde a la operación de construcción del conjunto potencia denotado en este texto como $\mathcal{P}(\_)$; este operador denota el conjunto de todos los subconjuntos de un conjunto dado. El Teorema 5.5.2 presenta resultados asociados al conjunto potencia de un conjunto, finito o infinito.

Teorema 5.5.2: Cardinal del conjunto potencia

Para cualquier conjunto X se tiene:

1. $\text{card}(\mathcal{P}(X)) = 2^{\text{card}(X)}$.
2. $\text{card}(X) < \text{card}(\mathcal{P}(X))$.

Al igual que sucede con la demostración del Teorema 5.5.1, la demostración del Teorema 5.5.2 está fuera del alcance de este texto y se propone como ejercicio a lectores interesados en el tema. Intuitivamente, el tamaño del conjunto partes de un conjunto infinito cumple la misma ley de exponenciación para conjuntos finitos. Además, la cantidad de subconjuntos de un conjunto dado X, finito o infinito, es estrictamente mayor que la cantidad de elementos de X.

5.5.3. ¿Cuántos lenguajes hay? Como cada lenguaje identifica un problema algorítmico de decisión, determinar el tamaño del conjunto de lenguajes también permite concluir cuántos problemas de decisión hay. Se establece un resultado clave en el camino hacia este objetivo intermedio al determinar el tamaño de $\{0,1\}^*$.

Teorema 5.5.3: $\{0,1\}^*$ es contable

El tamaño de Σ^* es el infinito contable, i.e., $\Sigma^* \sim \mathbb{N}$.

Demostración

Considere los siguientes conjuntos:

$$X_0 = \{w \mid w \in \Sigma^* \wedge \text{“}w \text{ inicia con } 0\text{”}\}$$
$$X_1 = \{w \mid w \in \Sigma^* \wedge \text{“}w \text{ inicia con } 1\text{”}\}.$$

Note que X_0 y X_1 son disyuntos, y junto con $\{\lambda\}$ forman una partición de Σ^*.

Observación 1: $X_0 \sim X_1$. Considere el siguiente mapa para cualquier $x \in \Sigma^*$:

$x \mapsto$ “inversión, punto a punto, de los bits de x”.

Por ejemplo, este mapa convierte la cadena 0001101 en 1110010. Demostrar que este mapa induce una función biyectiva es fácil y se propone como ejercicio al lector.

Observación 2: $X_1 \sim \mathbb{N}$. Considere el siguiente mapa para cualquier $x \in \Sigma^*$:

$x \mapsto$ “representación en base 10 de x” $- 1$.

Por ejemplo, este mapa convierte la cadena 110 en 5 y la cadena 1 en 0. Demostrar que este mapa induce una función biyectiva entre X_1 y $\mathbb{N}$ es fácil, y se propone como ejercicio al lector.

Con base en estas dos observaciones, se tiene que:

$$\begin{aligned}
\text{card}(\Sigma^*) &= \text{card}(X_0 \cup X_1 \cup \{\lambda\}) && \text{(partición de } \Sigma^*) \\
&= \text{card}(X_0) + \text{card}(X_1) + \text{card}(\{\lambda\}) && (X_0, X_1, \{\lambda\} \text{ son disyuntos}) \\
&= \text{card}(X_0) + \text{card}(X_1) + 1 && \text{(definición de card)} \\
&= \text{card}(X_0) + \text{card}(X_1) && \text{(Teorema 5.5.1)} \\
&= \text{card}(X_1) + \text{card}(X_1) && \text{(Observación 1)} \\
&= \text{card}(X_1) && \text{(Teorema 5.5.1)} \\
&= \text{card}(\mathbb{N}) && \text{(Observación 2).}
\end{aligned}$$

En conclusión, $\Sigma^* \sim \mathbb{N}$.

El Teorema 5.5.3 establece que no hay más secuencias binarias que números naturales, lo cual puede parecer extraño. Esta observación es consecuencia del resultado anterior. Es más, aún si el tamaño del alfabeto Σ es el inifito contable, la veracidad de este teorema también se mantiene.

Dado que un problema algorítmico es subconjunto de Σ^*, para razonar acerca de la cantidad de lenguajes es necesario razonar acerca del tamaño del conjunto $\mathcal{P}(\Sigma^*)$, es decir, del conjunto partes de Σ^*. Para ello se usan los resultados del Teorema 5.5.2.

Teorema 5.5.4: Cantidad de lenguajes

Hay $2^{\aleph_0}$ lenguajes.

Demostración

Hay tantos problemas algorítmicos como subconjuntos tiene $\mathcal{P}(\Sigma^*)$. Es decir, la cantidad de problemas algorítmicos es $\text{card}(\mathcal{P}(\Sigma^*))$. Observe que:

$$\begin{aligned} \text{card}(\mathcal{P}(\Sigma^*)) &= 2^{\text{card}(\Sigma^*)} && \text{(por el Teorema 5.5.2)} \\ &= 2^{\text{card}(\mathbb{N})} && \text{(por el Teorema 5.5.3)} \\ &= 2^{\aleph_0} && \text{(por el tamaño de } \mathbb{N}\text{).} \end{aligned}$$

Nota 5.5.3

El cardinal $2^{\aleph_0}$ es especial y se identifica con la letra c, dado que corresponde al cardinal del continuo: $2^{\aleph_0} = c = \text{card}(\mathbb{R})$. Por el Teorema 5.5.2 se tiene que $\aleph_0 < c$, es decir, hay (muchos) más números reales que naturales.

De manera aparte, pero como tema complementario, es importante llamar la atención a que c es interesante porque permite plantear uno de los problemas más desafiantes de las matemáticas, la Hipótesis del Continuo de Rienman (HCR):

> No existe conjunto alguno cuyo tamaño esté estrictamente entre el tamaño de los naturales y el tamaño de los reales.

Dicho de otra manera, la HCR plantea la imposibilidad de que exista un cardinal entre $\aleph_0$ y $2^{\aleph_0}$. Si la HCR es cierta, entonces se tendría que

$$\aleph_1 = 2^{\aleph_0} = c.$$

5.5.4. ¿Cuántas funciones computables hay? Es necesario fijar un alfabeto de programación para determinar la cantidad de funciones computables. Se supondrá que Π es el alfabeto de símbolos usados para programar una función computable en cualquier lenguaje de programación. De esta forma, se está tomando la dirección de tratar de identificar cada función computable con al menos una cadena de símbolos en Π que representa su codificación en un lenguaje de programación. Esto es lo que sucede en la realidad pues se pueden implementar tantos procedimientos de decisión como secuencias correctas de instrucciones escritas con símbolos en Π haya (en el lenguaje de programación de elección). De esta forma, $\text{card}(\Pi^*)$ impone una cota superior sobre la cantidad de funciones implementables en cualquier lenguaje de programación. Se supone que cada programa es finito y que la cantidad de lenguajes de programación también lo es, lo cual es cierto en la práctica.

Teorema 5.5.5: Cantidad de funciones computables

La cantidad de funciones computables es $\aleph_0$.

Demostración

La cantidad de funciones computables que se pueden programar con el alfabeto Π está acotada por $\text{card}(\Pi^*)$: el tamaño de todas las secuencias sobre Π (un programa tiene una descripción finita, luego debe ser una cadena en Π^*). Siguiendo un razonamiento similar al de la demostración del Teorema 5.5.3, se tiene que $\Pi^* \sim \mathbb{N}$. Luego, la cantidad de funciones computables es, a lo sumo, $\aleph_0$.

Saber que $\aleph_0$ es una cota superior sobre la cantidad de funciones computables es importante. Por una parte, se establece que esta cota se sostiene independientemente de cual sea el alfabeto con que se implementen las soluciones algorítmicas, mientras este alfabeto sea contable. Segundo, es la última pieza que se requiere para demostrar que hay problemas algorítmicos para los cuales no existe programa de computador alguno que los resuelva.

5.5.5. Hay problemas que no se pueden decidir. Este aparte, finalmente, presenta el princial resultado de esta sección. Es decir, demuestra que existen problemas de decisión que no se pueden resolver mecánicamente con un computador, sin importar qué lenguaje de programación se use. Como se mencional al inicio de la sección, es un resultado negativo porque muestra la imposibilidad de usar los computadores como máquinas universales que resuelvan cualquier problema (de

decisión). Sin embargo, y como se puede comprobar con situaciones diarias, los computadores son extensamente utilizados para resolver problemas y asisitir a las personas en la solución de otros tantos. Por ello, a pesar de que teóricamente se establece dicha imposibilidad de solución de problemas macánicamente, en la práctica cada día se depende más de los computadores.

Teorema 5.5.6

Existe al menos un problema de decisión para el cual no hay una función computable que lo resuelva. Es decir, hay al menos un problema que un computador no puede resolver.

Demostración

Note que hay $2^{\aleph_0}$ lenguajes y $\aleph_0$ funciones computables, por los Teorema **??**, respectivamente. Por el Teorema 5.5.2, se tiene que $\aleph_0 < 2^{\aleph_0}$. En consecuencia, hay al menos un problema de decisión para el cual no es posible implementar un programa de computador que lo decida.

El Teorema 5.5.6 no riñe con el resultado del Teorema 5.3.2. Una cosa es entender que para cada problema hay exactamente una función total que lo decide y otra que dicha función sea pueda mecanizar en un computador. Dicho de otra forma, en un computador no se pueden definir todas las funciones con dominio en Σ^* y rango en Σ. Así es como surge la existencia de problemas indecidbles: son aquellos problemas para los cuales no existe una implementación de su función de decisión asociada. La demostración del Teorema 5.3.2 no es constructiva y por ello no permite identificar concretamente uno de estos lenguajes indecidibles que sirva como testigo para corroborar su conclusión (no por ello deja de ser cierto).

Nota 5.5.4

El Teorema 5.5.6 presenta una realidad abrumadora: en relación con la totalidad de problemas de decisión, son muy pocos (pero muy pocos) aquellos que un computador puede resolver. Aún más sorprendente es que, seguramente, pocas veces los problemas indecidibles surgen en la práctica de la programación. Por ejemplo, en este texto solo se han diseñado algoritmos de decisión implementables.

Ejercicios

1. Demuestre que $\_ \sim \_$ es una relación de equivalencia. Es decir, reflexiva, simétrica y transitiva.
2. Demuestre que $\_ \leq \_$ (sobre cardinales infinitos) es un orden parcial. Es decir, reflexiva y transitiva.
3. Use la propiedad de R. Dedekind en la Nota 5.5.1 para concluir que hay una cantidad inifita de números primos. ¿Es esta cantidad contable o no contable?
4. Demuestre que el mapa que invierte punto a punto los bits de las cadenas binarias en la Observación 1 en la demostración del Teorema 5.5.3 induce una función biyectiva entre X_0 y X_1.
5. Demuestre que el mapa que interpreta cadenas binarias en números en base 10 en la Observación 2 en la demostración del Teorema 5.5.3 induce una función biyectiva entre X_1 y $\mathbb{N}$.
6. Elabore en más detalle la demostración del Teorema 5.5.5. En particular, diseñe una biyección entre Π^* y $\mathbb{N}$. Ayuda: como Π es finito, suponga que tiene $n \geq 1$ elementos. Generalice el razonamiento, pasando de base 2 a base n, de la demostración del Teorema 5.5.3.

Notas del capítulo y referencias

Los problemas de decisión han sido el principal objeto de estudio de la teoría de la computabilidad. Textos como el de M. Davis [**Dav85**] explican de manera detallada por qué es razonable restringir el estudio de la computabilidad e indecidibilidad a problemas de decisión. Explicaciones similares se pueden encontrar en [**CLRS22, Eri19**]. Basar el estudio de problemas de decisión en el marco de lenguajes formales es parte de la influencia del texto de Cormen et al. [**CLRS22**] sobre este texto. Sin embargo, la definición 'genérica' de problema de decisión, y las definiciones de aceptación y decisión de lenguajes basada en funciones parciales y totales no es la usual (secciones 5.1 y 5.3). En particular, la relación entre función total y decisión sobre lenguajes (Teorema 5.3.2) es novedosa. Se optó por combinar lenguajes y funciones porque permite abordar el objeto de estudio en cuestión a alto nivel, identificando los concpetos clave y sin perder generalidad. Las funciones parciales y totales son conceptos básicos de matemáticas, y son familiares para un grupo amplio de lectores.

La noción de computabilidad presentada en este capítulo es la usual y tratamientos similares se pueden encontrar en [**Dav85, Har12**]. Incluir definiciones de

computabilidad a nivel de funciones y de problemas es conveniente dado que permite entender el concepto de manera más general. De igual forma, es importante indicar las relaciones de conjunto computablemente enumerable y computable con las nociones de semidecidibilidad y decidibilidad, que son también comunes en literatura de ciencias de la computación. En la introducción del capítulo se indica que estas definiciones se dan sin necesidad de apelar a un modelo de computación en particular. Esto es posible porque las nociones presentadas coinciden en los principales modelos de computación. Hay tres categorías principales de modelos de computación: modelos secuenciales que incluyen las máquinas de Turing, las máquinas de Post y las máquinas de acceso aleatorio; modelos funcionales que incluyen el cálculo λ, las funciones recursivas generales, los sistemas de reescritura abstractos y la lógica combinatoria; y modelos concurrentes como el modelo de actores, las redes de Petri, la lógica de reescritura y los autómatas celulares. Material adicional de sistemas de reescritura abstractos se puede encontrar en [**Ohl02**] y de lógica de reescritura en [**CDE$^+$07**]. Los demás modelos cuentan con literatura abundante que puede ser encontrada en textos clásicos o en internet.

El tratamiento que se hace de la existencia de problemas indecidibles o funciones no computables no es el común, ni para pregrados de computación ni de matemáticas. Esto puede ser así por la necesidad de usar aritmética cardinal que no es un tema estudiando, generalmente, en ninguno de estos currículos. La principal justificación para dar este tratamiento en el capítulo tiene que ver con la claridad del argumento (no constructivo): hay al menos una función que no puede ser calculada algorítmicamente. Los resultados acerca de cardinales utilizados en la Sección 5.5, incluyendo su aritmética, pueden ser constatados y profundizados en textos de teoría de conjuntos intuitiva como [**Bre06, Hal11**] y axiomática como [**Kun11**]. El texto [**Ohl02**] de E. Ohlebush es una buena fuente de problemas indecidibles, como terminación y confluencia, en el marco de los sistemas de reescritura abstractos y de reescritura de términos, y de los problemas indecidibles clásicos de E. Post.

Bibliografía

[BB88] Gilles Brassard and Paul Bratley, *Algorithms: Theory and practice*, Prentice Hall, 1988.

[Bei13] Wolfgang Bein, *Advanced techniques for dynamic programming*, pp. 41–92, Springer, New York, NY, 2013.

[Bel84] Richard Bellman, *Eye of the hurricane: an autobiography*, World Scientific, 1984.

[Bha15] Harsh Bhasin, *Algorithms: Design and analysis*, Oxford University Press, 2015.

[Boh06] Jaime Bohórquez, *Diseño efectivo de programas correctos*, Escuela Colombiana de Ingeniería, 2006.

[Bre06] Joseph Breuer, *Introduction to the theory of sets*, 1st ed., Dover Publications, 2006.

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott, *All about maude*, 1st ed., Springer, 2007.

[CLRS22] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein, *Introduction to algorithms*, 4th ed., MIT Press, 2022.

[Coh90] Edward Cohen, *Programming in the 1990s: an introduction to the calculation of programs*, Springer-Verlag, 1990.

[Dav85] Martin Davis, *Computability and unsolvability*, Dover Publications, 1985.

[Dij76] Edsger Wybe Dijkstra, *A discipline of programming*, Prentice-Hall, 1976.

[Eri19] Jeff Erickson, *Algorithms*, 2019.

[Gri81] David Gries, *The science of programming*, Springer-Verlag, 1981.

[Hal11] Paul R. Halmos, *Naive set theory*, 1st ed., Martino Fine Books, 2011.

[Har12] David Harel, *Algorithmics: The spirit of computing*, 3rd ed., Springer, 2012.

[HHE20] Steven Halim, Felix Halim, and Suhendry Effendy, *Competitive programming 4: Book I*, LuLu, 2020.

[Kal90] Anne Kaldewaij, *Programming: the derivation of algorithms*, Prentice-Hall, 1990.

[KET06] Jon Kleinberg and Éva Tardos, *Algorithm design*, Pearson, 2006.

[Knu88] Donald Knuth, *The art of computer programming: Volume 3 / sorting and searching*, 2nd ed., Addison-Wesley, 1988.

[Kun11] Kenneth Kunen, *Set theory*, 2nd ed., College Publications, 2011.

[Lev12] Anany Levitin, *Introduction to the design and analysis of algorithms*, 3rd ed., Pearson, 2012.

[Ohl02] Enno Ohlebusch, *Advanced topics in term rewriting*, 2nd ed., Springer, 2002.

[Rou17] Tim Roughgarden, *Algorithms illuminated I: The basics*, 2017.

[Rus08] John Rust, *Dynamic programming*, pp. 1–26, Palgrave Macmillan, 12 2008.

[SF13] Robert Sedgewick and Philippe Flajolet, *An introduction to the analysis of algorithms*, 2nd ed., Addison-Wesley Professional, 2013.

[Ski08] Steven S. Skiena, *The algorithm design manual*, Springer London, 2008.

Índice alfabético