Introducción al Pensamiento Algorítmico

Guía de Estudio

Licenciatura en Inteligencia Artificial y Licenciatura en Física y Matemáticas

Jorge Hermosillo Valadez, Daniel Rivera López
Centro de Investigación en Ciencias
Universidad Autónoma del Estado de Morelos (UAEM)

26 de junio de 2025

Índice general

Prefacio

Esta obra es una guía introductoria al Pensamiento Algorítmico. Forma parte del curso propedéutico de la Licenciatura en Inteligencia Artificial y de la Licenciatura en Física y Matemáticas, del Instituto de Ciencias Básicas y Aplicadas, de la Universidad Autónoma del Estado de Morelos (UAEM).

El pensamiento algorítmico se refiere a la habilidad de analizar problemas de manera a que puedan ser resueltos mediante una secuencia de pasos específicos: un algoritmo. Aunque es una habilidad obligatoria para cualquier profesional de las ciencias computacionales, es también uno de los pilares del método científico: cualquier profesional en ciencias que pretenda resolver problemas en su disciplina, y particularmente mediante una computadora, debe contar con esta habilidad. En esta guía de estudio, se abordan brevemente algunas bases sobre arquitecturas de cómputo y se introduce al lector al mundo del análisis de problemas bajo el enfoque del pensamiento computacional y al diseño básico de algoritmos.

1 Introducción a la Noción de Algoritmo

Jorge Hermosillo Valadez, Daniel Rivera López

En este capítulo, comenzamos con un breviario cultural sobre la historia de las computadoras. Esto nos lleva a comentar muy brevemente sobre las nociones de byte como unidad de información y de arquitectura de una computadora. Luego definimos el concepto de algoritmo, analizamos sus características y definimos nociones importantes como estado y variable. Concluimos el capítulo introduciendo dos instrucciones básicas para el control de flujo de un algoritmo.

1.1. Orígenes de la computación moderna

1.1.1. Breve reseña histórica

En esta sección hacemos una revisión concisa de los orígenes de la computación y en particular del hito que marcó el inicio de la era moderna en este campo. La importancia de este hecho histórico radica en el impacto que tuvo en el desarrollo de las computadoras, que propició el surgimiento y crecimiento exponencial de métodos, herramientas, conocimiento y tecnología que hoy son materia de estudio de la ciencia computacional.

COMPUTACIÓN

La ciencia de la computación tiene su origen en el cálculo, es decir, en la preocupación del ser humano por encontrar formas de realizar operaciones matemáticas de forma cada vez más rápida y eficiente. La historia de la computación se remonta a unos 500 años A.C. con el surgimiento del ábaco. Posteriormente, con el nacimiento del álgebra, se introduce la palabra algoritmo hacia el siglo IX de nuestra era. No obstante, la primer máquina computadora aparece 800 años después, en el siglo XVII. Es entonces, cuando nace la noción moderna de computación, con el diseño de aparatos y máquinas que pueden realizar operaciones aritméticas de forma automática.

Ciento cincuenta años más tarde, surge el álgebra booleana, indispensable para diseñar circuitos digitales. Pero no fue sino hasta la primera mitad del siglo XX, con el advenimiento del bulbo electrónico, que la humanidad conoce la primera computadora electrónica, cuyo principio básico de operación rige el diseño de nuestras computadoras de hoy. Vemos pues, que a pesar de que el término algoritmo había surgido alrededor del año 830 de nuestra era, hubo que esperar hasta los albores del siglo XX—mil años después—para conocer los principios matemáticos que rigen la teoría de la computación moderna.

1.1.2. ENIAC: Nace la computación moderna

Existe mucha información publicada acerca de la historia de las computadoras, por lo que no haremos aquí un recuento detallado de su evolución. Sin embargo, existe un hito en este proceso que vale la pena recordar por el impacto que tuvo en el diseño de las computadoras, produciendo un cambio que revolucionó la concepción

de la computación, habilitando la posibilidad de crear un sinnúmero de soluciones a problemas de diversa índole. Se trata de la primera computadora electrónica, que funcionaba con tubos al vacío, llamada Electronic Numerical Integrator And Calculator (ENIAC).

ENIAC

La ENIAC se construyó en la Universidad de Pennsylvania en 1947, por un equipo de diseño encabezado por los ingenieros John Mauchly y John Eckert. En el diseño de la ENIAC fueron incluidas nuevas técnicas de la electrónica que permitían minimizar el uso de partes mecánicas. Esto trajo como consecuencia un incremento significativo en la velocidad de procesamiento: podía efectuar 5,000 sumas o 500 multiplicaciones en un segundo y permitía el uso de aplicaciones científicas en astronomía, meteorología, etc. Esta máquina tenía más de 18,000 tubos de vacío, consumía 200 KW de energía eléctrica y requería todo un sistema de aire acondicionado. El proyecto, auspiciado por el departamento de Defensa de los Estados Unidos, culminó dos años después, cuando se integró a ese equipo el ingeniero y matemático húngaro-estadounidense John von Neumann (1903 - 1957).

Durante el desarrollo del proyecto ENIAC, von Neumann propuso mejoras que posibilitaron el desarrollo de las computadoras actuales:

1. Utilizar un **sistema de numeración de base dos (binario)** en vez del sistema decimal tradicional.
2. Hacer que las instrucciones de operación estén en la memoria, al igual que los datos. De esta forma, memoria y programa residirán en un mismo sitio.

Con estos cambios, la ENIAC revolucionó el mundo de las computadoras, y con ellos, nace la era de la computación moderna.

1.1.3. Evolución de las computadoras

A lo largo de las décadas, la evolución de las computadoras ha sido impresionante, dividida en distintas generaciones que han marcado avances tecnológicos clave. Cada una de estas generaciones trajo consigo nuevas capacidades y avances importantes, pero también enfrentó diversas limitaciones.

▸ Primera generación (1940 – 1956)

La ENIAC marcó la primera generación de computadoras, que estuvo marcada por el uso de tubos de vacío o bulbos como componentes principales. Estas máquinas eran extremadamente grandes, ocupaban habitaciones enteras y consumían grandes cantidades de energía. Eran computadoras lentas y propensas a fallos frecuentes debido al sobrecalentamiento de los bulbos.

A pesar de estas limitaciones, representaron un avance notable al ser las primeras computadoras digitales, capaces de realizar cálculos automáticos. El almacenamiento de datos en cintas o tambores magnéticos y el uso del lenguaje de máquina (binario) para procesar instrucciones fueron algunas de las innovaciones de esta época.

▸ Segunda generación (1956 – 1963)

La segunda generación de computadoras sustituyó los bulbos por transistores, lo que supuso un gran avance en términos de tamaño, velocidad y eficiencia. Las computadoras de esta generación eran más pequeñas y

consumían menos energía. Además, comenzaron a utilizarse lenguajes de programación de alto nivel, como FORTRAN y COBOL, lo que facilitó el desarrollo de software más complejo.

Aunque los transistores permitieron la creación de máquinas más fiables y rápidas, el costo seguía siendo elevado y estas computadoras todavía no eran accesibles para la mayoría de las personas.

▸ Tercera generación (1964 – 1971)

Las computadoras se hicieron aún más compactas gracias a la invención de los circuitos integrados. Estos dispositivos, que agrupaban varios transistores en un solo chip, incrementaron la capacidad de procesamiento y permitieron la creación de sistemas operativos que gestionaban múltiples tareas a la vez. No obstante, aunque esta generación mejoró notablemente la eficiencia, el acceso seguía estando restringido debido a los altos costos.

▸ Cuarta generación (1971 – 1981)

La cuarta generación se caracteriza por el uso de microprocesadores. Estos pequeños chips, que contienen millones de transistores, han permitido la creación de computadoras personales accesibles para el público general. El Intel 4004, lanzado en 1971, fue uno de los primeros microprocesadores que revolucionaron la industria informática. La aparición de computadoras personales como el Apple II y la IBM PC democratizó el acceso a la informática, impulsando su masificación. Además, los avances en sistemas operativos gráficos facilitaron la interacción entre el usuario y la máquina, permitiendo que personas sin conocimientos técnicos avanzados pudieran usar computadoras.

▸ Quinta generación (1981 – presente)

Finalmente, la quinta generación de computadoras, que está en curso, se define por el desarrollo de sistemas de integración a gran escala. La iMac fue el primer ordenador "todo en uno", combinando hardware y software en un solo dispositivo con un diseño innovador. Este enfoque simplificó el uso y mantenimiento de las computadoras, haciendo que fueran más accesibles para el público general.

Hoy en día el nivel de integración se está haciendo cada vez más fuerte con el advenimiento de la Inteligencia Artificial. Los avances en procesamiento paralelo, machine learning, y redes neuronales han llevado a la creación de máquinas capaces de realizar tareas sumamente complejas, como el reconocimiento de imágenes, el procesamiento de lenguaje natural y la toma de decisiones autónomas.

Las computadoras actuales no solo son más rápidas y potentes, sino que también son capaces de aprender y adaptarse a nuevos problemas, lo que abre nuevas posibilidades en áreas como la robótica y la automatización. Sin embargo, esta generación plantea importantes desafíos éticos y de privacidad, ya que la dependencia de sistemas autónomos plantea preguntas sobre el control y la toma de decisiones sin intervención humana.

En resumen, la evolución de las computadoras desde la primera generación, basada en tubos de vacío, hasta la quinta generación actual, centrada en la inteligencia artificial, ha transformado la forma en que interactuamos con la tecnología y el mundo que nos rodea. Cada generación ha superado las limitaciones de la anterior, permitiendo avances significativos en velocidad, eficiencia y accesibilidad. No obstante, el progreso ha traído consigo nuevos desafíos, especialmente en la intersección entre la tecnología y los valores humanos.

1.2. Números binarios como unidades de información

Como hemos dicho, la computadora sólo es capaz de manipular dígitos binarios. Por lo tanto, cualquier tipo de información —números, letras, imágenes, etc.— está codificada en una computadora mediante 1s y 0s.

Para referirnos a cualquiera de estos dígitos binarios, se utiliza el término **'bit' (binary digit)**. Un bit permite entonces codificar un valor numérico de 0 o 1.

Definición 1.2.1. **Un byte** es una secuencia ordenada de 8 bits (Figura 1.1). Es la unidad básica de información en términos de procesamiento y almacenamiento computacional.

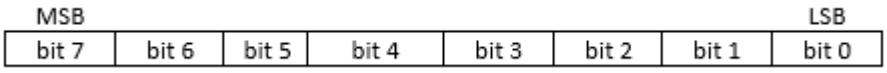


Figura 1.1: Típica organización de 1 Byte como una palabra de 8 bits.

No ahondaremos en nociones de sistemas numéricos, sin embargo, el lector debe saber que, al igual que nuestro sistema decimal, el sistema binario es un sistema posicional, por lo que cada bit en un byte tiene un peso que se incrementa de derecha a izquierda; de ahí los términos "Bit Menos Significativo" (LSB: Less Significant Bit) y "Bit Más Significativo" (MSB: Most Significant Bit).

1.3. Noción de arquitectura de una computadora y de programa

La arquitectura básica de una computadora se ilustra en forma esquemática en la Figura 1.2. Incluye un microprocesador (el "cerebro" de la computadora) y lo que se conoce como el "sistema de memoria": un conjunto de dispositivos en los cuáles se puede almacenar información en unidades de bytes. El almacenamiento puede ser permanente, como en la memoria ROM—READ ONLY MEMORY— o en discos duros que pueden ser mecánicos o de silicio (también llamados de estado sólido). También puede ser temporal, como en la memoria RAM—RANDOM ACCESS MEMORY— o en la memoria interna del microprocesador que incluye la memoria cache y sus registros, donde la información se guarda mientras la computadora está encendida y durante la ejecución de programas.

Contra toda apariencia, el microprocesador (micro) es un dispositivo bastante "tonto". Se trata de lo que se conoce en ciencia computacional como un autómata: una máquina que realiza siempre las mismas acciones en forma repetitiva, una y otra vez, en lo que se conoce como *ciclo de operación*. El ciclo de operación de un micro consta de 3 etapas: 1) lectura de la siguiente instrucción, 2) decodificación de la instrucción y 3) ejecución de la instrucción. Por instrucción debemos entender cualquier **acción** que sea parte de las cosas que puede realizar el micro:

- Leer de memoria
- Escribir a memoria
- Calcular (sumar, restar, multiplicar, dividir, etc.)

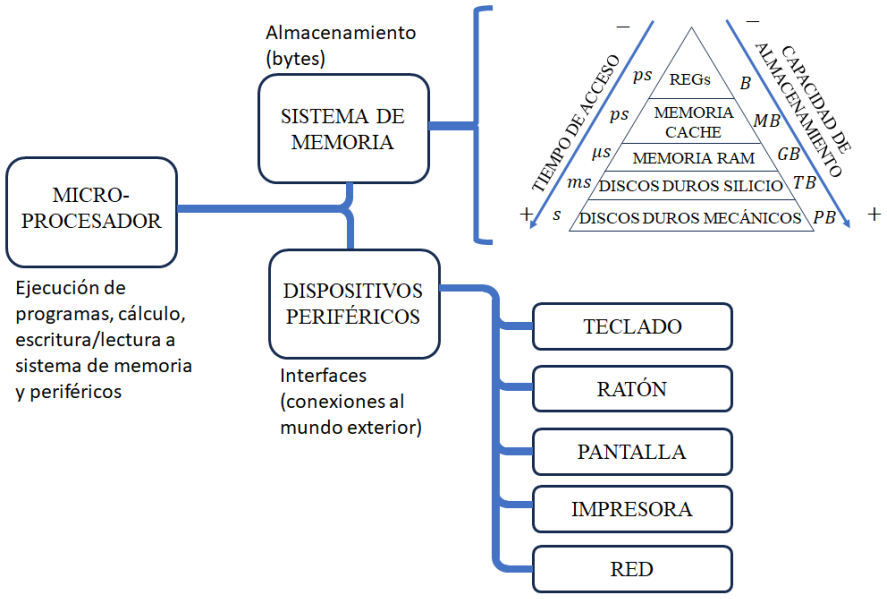


Figura 1.2: Arquitectura esquemática de una computadora. Se muestran el microprocesador (el "cerebro" de la computadora), el sistema de memoria (almacenamiento de datos en unidades de información) y el sistema de dispositivos periféricos (la forma en que el microprocesador intercambia datos con el mundo exterior).

- Leer/Escribir a otros dispositivos periféricos (teclado, pantalla, ratón, impresora, la red, etc.).

Por lo tanto, cuando hablamos de la ejecución de un programa, hablamos de todas las cosas que suceden (una a una) cuando escribimos un programa de computadora. Todas las cosas que ocurren como parte de este ciclo de operación del micro.

Para fines prácticos, diremos que un programa es esencialmente un algoritmo que se escribe usando un lenguaje de programación (p.ej. C, PASCAL, JAVA, PYTHON, etc.). Por lo tanto, antes de aprender a programar es necesario aprender a diseñar algoritmos.

1.4. Conceptos básicos sobre algoritmos.

Para introducir los conceptos sobre algoritmos, analizamos primero un problema de manera informal. Esto nos permitirá identificar algunos aspectos básicos que formalizamos a continuación.

1.4.1. ¿Cómo cruzar el río?

En esta sección comenzaremos por presentar un problema clásico para el que trataremos de dar una solución de manera informal por ahora.

Problema 1.4.1: ¿Cómo cruzar el río?

Un granjero con un zorro, un ganso y un saco de maíz necesita cruzar un río. El granjero tiene un bote de remos, pero sólo caben él y uno de sus tres objetos. Por desgracia, tanto el zorro como el ganso tienen hambre. El zorro no puede quedarse solo con el ganso, o el zorro se comerá al ganso. Del mismo modo, el ganso no puede quedarse solo con el saco de maíz, o se lo comerá. ¿Cómo hace el granjero para llevarlo todo al otro lado del río?

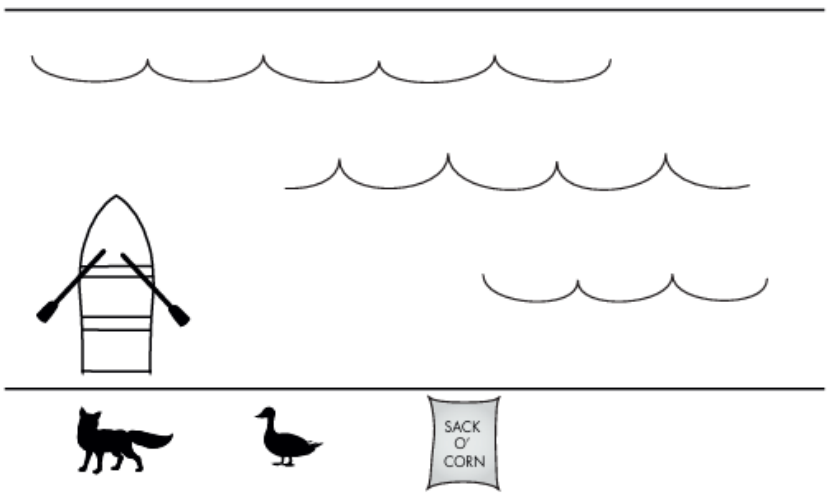


Figura 1.3: Situación inicial a partir del problema 1.4.1

La Figura 1.3 ilustra la configuración inicial planteada por el problema 1.4.1. Como el granjero solo puede llevar una cosa a la vez en su bote, tendrá que hacer varios viajes para poder cruzar el río con todo.

Ahora bien, no puede llevar lo primero que se le ocurra, porque ciertas combinaciones pueden tener consecuencias desastrosas:

- Si en el primer viaje se lleva al zorro, entonces el ganso se quedará solo con el saco de maíz, y eso es un problema: ¡el ganso se comería el maíz!
- Si en cambio se lleva el saco de maíz primero, dejaría al zorro con el ganso, y entonces el zorro se lo comería.

Por eso, la mejor opción para el primer viaje es llevarse al ganso. De esa manera, evita cualquier situación peligrosa y comienza a resolver el problema de forma segura. Esta decisión nos lleva a la configuración que se muestra en la figura.

Hasta este punto, todo va bien: el granjero ya ha cruzado una vez con el ganso, y lo ha dejado en la orilla lejana.

Pero ahora viene el segundo viaje, y el granjero tiene que decidir si lleva el zorro o el saco de maíz.

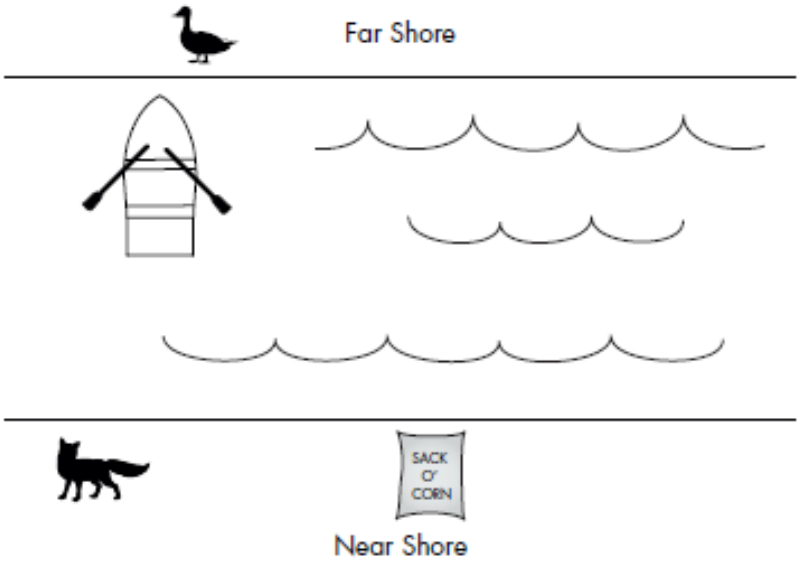


Figura 1.4: Situación conveniente

Aquí surge el problema: Cualquiera que sea el objeto que lleve, al llegar a la orilla lejana tendrá que dejarlo con el ganso mientras él regresa solo a buscar lo que falta.

Entonces pasan dos cosas posibles:

- Si lleva el zorro, lo deja con el ganso... ¡y el zorro se lo comería!
- Si lleva el maíz, lo deja con el ganso... ¡y el ganso se lo comería!

En ambos casos, hay una situación peligrosa. Por eso, ninguna de estas opciones parece funcionar, y el problema empieza a parecer sin solución.

Como ya se explicó, en el primer viaje el granjero debe llevarse al ganso, para evitar que el zorro se lo coma o que él se coma el maíz.

En el segundo viaje, el granjero cruza de nuevo el río y esta vez se lleva al zorro. Pero aquí ocurre algo inteligente: no deja al zorro con el ganso, porque eso sería peligroso. En lugar de eso, cuando llega a la otra orilla, deja al zorro allí y se lleva de vuelta al ganso al otro lado del río (ver Figura 1.4).

En el tercer viaje, el granjero cruza llevando el saco de maíz, y lo deja con el zorro, que no se lo va a comer. Luego regresa una vez más (cuarto viaje) para buscar al ganso, y finalmente lo lleva con los otros.

Así, todos —el zorro, el ganso y el saco de maíz— cruzan el río sin que nadie se coma a nadie. Esta solución completa se muestra ilustrada en la Figura 1.5.

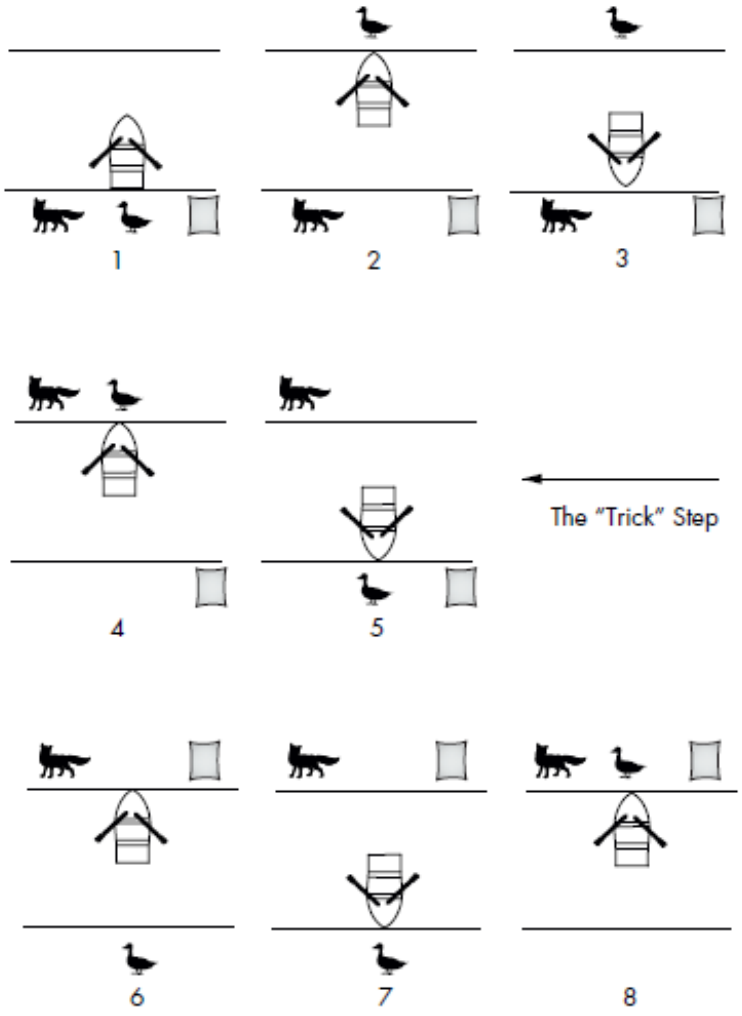


Figura 1.5: Solución al problema 1.4.1

Ejercicio Resuelto 1.4.1: Escribe la solución al problema 1.4.1

Escribe la secuencia de pasos que resuelven el problema 1.4.1, siguiendo la secuencia mostrada en la Figura 1.5.

1.4.2. ¿Qué es un algoritmo?

La sección anterior nos permite introducir la noción de algoritmo[1].

Definición 1.4.1 (Algoritmo)**.** Un algoritmo es un conjunto ordenado y finito de pasos individuales o instrucciones que definen una secuencia lógica de operaciones para resolver un problema o realizar una tarea específica.

Los algoritmos son una forma de especificar, claramente y sin ambigüedad, un proceso de resolución de un problema, para el cuál se ha de especificar los datos de entrada y la salida deseada. Si se sigue al pie de la letra, un algoritmo debe producir sistemáticamente el mismo resultado a la salida para un mismo dato de entrada.

La ciencia computacional es el arte de analizar y diseñar algoritmos correctos. Existe una disciplina formal que se encarga del estudio sistemático de los algoritmos, que nos ayuda a comprender exactamente cómo caracterizarlos. Esto es importante, ya que una comprensión profunda de lo que es un algoritmo correcto, es la base de cualquier solución computacional.

Por ahora, en esta sección, analizaremos las propiedades de un algoritmo, lo que nos permitirá entender cómo estructurar una solución algorítmica a un problema.

▸ Conjunto de pasos individuales

Volviendo a la DEFINICIÓN 1.4.1, la primera propiedad de un algoritmo es que se trata de un **conjunto de pasos individuales**. La analogía de esta propiedad es una receta de cocina, como los pasos a seguir para preparar café: “poner agua en la cafetera”, “agregar café”, “encender la cafetera”, etc. Cada paso implica una operación, una acción.

▸ Definitud

Las acciones deben estar perfectamente definidas. Cada paso de un algoritmo debe tener una sola interpretación: no puede haber ambigüedad. Volviendo a la analogía de la receta de cocina, no es lo mismo “agregar 10g. de café”, que “100g.”, o “poco café”, el resultado sería distinto o ambiguo. Por esta misma razón, el inicio y el fin de un algoritmo sirven para precisar el alcance de la secuencia de pasos.

▸ Secuencia ordenada

Un algoritmo es una secuencia ordenada de acciones: el orden es importante, ya que, si se intercambian las acciones, el resultado podría ser otro completamente distinto al deseado: ¡el café y el agua se ponen antes de encender la cafetera!

En lo sucesivo, escribiremos un algoritmo utilizando una sintaxis (escritura) particular, mostrada a continuación, que refleja sus características y propiedades como las hemos definido.

[1] Cormen, T.H. et al. (2009) Introduction to Algorithms.

Algoritmo 1.4.1. Estructura de un algoritmo

```
1   Inicio
2   acción/instrucción
3   acción/instrucción        /* comentario */
⋮   ⋮
k   Fin
```

Al respecto comentamos lo siguiente:

1. Como vemos, el algoritmo 1.4.1 muestra una secuencia ordenada de pasos, que van desde el paso 1, marcado por la palabra "Inicio" (punto de entrada) hasta un paso k donde termina el algoritmo con la palabra "Fin" (punto de salida).
2. La línea 2 muestra una acción o instrucción. Esto corresponde a una sola acción que se ejecuta en esa línea.
3. La siguiente línea del algoritmo muestra otra acción y a un costado un "comentario". Esto es un texto que no corresponde a ninguna acción o instrucción y que sólo sirve para propósitos de legibilidad o explicación al lector acerca del propósito de esa línea. Esto significa que lo que está entre los símbolos "/* */" no es ninguna acción, es sólo algo que escribimos para nosotros mismos, o para quien vaya a leer el algoritmo.

1.4.3. Noción de estado

ESTADO Un concepto estrechamente relacionado con la noción de secuencia de acciones es el de **estado**[2].

Definición 1.4.2 (Estado)**.** Un estado es la configuración actual de toda la información que mantiene un programa (o algoritmo) en un instante de tiempo.

Por ejemplo, el ciclo operativo básico de un microprocesador cuenta con tres estados: lectura de instrucción, decodificación de instrucción, y ejecución de instrucción.

La noción de estado es importante porque nos dice que nada es estático en una computadora: un algoritmo fluye en el tiempo, cambia de estado. Por lo tanto, **no hay una visión global** cuando se trata de algoritmos. Esto significa que, en cada instante de tiempo, el entorno dentro del cual se ejecuta un algoritmo existe en un estado particular; pero, para el momento en el que el siguiente paso se ejecuta, algo pudo haber cambiado. Esto es precisamente lo que se desea: un cambio de estado inicial a estado final; tal cual la receta del café: al final no hay más ingredientes, sino la bebida.

Desde el punto de vista del diseño de algoritmos, esta visión local de estado es fundamental, ya que los pasos de un algoritmo se ejecutan uno a uno, y sólo un paso a la vez puede considerarse en cada instante de tiempo.

[2] Hopcroft, J.E., Motwani, R. and Ullman, J.D. (2006). Introduction to Automata Theory, Languages, and Computation (3rd Edition).

En otras palabras, **a menos que se almacene el estado actual, todo lo que se hizo una vez ejecutado un paso se pierde en el paso siguiente**. Por esta razón debemos hablar del concepto de **variable**.

1.4.4. Noción de variable

Definición 1.4.3 (Variable)**.** Una variable es un contenedor de información que la computadora mantiene en algún lugar físico de la memoria. Un algoritmo puede escribir información en el contenedor, lo que se conoce como asignación a la variable, y el contenido de ésta permanece guardado mientras no haya otra asignación.

Las variables son indispensables en cualquier algoritmo, ya que, el valor que guardan en un instante de tiempo es parte el estado de un programa. Si se altera el valor de una variable, se altera el estado del programa. Si se desea conservar un estado de cosas, las variables nos permiten almacenar en la memoria de la computadora los datos que conforman ese estado de cosas, de manera que tenemos el control total de lo que ocurre con ellos. Ese estado de cosas puede cambiar si deseamos alterarlo durante la ejecución de un algoritmo.

▸ Asignación a una variable

Para darle un valor concreto a una variable usamos una instrucción (acción) que llamamos **asignación**. Esta acción la representamos con el símbolo $\leftarrow$. Por ejemplo, para decir que a una variable x le asignamos el valor 2, escribimos:

$$x \leftarrow 2 \quad \text{/* se asigna el valor 2 a la variable } x \text{ */}$$

▸ Lectura de una variable

La instrucción $x \leftarrow 2$ asignó el valor de 2 a la variable x, por lo que ahora **el contenido** de x es 2.

Supongamos que ahora queremos usar el valor que contiene x para hacer un cálculo. Por ejemplo, queremos sumar 3 a x. Para ello necesitamos **leer** el contenido de x y sumarle 3. Esto lo escribimos de la siguiente forma:

$$x + 3 \quad \text{/* se suma el valor 3 a la variable } x \text{ */}$$

Hasta aquí podríamos pensar que todo va bien. Si quisiéramos imprimir en pantalla el valor de x podríamos escribir:

$$\text{Imprime } x \quad \text{/* se imprime en pantalla el valor de } x \text{ */}$$

Sin embargo, nos llevaríamos una sorpresa al descubrir que el valor que se imprime es:

$$2.$$

¿Cómo? Si acabamos de sumarle a x un 3. Claro que hicimos la suma, pero este nuevo estado de cosas para x ¡no lo guardamos!

Es cierto que pudimos haber escrito:

$$\text{Imprime } x + 3 \quad \text{/* se imprime en pantalla el valor de } x + 3 \text{ */}$$

y desde luego que en pantalla veríamos 5. Sin embargo, esto tampoco altera el valor de x que seguiría siendo 2.

Veamos cómo hacerlo correctamente.

▸ Lectura y asignación simultáneas

Para que x tenga el nuevo valor deseado, necesitamos escribir:

$$x \leftarrow x + 3 \quad \text{/* se suma el valor 3 a la variable y se guarda el resultado en } x \text{ */}$$

En palabras, esta instrucción dice:

Lee el contenido de la variable x, sumale 3, y guarda la suma en la variable x.

Esto es: primero se lee el contenido de x, luego se calcula la suma y finalmente se asigna el valor de la suma a x nuevamente. Cuando decimos que hacemos una lectura-escritura "simultáneas" en realidad estamos diciendo que ambas operaciones se **escriben simultáneamente** pero se ejecutan en forma secuencial.

En resumen, una variable sirve para guardar un estado de cosas mediante una instrucción de asignación, y permanece así mientras no hagamos una nueva asignación a la variable.

Observación 1.1 (Asignaciones válidas e inválidas). Note que es posible realizar operaciones aritméticas al mismo tiempo que se hace una asignación a variable. Este tipo de asignaciones es válida. Sin embargo, una asignación como esta:

$$5 \leftarrow x \qquad \text{/* \textbf{inválida: no se puede asignar a una constante una variable} */}$$

es **inválida**, ya que estaríamos diciendo que a un valor constante (5) le asignamos una variable (x). Esto no tiene sentido y por lo tanto es una asignación inválida.

De manera similar, en el bloque de asginaciones siguiente, la tercera asignación es inválida:

$$\begin{aligned} x &\leftarrow 0 \qquad \text{/* \textbf{válida: asignación correcta: x=0} */} \\ y &\leftarrow x \qquad \text{/* \textbf{válida: asignación correcta: y=0} */} \\ z &\leftarrow n \qquad \text{/* \textbf{inválida: asignación incorrecta: z=?} */} \end{aligned}$$

No se puede asignar a una variable otra que no haya sido inicializada previamente.

Pasaremos ahora a otro tema muy importante en el que las variables juegan un papel fundamental. El ejercicio 1.4.1 nos mostró una secuencia en la que las decisiones importantes se tomaron durante la etapa de análisis del problema. Sin embargo, en muchas ocasiones debemos tomar decisiones sobre el curso de las acciones a seguir dentro del flujo del algoritmo. En otras palabras, hay problemas para los que es necesario **controlar el flujo de ejecución**— el estado de cosas— dentro de un algoritmo.

1.5. Control de ejecución de un algoritmo

Existen dos formas de controlar la ejecución de un algoritmo: la bifurcación y la iteración. En ambos casos el uso de variables permite construir condiciones para continuar con un curso de acción, o para realizar otro curso de acción distinto.

1.5.1. Bifurcación

La bifurcación, o selección, de ejecución en un algoritmo, es un mecanismo de **decisión** sobre qué acción ejecutar, con base en alguna **condición**.

Para ilustrar esta operación resolvamos el siguiente problema.

Problema 1.5.1: Saber si un número es par

Escribir un algoritmo que lea un número dado por el usuario y diga si el número es par.

Lo primero que se debe observar es que un número puede ser par o no serlo (número impar), y que el problema es escribir que el número es par **sólo cuando lo es**; en otras palabras, el problema implica una **decisión**. De esta forma, si el número es par, el algoritmo debe seguir un **curso de acciones distinto** de cuando el número es impar.

Más adelante analizaremos con mayor detalle una forma de construir algoritmos partiendo de un objetivo. Por ahora, diremos que el objetivo es replantear el problema de forma a identificar **el cálculo** que se requiere hacer.

Por definición, un número es par si es múltiplo de 2. Específicamente, un número n es par, si existe otro número p, tal que:

$$n = 2 \times p,$$

y por lo tanto $p = n \div 2$ es entero, es decir, la división es exacta. En otras palabras, si n es entero, entonces:

$$r = residuo(n \div 2) = 0.$$

El ALGORITMO 1.5.1 resuelve el problema.

Algoritmo 1.5.1. Algoritmo para resolver el PROBLEMA 1.5.1

```
1  Inicio
2  n ← un número dado por el usuario
3  SI residuo (n ÷ 2) = 0:          /* Instrucción condicional: punto de decisión */
4        Imprime "El número es par."   /* Se ejecuta sólo si la condición se cumple */
5  Fin
```

El algoritmo funciona de la siguiente forma.

1. La línea 1 marca el punto de entrada al algoritmo.
2. la segunda línea muestra la acción de **asignación a la variable** n de "un número dado por el usuario".
3. La instrucción 3 es algo nuevo. Se trata de una **INSTRUCCIÓN CONDICIONAL**. Se trata de un **punto de decisión**, donde se verifica la condición: $\mathbf{residuo}\ (\mathbf{n} \div \mathbf{2}) = \mathbf{0}$. Si el resultado de la operación residuo es 0, decimos que **la condición se cumple** o que la condición **es verdadera**[3]. En este caso, el algoritmo continúa su ejecución en la línea 4.
4. Esta línea sólo se ejecuta si la condición evaluada en la línea 3 se cumple. Note que la instrucción en la línea 4 tiene una identación (un espacio hacia adentro) mayor. Esto significa que esa instrucción forma parte de lo que se llama **el bloque condicional**, es decir todas las cosas que queremos que sucedan si la condición se cumple.

Note que el algoritmo termina en la línea 5. Esto es así se cumpla o no la condición en la línea 3. La única diferencia es que si se cumple, se ejecutará la línea 4, y pasaremos inmediatamente después a la línea 5. Si no se cumple la condición en la línea 3, el algoritmo pasa de esta línea directamente a la línea 5 donde el algoritmo termina.

La siguiente Figura 1.6 muestra la estructura básica de la bifurcación.

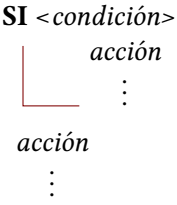


Figura 1.6: *Estructura básica de la bifurcación.*

Lo que muestra la Figura 1.6 es que, si la **condición se cumple (es verdadera)**, el flujo de ejecución sigue el bloque que está debajo de la **instrucción SI**, indicado por la línea terminando en ángulo recto; decimos que las acciones del bloque están **identadas**. Una vez que termina este flujo, la ejecución continúa a la misma altura de la instrucción **SI**. Si la **condición no se cumple (es falsa)**, la ejecución evita el bloque debajo de SI, continuando directamente con las acciones justo debajo del bloque **SI**. De esta forma, decimos que **la ejecución del algoritmo se bifurca dentro del bloque si la condición es verdadera**.

[3] Cuando decimos que una condición o variable es `verdadera` (resp. `falsa`) estamos diciendo que tiene un valor de 1 (resp. 0). Estos son `valores de verdad` y son muy importantes en programación. Si deseas saber más sobre esto puedes consultar el Apéndice A.

Observación 1.2 (COMPARACIÓN como CONDICIÓN). Note que la **condición** de la línea 3 en el Algoritmo 1.5.1 es una operación de **comparación**. Esto se da con mucha frecuencia, por lo que es importante tomar nota de los operadores típicos de comparación que son: $\{=, \neq, >, <, \geqslant, \leqslant\}$.

Se muestran algunos ejemplos con sus respectivos valores de verdad:

Igualdad ($=$)		Desigualdad ($\neq$)	
$2 = 1 + 1$	VERDADERO	$2 \neq 1 + 1$	FALSO
$3 = 4$	FALSO	$3 \neq 4$	VERDADERO
$4 = 4$	VERDADERO	$4 \neq 8 \div 2$	FALSO
Mayor que ($>$)		Menor que ($<$)	
$2 > 1 + 1$	FALSO	$2 < 2 + 1$	VERDADERO
$4 > 4$	FALSO	$3 < 4$	VERDADERO
$4 > 3$	VERDADERO	$4 < 8 \div 2$	FALSO
Mayor o igual que ($\geqslant$)		Menor o igual que ($\leqslant$)	
$2 \geqslant 1 + 1$	VERDADERO	$2 \leqslant 2 + 1$	VERDADERO
$4 \geqslant 4$	VERDADERO	$3 \leqslant 2$	FALSO
$4 \geqslant 5$	FALSO	$4 \leqslant 8 \div 2$	VERDADERO

Observación 1.3 (ASIGNACIÓN **NO ES** COMPARACIÓN)**.** Se debe hacer notar que la asignación a una variable, como por ejemplo escribir $x \leftarrow 0$ no es una comparación, por lo tanto una asignación **no se puede usar como condición**.

El algoritmo 1.5.1 plantea una pregunta importante: ¿Qué sucedería si quisiéramos elegir entre un bloque de acciones, en caso de que la condición fuera verdadera, y otro bloque distinto, en caso contrario, es decir cuando la condición fuera falsa? Por ejemplo, en este caso, si el número no fuera par.

El siguiente problema ilustra esta situación:

Problema 1.5.2: PROBLEMA 1.5.1 extendido

Escribir un algoritmo que lea un número dado por el usuario y diga si el número es par o impar.

Nótese que esta vez debemos decidir entre dos posibles condiciones: cuando el número es par, debemos enviar un mensaje; cuando el número es impar, debemos enviar otro mensaje, distinto del anterior. Como un número no puede ser par o impar a la vez, estas dos condiciones son **mutuamente excluyentes**.

En palabras, desearíamos poder escribir algo como:

- SI A se cumple (es verdadera), entonces ejecuta X;
- SI A no se cumple (es falsa), entonces ejecuta Y.

El ALGORITMO 1.5.2 resuelve este problema.

Algoritmo 1.5.2. Algoritmo para resolver el PROBLEMA 1.5.2

```
1  Inicio
2  n ← un número dado por el usuario
3  SI residuo (n ÷ 2) = 0:
4        Imprime "El número es par."
5  DE LO CONTRARIO:
6        Imprime "El número es impar."
7  Fin
```

En este caso la línea 3 es una instrucción condicional, que, en caso de que la CONDICIÓN SEA VERDADERA, se ejecutará la línea 4; pero, en caso de que la CONDICIÓN SEA FALSA, lo cuál **se confirma con la línea 5**, se ejecutará la línea 6. Así, la bifurcación está dada por un bloque **SI–DE_LO_CONTRARIO** como se muestra en la Figura 1.7.

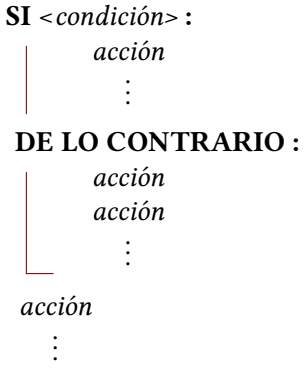


Figura 1.7: *Estructura de la bifurcación: SI - DE_LO_CONTRARIO.*

En este caso, si la **condición SI se cumple** (es verdadera), el algoritmo ejecuta lo que está debajo de ésta, y **no ejecuta** lo que está debajo de "**DE LO CONTRARIO**".

Inversamente, si la **condición no se cumple** (es falsa), **la condición DE LO CONTRARIO es verdadera automáticamente**, y entonces el algoritmo ejecuta lo que está debajo de ésta, sin pasar por lo que está debajo de la primera. Una vez que el algoritmo llega a la última acción del conjunto de acciones debajo de la primera condición, o debajo de la segunda, continúa su curso de ejecución inmediatamente después de que concluye el bloque **SI–DE_LO_CONTARIO**.

Observación 1.4. Hacemos notar que es posible "combinar" bloques SI en forma de anidamientos o de secuenciación de bloques de decisión, pero cada bloque debe estar correctamente definido en su estructura. La siguiente tabla muestra esquemas de anidamiento y secuenciación válidos e inválidos.

<table>
<tr><th>Construcciones CORRECTAS</th><th>Construcciones INCORRECTAS</th></tr>
<tr><td colspan="2">• Las acciones entre '[ ]' son acciones opcionales; es decir, pueden o no existir.
• Las acciones subrayadas son obligatorias; es decir, debe haber al menos una.</td></tr>
<tr><td>SI <condición> :
[acción]
SI <condición> :
<u>acción</u>
⋮
[acción]
⋮
[acción]
⋮</td><td>SI <condición> :
<u>acción</u>
SI <condición> :
acción
⋮
[acción]
⋮</td></tr>
<tr><td>SI <condición> :
<u>acción</u>
⋮
SI <condición> :
<u>acción</u>
⋮
[acción]
⋮</td><td>SI <condición> :
SI <condición> :
<u>acción</u>
⋮
[acción]
⋮</td></tr>
</table>

1.5.2. Iteración, ciclo o repetición

Así como pueden guardar información relevante para un problema, las variables también pueden ayudar a controlar la ejecución de un algoritmo. Otra forma básica es la iteración, también llamada **bucle** o **ciclo**. Un ciclo permite la ejecución reiterada de una secuencia de acciones, de tal forma que dicha secuencia se ejecuta una y otra vez, sin que sea necesario reescribirlas una y otra vez. Por ejemplo, tomemos el clásico verso infantil del elefante que se balanceaba. Dice algo como:

1 elefante se balanceaba sobre la tela de una araña.
Como veían que resistía fueron a buscar otro elefante.
2 elefantes se balanceaban sobre la tela de una araña.
Como veían que resistía fueron a buscar otro elefante
3 elefantes se balanceaban sobre la tela de una araña.
....

Es evidente que si deseamos que un algoritmo escriba estos versos resultaría tedioso, e ineficiente, escribirlos uno por uno. Lo ideal sería escribir una sola vez las líneas que no cambian, y usar una variable para modificar

el valor que cambia. Para el ejemplo, la idea es usar una variable **x** en un algoritmo donde se escriba una sola vez el verso:

> **x** *elefantes se balanceaban sobre la tela de una araña.*
> *Como veían que resistía fueron a buscar otro elefante.*

y se itere, tantas veces como se desee, sobre este verso, cambiando el valor de **x** en cada iteración. El algoritmo podría ser como el mostrado en **ALGORITMO** 1.5.3:

Algoritmo 1.5.3. Algoritmo para repetir 3 veces un verso

```
1  Inicio
2  Imprime "1 elefante se balanceaba..."
3  x ← 2                                  /* Inicialización de x */
4  SI x ⩽ 4:                              /* Inicia ciclo de repetición */
5      Imprime x "elefantes se balanceaban ..."
6      x ← x + 1                          /* Cambio de estado */
7      REPITE desde el paso 4             /* Instrucción de bucle */
8  Fin
```

El algoritmo funciona de la siguiente forma:

1. En la línea 1 se marca el inicio (el punto de entrada) y en la 8 el final (el punto de salida).
2. Se ejecuta la línea 2, por lo que se imprime en pantalla el texto: "1 elefante se balanceaba...".
3. Luego se ejecuta la línea 3, donde se realiza una asignación del valor 2 a la variable x. Esto significa que en la línea 3 **la variable x se inicializa con el valor 2**.
4. En la siguiente línea, se valida la condición del bucle o ciclo, que es la comparación $x \leqslant 4$. Esta es **VERDADERA** por lo que entramos al bloque condicional.
5. En la línea 5 se imprime el valor de x, que es 2, junto con el verso en plural.
6. La línea 6 modifica el valor que contiene x, sumándole un 1 y almacenando el resultado de la suma de vuelta en x. Por lo tanto, x **contiene ahora el valor 3**.
7. (**ITERACIÓN 1**) La línea 7 construye **el bucle, el ciclo de repetición**. En este caso, **se instruye repetir la ejecución desde el paso 4**, por lo que el algoritmo nos lleva de vuelta a verificar la condición. Como x es ahora 3 la condición $x \leqslant 4$ es **VERDADERA** por lo que continuamos con la línea 5.
8. En la línea 5 se imprime el valor de x, que es ahora 3, junto con el verso en plural.
9. Continúa en la línea 6, donde se modifica nuevamente el valor de x que ahora es 4.
10. (**ITERACIÓN 2**) La línea 7 vuelve a cerrar el bucle sobre la línea 4, donde la condición es **VERDADERA**, por lo que el algoritmo continúa con la línea 5, sólo que ahora el valor de x es 4.

11. Continúa en la línea 6, donde se modifica nuevamente **el valor de x que ahora es 5**.

12. (**Iteración 3**) La línea 7 vuelve a cerrar el bucle sobre la línea 4. Como $5 \leqslant 4$ es **falsa**, decimos que **"se rompe el ciclo"** (ya no se entra de nuevo) y el algoritmo continúa en la línea 8 llegando a su fin.

13. Al final el ciclo se repitió 3 veces y el valor final de x es 5.

La siguiente Tabla muestra lo que se conoce una **prueba de escritorio** del ALGORITMO 1.5.3:

Tabla 1.1: Prueba de escritorio para el Algoritmo 1.5.3.

Inicialización				
Imprime "1 elefante..."				
$x \leftarrow 2$				
Ciclo				
iteración	x	$x \leqslant 4$	Imprime	$x \leftarrow x + 1$
-	**2**	Sí	2 "elefantes..."	**3**
1	**3**	Sí	3 "elefantes..."	**4**
2	**4**	Sí	4 "elefantes..."	**5**
3	**5**	No		
Fin				

El ALGORITMO 1.5.3 muestra dos cosas:

1. Cómo una **variable** (**x**) permite **controlar el bucle**, es decir, el número de repeticiones.

2. Cómo es necesaria una **condición de paro**; en este caso, el hecho de que **x** *sea mayor* a 4.

Si deseas saber más sobre validación de condiciones, puedes consultar el Apéndice B.

Hacemos una última observación importante en este capítulo.

Observación 1.5 (Prioridad de operadores aritméticos). En computación, la prioridad de operadores aritméticos determina el orden en que se evalúan las operaciones en una expresión. En general, se sigue el orden de precedencia PEMDAS: Paréntesis/Llaves, Exponentes/Potencias, Multiplicación y División (de izquierda a derecha), Suma y Resta (de izquierda a derecha).

- **Prioridad de Operadores Aritméticos Comunes:**

1. Paréntesis/Llaves: Las expresiones dentro de paréntesis o llaves se evalúan primero.
2. Exponentes/Potencias: Las operaciones de exponenciación se ejecutan después de los paréntesis, pero antes de la multiplicación y división.
3. Multiplicación y División: Se evalúan de izquierda a derecha en el orden en que aparecen.
4. Suma y Resta: Se evalúan de izquierda a derecha en el orden en que aparecen, después de la multiplicación y división.

- **Ejemplo:**

En la expresión $2 + 3 \times 4$, la multiplicación 3×4 se realiza primero (prioridad 3), resultando en $2 + 12$. Luego, se realiza la suma $2 + 12$, dando como resultado final 14.

- **Uso de Paréntesis:**

Los paréntesis se utilizan para forzar una secuencia de evaluación diferente a la predeterminada. Por ejemplo, en la expresión $(2 + 3) \times 4$, la suma dentro del paréntesis se realiza primero, lo que resulta en $5 \times 4 = 20$.

1.6. Ejercicios y Problemas

1.6.1. Sistemas de Cómputo

1. ¿Cuál es la unidad fundamental de memoria ?

   a) el hexabyte
   b) el bit
   c) el Byte
   d) el pixel
   e) el Megabyte

2. Son las funciones principales de una computadora

   a) Guardar datos en la memoria, sacar datos de la memoria, hacer operaciones, controlar esas operaciones
   b) Guardar datos en la memoria, hacer operaciones, controlar esas operaciones, hacer programas
   c) Guardar datos en dispositivos externos, controlar la memoria, hacer operaciones con la memoria, controlar los dispositivos de la memoria

3. ¿Cuáles son los estados que existen para un bit?

   a) 0 y 1
   b) 0, 1, 2, 3, 4, 5, 6, y 7
   c) V y F
   d) 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9

4. ¿Cuál es el tamaño de un Byte?

   a) 2 bits (0 y 1)
   b) 8 bits
   c) 16 bits
   d) 1 Mbit
   e) ninguna de las anteriores

1.6.2. Bifurcaciones

1. El siguiente algoritmo tiene un error. ¿Cuál es?

```
1  Inicio
2  x ← 0
3  n ← 1
4  SI (n + x) ← 2:
5        x ← 1
7  Fin
```

2. El siguiente algoritmo tiene un error. ¿Cuál es?

```
1  Inicio
2  2 ← x
3  n ← 1
4  SI (n + x) ⩾ 2:
5        x ← 1
7  Fin
```

3. El siguiente algoritmo tiene un error. ¿Cuál es?

```
1  Inicio
2  x ← 2
3  n ← z
4  SI (n + x) ⩾ 2:
5        x ← 1
7  Fin
```

4. El siguiente algoritmo tiene algunos errores. Identifique las líneas donde ocurren y diga en qué consiste cada uno.

```
1  Inicio
2  x ← 2
3  n ← 3
4  SI x ⩽ n:
5  x ← x, 1
6  SI 2 ⩽ (n + x):
7        x = 1
8  Fin
```

5. ¿Cuál es el valor de la variable `suma` que se imprime al terminar el siguiente algoritmo?

```
1  Inicio
2  suma ← 0
3  n ← 6
4  SI n ÷ 3 = 2:
5        suma ← suma + (n ÷ 2)
6  Imprime suma
7  Fin
```

6. Considere el problema: "*Dados tres números* a, b *y* c *determine si* $c = a + b$ o no" y observe el algoritmo siguiente que pretende resolver este problema. Responda a lo que se pide.

```
1   Inicio
2   a ← dato dado por el usuario
3   b ← dato dado por el usuario
4   c ← dato dado por el usuario
5   SI c = a + b:
6         x ← 1
7   SI x = 0:
8         Imprime "c es igual a a + b"
9   DE LO CONTRARIO:
10        Imprime "c no es igual a a + b"
11  Fin
```

a) Suponga que los valores de a, b y c son los siguientes: $c = 5$, $a = 3$ y $b = 2$. Por lo tanto se cumple la condición de la línea 5 y la línea 6 se ejecuta.

   1) Diga cuál es el problema en lo que sigue del algoritmo.
   2) Corrija este problema.

b) Suponga ahora que $c = 7$, $a = 3$ y $b = 2$. Diga que ocurre a partir de la línea 5.

c) ¿Cómo resolvería el problema que surge de la situación anterior?

7. Observe el siguiente algoritmo y responda lo que se pide.

```
1   Inicio
2   a ← entero positivo
3   b ← entero positivo
4   x ← 0
5   SI residuo(a ÷ b) = 0:
6       x ← 1
7       SI b = a × 3:
8           x ← 2
9   Imprime x
10  Fin
```

a) Diga cuál es valor de x al terminar el algoritmo si $b = 27$ y $a = 9$.

b) Diga cuál es valor de x al terminar el algoritmo si $a = 27$ y $b = 9$.

c) Diga qué valores deben tener a y b para que $x = 2$.

8. Diga cuál es valor de x al terminar el siguiente algoritmo.

```
1   Inicio
2   a ← 32
3   b ← 4
4   x ← 8
5   SI x = a ÷ b:
6       SI b = a × 2:
7           x ← x + b
8       DE LO CONTRARIO:
9           x ← x + a
10  Imprime x
11  Fin
```

9. Analice el siguiente algoritmo y responda lo que se indica:

```
1   Inicio
2   m ← 12
3   n ← 4
4   x ← 24
5   SI (x ÷ (m − 4) > 1):
6       SI (n × 2 = m):
7           x ← x + m − n
8       DE LO CONTRARIO:
9           x ← x − m ÷ n
10  DE LO CONTRARIO:
11      x ← x × 2 + 1
12  Imprime x
13  Fin
```

a) Diga qué valor se imprime.

b) Diga que valor se imprime si la condición en la línea 5 se escribe: SI $(x \div (m - 4) < 1)$:

10. Se desea resolver el siguiente problema: "*Dado un número entero positivo, escribir Fizz si el número es múltiplo de 3, escribir Buzz si es múltiplo de 5 o escribir el número si ninguna de las condiciones anteriores se cumple*". Observe los siguientes algoritmos y responda lo que se pide.

	Algoritmo 1		Algoritmo 2
1	Inicio	1	Inicio
2	$x \leftarrow$ entero positivo	2	$x \leftarrow$ entero positivo
3	SI `residuo`$(x \div 3) = 0$:	3	SI `residuo`$(x \div 3) = 0$:
4	Imprime "Fizz"	4	Imprime "Fizz"
5	DE LO CONTRARIO:	5	SI `residuo`$(x \div 5) = 0$:
6	SI `residuo`$(x \div 5) = 0$:	6	Imprime "Buzz"
7	Imprime "Buzz"	7	DE LO CONTRARIO:
8	DE LO CONTRARIO:	8	Imprime x
9	Imprime x	9	Fin
10	Fin		

a) ¿Cuál de los dos algoritmos resuelve el problema planteado?

b) Suponga que al planteamiento del problema se le agrega lo siguiente:"*En caso de que el número dado sea múltiplo de 3 y de 5 escribir FizzBuzz*".¿Cuál de los dos algoritmos resuelve el problema ahora?

c) Escriba la secuencia que se imprime si x toma como valor cada uno de los siguientes números: $\{1, 2, 3, 4, 5, 8, 9, 12, 15, 50, 60\}$.

1.6.3. Ciclos

1. ¿Cuántas veces se repite el ciclo definido de las líneas 5 a 8?

```
1  Inicio
2  suma ← 0
3  n ← 1
4  u ← 10
5  SI n < u:
6      suma ← n
7      n ← n + 1
8      REPITE desde el paso 5
9  Imprime suma
10 Fin
```

2. Para el algoritmo del Ejercicio 1, responda lo siguiente:

a) ¿Qué pasaría si la línea 7 no existiera?

b) Suponga que la condición en la línea 5 cambia por $n > u$. ¿Cuántos ciclos se ejecutan?

c) ¿Qué pasaría si ahora cambiamos la condición por $u > n$?

3. ¿Qué valor se imprime en la línea 9 del Ejercicio 1.

4. Suponga que la línea 6 del algoritmo del Ejercicio 1 se cambia por la siguiente:

   6 suma $\leftarrow$ suma $+ n$

   ¿Qué valor se imprime ahora en la línea 9?

5. Modifique el valor de u en el algoritmo del Ejercicio 1 para que el ciclo se realice 15 veces. ¿Cuál es el valor de n al terminar el ciclo?

6. Para el algoritmo del Ejercicio 1, suponga que se intercambian de lugar las líneas 6 y 7. Responda lo siguiente:

   a) ¿Cambia el número de ciclos que se ejecutan?

   b) ¿Qué valor se imprime en la línea 9?

7. Observe el algoritmo siguiente y conteste lo que se pide:

```
1   Inicio
2   cuenta ← 0
3   x ← entero positivo
4   n ← 2
5   SI n ⩽ x:
6       SI residuo(x ÷ n) = 0:
7           cuenta ← cuenta + 1
8       n ← n + 1
9       REPITE desde el paso 5
10  Imprime cuenta
11  Fin
```

   a) ¿Cuál es el valor de `cuenta` al final del algoritmo si $x = 5$ y $x = 4$?

   b) Determine el valor de `cuenta` al final del algoritmo para $x = \{7, 11\}$ y compruebe que en ambos casos `cuenta` $= 1$.

   c) Determine el valor de `cuenta` al final del algoritmo para $\{15, 21\}$ y compruebe que en ambos casos `cuenta` > 1.

   d) A partir de la información anterior, trate de inferir qué está calculando el algoritmo.

8. Observe el siguiente algoritmo y responda lo que se pide.

```
1   Inicio
2   x ← entero positivo
3   n ← 0
4   incognita ← n
5   SI x ⩾ n:
6       n ← n + 2
7       incognita ← incognita + n × n
8       REPITE desde el paso 5
9   Imprime incognita
10  Fin
```

a) Suponga que en el algoritmo la línea 8 se cambia por:

```
8       REPITE desde el paso 4
```

¿Obtenemos el mismo resultado?

b) Explique por qué este cambio es problemático.

9. Suponga que en el algoritmo del Ejercicio 8 se elimina la línea 6 y la línea 5 se reescribe de la siguiente forma:

```
5   SI x ⩾ n + 2:
```

¿Funcionaría igual que en el Ejercicio 8? Explique.

10. Suponga que se dispone de una serie de números enteros que van del 10 al 20 **inclusive**. Usted desea saber cuántos números pares hay en ese rango. ¿Cuál de los siguientes algoritmos resuelve el problema?

Algoritmo 1	
1	Inicio
2	cuenta ← 0
3	$i \leftarrow 10$
4	$n \leftarrow 20$
5	SI $i \leqslant n$:
6	SI `residuo`$(i \div 2) = 0$:
7	cuenta ← cuenta + 1
8	$i \leftarrow i + 1$
9	REPITE desde el paso 5
10	Imprime cuenta
11	Fin

Algoritmo 2	
1	Inicio
2	cuenta ← 0
3	$i \leftarrow 10$
4	$n \leftarrow 20$
5	SI $i \leqslant n$:
6	SI `residuo`$(i \div 2) = 0$:
7	cuenta ← cuenta + 1
8	$i \leftarrow i + 1$
9	REPITE desde el paso 5
10	Imprime cuenta
11	Fin

11. Suponga que el Algoritmo 1 del Ejercicio 10 se escribe ahora de la siguiente forma. Diga que ocurre ahora con el algoritmo.

```
1   Inicio
2   cuenta ← 0
3   i ← 10
4   n ← 20
5   SI i ⩽ n:
6       SI residuo(i ÷ 2) = 0:
7           cuenta ← cuenta + 1
8           i ← i + 1
9       REPITE desde el paso 5
10  Imprime cuenta
11  Fin
```

12. Modifique el algoritmo correcto del Ejercicio 10 para saber cuántos números pares hay en el rango a a b, modificando únicamente instrucciones de las líneas 5 a 9, para los siguientes casos:

    a) Excluyendo el valor inicial a

    b) Excluyendo el valor final b

    c) Excluyendo ambos

2 Introducción al Análisis de Problemas y el Diseño de Algoritmos

Jorge Hermosillo Valadez

Diseñar algoritmos para resolver un problema particular puede resultar una tarea compleja. Además de involucrar herramientas del pensamiento lógico para identificar variables y relaciones lógicas entre ellas, el diseño de algoritmos requiere la organización en el tiempo de la secuencia de pasos a seguir para alcanzar un objetivo o resolver un problema.

Aún con problemas que se podrían categorizar como "pequeños", cada paso de un algoritmo representa una meta a alcanzar antes de continuar con la siguiente. En este sentido, el diseño de algoritmos pasa primeramente por el análisis del problema a resolver, que se beneficia en muchas ocasiones de la identificación de subproblemas.

Es así que, en este capítulo proponemos una estrategia para analizar problemas en subproblemas, con el fin de identificar los elementos del algoritmo y las acciones concretas que habría que organizar en el tiempo para alcanzar un objetivo.

2.1. Analizando problemas

Comenzamos este capítulo tratando de analizar el siguiente problema.

Ejercicio 2.1.1: Resuelve el siguiente problema

Escribe un algoritmo para resolver el problema 2.1.1.

Problema 2.1.1: Rotula correctamente

Hay tres platos de fruta en un estante cubiertos por cajas (no puedes ver que plato está en cada caja). Un plato contiene sólo manzanas, otro plato contiene sólo naranjas y otro plato contiene manzanas y naranjas. Cada caja tiene enfrente uno de los siguientes rótulos: MANZANAS, NARANJAS, o MANZANAS Y NARANJAS. Sin embargo, cada caja tiene el rótulo equivocado. Tu misión es seleccionar una caja y tomar una fruta. Al hacer esto y con la información anterior, ¿puedes rotular correctamente cada plato?

Veamos ahora una estrategia para analizar problemas.

2.2. Enfoque de George Polya

Para encontrar una solución a un problema, es necesario analizarlo primero. Intentar resolver problemas es una tarea que requiere de creatividad e ingenio. La pregunta es: ¿se puede guiar un proceso creativo? Afortunadamente, la respuesta es sí. Existen varios esfuerzos que se han emprendido en este sentido, que son dignos de ser tratados en una obra aparte[1]. En este capítulo, damos algunas pautas que han dado buenos resultados en el pasado. Comenzaremos con un enfoque sistemático que ha sobrevivido a lo largo del tiempo, por tratarse de un método intuitivamente sencillo y muy útil. Se trata del método propuesto por George Polya[2].

El libro de George Polya toma los principios del razonamiento heurístico, aplicables a problemas en todos los campos, y muestra cómo se pueden aplicar los principios correctos de planteamiento y resolución de problemas, con una participación activa y no con la mera aceptación de los resultados obtenidos por otros. En este sentido, más allá de lograr resolver un problema, se trata de adueñarse del proceso de desarrollo de una solución al problema, con el fin de extender la metodología a otras situaciones. En este sentido, Polya hace ver a las matemáticas como un verdadero proceso de invención e inducción, donde se experimenta de manera iterativa para construir analogías que hagan factible la resolución de problemas.

Para Polya, todo intento por resolver un problema debe seguir el siguiente proceso:

1. Entiende el problema
2. Configura un plan
3. Ejecuta el plan
4. Revisa y extiende

2.2.1. Entender el problema

Para entender un problema Polya sugiere hacerse preguntas básicas como:

- ¿Cuál es la incógnita?
- ¿Cuáles son los datos?
- ¿Cuál y cómo es la condición?
- ¿Es la condición suficiente para determinar la incógnita?

A través de estas preguntas es posible contextualizar el problema y entender su naturaleza. En general, entender el problema es una de las etapas más difíciles, y muchas veces se proponen procedimientos antes de entender la naturaleza del problema. A medida que los problemas son más complicados, esta etapa es crucial. Desde el punto de vista computacional, podemos decir que estas preguntas se traducen en las siguientes:

[1] Un ejemplo es TRIZ. La teoría de la solución de problemas de inventiva (TRIZ, por sus siglas en ruso) fue desarrollada por Genrich Altshuller y sus colegas en la ex URSS a comienzos de 1946, y, actualmente, se desarrolla y practica en todo el mundo.

[2] Polya, G. (1945). How to solve it; a new aspect of mathematical method. Princeton University Press.

- ¿Cuál es el objetivo?
- ¿Cuáles son los datos de entrada?
- ¿Cuál es la salida deseada?
- ¿Cuál es la condición?
- ¿Es la condición suficiente para saber cómo procesar la entrada?
- ¿Existe alguna contradicción en la condición?

Estas preguntas podrían ser o no suficientes para resolver el problema. Pero no hay que olvidar que en esta etapa estamos en el *qué*, no en el *cómo*. Lo importante aquí es plantearse las preguntas que nos hagan sentido para **entender qué es lo que tenemos que resolver**. Para ello, otra estrategia es escribir el problema en nuestras palabras, o incluso hacer diagramas, dibujos o cualquier cosa que nos ayude a dar sentido al problema desde nuestra perspectiva. De esta forma, nos podremos dar cuenta de:

- Qué información tengo: *lo que sí sé (especifica)*.
- Qué información me falta: *lo que no sé (especifica)*.
- Qué debo lograr: *cómo sé que resolví el problema (especifica)*.
- Qué cosas deben ocurrir y en qué orden: *cómo debería funcionar*.

2.2.2. Configurar un plan

Una vez entendida la naturaleza del problema, Polya sugiere hacerse más preguntas para configurar un plan. Retomamos aquí algunas de ellas:

- ¿Te has encontrado con un problema semejante? ¿O has visto el mismo problema planteado en forma ligeramente diferente?
- ¿Conoces algún problema relacionado con éste?
- ¿Puedes utilizarlo? ¿Puedes utilizar su resultado? ¿Puedes emplear su método? ¿Te hace falta introducir algún elemento auxiliar a fin de poder utilizarlo?
- ¿Puedes enunciar al problema de otra forma? ¿Puedes plantearlo en forma diferente nuevamente?
- Si no puedes resolver el problema propuesto, trata de resolver primero algún problema similar. ¿Puedes imaginarte un problema análogo un tanto más accesible? ¿Un problema más general? ¿Un problema más particular? ¿Un problema análogo? ¿Puedes resolver una parte del problema?

Estas preguntas están relacionadas con el *pensamiento lateral*[3]. La idea es salirse de la "ruta vertical", de lo que podría llamarse "el camino habitual": se trata de abrir las posibilidades. Una vez que se ha divisado una estrategia de solución, es necesario plantear la ruta específica que habría que seguir para validar la hipótesis de que se ha resuelto el problema.

Hablamos bien de **una** solución y **no la** solución al problema. Existen muchas soluciones posibles, y no existe una sola ruta para resolver problemas (científicos, tecnológicos, etc.). Podemos hablar de soluciones óptimas, pero cualquier solución que no es óptima es una **heurística**, es decir, un procedimiento específico que proporciona una solución adecuada al problema.

2.2.3. Ejecutar el plan, Revisar y Extender

Podría parecer obvio, pero al ejecutar el plan de la solución, es necesario comprobar cada uno de los pasos: ¿Puedes ver claramente que el paso es correcto? ¿Puedes demostrarlo? Por último, una vez hecho esto, es necesario verificar y extender: ¿Puedes verificar el resultado? ¿Puedes comprobar el razonamiento? ¿Puedes obtener el resultado en forma diferente? ¿Puedes verlo de golpe? ¿Puedes emplear el resultado o el método en algún otro problema?

Si habiendo seguido estos pasos, no se logra encontrar una solución al problema, la sugerencia es: intenta descomponer el problema.

2.3. Diseñando algoritmos

Veamos dos ejemplos donde aplicaremos los conceptos vistos hata ahora.

2.3.1. Contar números pares en un rango

En este apartado analizaremos el siguiente problema:

Problema 2.3.1: Contar números pares en un rango

Escribir un algoritmo que cuente cuántos números pares hay en un rango predeterminado de números enteros.

Usaremos cada una de las estrategias propuestas para proponer al final un algoritmo.

▸ Análisis de George Polya

Usaremos ahora la estrategia de Polya para adentrarnos un poco más en nuestra comprensión del problema. La Tabla 2.1 retoma los elementos del árbol para estructurar mejor el análisis del problema.

[3] De Bono Edward. 2006. El Pensamiento Lateral. Editorial Paidós Ibérica S.A.

Tabla 2.1: Análisis de Polya para el PROBLEMA 2.3.1

Pregunta	Respuesta (análisis)
¿Cuál es el objetivo?	▪ Contar números pares en un rango
¿Cuáles son los datos de entrada?	▪ Un rango de números enteros
¿Cuál es la salida deseada?	▪ El conteo de los números pares dentro del rango
¿Cuál es la condición?	▪ Necesitamos conocer los números que definen el rango y saber cuando un número es par.
¿Es la condición suficiente para saber cómo procesar la entrada?	▪ Sí, a condición de conocer los números que definen el rango. ▪ Suponemos que son números enteros.
Qué información tengo	▪ Los números que definen el rango
Qué información me falta	▪ Saber cuando un número es par ▪ Llevar un contador de números pares
Qué debo lograr	▪ Contar los números pares
Qué cosas deben ocurrir y en qué orden	1. Leer los números del rango. 2. Ordenarlos para determinar el menor de ambos. 3. Leer uno a uno los números, comenzando por el menor. 4. Determinar si el número leído es par. 5. Actualizar un contador en caso de que el número leído sea par. 6. Devolver el valor final del contador.
¿Existe alguna contradicción en la condición?	▪ No, si nuestras suposiciones acerca de que se trata de números enteros es correcta.

La Tabla 2.1 nos permite escribir el ALGORITMO 2.3.1

Algoritmo 2.3.1. Algoritmo para resolver el PROBLEMA 2.3.1

```
1  Inicio
2  c ← 0
3  a ← primer número del rango
4  b ← segundo número del rango
5  SI a > b:                    /* Ordenamos para que a<b */
6      j ← a
7      a ← b
```

```
8        b ← j
9    x ← a                          /* x tiene el valor más pequeño */
10   SI residuo(x ÷ 2) = 0:          /* si x es par */
11       c ← c + 1                  /* actualizamos el contador de números pares */
12   x ← x + 1                      /* vamos al siguiente número */
13   SI x ⩽ b:                      /* si faltan números por recorrer */
14       REPITE desde el paso 10
15   Imprime c
16   Fin
```

En este algoritmo hacemos notar dos cosas:

1. Note que nos aseguramos de tener el número más pequeño dado por el usuario en a, y el mayor en b (líneas 5 a 8).
2. La línea 13 condiciona la iteración del ciclo:

   Si x no alcanza todavía el valor de b, entonces $REPITE$ desde el paso 10 donde preguntamos si x es par.

   En otras palabras, el bucle sólo se repetirá un cierto número de veces. ¿Cuántas? Pues el número que se lleve en tanto la condición 13 se siga cumpliendo. En este caso, la condición es que el valor de x sea menor o igual al valor de b.

2.3.2. Sumar los dígitos de un número

Problema 2.3.2: Sumar los dígitos de un número

Escribir un algoritmo que sume los dígitos de un número entero positivo N. Por ejemplo, si $N = 521$ el algoritmo debe imprimir $5 + 2 + 1 = 8$.

Mostramos a continuación el algoritmo. Dejamos como ejercicio para el lector el análisis de Polya.

Algoritmo 2.3.2. Algoritmo para resolver el PROBLEMA 2.3.2

```
1    Inicio
2    suma ← 0                          /* Incialización de suma */
3    N ← leer el número entero positivo
4    R ← residuo(N ÷ 10)               /* R tiene siempre el valor a sumar */
5    N ← parteEntera(N ÷ 10)           /* Actualizamos el valor de N */
6    suma ← suma + R
7    SI N ≠ 0:
8        REPITE desde el paso 4        /* Si N es 0 entonces terminamos */
9    Imprime suma
10   Fin
```

En este algoritmo hacemos notar dos cosas: 1) el hecho de que el estado del programa cambia con cada iteración, dado que el valor de N se actualiza en cada iteración, con la parte entera de la división entre 10, observando que antes de hacer esta actualización calculamos el valor de R con el valor de N del ciclo anterior; 2) la importancia de llevar el control preciso del flujo con las líneas 5 y 7, ya que, como vimos sólo es necesario sumar el valor de R y continuamos mientras nuestra condición de paro no se cumpla (N=0), de lo contrario repetimos el ciclo mientras el valor de N siga siendo no nulo.

La siguiente Tabla muestra una **prueba de escritorio** del ALGORITMO 2.3.2 para el caso $N = 1005$:

Tabla 2.2: Prueba de escritorio para el Algoritmo 1.5.3 con $N = 1005$. Resultado esperado: 6

Inicialización				
$suma \leftarrow 0$				
$N \leftarrow 1005$				
Ciclo				
iteración	**R** $\leftarrow$ `residuo`$(N \div 10)$	**N** $\leftarrow$ `parteEntera`$(N \div 10)$	**suma** $\leftarrow suma + R$	$N \neq 0$
-	`r`$(1005 \div 10)$: **5**	`pE`$(1005 \div 10)$: **100**	$0 + \mathbf{5} = 5$	Sí
1	`r`$(100 \div 10)$: **0**	`pE`$(100 \div 10)$: **10**	$5 + \mathbf{0} = 5$	Sí
2	`r`$(10 \div 10)$: **0**	`pE`$(10 \div 10)$: **1**	$5 + \mathbf{0} = 5$	Sí
3	`r`$(1 \div 10)$: **1**	`pE`$(1 \div 10)$: **0**	$5 + \mathbf{1} = \mathbf{6}$	**No**
Imprime $suma$				
6				
Fin				

2.3.3. Adivina el número

Ejercicio 2.3.1: Resuelve el siguiente problema

Intenta resolver el siguiente problema

Problema 2.3.3: Adivina el número

Imagina que tenemos una lista ordenada de números, por ejemplo, del 1 al 100. Una persona elige en secreto uno de esos números, lo escribe en un papel y lo guarda en una caja. Nuestro desafío es adivinar qué número eligió. Para lograrlo, podemos ir proponiendo números. Cada vez que hacemos una propuesta, la persona nos dirá si el número que dijimos es:

- mayor que el número secreto,
- menor que el número secreto, o
- exactamente el número que ella eligió.

Usando esta información, ¿podemos encontrar el número correcto?

Primero exploremos la estrategia más simple.

Estrategia número uno

Para adivinar el número seguiremos los siguientes pasos:

- Empieza proponiendo el primer elemento de la lista, en nuestro caso el 1.
- Si no es igual al número secreto avanzamos al siguiente elemento en la lista.
- Repite el proceso hasta adivinar el número.

Estrategia número dos

En está segunda estrategia seguiremos los siguientes pasos:

- Empieza proponiendo el elemento de la mitad de la lista (en este caso 50). Si el número secreto es mayor que 50 eliminamos en una sóla propuesta la mitad de los números $(1, 2, \ldots, 50)$ de forma similar si el número secreto es menor, entonces eliminamos los números $50, 51, \ldots, 100$.
- Si el número secreto es mayor que 50, sabemos que el número secreto esta entre 51 y 100 y proponemos a 75.
- Si el número secreto es menor que 50, sabemos que el número secreto esta entre 1 y 49 y proponemos a 25.

Hagamos un ejemplo, supongamos que el número secreto es 74.

- Proponemos $50 \longrightarrow$ "Tu número es menor" $\longrightarrow$ nuevo rango: $51 \breve{} 100$
- Proponemos $75 \longrightarrow$ "Tu número es mayor" $\longrightarrow$ nuevo rango: $51 \breve{} 74$
- Proponemos $62 \longrightarrow$ "Tu número es menor" $\longrightarrow$ nuevo rango: $63 \breve{} 74$
- Proponemos $68 \longrightarrow$ "Tu número es menor" $\longrightarrow$ nuevo rango: $69 \breve{} 74$
- Proponemos $71 \longrightarrow$ "Tu número es menor" $\longrightarrow$ nuevo rango: $72 \breve{} 74$
- Proponemos $73 \longrightarrow$ "Tu número es menor" $\longrightarrow$ nuevo rango: 74
- Encontramos el número secreto, 74

Puede que no estés del todo consciente de que lo que acabamos de hacer con este último problema es dar un ejemplo, o lo que se conoce también como **generar una instancia**, es decir, un caso concreto del problema. A partir de esta instancia hicimos el análisis de la segunda estrategia.

2.4. Algunas recomendaciones para diseñar algoritmos

El primer paso, en la resolución de problemas con algoritmos, es declarar con la mayor precisión posible el objetivo. El objetivo define el problema. Si no hay claridad en el objetivo, no la habrá tampoco en el planteamiento del problema. En todo este proceso, el pensamiento lógico es de suma importancia. Cuando las cosas no salen como previstas, lo primero que habría que hacer es revisar la lógica del algoritmo: la lógica de la secuencia de pasos: ¿En qué orden deben ir las acciones?; y, la lógica de las bifurcaciones y ciclos: ¿Qué condición debe controlar el flujo de las acciones? Son dos cosas distintas, pero íntimamente relacionadas.

Por un lado, una condición lógica puede cambiar el curso de las acciones, y en este sentido, cambiar el orden de ejecución de las mismas; por otro lado, un curso de acción puede cambiar el valor de la variable requerida en una condición, y entonces cambiar la bifurcación o el ciclo donde ésta interviene.

El pensamiento algorítmico se refiere a la capacidad de ordenar en el tiempo la secuencia de acciones necesarias para resolver un problema. Se refiere al proceso creativo de construir, de manera iterativa, una solución algorítmica, que toma una entrada, realiza un proceso sobre la entrada, y devuelve la salida deseada. En este sentido, resolver un problema requiere de pensar en términos de variables y estados. Variables que almacenan información, que puede variar en el tiempo. Estados que representan una fotografía del proceso en un instante de tiempo. Por lo tanto, crear una solución a un problema mediante un algoritmo, es pensar en un proceso dinámico que modifica el estado de cosas inicial, para lograr otro estado de cosas final.

En la práctica, resolver un problema mediante una serie ordenada de pasos suele no ser una tarea tan trivial, como la de preparar café, por ejemplo, y se trata de un proceso creativo. La buena noticia es que se puede estructurar este proceso.

Comprender un problema en términos de acciones concretas, específicas, significa plantearlo en términos del cómputo que se requiere hacer para resolverlo. Una estrategia para adentrarse en la comprensión del problema es usar la técnica de George Polya, que, mediante ciertas preguntas, busca reformular el problema de manera a comprender mejor su naturaleza. Para esto, es importante que podamos expresar el problema con nuestras propias palabras. En este sentido, es interesante darse cuenta de cómo los sustantivos, adjetivos y verbos, nos ayudan a dar claridad y especificidad a la naturaleza del cómputo necesario para abordar un problema. Cuanto más preciso sea un verbo (una acción), más clara será la naturaleza del problema, y de la solución. Cuanto más claro sean los sustantivos (las variables), mejor podremos ver la interacción entre ellas. Cuanto más específico sea un adjetivo (condición) mejor podremos evaluar el tipo de instrucción que necesitamos para el control del flujo de las acciones. También es posible hacer una descomposición del problema en subproblemas, o subtareas. Aquí podemos usar árboles, o diagramas a bloques. Estos diagramas son modelos estáticos del problema, que nos permiten identificar subproblemas que podemos jerarquizar, unir o separar, ya sea porque comparten (o no) ciertas propiedades, o bien, porque podemos relacionarlos fácilmente. Por último estrategias del tipo “divide y vencerás” como la recursión son técnicas avanzadas de análisis de problemas. Para el lector interesado, en el Apéndice C se discuten brevemente estos aspectos.

2.5. Ejercicios y Problemas

2.5.1. Análisis de Problemas

1. Haga un análisis de descomposición para la planeación de una fiesta de cumpleaños.

2. Haga el análisis de Polya del problema 2.3.2.

3. Haga un análisis del problema de calcular la edad de una persona dada su fecha de nacimiento y la fecha actual.

4. Haga un análisis de Polya para el problema de determinar si un estudiante acredita el año escolar o no, con base en su calificación final promedio que debe ser mínimo de 6.0. Suponga que puede leer 6 calificaciones en forma de lista: $L := [c_1, c_2, c_3, c_4, c_5, c_6]$. Usted puede acceder a cada calificación usando un índice sobre la lista. Por ejemplo, para acceder al primer elemento de la lista usted escribe: $L[1]$ y esto arroja el valor c_1. Si desea el valor c_4 usted escribiría $L[4]$. De esta forma, usted puede usar una variable i para indexar la lista. Así, si $i = 1$, $L[i]$ es c_1.

2.5.2. Diseño de Algoritmos

1. Escriba un algoritmo que lea dos números enteros positivos a y b y escriba si el número mayor es múltiplo del menor. El algoritmo debe dar una respuesta correcta en caso de que ambos números sean iguales.

2. Haga una prueba de escritorio del algoritmo del inciso anterior para a=18, b=3, para a=5, b=25 y para a=1, b=1.

3. Diseñe un algoritmo que reciba dos números enteros ‘a‘ y ‘b‘, y calcule el máximo común divisor.

4. Escriba un algoritmo para resolver el problema del ejercicio 4.

A Principios Básicos de la Lógica Booleana

A.1. Operaciones lógicas básicas

A.1.1. Valores lógicos o valores de verdad

En una computadora, todo es cálculo sobre números binarios. Como sólo podemos manipular 1's y 0's, se dice que manipulamos valores lógicos. De hecho, la computadora sólo realiza operaciones lógicas (¡incluso cuando hace operaciones aritméticas!). De esta forma, las **operaciones lógicas producen valores lógicos**, también llamados **valores de verdad**:

valor lógico	valor de verdad
1	`verdadero` (V)
0	`falso` (F)

Existen 3 operaciones lógicas básicas: NO (NEGACIÓN), Y (CONJUNCIÓN) y O (DISYUNCIÓN). Cada una de ellas opera como una función que recibe uno o más valores de verdad a la entrada y produce un sólo valor de verdad a la salida. A continuación describimos estas operaciones.

A.1.2. Negación (NO)

La NEGACIÓN (NOT en inglés), es una función que toma un sólo valor de verdad como entrada y produce su negación (valor contrario) a la salida. Usaremos la siguiente escritura para representar la negación de A : $\neg A$. La Figura A.1 presenta su simbología esquemática y la analogía de su operación mediante un interruptor normalmente cerrado.

a) NEGACIÓN (NOT)

b) Un interruptor normalmente cerrado, equivalente a la operación NOT

Figura A.1: a) Compuerta NEGACIÓN (NOT). b) Interruptor normalmente cerrado como representación de la operación de la compuerta. El estado normalmente cerrado indica que la operación de la compuerta es tal que la salida es 1 cuando A=0 (cerrado), y es 0 cuando A=1 (abierto).

Para expresar todos los valores posibles que puede arrojar la función, usamos **una Tabla de Verdad**.

Definición A.1.1. Tabla de Verdad. Una tabla de verdad es una tabla donde colocamos todas las combinaciones posibles de valores lógicos que recibe una función lógica (entrada) y el valor lógico que produce la función como resultado de la operación lógica sobre cada entrada (salida).

La tabla de verdad de la NEGACIÓN es la siguiente:

Tabla A.1: Tabla de verdad de la NEGACIÓN.

entrada	salida
A	$\neg A$
V	F
F	V

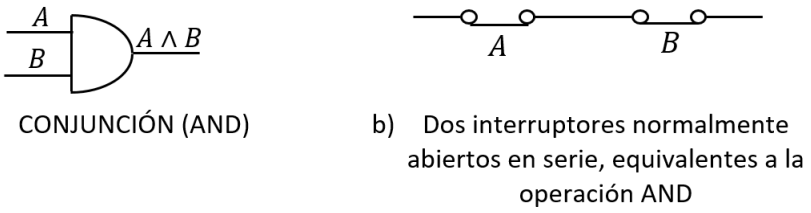
Ejemplo A.1.1

Por ejemplo, podemos aplicar la negación a un número binario:

NO (11001011) = 00110100

A.1.3. Conjunción (Y)

La operación lógica de la conjunción (Y) recibe dos valores lógicos a la entrada y produce un sólo valor a la salida. Usaremos la siguiente escritura para representar la conjunción (A Y B) : $A \wedge B$. Su operación está fundamentada en que, si ambos valores de entrada son verdaderos, entonces el valor que resulta de la conjunción también es verdadero. La Figura A.2 muestra su diagrama esquemático y analogía gráfica usando interruptores.



a) CONJUNCIÓN (AND)

b) Dos interruptores normalmente abiertos en serie, equivalentes a la operación AND

Figura A.2: a) Compuerta CONJUNCIÓN (Y). b) Dos Interruptores normalmente abiertos, conectados en serie como representación de la operación de la compuerta. La seriación indica que la operación de la compuerta es tal que la salida es 1, sólo cuando A=1 (cerrado) y B=1 (cerrado).

La tabla de verdad de la CONJUNCIÓN es la siguiente:

Tabla A.2: Tabla de verdad de la CONJUNCIÓN.

entrada $A\ B$	salida $A \wedge B$
$V\ V$	V
$V\ F$	F
$F\ V$	F
$F\ F$	F

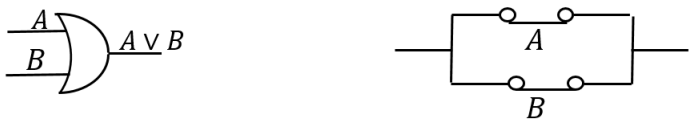
Ejemplo A.1.2

Por ejemplo, podemos calcular la conjunción de dos números binarios:

$$\begin{array}{ll} & 11110010 \\ Y & 11001011 \\ \hline & 11000010 \end{array}$$

A.1.4. Disyunción (O)

La operación lógica de la disyunción (O), está fundamentada en que, dados dos valores lógicos a la entrada, si al menos uno de ellos es verdadero, entonces su disyunción es verdadera. Usaremos la siguiente escritura para representar la disyunción (A O B) : $A \vee B$. La Figura A.3 muestra su diagrama esquemático y analogía gráfica usando interruptores.



c) DISYUNCIÓN (OR)

d) Dos interruptores normalmente abiertos en paralelo, equivalentes a la operación OR

Figura A.3: a) Compuerta DISYUNCIÓN (O). b) Dos Interruptores normalmente abiertos, conectados en paralelo como representación de la operación de la compuerta. El paralelismo del estado normalmente abierto indica que la operación de la compuerta es tal que la salida es 1, en aquellos casos cuando A=1 (cerrado) o B=1 (cerrado).

La tabla de verdad de la DISYUNCIÓN es la siguiente:

Tabla A.3: Tabla de verdad de la DISYUNCIÓN.

entrada		salida
A	B	$A \vee B$
V	V	V
V	F	V
F	V	V
F	F	F

Ejemplo A.1.3

11110010
O *11001011*
11111011

Ejercicio Resuelto A.1.1: Operaciones lógicas sobre números binarios

Determine el resultado de las siguientes operaciones lógicas sobre números binarios.

1. NO (0111 1011)
2. 1110 Y 0111
3. 110001 O 001110
4. 001100 Y 01110

SOLUCIÓN

1. NO (0111 1011) = 1000 0100
2. 1110 Y 0111 = 0110
3. 110001 O 001110 = 111111
4. 001100 Y 01110 = 001100 Y 0|01110 = 001100

A.1.5. Operaciones compuestas

Las operaciones lógicas básicas se pueden usar de manera composicional o compuesta. Es decir, que podemos realizar dos o más de ellas de forma estructurada. Veamos un ejemplo.

Ejercicio Resuelto A.1.2: Composición de operaciones lógicas

Para cada uno de las siguientes expresiones lógicas, determine el valor de verdad resultante. Suponga que las letras mayúsculas ($A, B, \ldots$) se refieren a cualquier variable binaria tales que $A \leftarrow 1$, $B \leftarrow 0$, $C \leftarrow 1$ y $D \leftarrow 0$.

1. (NO(A)) O (NO(B))
2. (B Y (NO(A) O C))
3. NO(A Y C) Y NO(D O B)
4. NO(NO(A) Y B) Y C
5. NO ((A Y B) Y C) Y D

SOLUCIÓN

1. NO(A) $\leftarrow$ 0; NO(B) $\leftarrow$ 1; RESP.: 1
2. NO(A) $\leftarrow$ 0; (NO(A) O C)$\leftarrow$ 1; RESP. 0
3. (A Y C)$\leftarrow$ 1; NO(A Y C) $\leftarrow$ 0; (D O B) $\leftarrow$ 0; NO(D O B) $\leftarrow$ 1; RESP.: 0
4. (NO(A) Y B) $\leftarrow$ 0; NO((NO(A) Y B)) $\leftarrow$ 1; RESP.: 1
5. (A Y B) Y C: $\leftarrow$ 0; NO((A Y B) Y C) $\leftarrow$ 1; RESP.: 0

Las operaciones lógicas son muy importantes a la hora de diseñar algoritmos. Pasamos ahora al tema central de este capítulo.

B Validación de condiciones

B.1. Sobre la importancia de construir condiciones correctas

Los ejemplos sobre chequeo de condiciones deberían dejar en claro la importancia de elegir correctamente la *condición*, ya que el valor de verdad de la misma es lo que determina que el flujo de un algoritmo se bifurque en una dirección o en otra.

En apariencia, podríamos pensar que se trata de un problema relativamente simple. La realidad es que no siempre lo es, y establecer la condición correcta puede llegar a ser un verdadero reto para ciertos problemas, incluso para los programadores más experimentados. Ante la duda, la construcción de tablas de verdad puede ser de ayuda.

Como una guía, podemos enunciar los siguientes aspectos que conviene tener en mente:

1. ***Distinguir entre condiciones disjuntas y condiciones atómicas (indisociables).*** Dijimos que una condición es una proposición, que puede ser atómica o una expresión compuesta. Las expresiones compuestas involucran una o más variables que se pueden combinar utilizando los conectivos lógicos que hemos visto (con excepción de la implicación). De esta forma, es importante reconocer cuando una proposición involucra la negación, la conjunción o la disyunción de dos o más condiciones, de cuando la condición no se puede componer usando conectivos. Por ejemplo, la proposición siguiente involucra la conjunción de dos condiciones: "X es un cuadrado" **y** "X es de color rojo".

   "La figura es un cuadrado **y** es de color rojo"

   Se habla en efecto de dos propiedades de una figura: una de forma y otra de color, y se busca la conjunción de estas dos propiedades.

   Pero la siguiente proposición tiene una trampa:

   "Mi hermana quiere un gato blanco y negro"

   En efecto, se trata de un solo gato cuya propiedad es ser de color blanco y negro, por lo que la condición del color no se puede concebir realmente como una conjunción de dos condiciones: "el color es blanco" **y** "el color es negro", sino que se trata de una sola proposición: "el color es blanco y negro".

2. ***Una condición compuesta no representa la temporalidad o secuencia de dos o más condiciones.*** Las condiciones involucran variables que pueden cambiar de valor en el tiempo, pero no pueden expresar por sí mismas una secuencia de eventos. Por ejemplo, la siguiente proposición expresa una secuencia de eventos:

"El archivo llega y se envía inmediatamente"

Se podría pensar que esta proposición se puede expresar mediante la conjunción de dos condiciones: "el archivo llega" **y** "el archivo se envía". Esto equivaldría a verificar que ambas son verdaderas por acciones que ocurren simultáneamente, o por acciones que ocurrieron a destiempo, pero que ahora se verifican simultáneamente. Pero esto es muy distinto a expresar un encadenamiento de hechos, es decir, que deben ocurrir uno después del otro. La primera situación queda perfectamente expresada en la conjunción, la segunda no.

3. ***Considerar acciones cuando la condición es falsa.*** Esto podría parecer una obviedad. La realidad es que no lo es. Muchas veces olvidamos que la computadora no puede tomar decisiones por nosotros: no tienen sentido común. Por ejemplo, si el valor de una variable debe cambiar cuando una condición es verdadera, ¿qué pasaría con ella si la condición es falsa? Se podría pensar que el valor no debe cambiar, y si eso es correcto, entonces debemos al menos asegurarnos de que tiene un valor válido, pero si no lo es, debemos hacer algo con ella cuando la condición es falsa. Por ejemplo, supongamos que un usuario nos plantea un problema como sigue:

   "Si la evaluación del examen es mayor a 5 entonces poner calificación de aprobado."

   Se podría pensar que un algoritmo que incluya lo siguiente:

   ```
   SI X > 5:
      CALIF ← "APROBADO"
   ```

   sería suficiente, pero ¿qué sucede si la condición es falsa? En ese caso, dado que el algoritmo no contempla una acción para la condición falsa, la variable CALIF no tendría valor, y en una computadora, eso se traduce en un error. A la hora de querer consultar los alumnos reprobados, no se podría saber, ya que el valor de CALIF sería (en el mejor caso) "basura". En el peor caso, el programa se corrompería, ya que la variable CALIF ¡ni siquiera existiría!, puesto que el bloque debajo de SI nunca se ejecutaría.

   Habría dos soluciones para esta situación. La primera es usar un bloque SI - DE_LO _CONTRARIO. La solución sería la siguiente:

   ```
   SI X > 5:
      CALIF ← "APROBADO"
   DE LO CONTRARIO:
      CALIF ← "REPROBADO"
   ```

   La segunda alternativa es asegurarse de que la variable CALIF cuente con un **valor por defecto**.

   ```
   CALIF ← "REPROBADO"
   SI X > 5:
      CALIF ← "APROBADO"
   ```

donde nos hemos asegurado que el valor por defecto de la variable CALIF es “REPROBADO”. De esta forma, si la condición se cumple el valor de CALIF cambia, y si no, conserva su valor inicial.

4. ***Validar una condición mediante el análisis de casos***. Por último, mencionaremos la importancia que tiene tomarse un momento para estructurar una condición compleja, que involucre varias operaciones lógicas. Por ejemplo, supongamos el siguiente problema:

   “La medida de la pieza debe ser 10 o 15, con una tolerancia de +/- 1 para ser aprobada, fuera de esas medidas y tolerancias debe desecharse.”

   Consideremos las siguientes variables, X recibe la medida de la pieza y A tiene un valor inicial de VERDADERO (pieza aprobada de inicio):

   X ← medida de la pieza;
   A← VERDADERO

   ¿Cuál de las siguientes condiciones es la correcta para desechar una pieza?:

   a) SI (X<9 O X>11) O (X<14 O X>16):
      A← FALSO

   b) SI NO (X>=9 Y X<=11) Y NO (X>=14 Y X<=16):
      A← FALSO

   La respuesta correcta es (b). A pesar de que la condición (a) podría parecer digna del sentido común, ésta falla, ya que la disyunción es verdadera cuando al menos una proposición es verdadera. El problema es que si X>11 es verdadera, automáticamente toda la condición es verdadera. Por lo tanto, una pieza con medida X=15 > 11 sería desechada. La condición (b) pone en conjunción dos proposiciones: la negación de que las medidas de la pieza estén dentro del primer rango tolerado con la negación de que las medidas estén dentro del segundo rango de valores tolerados. Si ambas (negaciones) son verdaderas, es decir, si la medida de la pieza no está en ninguno de los rangos de tolerancia, la pieza es desechada.

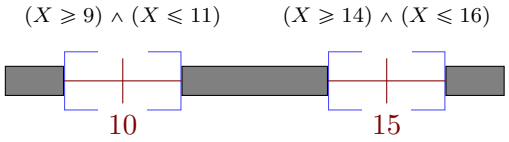


Figura B.1: *La conjunción de dos condiciones negadas se representa con las regiones en gris.*

Esta condición se puede ilustrar como en la Figura B.1, que muestra los valores (recuadros en color gris) que considera la condición (b) para desechar una pieza.

C Algunas Herramientas Adicionales de Análisis de Problemas

C.1. Descomposición de problemas en subproblemas

La descomposición de problemas en subproblemas es una estrategia que da muy buenos resultados en ciencias computacionales. Los científicos de la computación generalmente deben tratar con problemas complejos, que involucran muchos componentes interrelacionados. No sólo eso, sino que, además los recursos de que disponen (capacidad de cómputo, almacenamiento, velocidad) son finitos y muchas veces limitados. Por lo tanto, la solución a un problema complejo, pasa muchas veces por la descomposición en problemas más pequeños.

C.1.1. Estructuras de árbol

Una herramienta muy útil en una descomposición es la **estructura de árbol**, **estructura arborescente** o simplemente **un árbol**. Un árbol representa una colección de entidades organizadas en relaciones jerárquicas descendentes: cada entidad "*padre*", puede tener un cierto número de entidades "*hijo*", o ninguno. El primer nodo se llama nodo raíz y los nodos terminales se les llama hojas. De esta forma, un árbol puede representar cualquier tipo de estructura de dependencias, como se muestra en las Figuras C.1, C.2 y C.3, que ilustran respectivamente la realización de una obra de teatro, una compra por internet, o la conducción de un vehículo.

Los árboles permiten también la colaboración. Discutir la descomposición de tareas en subtareas es, muchas veces, una necesidad. Para cierto tipo de problemas, un árbol no sólo permite la discusión colaborativa, sino también la distribución de las tareas dentro del equipo de trabajo.

Además, se especifican las entradas y las salidas. De esta forma, un árbol permite aclarar los requerimientos de información en cada tarea, por lo que facilita la especificación de las interfaces; es decir, qué debe entrar y qué debe salir en cada parte del proceso.

C.1.2. Descomposición como herramienta de simulación

La descomposición mediante un árbol, si bien es útil para identificar los elementos o componentes de un problema, no nos da una solución completa, es decir, un algoritmo.

En otras palabras, nos dice el qué, pero no nos dice el cómo. No obstante, se trata de un buen punto de partida y el análisis de entradas y salidas puede ser útil en la identificación de la secuencia que debemos seguir para dar solución al problema completo.

Por ejemplo, en el caso de la conducción de un vehículo (Figura C.3), no se puede proponer una ruta alternativa si no conoce el destino, o no se pueden resolver situaciones imprevistas sin vigilar el entorno. Sin embargo,

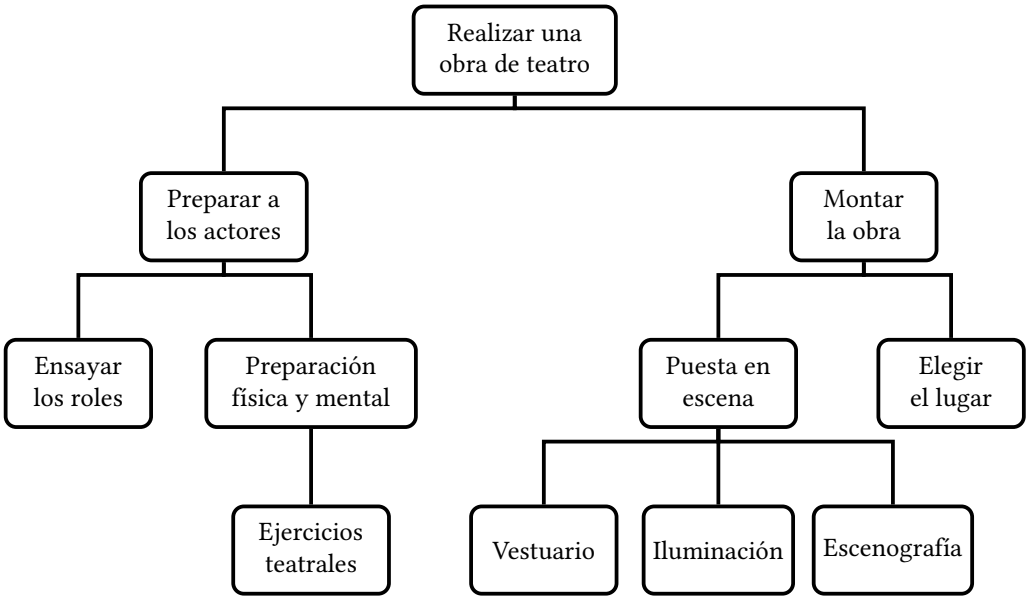


Figura C.1: *Descomposición en subproblemas para realizar una obra de teatro.*

esta discusión no tendría lugar, de no haber tenido una herramienta que la habilitara, de ahí la utilidad del árbol.

Finalmente, diremos que la descomposición mediante árboles también ayuda a la reflexión crítica y a la validación en papel de posibles soluciones: se trata de usar la herramienta como un mecanismo de simulación de instancias reales de un problema. Esto es un aspecto fundamental en computación. Podemos imaginar estos elementos de descomposición de un problema, pero hasta no "correr" una instancia concreta del problema, no sabremos si la propuesta es funcional o no.

Simular escenarios o situaciones concretas o contar con ejemplos reales de un problema, nos ayuda a visualizar el posible flujo de la información, evaluar la relevancia de las tareas, y, en su caso, detectar la falta de una tarea importante.

C.2. Entidades, Verbos y Adjetivos

La simulación de ejemplos concretos mediante la descomposición de problemas en subproblemas también ayuda de detectar patrones que se repiten, lo que puede ser muy útil para reflexionar sobre descomposiciones alternativas que tomen en cuenta estos patrones.

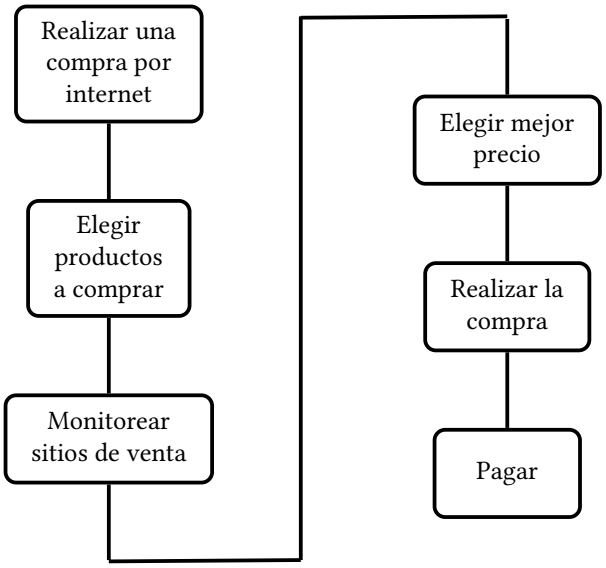


Figura C.2: *Descomposición en subproblemas para realizar una compra por internet.*

Por ejemplo, supongamos un problema en el que se nos pide dibujar el plano bidimensional de una casa. Esto lo podemos hacer usando formas geométricas, como rectángulos para dibujar muros y puertas, o círculos para dibujar mesas o bancos, como se ilustra en la Figura C.4.

Si observamos bien la Figura C.4, vemos que no tendría sentido descomponer el problema en una tarea para cada tipo de objeto en el dibujo, ya que hay un patrón en la repetición de las figuras geométricas, con posibles variaciones en las dimensiones (ancho y largo). En este caso, bastaría con hacer un algoritmo especializado en formas rectangulares y circulares, que son las formas que se repiten como parte de una solución al problema. De esta forma, podríamos representar la solución mediante un árbol como se ilustra en la Figura C.5.

Otro tipo de patrón es el que se refiere al comportamiento. Por ejemplo, en el caso del sistema de compras en línea, podría existir un patrón estacional: las ofertas siguen un patrón de fin de temporada, o bien algunos sitios suelen colocar ofertas cerca de fin de mes. En este caso, se podría refinar el árbol de descomposición de este problema, para incluir una tarea que identifique estos patrones y agrupe los sitios de compra en función de estos. En el caso de la conducción de un vehículo, el patrón podría ser que buscamos alternativas de ruta cercanas al itinerario original. Así, la tarea que evalúa rutas alternativas, podría considerar las condiciones del tráfico en un radio limitado alrededor de la ruta original, lo que haría que la tarea fuera menos costosa.

¿Cómo podemos reconocer patrones? Proponemos una estrategia a continuación.

Si observamos el árbol de la Figura C.5, vemos que cada bloque contiene esencialmente tres tipos de palabras: sustantivos, adjetivos y verbos.

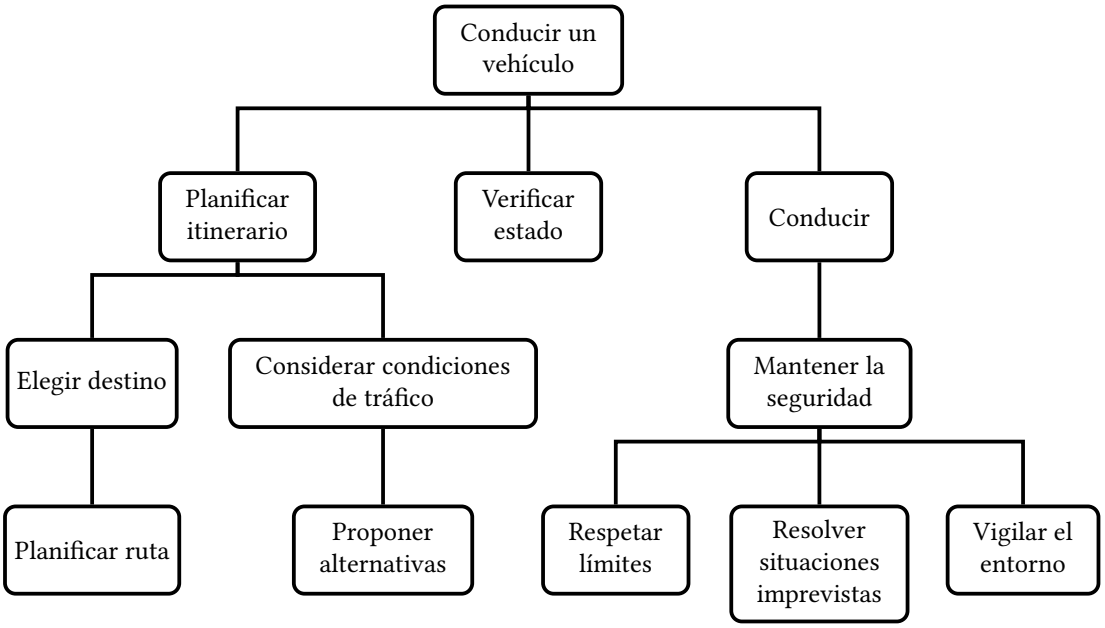


Figura C.3: *Descomposición simple para la conducción de un vehículo.*

C.2.1. Sustantivos

Los **sustantivos** pueden ser *entidades nombradas*, que responden al "qué", "quién", "cuándo", como son nombres propios o de instituciones, lugares, objetos o días (Juan, Ana, Amazon, Notaría Diez, Jardín Botánico, casa, perro, lápiz, lunes, viernes). Los sustantivos nos permiten identificar lo que llamamos **entidades del problema**. Son en cierta forma las *variables del problema*. Por ejemplo, en las figuras mencionadas tenemos sustantivos como: *Internet, actores, roles, vehículo, seguridad, destino, figuras, diámetro, orientación*. Son entidades de cada problema, cosas que nos interesan en cada problema. Los sustantivos nos permiten agrupar cosas por algún criterio que las relacione. Por ejemplo, *diámetro* y *orientación* son *características* de una figura, por tanto, se podría concebir un bloque de procesamiento dedicado capturar o modificar el tipo de figura y sus características.

C.2.2. Adjetivos

Los adjetivos modifican los sustantivos. Por ejemplo, en las figuras de los árboles mencionados tenemos adjetivos como: *Mejor* (precio), *Imprevistas* (situaciones), *Rectangulares* o *Circulares* (figuras). Los adjetivos nos hablan de **modificadores** o **propiedades de nuestras entidades**. Estas propiedades nos dan pistas sobre grupos o conjuntos de entidades. Nos permiten identificar **patrones de semejanza** entre los objetos del pro-

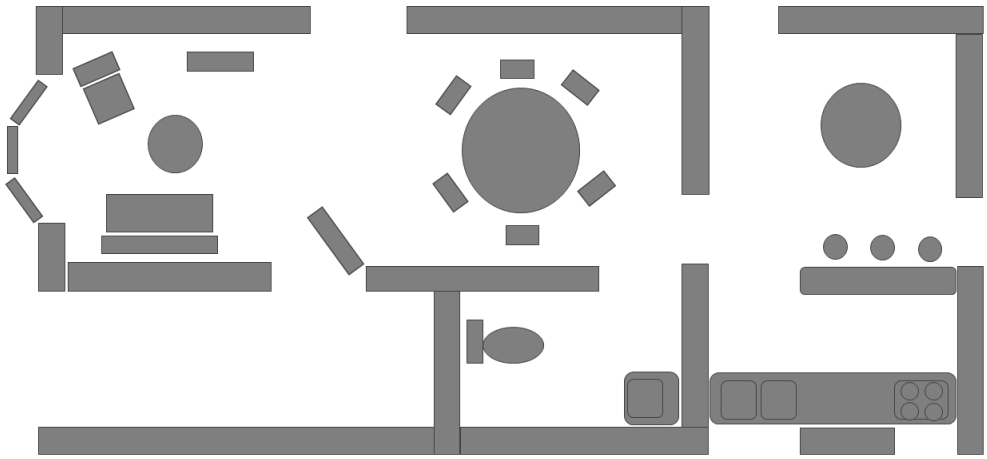


Figura C.4: *Plano 2D de una casa utilizando formas geométricas.*

blema. De esta forma, podemos diseñar soluciones, especialmente concebidas para cierto tipo de propiedades, como fue el caso de las figuras geométricas para diseñar un plano 2D de una casa.

C.2.3. Verbos

Por último, los **Verbos** definen ***acciones***, y están generalmente conjugados en infinitivo. Los verbos definen **lo que queremos hacer con nuestras entidades**: definen **tareas**. En este sentido, puede que no especifiquen una sola acción, sino una serie de acciones (secuencia de ejecución) que conforman un patrón de acción, como parte de una solución a un problema.

Por ejemplo, en las figuras mencionadas tenemos verbos como: *Realizar* (una compra), *Vigilar* (el entorno), *Elegir* (productos), *Preparar* (actores), *Respetar* (límites), *Resolver* (situaciones), *Ubicar* (figuras). Estas tareas o **serie de acciones**, **podrían ser parte de un ciclo** (repetición de una sola acción), o **ejecutarse dentro de una bifurcación**, como parte de una decisión.

Tomemos como instancia el verbo "vigilar"; que significa: "*observar atentamente una cosa y estar pendiente de ella para que se desarrolle u ocurra como se desea o para seguir su evolución o desarrollo*". Esta definición podría estar bien, si quisiéramos explicarle a un humano en qué consiste la tarea de "vigilar". Sin embargo, para fines de resolver la tarea de "vigilar el entorno", en el contexto del problema de la conducción de un vehículo, esta definición no dice nada respecto de lo que hay que poner en concreto en un algoritmo (si quisiéramos crear uno que conduzca el automóvil).

Para ilustrar el problema, la Figura C.6 muestra una representación pictórica de "vigilar el entorno", que consiste en obtener observaciones de lo que ocurre alrededor del vehículo central, mediante el uso de sensores de varios tipos, que tienen alcances y zonas de percepción distintas.

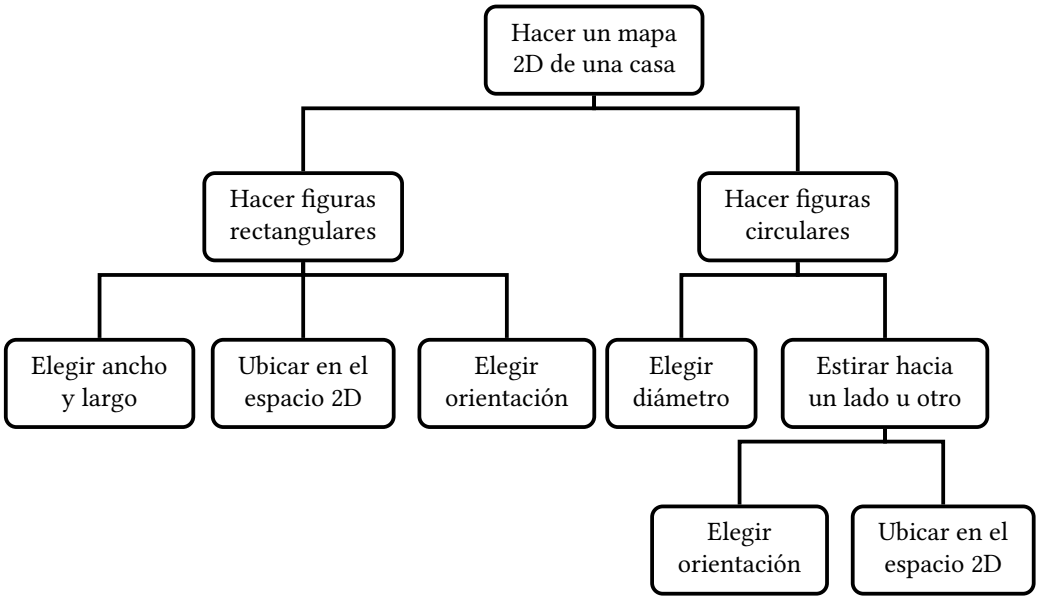


Figura C.5: *Descomposición del problema para hacer la Figura C.4*

Figura C.6: *Imagen que ilustra una tarea de "vigilar el entorno" como un conjunto de observaciones que se hacen mediante sensores de varios tipos, que tienen alcances y zonas de percepción distintas.*

Evidentemente, esta tarea se prevé más compleja cuantos más sensores tengamos a nuestra disposición para "observar atentamente". Sin embargo, **un análisis más específico** de lo que significa "vigilar el entorno",

nos permite identificar algunas acciones que podemos estructurar alrededor de la tarea, como se ilustra en la Figura C.7.

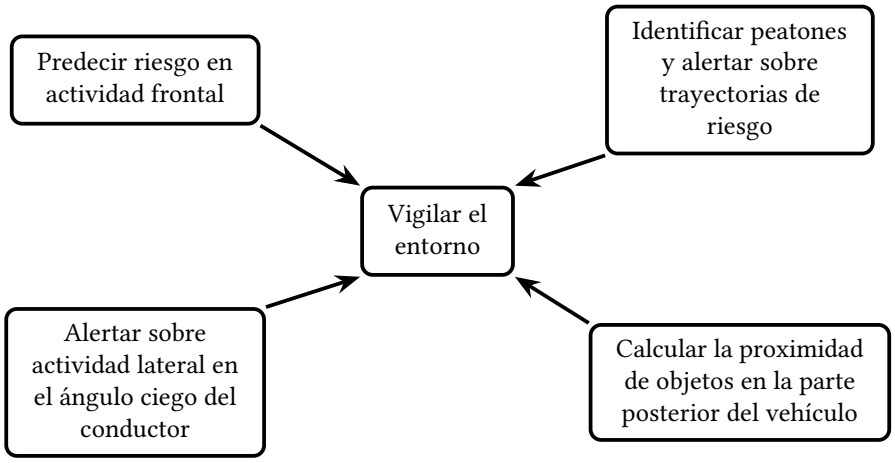


Figura C.7: *La tarea de vigilar el entorno se puede a su vez descomponer en subtareas, definidas por verbos más específicos.*

La Figura C.7 muestra varias cosas que es importante resaltar. Por un lado, vemos que podemos estructurar la tarea de "vigilar el entorno" en varias subtareas, que a su vez están definidas con verbos de acción. Esta vez, las acciones son más específicas; en otras palabras, "vigilar" se ha especificado en los verbos: *Predecir* (riesgo), *Identificar* (peatones), *Alertar* (sobre actividad), *Calcular* (proximidad).

Estos verbos más específicos, permiten no sólo ir anclando el problema en tareas concretas, sino que permiten reconocer acciones que presentan patrones que podemos asociar o diferenciar. Por ejemplo, identificar peatones, supone un patrón de reconocimiento de formas y de movimientos muy distinto al patrón de reconocimiento de vehículos o bicicletas. Predecir el riesgo de colisión implica un cálculo de proximidad, que a su vez puede ser útil para alertar sobre actividad cercana. Estas acciones podrían tener lugar sólo para un subconjunto de sensores, por ejemplo, de tal suerte que se pueden resolver los subproblemas por patrones de actividad en cada subconjunto de sensores.

Esta descomposición es sin duda más específica, no obstante, todavía no nos da una secuencia de acciones, que podrían formar parte de un algoritmo. Para ello, podríamos recurrir a un **modelo: una representación de una solución, donde figura la organización de las subtareas por niveles de abstracción y la interacción entre ellas**.

C.3. Recursión

La recursión se define como: una solución que depende de la solución de pequeñas instancias del mismo problema. Un algoritmo recursivo es un algoritmo que expresa la solución de un problema en términos de una llamada a sí mismo. La llamada a sí mismo se conoce como llamada recursiva o recurrente. La mayoría de los lenguajes de programación dan soporte a la recursión permitiendo a una función llamarse a sí misma desde el texto del programa.

Esta estrategia se conoce como "*divide y vencerás*". El método está basado en la resolución **recursiva** de un problema dividiéndolo en dos o más subproblemas de igual tipo o similares. El proceso continúa hasta que éstos llegan a ser lo suficientemente sencillos como para que se resuelvan directamente. Al final, las soluciones a cada uno de los subproblemas se combinan para dar una solución al problema original. Esta técnica es la base de los algoritmos eficientes para casi cualquier tipo de problema como, por ejemplo, algoritmos de ordenamiento (*quicksort*, *mergesort*, entre muchos otros), multiplicar números grandes (*Karatsuba*), análisis sintácticos (análisis sintáctico *top-down*) y la *transformada discreta de Fourier*.

El nombre divide y vencerás también se aplica a veces a algoritmos que reducen cada problema a un único subproblema, como la búsqueda binaria para encontrar un elemento en una lista ordenada. Estos algoritmos pueden ser implementados más eficientemente que los algoritmos generales de "divide y vencerás"; en particular, si es usando una serie de recursiones que lo convierten en simples bucles. Bajo esta amplia definición, sin embargo, cada algoritmo que usa recursión o bucles puede ser tomado como un algoritmo de "divide y vencerás".

Índice Alfabético