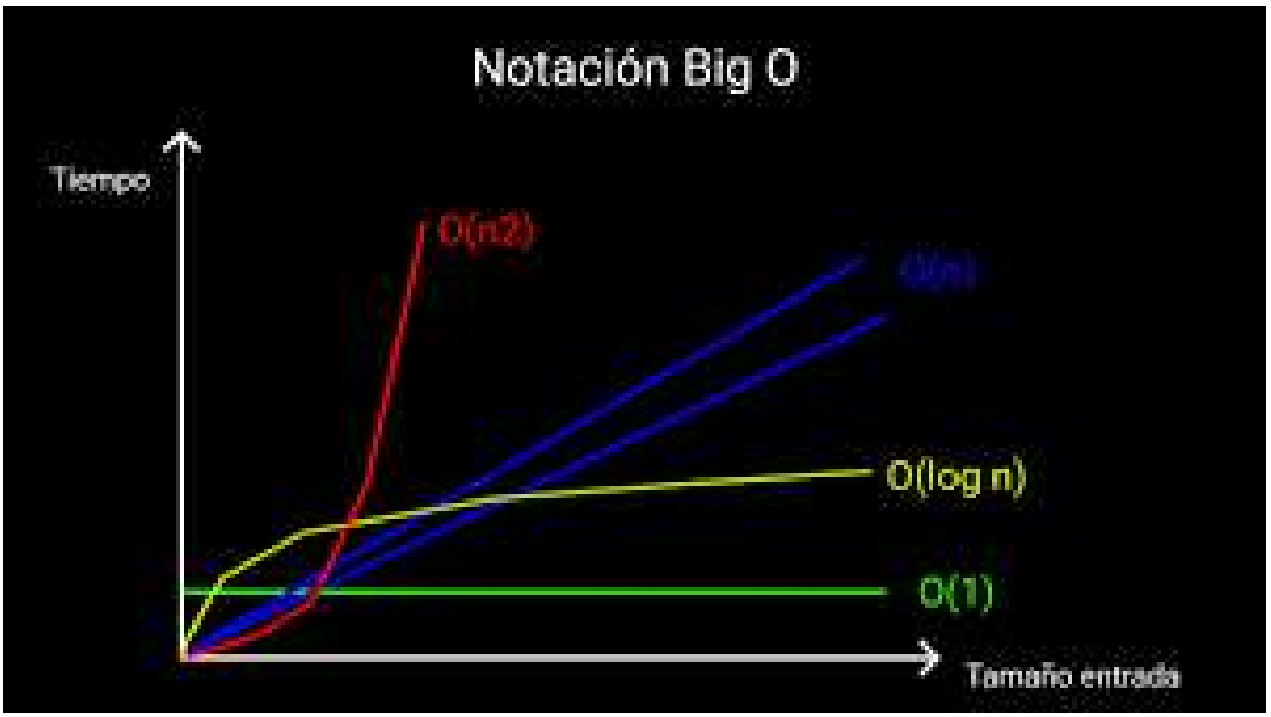


NOTAS DE CLASE
ANÁLISIS DE ALGORITMOS

{Con los lenguajes de programación Java y Python}

/* ****** **Jaime E. Montoya M.** ****** */

NOTAS DE CLASE
ANÁLISIS DE ALGORITMOS

{Con los lenguajes de programación Java y Python}

```
/**
* Versión 3.0
* Año: 2026
* Licencia software: GNU GPL
* Licencia doc: GNU Free Document License (GNU FDL)
*/


class Author {
    String name = “Jaime E. Montoya M.”;

    String profession = “Ingeniero Informático”;

    String employment = “Docente y desarrollador”;

    String city = “Medellín - Antioquia - Colombia”;

    int year = 2026;
}
```

Tabla de contenido

Introducción

El análisis de algoritmos tiene como objetivo desarrollar en el estudiante un pensamiento lógico y estructurado a partir de conocimiento técnico, análisis de casos de estudio y desarrollo lógico de algoritmos que cumplan con requerimientos de eficiencia computacional.

El contenido se centra en el "análisis de algoritmos", término acuñado por Donald Knuth[1] que lo concibió como el análisis teórico de algoritmos para estimar su complejidad en sentido asintótico, es decir, estimar la función de complejidad para entradas arbitrariamente grandes. El análisis de algoritmos es una parte importante de la teoría de la complejidad computacional, que proporciona una estimación teórica de los recursos necesarios de un algoritmo para resolver un problema computacional específico. La mayoría de los algoritmos están diseñados para trabajar con entradas de longitud arbitraria. El análisis de algoritmos consiste en determinar la cantidad de recursos de tiempo y espacio necesarios para ejecutarlo, lo que se conoce como eficiencia computacional.

Complementariamente, se pretende que el estudiante reconozca los problemas según su planteamiento y solución, y conozca las técnicas de diseño de algoritmos que pueden mejorar la eficiencia computacional, o si se entrega una solución con un grado de eficacia aceptable.

Este documento es un complemento a las clases presenciales y virtuales, y está basado en la bibliografía del curso, así como de otras fuentes adicionales que se indican a lo largo del texto, además de la experiencia del autor en su función docente en las áreas de programación en instituciones educativas y como desarrollador en distintas empresas. No se pretende reemplazar los textos con este manual, sino servir de ayuda didáctica y apoyo académico a los estudiantes.

La guía incluye, además de los conceptos teóricos, ejemplos y gráficas, desarrollos en clase, y al final de cada capítulo, unas preguntas y ejercicios, con los cuales se espera que permitan reforzar los conceptos y promover la práctica y el estudio de los conceptos vistos.

Al final de este manual, se indican fuentes y referencias adicionales que el estudiante puede consultar.

Los ejemplos se ilustran principalmente en pseudocódigo, en Java (o pseudo Java) y Python, y quizá ocasionalmente con el uso de otros lenguajes de programación como C/C++, PHP, C#, etc.; en algunos casos se pueden mostrar diferentes técnicas de diagramación, entre ellas, los *flujogramas* o *diagramas libres* y los *diagramas rectangulares* o de *Nassi-Schneiderman*, aunque estos diagramas no aportan nada nuevo a la teoría, solo sirven para ilustrar algún desarrollo.

[1] Donald Ervin Knuth es un reconocido experto en ciencias de la computación estadounidense y matemático, famoso por su fructífera investigación dentro del análisis de algoritmos y compiladores. Es Profesor Emérito de la Universidad de Stanford. Donald Knuth - Wikipedia, la enciclopedia libre

Por último, se asume que el estudiante posee conocimientos básicos en los lenguajes Java y Python en cuanto a la sintaxis de las instrucciones comunes como entrada/salida, asignaciones y operaciones aritmético/lógicas, manejo de las estructuras de control, manipulación de cadenas de caracteres, así como la creación y gestión de clases y objetos, entre otros aspectos, lo cual implica que conozca también los principales conceptos de la POO. El capítulo 0 repasa algunos de estos conceptos con el fin de nivelar al lector con algunos conocimientos básicos requeridos para el estudio del presente texto; este capítulo también contiene un repaso de algunos aspectos matemáticos que se requieren en el curso.

Se recomienda al lector revisar los ejemplos y realizar los ejercicios propuestos para afianzar los conceptos mediante la práctica. Tanto en los ejemplos como en los ejercicios se encuentran conceptos adicionales a la teoría y se explican formas de enfrentar los problemas.

Recursos

Adicional a la bibliografía disponible al final del texto y en las notas al pie de página que contienen referencias a otras lecturas, para este curso se debe contar con un equipo de cómputo que soporte los lenguajes de programación actuales, entre ellos Java y Python, con el fin de realizar las prácticas. Se asume que el lector ya está familiarizado con dichos lenguajes de programación. El sistema operativo es a gusto del estudiante, puede trabajar bajo Windows, Linux, Mac u otro.

Además de disponer de conexión a Internet, debe contar también con un IDE o editor (se sugiere Visual Studio Code, pero no es obligatorio) para la edición de los programas.

Con una cuenta de Google se puede hacer uso de Colab, una herramienta para la edición y ejecución de partes de código Python.

El sitio web Programiz ofrece una interfaz para el desarrollo de código con la opción de poder seleccionar entre varios lenguajes de programación, tales como Java, Python, C, C++, C#, PHP, R y Javascript, entre otros.

Lista de abreviaturas

ASCII: American Standard Code for Information Interchange

BD: Base de Datos

CF: Contador de Frecuencias

DyV: Divide y Vencerás

IDE: Integrated Development Environment (Entorno de Desarrollo Integrado)

LSL: Lista Simplemente Ligada

LSLC: Lista Simplemente Ligada Circular

LDL: Lista Doblemente Ligada

LDLC: Lista Doblemente Ligada Circular

MCD: Máximo Común Divisor

PHP: PHP Hypertext Preprocessor

SI: Sistema de Información

SO: Sistema Operativo

Capítulo 0. Preliminares

Este capítulo comprende un repaso de conceptos ya vistos por los estudiantes, entre los que se incluyen definiciones fundamentales de programación, entre ellos, aspectos generales de lenguajes; también se tratan algunos temas de matemáticas que ya se deben manejar. Si el lector considera que se siente bien en dicha teoría, puede pasar sin problemas al capítulo 1, dando una mirada a los títulos que se mencionan.

¿Qué es un algoritmo?

Aunque es un término bastante empleado tanto en matemáticas como en las ciencias computacionales, es común encontrar variantes en la definición. Sin embargo, el consenso general en ciencias, permite definir un ***algoritmo*** como *un conjunto de pasos finitos y ordenados que buscan la solución de un problema*. Este nombre al parecer tuvo influencia en el matemático persa Al-Juarismi, que en latín antiguo se conocía como *Algorithmi*.

En la antigüedad hubo desarrollos de procesos algorítmicos para resolver problemas, entre ellos se encuentra uno de los más famosos conocido como *Algoritmo de Euclides* para hallar el *Máximo Común Divisor (MCD*) de dos enteros.

Puede entenderse un algoritmo como un *proceso mecánico* para resolver algún problema.

Un algoritmo se puede escribir siguiendo una serie de reglas sintácticas que permiten crear un **pseudocódigo** basado en él y que puede llevarse luego a un computador, en otras palabras, se puede escribir de una forma muy parecida a como un computador entendería cada paso del algoritmo.

Un algoritmo puede describirse gráficamente así:

Figura 0.1. Representación gráfica del proceso algorítmico

Programa

A partir del pseudocódigo de un algoritmo, podemos construir un **programa** para llevarlo a un computador. Un programa es por tanto un *conjunto de instrucciones finitas dispuestas en orden para solucionar un problema y que las entiende un computador para ser ejecutadas*. Esta definición coincide con la de algoritmo, donde los pasos de éste equivalen a las instrucciones del programa. En otras palabras, un programa es un *formato especial* que representa la solución de un algoritmo y que comprende un procesador.

Las operaciones que conducen a expresar un algoritmo en forma de programa, se conoce como **programación** y los que escriben dichos programas se conocen como **programadores** (o **desarrolladores**).

El proceso de la programación

Consiste en proponer una solución para computador (programa) luego de tener un problema y haber llegado a una solución algorítmica. Una vez creado el programa, éste es *ejecutado* para dar solución al problema propuesto.

Gráficamente, podemos verlo así:

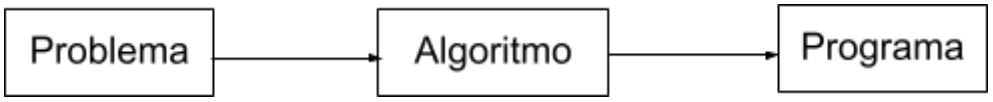


Figura 0.2. Representación gráfica del proceso de la programación

Los lenguajes de programación

Un **lenguaje de programación** es un *programa para crear programas*. Como tal, debe ser capaz de *interpretar* el algoritmo, lo que significa que comprenda cada paso de éste en forma de instrucciones y realice las operaciones correspondientes sin intervención humana. Un programa es en últimas un formato del algoritmo escrito de tal forma que sea entendido por una máquina (procesador) y que debe ser escrito en un lenguaje de programación en particular, proceso que también se conoce como **codificación**, y al texto generado, **código de programación**.

Tipos de lenguajes

Cada lenguaje de programación tiene un **propósito**, esto es, tiene un fin para el que fue creado, lo cual significa que no todo problema algorítmico puede tratar de resolverse en cualquier lenguaje. Algunos lenguajes son considerados de propósito general, lo que en teoría significa que puede resolverse cualquier algoritmo en estos lenguajes, otros son para el desarrollo Web, para el desarrollo móvil, para el "*backend*" o el "*frontend*", para bases de datos, para cálculos matemáticos, para el diseño gráfico, para los componentes físicos, etc. Podemos clasificar los lenguajes en tres grandes grupos:

Lenguajes de máquina

Utilizan el lenguaje binario (1s y 0s) para cada instrucción y operación, por lo que suele conocerse también como **código** o **lenguaje máquina**. Este código depende de la máquina (fabricante del hardware). Estos lenguajes cargan los programas directamente sin requerir "traductores", y al ser operaciones en código nativo, la velocidad y rendimiento es mayor, pero tienen las desventajas de la complejidad, coste en tiempo, dificultad para detectar errores. Una mala programación en lenguaje máquina de un componente puede generar daños costosos o incluso dejarlo inutilizable.

Lenguajes de bajo nivel

El Lenguaje Ensamblador (*Assembly Language*), así como los lenguajes PLC (*Programmable Logic Controller*) hacen parte de los lenguajes que componen este grupo de lenguajes de bajo nivel. Son un poco más fáciles de usar que los lenguajes de máquina, pero son dependientes de la máquina. El lenguaje Ensamblador tiene múltiples aplicaciones para la programación de dispositivos físicos, pero depende de la fabricación de estos, ya que las especificaciones de cada fabricante varía entre componentes del mismo tipo. PLC es muy utilizado en entornos industriales para desarrollos mecánicos, mecatrónicos y de otras aplicaciones donde es necesario utilizar un lenguaje binario para programar algunos tipos de máquinas. Estos lenguajes no se ejecutan directamente, requieren ser **traducidos** al lenguaje de máquina (o simplemente, lenguaje máquina). El programa escrito en lenguaje ensamblador se conoce como **código (programa) fuente** y el traducido como **código (programa) objeto**. El traductor de código está presente en la mayoría de computadores y es un programa que se conoce como *Ensamblador* (*Assembler*).

Lenguajes de alto nivel

Son los más utilizados por los programadores en general y están diseñados para que el programador se pueda "entender" más fácilmente con la máquina, ya que permiten el ingreso de las instrucciones usando palabras con caracteres del idioma inglés[2], incluyendo los caracteres especiales de dicha lengua y del alfabeto latino en general. Esto los hace más atractivos y su campo de aplicación abarca prácticamente todas las áreas, además que en general, son independientes de la máquina (no dependen de las características del hardware), lo cual hace a los programas escritos *portables* a otras plataformas. Algunos permiten interactuar con lenguajes de bajo nivel y de máquina, lo cual facilita el trabajo en dichos entornos de trabajo.

Con respecto a los otros tipos de lenguajes, los de alto nivel tienen algunas desventajas, como por ejemplo un mayor consumo de memoria, mayor tiempo de ejecución y subutilización de los recursos del hardware, entre otras.

Desde la invención de los primeros computadores electrónicos, han sido muchos los lenguajes que se han creado, muchos han desaparecido, surgiendo nuevas de estas herramientas según las necesidades creadas con los desarrollos actuales tecnológicos. Veamos algunos de los más destacados a nivel histórico y relevantes en el nuevo milenio.

1. COBOL (Common Business Oriented Language)
2. Fortran (Formula Translation)
3. Pascal
4. Delphi
5. Ada
6. SmallTalk
7. DBase
8. Foxpro
9. Visual Foxpro

[2] Esto se debe a que la mayoría de lenguajes de programación han sido creados en Estados Unidos, país de habla inglesa.

10. Basic
11. Visual Basic
12. Visual Basic for Applications (VBA)
13. Visual Basic Script (VBS)
14. Visual Basic .NET (VB .NET)
15. Matlab
16. Mathematica
17. Java
18. Javascript
19. Kotlin
20. Go
21. PHP
22. Perl
23. Python
24. Ruby
25. C#
26. C/C++
27. R
28. Prolog
29. LabVIEW
30. PLC

Traductores de lenguaje

Son programas incorporados en los lenguajes de programación y que se encargan de traducir el código fuente a código máquina. Hay dos tipos de traductores:

- Intérpretes
- Compiladores

Intérpretes

Leen línea a línea del programa y lo ejecutan directamente si no detectan errores. Algunos lenguajes interpretados, son:

- PHP
- Python
- Javascript
- Perl
- Ruby

Compiladores

Son programas que verifican los errores sintácticos y luego transforman el *código fuente* en *código objeto* que es ejecutado posteriormente por la máquina. Algunos lenguajes compilados, son:

- C/C++
- Java

- COBOL
- Ada
- Go

Lenguajes de programación fuertemente tipados

Exigen estrictamente declarar todas las variables especificando el tipo de dato de cada una de éstas; de no seguir esta regla, generan errores de ejecución. Algunos de estos lenguajes exigen incluso indicar el tipo de dato de las constantes, aunque la mayoría opta por asumir de forma implícita el tipo de dato de acuerdo al tipo de dato del valor asignado, con lo cual que definida la constante. Algunos lenguajes de este tipo son C/C++, Java, C#, Python y Ada.

Lenguajes de programación débilmente tipados

Son flexibles en cuanto a la declaración de variables y la especificación de sus correspondientes tipos de datos. En algunos lenguajes se tiene la posibilidad de declarar las variables de forma opcional sin especificar el tipo de dato, el cual se asigna a la variable de manera implícita con el primer valor que se le asigne a éstas. Esto permite que las variables luego puedan tomar valores de otros tipos de datos sin generar errores de ejecución; sin embargo, esto puede llevar a confusiones y malas interpretaciones si no se tiene el cuidado pertinente. Algunos lenguajes de este tipo son PHP, Javascript y Visual Basic.

Si el lenguaje permite la declaración de variables, así sea débilmente tipado, es una buena técnica hacerlo a la hora de escribir programas; esto ayudará a mantener un mayor orden, una mejor estructura y facilitará futuros mantenimientos y migraciones.

Fases de la compilación - ejecución

- Problema (enunciado, planteamiento)
- Algoritmo (pseudocódigo, diagramas)
- Codificación (programa fuente)
- Análisis por parte de los traductores de lenguaje (verificación de errores, traducción a código máquina, programa objeto)
- Ejecución del programa

La unidad de memoria

Es un dispositivo encargado de recordar órdenes que indica un agente externo por medio de un dispositivo de entrada, así como la información dividida en datos dados a la máquina y/o que ésta genera a través de algún proceso determinado; además almacena las instrucciones que contienen las operaciones que debe realizar la máquina para que pueda trabajar sin intervención humana.

Esta unidad está integrada en la actualidad por millones de circuitos electrónicos invisibles al ojo humano, donde hablamos de unidades del orden 10^{-9}m o nanómetros (1nm = 10^{-9}m) que requiere intervención tecnológica con láser y otras técnicas para la manipulación de los componentes. Cada uno de estos elementos tiene la propiedad física de ser **biestable**, esto es, posee uno de dos estados que puede tener en cualquier momento (pero no ambos a la vez). Las convenciones para llamar estos estados binarios puede variar de acuerdo al autor; algunos de los que han sido usados son:

- Encendido - apagado
- On - off
- Si - no
- Yes - no
- Verdadero - falso
- True - false
- 1 - 0

Los valores 1 y 0 al ser numéricos, permiten construir toda una aritmética binaria, base matemática de los sistemas digitales y particularmente del computador, y que a su vez abarca cualquier nombre que quiera darse a cada estado.

Manejo de la memoria

La memoria está conformada por una gran cantidad de elementos que pueden manipularse uno a uno; sin embargo, esto no es algo práctico y resulta mejor agrupar varios de estos elementos para manipularlos, sin importar cuantos conformen dicha agrupación. Dichos grupos se conocen como **campos** y se distinguen por un *nombre único* que los identifica. El *tamaño* de cada campo está dado por el número de elementos biestables que lo conforman. Un campo por tanto almacena información de una forma más organizada.

La siguiente figura muestra una representación de la memoria y de grupos de elementos biestables (campos) marcados con colores.

1	0	0	0	1	1	1	0	0	0	0	0	0	1	1	1	0	1	1
0	0	0	0	1	1	0	1	1	0	0	1	1	0	1	1	0	0	0
1	0	1	1	1	0	0	0	1	1	0	1	0	1	1	0	0	1	0
1	1	0	0	1	0	1	1	0	0	0	1	1	1					

Figura 0.3. Representación de la memoria y de grupos de elementos biestables o campos

Tipos de campos

Un campo puede ser **constante** o **variable**. Un campo variable puede cambiar su contenido durante el tiempo de ejecución del programa; por otro lado, un campo constante siempre mantendrá su valor de forma rígida mientras el programa se esté ejecutando; si se trata de cambiar el valor de un campo constante en tiempo de ejecución, se generará un error. Una constante también puede estar representada por un valor fijo y no necesariamente con un campo, como por ejemplo el literal de cadena "lógica de programación" (encerrado entre comillas), o el número 215, o el valor booleano Falso, entre otros.

Recordemos que un campo se identifica de manera única en la memoria utilizada por un programa con un nombre. Al hacer referencia a este nombre, estamos en realidad accediendo a su información o contenido, ya sea para leerla o modificarla.

Al nombrar un campo, se deben seguir unas cuantas reglas que se establecen en la lógica de programación, así como en la mayoría de lenguajes de programación:

- Puede contener caracteres alfanuméricos, pero debe iniciar con una letra
- No puede contener caracteres especiales, a excepción del guión bajo (**_** *underline*), con el cual también puede iniciar el nombre del campo (esto evita confusiones con las operaciones que realiza el computador)
- Muchos lenguajes de programación son sensibles a los caracteres, lo que significa que distinguen entre mayúsculas y minúsculas, por lo que un campo con el nombre *A* es diferente de otro con el nombre *a*.[3]

Ejemplos
Los siguientes son nombres válidos para campos:
a, x, z2, Placa, EDAD, nom, nombre_apellido, salarioEmpleado, _num3, NotaAsignatura

Los siguientes son nombres para campos no válidos:
5B, nombre-persona, num 1, z*, miedad}, mamá

Datos y tipos de datos

En la memoria se pueden agrupar varios elementos biestables para formar campos para hacer más fácil y manejable el tratamiento de la información. Al hacer referencia al nombre de un campo, hacemos mención a su contenido. El contenido (información) de un campo es conocido como **dato**.

Los lenguajes de programación permiten realizar abstracciones para no preocuparnos por los estados de cada elemento biestable o por las cadenas binarias que forman el dato del campo, pudiendo ignorar los detalles de implementación interna. Así, podemos referirnos al campo **`nombre`** cuyo contenido es **`"Ana Gil"`** o al campo **`numero`** cuyo contenido es **`10`**.

[3] En este curso asumiremos los nombres de campos sensibles a los caracteres

Cada dato contiene un *tipo* de información específica, que puede ser numérica, alfabética u otra y que obedece a la forma como haya sido definido el campo en el programa. A un campo por tanto se le asocia un **tipo de dato**, el cual le indica al programa qué tipo de información puede aceptar éste. Un tipo de dato puede ser **simple** (**primitivo**) o **compuesto** (**estructurado**). Un campo definido como tipo de dato simple solo permite almacenar un valor, mientras que en un campo compuesto se pueden almacenar varios valores y su definición es en función de los tipos de datos simples; éstos se tratarán más adelante.

Los **tipos de datos primitivos** en los lenguajes de programación, son:

1. Numéricos
    a. Enteros (*integer*)
    b. Reales (*real*)
2. Carácter, Cadena (*char*, *string*)
3. Lógicos (*boolean*)

Los lenguajes de programación modernos admiten una gran cantidad de tipos de datos primitivos para almacenar datos, entre ellos se destacan los tipos:

- Fecha (date)
- Hora (time)
- FechaHora (datetime)
- Entero largo (long int)
- Entero corto (smallint)
- Entero muy corto (tinyint)
- Real en punto flotante precisión simple (float)
- Real en punto flotante precisión doble (double)

Las fechas se suelen tratar como cadenas de caracteres cuando el tipo de dato no está disponible en el lenguaje y las horas como números enteros, aunque también pueden ser tratadas como un dato compuesto por varios números/cadenas.

Tipos de datos numéricos

Representa el conjunto de valores numéricos, que pueden ser enteros o reales (en punto flotante). Los números enteros son un subconjunto de los reales, son números que no tienen parte decimal y pueden ser negativos, positivos o cero: -9, 5, 4, 0, 500, -1005.

Los números reales a nivel computacional son en realidad aproximaciones, ya que no podemos escribir infinitas cifras decimales por las limitaciones tecnológicas. Un número real o en punto flotante está compuesto de una parte entera y posiblemente una parte decimal (mantisa): 1, -5.6, 9.66666, -0.0001, 350, 41.2. Para acortar la escritura de ciertos números muy grandes o muy pequeños, se puede utilizar la **notación científica**: `3E10` $= 3 \times 10^{10}$, `5.4e-5` $= 5.4 \times 10^{-5}$. Vale aclarar que el carácter utilizado en esta notación para separar las cifras decimales es el **punto** (**.**) y no la *coma* (,).

Tipo de dato carácter/cadena

Es el conjunto finito de todos los caracteres disponibles en el computador:

- Caracteres alfabéticos = {a, b, c, …, z, A, B, C, …, Z }
- Caracteres numéricos = {0, 1, …, 9}
- Caracteres especiales = {+, -, *, /, (,), @, ^, $, #,…}

Los datos tipo carácter están delimitados entre *comillas dobles* o *simples* (apóstrofos) y el número de caracteres que contenga determina su **longitud**: "Hola", 'Pedro Zapata', "Hoy es martes". Estas expresiones encerradas entre comillas se conocen como **literales de cadena**.

Una **cadena de caracteres** es una secuencia compuesta por caracteres del código **ASCII** (*American Standard Code for Information Interchange*) y que están disponibles en todas las distribuciones comerciales en los distintos idiomas en que están los teclados.

Tipo de dato lógico (booleano)

Permite almacenar un valor binario. Algunos lenguajes antiguos no incluían este tipo de datos y hacían el tratamiento con campos de tipo entero, con los valores 1 y 0. Las constantes lógicas que usaremos en lógica de programación, son:

- **Verdadero o (v)** (***true***)
- **Falso o (f)** (***false***)

Tipos de datos primitivos en Java y su tamaño

Los tamaños en general son muy similares en los distintos lenguajes de programación y dicha característica también depende de las características de la máquina. En los computadores comerciales la mayoría de lenguajes manejan estos tamaños para los tipos que implementan. El tipo cadena no está presente, ya que las cadenas de caracteres son tratadas como objetos de una clase (llamada **String**); cada carácter de una cadena ocupa 1B de memoria.

- **byte**: representa un tipo de dato de 8 bits con signo. Puede almacenar valores numéricos en el rango de -128 a 127, incluyendo ambos extremos. Es útil para ahorrar memoria cuando se necesita almacenar valores pequeños.
- **short**: este tipo de dato utiliza 16 bits con signo y puede almacenar valores numéricos en el rango de -32768 a 32767. Se utiliza cuando se necesita un rango más amplio que el proporcionado por los bytes, pero aún se desea ahorrar memoria en comparación con los tipos de datos más grandes.
- **int**: es un tipo de dato de 32 bits con signo utilizado para almacenar valores numéricos. Su rango va desde -2147483648 (-2 ^ 31) hasta 2147483647 (2 ^

`31 - 1`). Es el tipo de dato más comúnmente utilizado para representar números enteros.

- **long**: este tipo de dato utiliza 64 bits con signo y puede almacenar valores numéricos en el rango de `-9223372036854775808` ($(-2)^{63}$) a `9223372036854775807` ($2^{63} - 1$). Se utiliza cuando se necesitan números enteros muy grandes.
- **float**: es un tipo de dato diseñado para almacenar números en punto o coma flotante con precisión simple de 32 bits. Se utiliza cuando se requieren números decimales con un grado de precisión adecuado para muchas aplicaciones. Está en un rango de `1.4 * 10 ^ –045 a 3.4 * 10 ^ 038`.
- **double**: este tipo de dato almacena números en coma flotante con doble precisión de 64 bits, lo que proporciona una mayor precisión qué el tipo float. Se usa en aplicaciones que requieren una alta precisión en cálculos numéricos. Está en un rango de `4.9 * 10 ** –324 a 1.8 * 10 ** 308`.
- **boolean**: sirve para definir tipos de datos booleanos que pueden tener solo dos valores: **`true`** o **`false`**. Aunque ocupa sólo 1 bit de memoria, generalmente se almacena en un byte completo por razones de eficiencia.
- **char**: es un tipo de datos que representa un carácter unicode sencillo de 16 bits. Se utiliza para almacenar caracteres individuales, como letras o símbolos en diferentes lenguajes y conjuntos de caracteres.

Constantes y Variables

Un campo puede ser variable o constante. De ahora en adelante, al referirnos a un campo, asumimos de acuerdo al contexto, que es sinónimo de constante o variable para referirnos a esos espacios de la memoria donde podemos guardar un dato; así, podemos definir alternativamente como una **constante** o **variable** a *un espacio de la memoria asociado a un tipo de dato que le asignamos un nombre único y donde almacenamos un dato*.

Expresiones

Son combinaciones de constantes, variables, operadores, paréntesis y funciones especiales formadas con el objetivo de solucionar algo:

- ab^2 `- 4 / 3 + c`
- `5 ^ 3 / 10 + 2 * 6 - 4`

Una expresión no necesariamente tiene que ser aritmética como la anterior, también hay expresiones lógicas y de cadenas:

- `4 > (a + 5b)`
- `“Luis” + “Arango”` (una *suma* de cadenas se conoce como **concatenación**, una operación que permite unirlas; para el ejemplo, dicha concatenación forma la cadena `“LuisArango”`)

Operadores fundamentales en matemáticas y computación

Un *operador* es un símbolo usado en matemáticas para representar una operación a realizar, la cual puede ser unaria (con un *operando*) o binaria (con dos *operandos*). En la aritmética y en el álgebra se cuenta, entre otros, con varios operadores elementales; cada operador tiene una *prioridad* asignada, lo cual significa que los de mayor prioridad, se ejecutarán primero. Se dividen en tres grupos, de los cuales se muestra su representación matemática, así como algorítmica.

Operadores aritméticos

Utilizados para realizar cálculos u operaciones matemáticas.

Nombre Operador	**Símbolo matemático y algorítmico (computacional)**	**Prioridad**
Negación aritmética unaria	$-$	Alta
Potencia	x^y $\wedge$ $**$	Alta - media
Raíz cuadrada	$\sqrt{x}$ $\|/$ $raiz2(x)$ $sqrt(x)$	
Multiplicación	$\times$ $*$ $.$ $()()$	Media
División	$\div$ $/$ $\frac{a}{b}$ a|div	
Módulo	$\%$ mod	
División entera	$\backslash$ div $//$	
Suma	$+$	Baja
Resta	$-$	

Notas

- La negación aritmética es una operación *unaria* que consiste en negar el símbolo del número (operando). Ejemplo: $-(+2)$, $-(8)$, $-(-5)$, -94
- La prioridad se refiere al orden en que los operadores se efectúan en una expresión aritmética: los de mayor prioridad se efectúan primero.
- Las operaciones encerradas entre paréntesis se efectúan primero, por lo que tienen mayor prioridad. Los paréntesis modifican la prioridad de los operadores en una expresión.
- Si hay dos operadores de igual prioridad, se ejecuta primero el que se encuentre más a la izquierda, esto es, se sigue el orden de izquierda a derecha.

Operadores relacionales o de comparación

Permiten realizar comparaciones entre dos expresiones; el resultado de una comparación es un valor booleano (verdadero (V) o falso (F); 1 ó 0, respectivamente); esto son:

Nombre Operador	Símbolo	Prioridad
Igual	= ==	Alta
Diferente	≠ <> !=	
Mayor que	>	Media
Menor que	<	
Mayor o igual que	≥ >=	Baja
Menor o igual que	≤ <=	

Ejemplo 0.1

1. $8 \neq 9 \rightarrow (V)$
2. $9 \geq 9 \rightarrow (V)$
3. $7 \neq 14 \div 2 \rightarrow (F)$
4. $9 \times 2 \leq 50 \div 10 \rightarrow (F)$
5. $-8 = 8 \rightarrow (F)$

Operadores lógicos o booleanos

Permiten conectar expresiones de comparación y realizar operaciones lógicas. El valor devuelto (verdadero o falso) depende del conectivo lógico utilizado, según las leyes del álgebra proposicional y booleana; estos son algunos:

Nombre Operador	Símbolo	Prioridad
Negación lógica unaria	¬ ~ − ! ' no not	Alta ↓ (mayor a menor)
Conjunción	∧ && • × y and	
Disyunción	∨ || + o or	
Disyunción exclusiva	V ⊕ W 'o bien' xor eor	

Operación de asignación

Esta operación consiste en darle un valor a un campo. Si el campo es constante, dicha asignación se realiza antes de poner en marcha el programa, ya que el valor de éste no podrá cambiarse en tiempo de ejecución. También se conoce como *sentencia* o *instrucción de asignación*.

El operador utilizado en lógica de programación es una *flecha* apuntando hacia la izquierda: (←) También se suele utilizar el símbolo *igual* (=). Los lenguajes de programación utilizan en general el operador igual para la asignación. La forma de su uso se indica a continuación:

Sintaxis

En lógica podemos usar estas formas:

$constante \leftarrow valofijo$
$constante = valofijo$
$variable \leftarrow expresión$
$variable = expresión$

Ejemplo 0.2

1. $x \leftarrow 3$
2. $nombre \leftarrow "Diana\ María"$
3. $b = Verdadero$
4. $z \leftarrow 5 * x / 4$
5. $mensaje = 'Bienvenido\ a\ la\ programación'$

Nota
El operador de asignación es un operador ***asimétrico*** (a diferencia de los operadores binarios aritméticos, lógicos o relacionales que son *simétricos*), ya que la máquina primero debe evaluar la expresión a la derecha del operador de asignación y luego tomar el resultado para asignarlo a la variable que se encuentra a la izquierda de éste.

La clase Math

Muchas operaciones matemáticas no pueden ser realizadas con los operadores elementales, por lo que sería necesario desarrollar el método de efectuar dichos cálculos. Para ello los lenguajes cuentan con una serie de funciones matemáticas incorporadas para facilitar las tareas de programación. Los lenguajes que provienen del compilador de C/C++ mantienen las funciones para el tratamiento matemático, algunos haciendo uso de éstas como métodos de una clase. Lenguajes como Java y Python, entre otros, disponen de métodos similares agrupados en una clase estática (llamada **`Math`** en Java y **`math`** en Python) para realizar distintas operaciones matemáticas. A lo largo del texto se comentarán distintas funciones matemáticas que implementan estos lenguajes con dicha clase. Vale anotar que en Python varias de las funciones matemáticas se implementan por fuera de la clase math como *funciones libres* o dentro de otras clases. También hay que indicar que algunos grupos de funciones matemáticas también están incluidas en otras librerías en ambos lenguajes, lo cual se ilustrará en ejemplos más adelante.

Notaciones

Expresiones matemáticas vs. expresiones algorítmicas

Al trabajar con computadores y lenguajes de programación, algunos símbolos matemáticos son difíciles de obtener desde los caracteres estándar del teclado. Es por ello que las expresiones matemáticas deben ser reescritas cuando las llevamos a un lenguaje de programación utilizando para ello una notación algorítmica comprensible por la máquina.

En las explicaciones matemáticas que se abordarán en el texto, se utilizará ya sea la notación matemática o la notación algorítmica.

Para convertir de una notación matemática a algorítmica o viceversa, nos basaremos en los operadores vistos anteriormente y su prioridad, así como en las propiedades del álgebra para los números reales observando cuáles de estos operadores pueden ser usados en un lenguaje determinado. En lógica de programación, el tema de los operadores puede flexibilizarse, siempre manteniendo la notación algorítmica. Veamos algunos ejemplos.

Ejemplo 0.3
Escribir en notación algorítmica las siguientes expresiones matemáticas

1. $ab + 3ac^3$
2. $\sqrt[3]{b^2} + \frac{a}{3}$
3. $\frac{a-2b+3c}{\sqrt{2}}$
4. $a \geq 0 \wedge b \neq (4 + 2ab^3) \vee [\neg(a + 2 < b) \wedge (-9 = c)]$

Solución

1. $a * b + 3 * a * c \wedge 3$
2. $b \wedge (2/3) + a / 3$
3. Veamos varias formas de escribir esta expresión
   a. $(a - 2 * b + 3 * c) / 2 \wedge (1/2)$
   b. $(a - 2 * b + 3 * c) / 2 \wedge 0.5$
   c. $(a - 2 * b + 3 * c) / raizc(2);\ donde\ raizc()\ es\ una\ función$
4. Veamos cómo escribir esta expresión que incluye todos los operadores. Para la conjunción podemos usar: **y**, **and**, ó **&&**, que son admitidos en lógica de programación y algunos lenguajes; análogamente para la disyunción podemos usar: o, or ó ||. Por último, podemos usar para la negación: **no**, **not** ó **!**. Recordemos que los operadores lógicos trabajan como conectivos.

   $a >= 0\ \&\&\ b <>\ (4 + 2 * a * (b \wedge 3))\ ||\ (!\,(a + 2 < b)\ \&\&\ (-9 = c))$

Nota
Observe que por la prioridad de los operadores, no es necesario usar paréntesis en algunas expresiones, a no ser que se quiera modificar ésta.

Notaciones comunes en programación

En la escritura en general es común encontrarnos con formas particulares de representar palabras o frases. Esto ha sido llevado y aplicado en la informática con excelentes resultados, ya que ha permitido mejorar la semántica del código de programación, así como mejorar los estilos y buenas prácticas; se aplica para los nombres de variables, funciones, procedimientos y objetos, entre otros. Veamos algunas de las más utilizadas.

Notación *camel case*[4]

[4] Puede consultar más sobre esta notación en: Camel case - Wikipedia, la enciclopedia libre

La notación **camel case** o **camelCase**, (*funda de camello*) es una forma de escritura inicialmente adoptada en lengua inglesa y más tarde extendida a otros idiomas y usada particularmente en la escritura de códigos de programación; es utilizada para escribir frases cortas compuestas de unas cuantas palabras todas pegadas y en donde la letra inicial de cada una de éstas se escribe en mayúscula y las demás letras en minúscula. Su nombre obedece a la similitud con la joroba de un camello.

Hay dos formas de *camel case*:

- *UpperCamelCase*: cada palabra en mayúscula inicial; por ejemplo: `ClaseLógicaProgramación.`
- *lowerCamelCase*: o simplemente *camelCase*, es similar a la anterior, pero con la diferencia que la primera letra está en minúscula; por ejemplo: `claseLógicaProgramación.`

Notación *snake case*[5]

Es un estilo de escritura donde cada espacio se reemplaza por un guión bajo o carácter de subrayado (*underline*) (*snake_case*). Aunque se indica que esta notación debe iniciar cada palabra en minúscula, en la práctica se combinan escrituras de acuerdo a las necesidades; por ejemplo:

- `Clase_Lógica_Programación`
- `clase_lógica_programación`
- `CLASE_LÓGICA_PROGRAMACIÓN`

Notación húngara[6]

Empleada en el campo de la programación, utiliza prefijos en los nombres de las variables, dando una descripción de ellas. Fue desarrollada por el ingeniero informático Charles Simonyi, nacido en Hungría y de allí su nombre. Por ejemplo, podemos utilizar esta notación para describir el tipo de objeto en una interfaz gráfica o el tipo de dato de una variable: `numEdad`, `cadNombre`, `lstLista`, `txt_cuadroTexto`, `btn_boton_mostrar`, `log_suiche`. Observe cómo en esta notación se combinan las anteriores, tales como el *camelCase* o *snake_case*, con el fin de proveer una mejor escritura de las variables y facilitar así su lectura.

Notas

- Las notaciones *camelCase* y *snake_case*, basan su importancia en la facilidad que brinda a la hora de leer varias palabras que se encuentran unidas sin espacios, algo muy común al nombrar variables en un programa.
- Las empresas generalmente adoptan estándares propios para facilitar el desarrollo de sus tareas, por tanto, es posible que se encuentre con combinaciones de estos tipos de notaciones empleadas en proyectos.

[5] Puede consultar más sobre esta notación en: Snake case - Wikipedia, la enciclopedia libre
[6] Puede consultar más sobre esta notación en: Convenciones de codificación de Windows - Win32 apps | Microsoft Learn y Notación húngara - Wikipedia, la enciclopedia libre

Conceptos matemáticos

Los conjuntos numéricos

La idea de conjunto se emplea bastante en matemáticas, por ser un concepto básico, no tiene una definición formal. Más adelante, se hablará un poco más acerca de los conjuntos y las operaciones que podemos realizar con ellos, por ahora veremos los conjuntos de números más conocidos.

Símbolos comunes en notación de conjuntos

Existen varios símbolos muy utilizados particularmente en el tratamiento de conjuntos.

$\in$: $Pertenece.\ Ejemplo\colon\ a \in A.\ Se\ lee\colon\ 'a'\ pertenece\ (es\ elemento\ de)\ a\ 'A'$
$\notin$: $No\ pertenece.\ Ejemplo\colon\ a \notin A.\ Se\ lee\colon\ 'a'\ no\ es\ elemento\ 'A'$
$\forall_x$: $Cuantificador\ universal.\ Ejemplo\colon \forall_x\ x > 9.\ Se\ lee\colon\ Para\ todo\ x\ se\ cumple\ que\ x\ es\ mayor\ que\ 9$
$\exists_x$: $Cuantificador\ existencial.\ Ejemplo\colon\ \exists_x\ x > 9.\ Se\ lee\colon\ Existe\ al\ menos\ un\ x\ tal\ que\ x\ es\ mayor\ que\ 9$
$\exists!_x$: $Cuantificador\ existencial\ único.\ Se\ lee\colon\ Existe\ exactamente\ un\ x$
$\subset$: $Contenido\ en,\ o\ es\ subconjunto\ de.\ Ejemplo\colon\ A \subset B.\ Se\ lee\colon\ A\ es\ subconjunto\ de\ B$
$\subseteq$: $Es\ subconjunto\ de\ o\ igual\ al\ conjunto.\ Ejemplo\colon\ A \subseteq B.\ Se\ lee\colon\ A\ es\ subconjunto\ o\ igual\ a\ B$

El conjunto de los números naturales **N**

Los números naturales surgen de la necesidad del hombre por contar. Es un conjunto infinito y lo simbolizamos así:
$\aleph = \{1, 2, 3, 4, 5,...\}$

El conjunto de los números enteros **Z**

Los números enteros incluyen a los números naturales, esto es a los números positivos, a sus respectivos negativos y al cero. Es un conjunto infinito y lo simbolizamos así:

$$Z = \{..., -5, -4, -3, -2, -2, 0, 1, 2, 3, 4, 5, ...\}$$

Podemos decir entonces que $N \subseteq Z$

El conjunto de los números racionales **Q**

Este conjunto incluye a los números enteros, las fracciones como $\frac{3}{4}$, $-\frac{9}{5}$, los números decimales conmensurables como 2.33, -5.99 y los números decimales inconmensurables periódicos como 0.333333…, 6.778877887788….Es un conjunto infinito y lo simbolizamos así:

$Q = \{\frac{p}{q} \mid p \in Z \ \wedge q \in Z;\ q \neq 0\}$

Por tanto $N \subseteq Z \subseteq Q$

El conjunto de los números irracionales **Q'**

Los números que no son racionales, se denominan irracionales, y son decimales inconmensurables que no son periódicos. Ejemplos de número irracionales, son $\pi = 3.141592\ldots, \sqrt{2},\ e = 2.718281\ldots, \sqrt[3]{7.},\ etc.$ Es un conjunto infinito y lo simbolizamos así:

$Q' = \{x \in R \mid x \notin Q\}$

El conjunto de los números reales **R**

El conjunto de los números reales está conformado por los números racionales y los irracionales. Ejemplos de número reales son todos los que hemos visto anteriormente. Es un conjunto infinito y lo simbolizamos así:

$\Re = Q \cup Q'$

El conjunto de los números complejos **C**

El conjunto de los números complejos está conformado por números de la forma

$z = a + bi;\ donde\ a,\ b \in R;\ i = \sqrt{-1}\ es\ la\ unidad\ imaginaria$

Ejemplos de número complejos son: $-54, 45 + 3i, 9 - 2i.$ Decimos que los reales están estrictamente contenidos en los complejos:

$R \subset C$

Propiedades fundamentales de los números reales

El estudio de las propiedades nos permite entender para qué fines sirven, reconocer sus implicaciones y poder derivar o concluir otras cosas de ellas, en otras palabras, cómo trabajar y usarlas en distintas situaciones. Vemos cuáles son:

Sean a, b, c número reales y vamos a aplicar las operaciones de suma y multiplicación.

Observación

La suma $+$ y la multiplicación $\times$ son operaciones fundamentales, mientras que la resta $-$ y la división $\div$ son derivaciones de éstas. Estas operaciones son *binarias*, en el sentido que involucra dos *operandos*.

Propiedad asociativa

La suma y la multiplicación son asociativas, esto es, los operandos se pueden agrupar de cualquier forma.

$a + (b + c) = (a + b) + c = a + b + c$
$a.(b.c) = (a.b).c = a.b.c$

Propiedad conmutativa

La suma y la multiplicación son conmutativas, es decir, el orden de los sumandos o los factores no altera ni la suma ni el producto, respectivamente.

$a + b = b + a$
$a.b = b.a$

Módulo de la suma y el producto. Elementos neutros

El módulo o elemento neutro, es un valor que al ser computado, no altera el resultado. El módulo o elemento neutro en la suma es el cero (0), ya que al sumar cualquier número por éste, obtenemos el mismo número. Similarmente sucede en la multiplicación, donde el uno (1) es el elemento neutro o módulo de esta operación. Los números 0 y 1 también son llamados *elementos identidad* para la suma y la multiplicación, respectivamente.

$a + 0 = 0 + a = a$
$a \times 1 = 1 \times a = a$

Inversos

Todo número real **a** tiene su *inverso aditivo* **-a** (llamado también el negativo de **a**) y satisface:

$a + (-a) = (-a) + a = 0$

Similarmente, todo número **a** distinto de cero tiene un único *inverso multiplicativo* $\mathbf{a^{-1}}$ que satisface:

$a.(a^{-1}) = (a^{-1}).a = 1,\ si\ a \neq 0$

Cabe recordar que $a^{-1} = \frac{1}{a}$.

Propiedad distributiva

$a.(b + c) = a.b + a.c$
$(b + c).a = b.a + c.a$

Resta y división

La suma y la multiplicación son operaciones básicas, mientras que la resta y la división derivan de éstas. La resta es la suma de un inverso aditivo y la división es la multiplicación por un recíproco. Tenemos entonces.

$$a - b = a + (-b)$$
$$\frac{a}{b} = a \div b = a.b^{-1} = a.\frac{1}{b}$$

Técnicas de demostración matemática

Proposición

Es un enunciado del cual se puede afirmar con exactitud si es falso o verdadero.

Axioma

Es una proposición que por lo evidente, no requiere demostración. También se puede decir acerca del axioma, que es toda proposición que no es deducida de otras.

Postulado

Es una proposición no evidente por sí misma ni demostrada, pero que se acepta como verdadera, ya que no existe otro principio al que pueda ser referida. En otras palabras, es una proposición cuya verdad se admite sin pruebas y que es necesaria como base en razonamientos ulteriores.

Demostración

Una demostración o prueba en matemáticas, y que es una herramienta fundamental en esta disciplina así como en las ciencias naturales y la ingeniería, es un argumento deductivo para asegurar la verdad de una proposición. En la argumentación se pueden usar otras afirmaciones previamente establecidas, tales como teoremas, definiciones, las afirmaciones iniciales (hipótesis) y los axiomas y postulados aceptados.

Existen básicamente cuatro formas de demostración en matemáticas, a saber: directa, reducción al absurdo, inducción matemática y el contraejemplo. Veamos primero algunos conceptos utilizados en la jerga matemática para luego ver en general cada tipo de demostración. En distintas partes del texto se ilustran estos métodos con ejemplos.

Teorema

Es una proposición verdadera que ha sido demostrada previamente por medio de axiomas, postulados y otros teoremas ya demostrados.

Hipótesis

Son las suposiciones iniciales o premisas que se consideran verdaderas. Es el punto de partida que se considera válido cuando se va a demostrar un teorema.

Tesis o conclusión

Es la proposición que se quiere demostrar como verdadera.

Nota

La demostración no solo usa lo que entrega la hipótesis, en los pasos lógicos y reglas de inferencia que se apliquen, es posible utilizar proposiciones que ya han sido demostradas (teoremas)[7], así como los axiomas y postulados aceptados. No sobra decir que una demostración no debería utilizar resultados de otros teoremas que aún no han sido demostrados.

Técnica de demostración: Demostración directa

Se trata, a partir de una hipótesis y los conocimientos de los axiomas, postulados, teoremas, leyes, etc. existentes, poder probar la veracidad de la tesis propuesta, utilizando una secuencia de pasos lógicos y reglas de inferencia válidas.

Técnica de demostración: Reducción al absurdo (*Reductio ad absurdum*)

También conocida como demostración por **contradicción**, se asume como verdad la negación o falsedad de la proposición que se quiere demostrar, y siguiendo reglas y argumentos lógicos, se busca llegar a una contradicción de lo que se asumió como verdad, lo cual permite inferir que la proposición original era verdadera.

Técnica de demostración: Inducción matemática

El método de inducción matemática es una técnica para demostrar proposiciones que involucran a los números naturales. Consiste en dos pasos: demostrar que la proposición es verdadera para el primer número natural (generalmente 1) y luego demostrar que si la proposición es verdadera para un número natural cualquiera, también lo es para el siguiente número natural.

Pasos de la demostración por inducción:

1. Base de la inducción: se verifica que la proposición es verdadera para el primer número natural (generalmente $n = 1$).
2. Hipótesis inductiva: se asume que la proposición es verdadera para un número natural arbitrario k (donde $k \geq 1$). Es decir, se asume que $P(k)$ es verdadera.

[7] Un teorema es una proposición que se ha demostrado que es verdadera, mientras que una proposición es una afirmación que puede ser verdadera o falsa.

3. Paso inductivo: se demuestra que si la proposición es verdadera para `k`, entonces también es verdadera para `k + 1`. Esto significa probar que `P(k)` implica `P(k+1)`.

Si se cumplen estos tres pasos, se puede concluir que la proposición es verdadera para todos los números naturales.

Cada una de las condiciones 1 y 2 tiene su propio significado especial. La condición 1 proporciona la base para la inducción; la condición 2 proporciona la justificación para generalizar a partir de esta base, o sea, la justificación para pasar de un caso especial hacia el siguiente, de `n` hacia `n + 1`. Si no se satisface la condición 1, entonces no existe base para aplicar el método de la inducción matemática, aún si se satisface la condición 2. Por otra parte, cuando sólo se satisface la primera condición y no la segunda, aunque se proporcione una base para la inducción, no existe justificación para hacer la generalización.

La inducción tiene amplias aplicaciones en las matemáticas, pero debe usarse con cuidado o puede conducir a conclusiones erróneas. En los siguientes ejemplos se ilustra cómo usar esta técnica de demostración.

Ejemplo

Demostrar que la suma de los primeros n números naturales es igual a `1 + 2 + 3 + ... + n = n * (n + 1) / 2`.

Demostración

Probemos inicialmente que la proposición es igual para `n = 1`: La suma de los números de 1 a 1 es 1 y con la fórmula se verifica fácilmente que: `1 * (2 + 1) / 2 = 1 * 2 / 2 = 2 / 2 = 1`.

Ahora supongamos que la proposición se cumple para `n = k: 1 + 2 + 3 + ... + k = k * (k + 1) / 2`.

Demostremos que la suma de los `k + 1` es `(k + 1) * (k + 1 + 1) / 2 = (k + 1) * (k + 2) / 2 = (k + 1) * (k / 2 + 1)`, con lo que se demuestra que si la proposición es verdadera para `n = k` lo será también para `n = k + 1`.

Se tiene que:

```
1 + 2 + 3 + ... + k + (k + 1) = k * (k + 1) / 2 + (k + 1)
```

Factorizando el miembro derecho de la expresión anterior, obtenemos

```
(k + 1) * (k / 2 + 1)
```

Con lo cual se demuestra que la proposición es verdadera para cualquier número natural.

Ejemplo

La sucesión de los números impares naturales está dada por `1, 3, 5, 7, 9, ...`

Encontrar la fórmula que exprese el número impar u_n en términos del índice n y la validez de dicha expresión en general.

Demostración
Tomemos el primer número de esta sucesión como u_1, el segundo como u_2, etc. así:
$u_1 = 1,\ u_2 = 3,\ u_3 = 5,\ u_4 = 7,\ u_5 = 9,\ \ldots$

$u_1 = 2 * 1 - 1 = 1$
$u_2 = 2 * 2 - 1 = 3$
$u_3 = 2 * 3 - 1 = 5$
$u_4 = 2 * 4 - 1 = 7$
$u_5 = 2 * 5 - 1 = 9$
…
$u_n = 2 * n - 1$

Veamos que la expresión $u_n = 2n - 1,\ n \in \mathbf{N}$ para encontrar cualquier número impar natural es válida en general.

Ya vimos que la fórmula se cumple para $n = 1,\ u_1 = 2 * 1 - 1 = 1$

Supongamos que la expresión se cumple para $n = k, u_k = 2 * k - 1, k \in \mathbf{N}$

Probemos que la expresión se cumple también para $n = k + 1$: $u_{k+1} = 2 * (k + 1) - 1 = 2 * k + 2 - 1 = 2 * k + 1$

Para obtener el $(k + 1)$-ésimo número impar basta con sumar 2 al k-ésimo número impar, el cual por hipótesis está dado por $u_k = 2 * k - 1$. Por tanto:
$u_{k+1} = (2 * k - 1) + 2 = 2 * k + 1$ y se concluye que $u_n = 2 * n - 1$

La inducción matemática puede aplicarse a los números enteros. Aunque comúnmente se asocia con los números naturales (enteros positivos), el principio de inducción matemática puede extenderse para probar afirmaciones sobre cualquier conjunto bien ordenado, incluyendo los enteros. La clave es adaptar el paso base y el paso inductivo para que sean válidos dentro del conjunto de enteros.

El principio de inducción matemática se basa en el concepto de que si una propiedad es verdadera para un caso base y si, asumiendo que es verdadera para un valor cualquiera, también es verdadera para el siguiente valor, entonces la propiedad es verdadera para todos los valores en el conjunto.

Ejemplo
Si n es un número entero, entonces $2n + 1$ es un número impar.

Demostración
Tomemos como caso base $n = 0$: $2(0) + 1 = 1$, que es un número impar

Supongamos que para $n = k$ se tiene que $2k + 1$ es un número impar
Probemos que para $n = k + 1$ se cumple que $2(k + 1) + 1$ es un número impar
Resolviendo
$2(k + 1) + 1 = 2k + 2 + 1 = 2k + 3$
Por hipótesis sabemos que $2k + 1$ es impar, por tanto, si sumamos 2 a este número, obtenemos el siguiente número impar: $2k + 1 + 2 = 2k + 3$
De esto concluimos que $2n + 1$ es un número impar

Ejemplo
La suma de los primeros n naturales impares es igual a n^2.
Por ejemplo, si $n = 4$: $1 + 3 + 5 + 7 = 16 = 4^2$.

Demostración
Para $n = 1$ se tiene que $1^2 = 1$.
Supongamos que si $n = k$ se cumple que la suma de los primeros k naturales es k^2.
Probemos que si $n = k + 1$ se cumple que la suma de los primeros $k + 1$ naturales es $(k + 1)^2$. Hasta el k-ésimo elemento la suma es k^2; basta con sumar 2 a este número para obtener el (k + 1)-ésimo elemento y sumarlo, pero $u_k = 2k - 1$ como se vio anteriormente, por tanto $k^2 + (2k - 1 + 2) = k^2 + 2k + 1 = (k + 1)^2$. Esto demuestra que la suma de los primeros n naturales impares es igual a n^2.

Técnica de demostración: Contraejemplo

Se trata de mostrar con un caso particular que la generalidad de una afirmación no es cierta, esto es, es falsa.

Ejemplo 0.
Afirmación: “Todos los números primos son impares”

Esta afirmación se puede contradecir fácilmente mostrando que el 2 es un número primo que no es impar. Este contraejemplo derrumba la afirmación anterior, por lo que la convierte en una afirmación falsa.

Paradojas matemáticas

Se llaman paradojas matemáticas a ciertos que conllevan a resultados falsos y que parecen deducirse de razonamientos rigurosos, pero durante los cuales se efectuaron operaciones sin sentido o erróneas.

Ejemplo 0.[8]
Demostrar que “*1 = 2*”.

Demostración

[8] Tomado de: Malba Tahan. El Hombre que Calculaba. Romance. Aventuras de un singular calculista persa. Segunda edición ampliada. Ed. Camacho Roldán. Bogotá, Colombia

Sean a, b dos números tales que $a = b$. Ahora, multipliquemos ambos miembros de esta ecuación por a:

$a^2 = ab$

Restemos en ambos miembros de esta nueva ecuación b^2:

$a^2 - b^2 = ab - b^2$

Factorizando:

$(a + b)(a - b) = b(a - b)$

Ahora, dividamos ambos miembros de esta ecuación por $(a - b)$, para lo cual obtenemos:

$(a + b) = b$. Como $a = b$, $podemos\ sustituir: (b + b) = b \Rightarrow 2b = b$.

Y dividiendo por b: $2 = 1$

Todo parece muy bien, pero dentro de las operaciones realizadas hubo un error lógico que pasó inadvertido, el cual se presenta cuando divide ambos miembros de la ecuación por $(a - b)$, operación que da 0 pues $a = b$, y sabemos que la división por 0 carece de sentido.

Este razonamiento incorrecto nos lleva por tanto a un resultado **absurdo**, y que nos dice acerca del cuidado riguroso que se debe tener al validar un enunciado, teorema, teoría o ley científica.

Números pares e impares

Un número ***par*** es un *número entero,* tal que puede escribirse como:

$2k,\ con\ k \in Z$

Decimos que los números pares son exactamente divisibles por 2 y también son múltiplos de 2.

El conjunto de los números pares es infinito:

$P = \{..., -6, -4, -2, 0, 2, 4, 6, ...\}$

Ejemplo 0.4

1. $0 = 2 \times 0,\ 0 \in Z$
2. $2 = 2 \times 1,\ 1 \in Z$
3. $4 = 2 \times 2,\ 2 \in Z$
4. $6 = 2 \times 3,\ 3 \in Z$
5. $-8 = 2 \times (-4),\ -4 \in Z$

Un número ***impar*** es un *número entero,* tal que puede escribirse como:

$2k + 1, \; con \; k \in Z$

El conjunto de los números impares también es infinito:

$I = \{..., -7, -5, -3, -1, 1, 3, 5, 7, \; ...\}$

Ejemplo 0.5

1. $1 = 2 \times 0 + 1, \; 0 \in Z$
2. $3 = 2 \times 1 + 1, \; 1 \in Z$
3. $5 = 2 \times 2 + 1, \; 2 \in Z$
4. $7 = 2 \times 3, \; 3 \in Z$
5. $-9 = 2 \times (-5) + 1, \;\; -5 \in Z$

Propiedades de las operaciones entre números pares e impares

Sean p_1, p_2 dos números pares e i_1, i_2 dos números impares. Se cumplen las siguientes operaciones entre ellos:

1. $p_1 + p_2 = 2n$
2. $p_1 . p_2 = 2n$
3. $p_1 + i_1 = 2n + 1$
4. $p_1 . i_1 = 2n$
5. $i_1 + i_2 = 2n$
6. $i_1 \times i_2 = 2n + 1$

Demostración

1. $p_1 + p_2 = 2a + 2b = 2(a + b) = 2c = 2n$
2. $p_1 . p_2 = 2a \times 2b = 2(a . 2b) = 2c = 2n$
3. $p_1 + i_1 = 2a + 2b + 1 = 2(a + b) + 1 = 2c + 1 = 2n + 1$
4. $p_1 . i_1 = 2a(2b + 1) = 2a \times 2b + 2a = 2(a . 2b) + 2a = 2(c + a) = 2n$
5. $i_1 + i_2 = 2a + 1 + 2b + 1 = 2(a + b + 1) = 2c = 2n$
6. $i_1 \times i_2 = (2a + 1)(2b + 1) = 2a \times 2b + 2a + 2b + 1 = 2(a . 2b) + 2a + 2b + 1$
   $= 2(2ab + a + b) + 1 = 2c + 1 = 2n + 1$

Nota: Paridad del cero

El cero (0) es un número par que cumple con las propiedades comentadas arriba. Se deja como ejercicio verificar esto.

Sumatoria, Productoria y Factorial

Sumatoria

Es una notación matemática utilizada para representar la suma de varios términos, o incluso infinitos, como es común encontrar en el cálculo, lo que simplifica la escritura de sumas grandes.

La sumatoria se representa con la letra griega Σ (*sigma* mayúscula).

Nota
La sumatoria es una suma, por tanto podemos utilizar ambos términos sin problema en este contexto.

Notación

$$\sum_{i=m}^{n} a_i = a_m + a_{m+1} + a_{m+2} + \ldots + a_n$$

Esto se lee: Sumatoria de *i*, desde *i* igual a *m* hasta *n*, de *a* sub *i*

Se debe cumplir, además, que:
$m \leq n$

Si $m = n$, entonces $\sum_{i=m}^{m} a_i = a_m$

Por definición, si $m > n$, entonces $\sum_{i=m}^{n} a_i = 0$

El número de términos a sumar es, por tanto: $n - m + 1$

Ejemplo 0.6

1. $\sum_{i=1}^{5} i = 1 + 2 + 3 + 4 + 5 = 15.\ Términos\ a\ sumar\colon 5 - 1 + 1 = 5$
2. $\sum_{i=3}^{8} i = 3 + 4 + 5 + 6 + 7 + 8 = 33.\ Términos\ a\ sumar\colon 8 - 3 + 1 = 6$
3. $\sum_{i=1}^{3} i^2 = 1^2 + 2^2 + 3^2 = 1 + 4 + 9 = 14$
4. $\sum_{i=1}^{\infty} i = 1 + 2 + 3 + 4 + 5 + \ldots = \infty.\ Términos\ a\ sumar\colon infinitos$

Productoria

Es una notación matemática utilizada para representar el producto de varios términos, o incluso infinitos, simplificando la escritura de grandes multiplicaciones.
La sumatoria se representa con la letra griega Π (*pi* mayúscula).

Nota
La productoria es una multiplicación, por tanto podemos utilizar ambos términos sin problema en este contexto.

Notación

$$\prod_{k=m}^{n} a_k = a_m + a_{m+1} + a_{m+2} + \ldots + a_n$$

Esto se lee: Productoria de *k*, desde *k* igual a *m* hasta *n*, de *a* sub *k*

Se debe cumplir, además, que:
$m \leq n$

Si $m = n$, entonces $\prod_{k=m}^{n} a_k = a_m$

Por definición, si $m > n$, entonces $\prod_{k=m}^{n} a_k = 1$

El número de términos a multiplicar es, por tanto: $n - m + 1$

Ejemplo 0.7

1. $\prod_{k=1}^{5} k = 1 \times 2 \times 3 \times 4 \times 5 = 120.\ Términos\ a\ multiplicar\colon 5 - 1 + 1 = 5$
2. $\prod_{k=5}^{8} k = 5 \times 6 \times 7 \times 8 = 1680.\ Términos\ a\ multiplicar\colon 8 - 5 + 1 = 4$
3. $\prod_{k=1}^{3} k^2 = 1^2 \times 2^2 \times 3^2 = 1 \times 4 \times 9 = 36$
4. $\prod_{k=1}^{\infty} k = 1 \times 2 \times 3 \times 4 \times 5 \times \ldots = \infty.\ Términos\ a\ multiplicar\colon infinitos$

Factorial

El *factorial* de un número entero positivo, se define como el producto de los números desde uno hasta dicho número. Se simboliza acompañando al número del signo de cierre de exclamación (*!*).

$n! = 1 \times 2 \times 3 \times \ldots \times (n-2) \times (n-1) \times n$
Y se lee: "*n factorial*" o "*factorial de n"*.

Dado que conocemos la notación de productoria, podemos escribir:

$$n! = \prod_{k=1}^{n} k$$

El factorial de *n* ($n \in Z_+ - enteros\ positivos\ -$) también se define de manera *recursiva*:

$n! = 1,\ si\ n = 0\ o\ n = 1$
$n! = n(n-1)!,\ si\ n > 1$

Esto se conoce como un proceso *recurrente* (*recursivo*), ya que se llama a sí mismo hasta llegar a un estado básico, el cual determina cuándo debe dejar de ser recurrente el proceso para no caer en un ciclo infinito; para el caso del factorial, sus estados básicos son 0 y 1.

Observe que por definición tenemos que:

$0! = 1$

Ejemplo 0.8

1. $5! = 5 \times 4! = 5 \times 4 \times 3! = 5 \times 4 \times 3 \times 2! = 5 \times 4 \times 3 \times 2 \times 1! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
2. $4! = 4 \times 3 \times 2 \times 1 = 24$
3. $7! = 5040$

Números primos

Un número primo es un número entero positivo que tiene solo dos divisores exactos distintos: el mismo número y la unidad.

Teorema (Euclides, propiedad XX libro IX Los Elementos)
Existen infinitos números primos

Demostración
Sea P el conjunto de los números primos. Supongamos que P es finito. P ≠ ∅, pues 2 es primo (2 ∈ P). Multipliquemos los elementos de P = {a_1, a_2, a_3, …, a_n,}:

x = a_1 . a_2, . a_3 … a_n.

Sea y = x + 1

Sea d ∈ Z | d es el menor entero tal que: d > 1 ^ y % d = 0.

Es claro que d existe, dado que y > 1 y puede pasar que d = y

Observemos que d es primo, pues en caso contrario todo divisor propio[9] de d dividiría también a 'y' y sería más pequeño que d (por ejemplo: y = 28, d = 4, 2 < d y divide a ambos), pero esto está en contradicción con la definición de d que debe ser el menor entero tal que y % d = 0 ^ d > 1.

(Obsérvese que en la justificación anterior hay una contradicción en sí misma anidada)

Ahora, sabemos por hipótesis que P es el conjunto finito de números primos, por tanto d ∈ P. Como $x = \prod_{i=1}^{n} P_i$ (productoria desde i = 1 hasta n de P_i), entonces d es divisor de x (x % d = 0), pero y = x + 1. Se tiene entonces que hay un entero d > 1 que divide exactamente a dos enteros consecutivos, lo cual es imposible. Por tanto la suposición inicial es falsa y se tiene que P es un conjunto infinito.

Observación

Esta demostración se puede convertir en un algoritmo: un conjunto finito P de números primos permite generar un nuevo número primo d que no está en P (d ∉ P)

Ejemplo 0.9

Los siguientes son algunos números primos

2, 3, 5, 7, 11, 13, 17, 23, 29, 31,...

Ejemplo 0.10

Programa para generar números primos usando el teorema de Euclides y otros métodos a partir de un conjunto de números primos y que utiliza dos clases para realizar las operaciones.

Operations.java

```
package com.packages.operations;

public class Operations
{
    public static long factorialIterative(long n)
    {
        long f = 1;
        for (long i = n; i >= 1; i--) {
            f *= i;
        }
        return f;
    }

    public static long factorialRecursive(long n)
    {
        long f = n;
```

[9] Un divisor propio de un número es cualquier divisor positivo de ese número, excluyendo el mismo número. Por ejemplo, los divisores propios de 12 son 1, 2, 3, 4 y 6.

```
        if (n == 0) {
            return 1;
        } else {
            return f * factorialRecursive(n - 1);
        }
    }
}
```

primes.py

```
# import Operations
from operations import Operations

class PrimeNumbers:

    # Constructor
    def __init__(self) -> None:
        pass

    # Generar un nuevo primo usando el teorema de Euclides
    def new_prime_Euclides(self, P) -> int:
        # prod = Operations.Operations()
        prod = Operations()
        x = prod.product_PI(P)
        y = x + 1
        d = 2
        while y % d != 0:
            d += 1
        return d

    # Generar un nuevo primo sin usar Euclides
    def new_prime(self, P) -> int:
        x = max(P) + 1
        while (not self.prime(x)):
            x = x + 1
        return x

    # Determinar si un número es primo
    def prime(self, n) -> bool:
        i = 2
        sw = True
        while i <= n ** 0.5 and sw:
        # while i <= n / 2 and sw:
            if n % i == 0:
                sw = False
            else:
                i = i + 1
```

```
        return sw

    def prime_improved(self, n) -> bool:
        if n % 2 == 0:
            sw = False
        else:
            i = 3
            sw = True # Supuesto: n es primo
            while i < n ** (1/2) and sw:
                if n % i == 0:
                    sw = False
                else:
                    i+=2
        return sw
```

operations.py

```
class Operations:

    # Constructor
    def __init__(self) -> None:
        pass

    # Sumatoria
    def sum_naturals_iterative(self, n) -> int:
        s = 0
        for i in range(1, n + 1, 1):
            s += i
        return s

    def sum_naturals_Gauss(self, n) -> int:
        s = n * (n + 1) / 2
        return s

    def sum_naturals_recursive(self, n) -> int:
        s = n
        if n == 1:
            return n;
        else:
            return s + sumNaturalsRecursive(n - 1)

    # Productoria
    def product_PI(self, P) -> int:
        prod = 1
        for i in P:
            prod *= i
        return prod
```

```
        # 38.709.183.810.570s 8 primeros elementos al ejecutar
        # new_prime_Euclides()
```

main_menu.py

```
from os import system
import time

from primes import PrimeNumbers

start = time.time()
prime_numbers = [2]
# prime_numbers = {2}
print(f"Números primos Euclides 1: {prime_numbers}")
for i in range(1, 8, 1):
    prime_numbers.append(
    prime.new_prime_Euclides(prime_numbers))
    # prime_numbers.add(prime.new_prime_Euclides(prime_numbers))
    print(f"Números primos Euclides {i + 1}: {prime_numbers}")
end = time.time()
print(f"Tiempo empleado: {end - start}\033[32ms\033[97m")
print("-" * 50)

start = time.time()
prime_numbers2 = [2]
# prime_numbers2 = {2}
# print(f"Números primos 1: {prime_numbers2}")
for i in range(1, 200, 1):
    prime_numbers2.append(prime.new_prime(prime_numbers2))
    # prime_numbers2.add(prime.new_prime(prime_numbers2))
print(f"Números primos ({i + 1}): {prime_numbers2}")
end = time.time()
print(f"Tiempo empleado: {end - start}\033[32ms\033[97m")
```

Potencias, exponentes y radicales

Potencia

Potencia de una expresión es la misma expresión o el resultado de tomarla como factor dos o más veces. La potencia es una operación basada en multiplicaciones sucesivas, donde se toma un número llamado *base* y se multiplica tantas veces indique otro número llamado *exponente*.

$a^b = c,\ donde$
$a\colon\ base$

b: $exponente$
c: $potencia$

Exponentes

Los exponentes están ligados a unas reglas que nos facilitan el trabajo de cálculo con potencias. Aunque las reglas se aplican en general a todos los números, los exponentes fraccionarios tienen una interpretación que veremos más adelante.

Exponentes enteros

Veamos las principales reglas para trabajar con exponentes enteros.

1. $a^m . a^n = a^{m+n}$
2. $(a^m)^n = a^{m.n}$
3. $a^0 = 1\ si\ a \neq 0,\ o\ lo\ que\ es\ equivalente\ a\ decir\colon\ a^{-n} = \frac{1}{a^n},\ si\ a \neq 0$
4. $\frac{a^m}{a^n} = a^{m-n},\ si\ a \neq 0$
5. $(a.b)^n = a^n . b^n$
6. $(\frac{a}{b})^n = \frac{a^n}{b^n},\ si\ b \neq 0$

Exponentes fraccionarios

Los exponentes fraccionarios tienen una interpretación propia, tal y como se enuncia en el texto Álgebra Elemental "*Toda cantidad elevada a un exponente fraccionario equivale a una raíz cuyo índice es el denominador del exponente y la cantidad subradical la misma cantidad elevada a la potencia que indica el numerador del exponente*"[10]. Esto es:

$a^{\frac{m}{n}} = \sqrt[n]{a^m}$

Radicales

Recordemos las partes de un radical:

$\sqrt[n]{a^m} = b\ ,\ donde$
$\sqrt{\ }\ símbolo\ \ o\ símbolo\ radical$
n: $índice\ del\ radical\ (si\ no\ aparece,\ se\ sobreentiende\ que\ es\ 2\ o\ raíz\ cuadrada)$
a^m: $cantidad\ subradical$
b: $raíz$

Propiedades de los radicales

$\sqrt[n]{a.b.c} = \sqrt[n]{a}.\sqrt[n]{b}.\sqrt[n]{c}$

[10] Álgebra de Elemental. Aurelio Baldor. Interpretación del exponente fraccionario. Página 402

$\sqrt[n]{\frac{a}{b}} = \frac{\sqrt[n]{a}}{\sqrt[n]{b}}\ si\ b \neq 0$

Del álgebra sabemos que el símbolo $\sqrt{a},\ a \geq 0$, se define como el único *x no negativo* tal que $x^2 = a$

Ejemplo 0.7

$\sqrt{4} = 2; \sqrt{0} = 0; \sqrt{\frac{9}{25}} = \frac{3}{5}$

Nota

$\sqrt{4} \neq -2,\ aun\ cuando\ (-2)^2 = 4,\ ya\ que\ \sqrt{4}\ denota\ \acute{u}nicamente\ la\ raíz\ positiva\ de\ 4$

De la definición de $\sqrt{a}$ se deduce que:

$\sqrt{x^2} = |x|$

Ejemplo 0.8

$\sqrt{5^2} = |5| = 5;\ \sqrt{(-3)^2} = |-3| = 3$

Cálculo de potencias y raíces en los lenguajes de programación Java y Python

Algunos lenguajes disponen de un operador, de una función o método, o de ambas. Java y Python cuentan con el método **pow**, que requiere de dos argumentos: base y exponente. Python también pone a disposición el operador ****** (doble asterisco) para efectuar potencias

Nota

Siempre es preferible elegir el operador sobre la función o método, si el lenguaje dispone de éste, por cuestiones de rendimiento, ya que el operador tiene implementación nativa y no requiere ejecutar líneas adicionales de código.

La raíz cuadrada se calcula con el método **sqrt**, también disponible en Java y Python. Para raíces de índice superior a dos, utilizamos la potencia con un exponente fraccionario.

En Java

Potencias

```
Math.pow(base, exponente)
```

Raíces

```
Math.sqrt(cantidad_subradical)
Math.pow(cantidad_subradical, exponente_fraccionario)
```

En Python

Potencias
math.pow(*base, exponente*)
*base ** exponente*

Raíces
math.sqrt(*cantidad_subradical*)
math.pow(*cantidad_subradical, exponente_fraccionario*)
*cantidad_subradical ** exponente_fraccionario*

Otras operaciones aritméticas

Orden y Valor absoluto

Orden
En términos *geométricos*, decimos que un número **a** es menor que otro número **b**, si **a** se encuentra a la izquierda de **b** en la recta numérica. Recordemos que todo número real, excepto el 0, es negativo o positivo; entonces, si tenemos estos números, decimos que:

$a < b \; si \; b - a > 0$, *esto es,* $b - a$ *es positivo*

1. $-3 < -2$ *ya que* $-2 - (-3) = -2 + 3 = 1 > 0$ *positivo*
   Observe además, que -2 *está a la derecha de* -3 *en la recta numérica*
2. $5 < 11$ *ya que* $11 - 5 = 6 > 0$ *positivo* (5 *está a la izquierda de* 11 *en la recta numérica*)

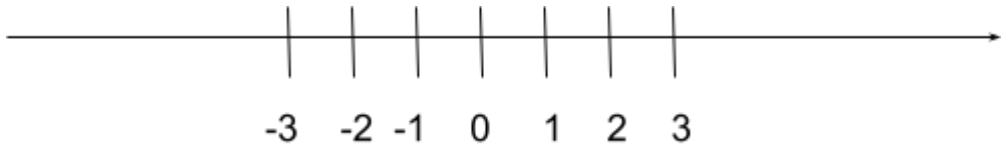
Valor absoluto
En términos *geométricos*, se define el **valor absoluto** de un número como la distancia que hay desde el cero (0) hasta dicho número en la *recta numérica*; en términos prácticos, la función valor absoluto tiene como fin convertir un número a positivo. Se simboliza encerrando el número entre dos barras: **| a |** donde ***a*** es cualquier número real.

Estrictamente, la función valor absoluto de un número se define así:

$|a| = a \; si \; a \geq 0$ ó $|a| = -a, \; si \; a < 0$

Observando la recta numérica, podemos ver que la distancia desde el 0 hasta el 3 y desde el 0 hasta el -3 es la misma, esto es, 3 unidades; aplicando la definición de valor absoluto, obtenemos de igual manera 3:

$|3| = 3$
$|-3| = -(-3) = 3$



Propiedades (teoremas y corolarios) del valor absoluto

Sean $a, b \in R$ Se cumplen las siguientes propiedades para el valor absoluto.

1. $||a|| = |a|$
2. $|-a| = |a|$
3. $|a.b| = |a|.|b|$
4. $\left|\frac{a}{b}\right| = \frac{|a|}{|b|} \; si \; b \neq 0$
5. $|a|^2 = a^2$
6. $|x| < a \; si \; y \; solo \; sí \; -a < x < a$
7. $|x| \leq a \; si \; y \; solo \; sí \; -a \leq x \leq a$
8. $|x| > a \; si \; y \; solo \; sí \; x > a \; o \; x < -a$
9. $|x| \geq a \; si \; y \; solo \; sí \; x \geq a \; o \; x \leq -a$
10. $|a + b| \leq |a| + |b| \; (desigualdad \; triangular)$

Java y Python disponen de la función **abs** para encontrar el valor absoluto de un número. En Java el método hace parte de la clase Math: `System.out.println("Abs: " + Math.abs(-60.17));`. Mientras que en Python es una función libre: `print(f"Abs: {abs(-60.17)}")`. En ambos casos devuelve `60.17`.

Módulo

Es una división entera que devuelve el residuo de ésta. Se representa con el símbolo **%** en Java, Python y otros lenguajes, o con la palabra **mod**, en algunos otros lenguajes.

División entera

La división entera se encarga de devolver solo la parte entera de dividir dos números enteros.

No todos los lenguajes implementan esta operación, pero contienen alternativas para redondear la división a un valor entero. Python implementa el operador **//** para la división entera. Los lenguajes fuertemente tipados devuelven la parte entera de una división entre dos números enteros.

Esta operación también puede verse como una operación de redondeo a enteros al dividir números no necesariamente enteros. Veremos más adelante cómo puede redondearse números reales a un valor entero o *truncado* a cierto número de decimales.

División entera y módulo por medio de restas sucesivas

Dados dos números enteros podemos encontrar su cociente entero y residuo a través de restas sucesivas.

Ejemplo 0.9

Encontrar la parte entera y el módulo de la división de 17 entre 4. Para esto tomamos el dividendo y le restamos el divisor, redefiniendo sucesivamente el dividendo hasta obtener un resultado inferior al divisor.

```
Resta 1: 17 - 4 = 13
Resta 2: 13 - 4 = 9
Resta 3: 9 - 4 = 5
Resta 4: 5 - 4 = 1
```

El número de operaciones (restas) da el resultado de la división entera (cociente entero), mientras que el resultado de la última operación da el módulo de la división. Es claro que `17 // 4 = 4 y sobra 1`.

Redondeo

Esta operación consiste en aproximar al siguiente o anterior valor (entero) según la cifra primera decimal: si es mayor o igual a cinco se aproxima al siguiente entero, en caso contrario, al anterior. Esta función puede extenderse para indicar a cuántos decimales se desea aproximar un número real.

```
redondeo(3.56) = 4
redondeo(4.7) = 5
redondeo(1.46) = 1
redondeo(-3.56) = -4
redondeo(5) = 5
redondeo(5.38, 1) = 5.4
```

Redondeo por debajo

Esta función se conoce también como **redondeo por el piso** y devuelve el mayor entero que es menor o igual al número a redondear: $redondeoPiso(x) = a \in Z \,/\, a \leq x$

```
redondeoPiso(3.56) = 3
redondeoPiso(4.7) = 4
redondeoPiso(1.46) = 1
redondeoPiso(-3.56) = -4
redondeoPiso(5) = 5
```

Redondeo por encima

Esta función se conoce también como **redondeo por el techo** y devuelve el menor entero que es mayor o igual al número a redondear: $redondeoTecho(x) = a \in Z \,/\, a \geq x$

```
redondeoTecho(3.56) = 4
```

```
redondeoTecho(4.7) = 5
redondeoTecho(1.46) = 2
redondeoTecho(-3.56) = -3
redondeoTecho(5) = 5
```

Para los casos anteriores, las funciones y métodos correspondientes en Java y Python son: **`round(x)`**, **`floor(x)`** y **`ceil(x)`** (`x ∈ double/float`).

```
// En Java
Math.round(3.56) // Devuelve 4
Math.round(-3.56) // Devuelve -4
Math.floor(3.56) // Devuelve 3
Math.floor(-3.56) // Devuelve -4

# En Python
math.ceil(3.56) # Devuelve 4
math.ceil(-3.56) # Devuelve -3
round(3.56) # Devuelve 4
round(-3.56) # Devuelve -4
round(1.56, 1) # El número se redondea a un decimal y devuelve 1.6
```

Nota

En Python **`round`** no hace parte de la clase math, es una función libre que admite un segundo parámetro de manera opcional cuando se desea redondear a un número determinado de decimales.

Para realizar una operación de aproximación de decimales en Java (operación también conocida como **truncamiento**), es necesario jugar un poco con la matemática haciendo algunos cálculos, también se podría tratar con cadenas. El siguiente ejemplo muestra una aproximación cortando (trucando) el número de decimales a un número determinado, pero sin tener en cuenta la regla del valor del decimal que le sigue al decimal hasta donde se desea truncar.

Ejemplo 0.10

Truncar un número real a un número determinado de decimales.

Programa Java

```
public double truncDecimals(double number, int nd)
{
    double diff = number - (int) number;
    double roundDec = (int) (diff * Math.pow(10, nd)) / Math.pow(10, nd);
    return (int) number + roundDec;
}
```

El llamado al método contiene un código como este:

```
Scanner input = new Scanner(System.in);
```

```
double datum;
Maths math = new Maths(); //Objeto de tipo Maths donde están los métodos
System.out.print("Ingrese un número real: ");
datum = Double.parseDouble(input.nextLine());
System.out.println("Truncamiento: " + math.truncDecimals(datum, 2));
```

El método mejorado considerando el decimal a aproximar es el siguiente:

```
public double truncDecimals(double number, int nd)
{
    double diff, roundDec, roundDecInt, divRoundDec;
    int lastDigit, divRound;
    diff = number - (int) number;
    roundDec = (int) (diff * Math.pow(10, nd + 1)) / Math.pow(10, nd + 1);
    roundDecInt = roundDec * Math.pow(10, nd + 1);
    lastDigit = (int) roundDecInt % 10;
    divRound = (int) roundDecInt / 10;
    divRoundDec = divRound / Math.pow(10, nd);
    if (lastDigit >= 5) {
        divRoundDec += Math.pow(10, -nd);
    }
    return (int) number + divRoundDec;
}
```

Si el número es negativo, pueden presentarse algunas inconsistencias en los cálculos, por lo que podemos solucionarlo usando el valor absoluto del número para realizar éstos y al final tener presente el signo del número original. Los resultados encontrados coinciden con los que entrega la función round de Python.

```
public double truncDecimals(double number, int nd)
{
    double diff, roundDec, roundDecInt, divRoundDec, number2;
    int lastDigit, divRound;
    number2 = Math.abs(number);
    diff = number2 - (int) number2;
    roundDec = (int) (diff * Math.pow(10, nd + 1)) / Math.pow(10, nd + 1);
    roundDecInt = roundDec * Math.pow(10, nd + 1);
    lastDigit = (int) roundDecInt % 10;
    divRound = (int) roundDecInt / 10;
    divRoundDec = divRound / Math.pow(10, nd);
    if (lastDigit >= 5) {
        divRoundDec += Math.pow(10, -nd);
    }
    return number >= 0 ?
        (int) number2 + divRoundDec : -((int) number2 + divRoundDec);
}
```

Logaritmos

Se define el **logaritmo** de un número x como el **exponente** al que hay que elevar una **base** b dada para obtener el número x.

$$log_b x = y \Leftrightarrow b^y = x\,(b \neq 1\ y\ positivo;\ x > 0)$$

Observe que aunque `b ≠ 1 y positivo, x > 0`, y sí puede ser menor que cero, y para ello se debe dar que `0 < x < 1`, como se verá en el próximo ejemplo.

Se puede definir cualquier base positiva mayor que 1, sin embargo las bases más comunes son:

- `e = 2.7182818285`: base de los logaritmos **naturales**; **e** se conoce como **Número de Euler** o **constante de Napier (Neper)**, es un número irracional, aquí aproximado a diez cifras significativas. El método disponible en la clase Math para calcular el logaritmo natural es **log(x)**. Por ejemplo en Java: `Math.log(100) // Devuelve: 4.605170185988092`
- `10`: base de los logaritmos "**vulgares**" o **decimales**. El método disponible en la clase math para calcular el logaritmo decimal es **log10(x)**. Por ejemplo en Python: `math.log10(100) # Devuelve: 2.0`
- `2`: base de los logaritmos **binarios** o **en base 2**, muy utilizada en computación y de especial atención en este curso. No se dispone de funciones para calcular bases diferentes a la base e ó 10, pero aplicando un cambio de base puede hallarse un logaritmo en una base distinta fácilmente.

De ahora en adelante, escribiremos estos logaritmos en el contexto de lógica de programación y matemático de la siguiente forma omitiendo la base, en otro caso si la especificaremos.

$$log_e x = ln(x) = lnx$$
$$log_{10} x = log(x) = logx$$
$$log_2 x = lg(x) = lgx$$
$$log_5 x = log_5(x)$$

Ejemplo 0.11

1. $ln(1) = 0,\ porque\ e^0 = 1$
2. $log(1000) = 3,\ porque\ 10^3 = 1000$
3. $lg32 = 5,\ porque\ 2^5 = 32$
4. $log_5 25 = 2,\ porque\ 5^2 = 25$
5. $log0.001 = -3,\ porque\ 10^{-3} = 0.001$

Propiedades de los logaritmos

El logaritmo de un producto es igual a la suma de los logaritmos de los factores
$log_b(xy) = log_b x + log_b y$

El logaritmo de un cociente es igual a la diferencia del logaritmo del numerador menos el logaritmo del denominador
$log_b \frac{x}{y} = log_b x - log_b y$

El logaritmo de una potencia es igual al producto entre el exponente y el logaritmo de la base de la potencia
$log_b x^y = y log_b x$

El logaritmo de una raíz es igual al producto entre la inversa del índice y el logaritmo del radicando. En realidad, es un caso particular del logaritmo de una potencia

$log_b \sqrt[n]{x} = \frac{1}{n} log_b x$

Cambio de base del logaritmo
Sean a, b ≠ 1 y positivos; x > 0. Se cumple la siguiente expresión para el cambio de base de logaritmos en una base b a una base a.

$log_b x = \frac{log_a x}{log_a b}$

Forma práctica para calcular logaritmos
Este método, usado con fines didácticos, también es muy útil a nivel algorítmico; permite encontrar el logaritmo exacto o aproximado al mayor entero menor o igual al logaritmo exacto (redondeo por debajo). El método consiste en realizar divisiones sucesivas del número por la base, hasta encontrar un cociente de 1.

Ejemplo 0.12
Encontrar lg16. Para esto tomamos el número dado (16) y lo dividimos por la base, 2 en este caso, redefiniendo sucesivamente el número dado hasta obtener un cociente de 1.

```
División 1: 16 / 2 = 8
División 2: 8 / 2 = 4
División 3: 4 / 2 = 2
División 4: 2 / 2 = 1
```

El número de operaciones (divisiones) da el logaritmo del número dado. Es claro que lg16 = 4, porque $2^4 = 16$.

Función exponencial

En matemáticas, una **función exponencial** es una función de la forma $f(x) = ab^x$ en el que el argumento x se presenta como un exponente. La función exponencial más conocida

es `f(x) = e^x`, con `e = 2.7182818285` (constante de Neper) de mucho uso en aplicaciones en ciencias. Del cálculo se sabe que este número equivale al límite: $e = \lim_{n \to \infty} (1 + \frac{1}{n})^n$ y que en general: $e^x = \lim_{n \to \infty} (1 + \frac{x}{n})^n$

En Java y en Python está disponible el método **`exp(x)`** (`x ∈ double/float`) para calcular el exponencial de `x`. Por ejemplo en Python: `math.exp(-2) # devuelve 0.1353352832366127`.

Conjuntos en Python

Python implementa funcionalidades para la creación de conjuntos y las operaciones entre ellos.

Un conjunto es un elemento similar a una lista, pero que tiene la misma propiedad de los conjuntos matemáticos, y es que cada elemento que lo compone es único. Los conjuntos admiten información heterogénea.

Sintaxis

1.
```
variableConjunto = {elementos}
```

2.
```
variableConjunto = set(iterable)
```

Donde:
elementos: es la secuencia o conjunto de elementos separados por comas. Si no se indica, el conjunto se crea vacío
iterable: objeto iterable reconocible por Python, tal como listas, tuplas o cadenas. Si no se especifica, el conjunto se inicializará vacío

Con los conjuntos de Python es posible realizar algunas de las operaciones matemáticas entre ellos. Algunas de las siguientes operaciones entre conjuntos se pueden realizar en Python:

Creamos los conjuntos a y b:

```
>>> a = {1, 2, 3}
>>> a
{1, 2, 3}
```

```
>>> b = set([0, 5, 1])
>>> b
{0, 1, 5}
```

Unión, intersección y diferencia de los conjuntos a y b:

```
>>> c = a.union(b)
>>> c
{0, 1, 2, 3, 5}
>>> c = a.intersection(b)
>>> c
{1}
```

```
>>> a
{0, 1, 3}
>>> b
{0, 1, 5}
>>> c = a | b
>>> c
{0, 1, 3, 5}
>>> c = a & b
>>> c
{0, 1}
>>> c = a - b
>>> c
{3}
```

Sean:
`a, b, c`: conjuntos
`dato`: una variable
`posicion`: variable que guarda un índice de la tupla

Se pueden aplicar los siguientes métodos a un conjunto en Python:

- Obtener el total de elementos con la función `len`: `len(a)`.
- Agregar un elemento: `a.add(dato)`.
- Buscar un elemento con el operador `in`: `sw = dato in a # True: si se encuentra`.
- Eliminar un elemento: `a.discard(dato)`.

```
>>> len(a)
3
>>> a.add(2)
>>> a
{1, 2, 3}
>>> a.add(0)
>>> a
{0, 1, 2, 3}
>>> a.discard(2)
>>> a
{0, 1, 3}
```

Límites, continuidad y la derivada

Límites

Veamos algunos de los resultados más importantes del cálculo de límites. Estos enunciados son teoremas que pueden utilizarse en los desarrollos, pero que no demostraremos en este texto.

1. Si m y b son dos constantes cualesquiera, entonces $\lim_{x \to a} (mx + b) = ma + b$

2. Si c es una constante, entonces para cualquier número a, $\lim_{x \to a} c = c$

3. $\lim_{x \to a} x = a$

4. Si $\lim_{x \to a} f(x) = L$ y $\lim_{x \to a} g(x) = M$, entonces $\lim_{x \to a} [f(x) \pm g(x)] = L \pm M$. Este teorema puede extenderse por inducción matemática a *n* funciones.

5. Si $\lim_{x \to a} f(x) = L$ y $\lim_{x \to a} g(x) = M$, entonces $\lim_{x \to a} [f(x) \cdot g(x)] = L \cdot M$. Este teorema puede extenderse por inducción matemática a *n* funciones.

6. Si $\lim_{x \to a} f(x) = L$ y *n* es cualquier entero positivo, entonces $\lim_{x \to a} [f(x)]^n = L^n$. Este teorema puede extenderse por inducción matemática a *n* funciones.

7. Si $\lim_{x \to a} f(x) = L$ y $\lim_{x \to a} g(x) = M$, entonces $\lim_{x \to a} \frac{f(x)}{g(x)} = \frac{L}{M}, si\ M \neq 0$

8. Si $\lim_{x \to a} f(x) = L$ y $\lim_{x \to a} g(x) = M$, entonces $\lim_{x \to a} [f(x) \cdot g(x)] = L \cdot M$

9. Si $\lim_{x \to a} f(x) = L$ y $\lim_{x \to a} g(x) = M$, entonces $\lim_{x \to a} \sqrt[n]{f(x)} = \sqrt[n]{L}$

10. **Límites unilaterales**. El $\lim_{x \to a} f(x)$ existe y es igual a L sí solo si $\lim_{x \to a^-} f(x)$ y $\lim_{x \to a^+} f(x)$ existen y son iguales a L.

11. **Límites infinitos**. Si *r* es cualquier entero positivo, entonces
    a. $\lim_{x \to 0^+} \frac{1}{x^r} = +\infty$
    b. $\lim_{x \to 0^-} \frac{1}{x^r} = \{+\infty, si\ r\ es\ par\ o\ -\infty\ si\ r\ es\ impar\}$

12. **Límites en el infinito**. Si *r* es cualquier entero positivo, entonces
    a. $\lim_{x \to +\infty} \frac{1}{x^r} = 0$
    b. $\lim_{x \to -\infty} \frac{1}{x^r} = 0$

Ejemplo 0.33
Determinar los siguientes límites cuando sea posible

a. $\lim_{x \to 2} (3x^2 - 4x + 5) = 3(2^2) - 4(2) + 5 = 12 - 8 + 5 = 9$
b. $\lim_{z \to -3} \frac{z^2-9}{z+3} = \frac{(-3)^2-9}{-3+3} = \frac{0}{0}$ Antes de concluir que el límite no existe, intentemos una factorización en el numerador:
$\lim_{z \to -3} \frac{z^2-9}{z+3} = \lim_{z \to -3} \frac{(z+3)(z-3)}{z+3} = \lim_{z \to -3} (z - 3) = -3 - 3 = -6$

Ejemplo 0.34
Determinar el siguiente límite sí es posible
$\lim_{x \to +\infty} \frac{3x^2+2x-5}{x^2+4}$ Dividimos cada término del numerador y denominador por x^2 y aplicamos el límite a cada uno de éstos usando los teoremas correspondientes; así obtenemos: $\lim_{x \to +\infty} \frac{3x^2+2x-5}{x^2+4} = 3$

Continuidad de una función en un número

Definición (continuidad)
Se dice que la función f es **continua** en el número a sí y sólo si se cumplen las 3 condiciones siguientes:

1. $f(a)$ existe
2. $\lim_{x \to a} f(x)$ existe
3. $\lim_{x \to a} f(x) = f(a)$

Si una de estas condiciones no se cumple para *a*, decimos que la función es **discontinua** en dicho número.

Ejemplo 0.35
Determine los números en los cuales es continua la función $g(x) = x / (x - 3)$.
$g(x)$ no existe en $x = 3$, por tanto $g(x)$ es continua en todos los reales, excepto el número 3: $\mathbb{R} - \{3\}$.

La derivada

La derivada de una función f es aquella función, denotada por f', cyo valor en un número cualquiera x del dominio de f está dada por $f'(x) = \lim_{\Delta x \to 0} \frac{f(x+\Delta x)-f(x)}{\Delta x}$, si este límite existe.

Teoremas sobre derivadas

Veamos algunos de los resultados más importantes del cálculo de derivadas. Estos enunciados son teoremas que pueden utilizarse en los desarrollos, pero que no demostraremos en este texto.

1. **Derivada de una constante**. Si c es constante $f(x) = c$ para toda x, entonces $f'(x) = 0$

2. **Derivada de una potencia**.
   a. Si n es un entero positivo y $f(x) = x^n$, entonces $f'(x) = nx^{n-1}$
   b. Si $-n$ es un entero negativo y $x \neq 0$, entonces $f'(x) = -nx^{-n-1}$

3. **Derivada de una constante por una función**. Si f es una función, c una constante y $g(x) = cf(x)$, entonces $g'(x) = cf'(x)$, si $f'(x)$ existe

4. **Derivada de una suma/resta de funciones**. Si f y g son funciones y $h(x) = f(x) \cdot g(x)$, entonces $h'(x) = f'(x) \pm g'(x)$, si $f'(x)$ y $g(x)$ existen

5. **Derivada de un producto de funciones**. Si f y g son funciones y $h(x) = f(x) \cdot g(x)$, entonces $h'(x) = f(x) \cdot g'(x) + g(x) \cdot f'(x)$, si $f'(x)$ y $g(x)$ existen

6. **Derivada de un cociente de funciones**. Si f y g son funciones y $h(x) = f(x)/g(x)$, donde $g(x) \neq 0$ entonces $h'(x) = \frac{g(x)f'(x)-f(x)g'(x)}{[g(x)]^2}$, si $f'(x)$ y $g(x)$ existen

Ejemplo 0.36

Obtener la derivada de la función dada

a. $f(x) = 5x^3 - 7x^2 + 2x - 3;\ f'(x) = 15x^2 - 14x + 2$
b. $f(x) = \frac{x^2}{4} + \frac{4}{x^2} = \frac{x^2}{4} + 4x^{-2};\ f'(x) = \frac{2x}{4} + (-2)4x^{-2-1} = \frac{x}{2} - 8x^{-3} = \frac{x}{2} - \frac{8}{x^3}$

Ejemplo 0.37

Obtener la derivada de la función $f(t) = (3t^2 - 4)(4t^3 + t - 1)$.

$g(t) = 3t^2 - 4;\ h(t) = 4t^3 + t - 1;\ f'(t) = g(t)h'(t) + h(t)g'(t)$
$g'(t) = 6t;\ h'(t) = 12t^2 + 1$
$f'(t) = (3t^2 - 4)(12t^2 + 1) + 6t(4t^3 + t - 1)$
$f'(t) = 36t^4 + 3t^2 - 48t^2 - 4 + 24t^4 + 6t^2 - 6t$
$f'(t) = 60t^4 - 39t^2 - 6t - 4$

Sucesiones y series

Sucesión

Una **sucesión** está conformada por elementos que se encuentran ordenados y que responden a una ley de formación. Por ejemplo, la siguiente es una sucesión: $a_n = a_1, a_2, a_3, \ldots, a_n, \ldots$, donde a_1 es el primer término, a_2 el segundo, a_3 el tercero, y así sucesivamente.

$a_n = 1, 2, 3, \ldots, n, \ldots \forall n \in N; a_n = n$

$b_n = 1, 1/2, 1/3, \ldots, 1/n, \ldots \forall n \in N; a_n = 1/n$

$c_n = -1, -1/4, -1/9, \ldots, -1/n^2, \ldots, \forall n \in N; a_n = -1/n^2$

$d_n = 5, 10, 15, \ldots, n, \ldots \forall n \in N; a_n = 5n$

$e_n = 1, 0, 1, \ldots, n, \ldots \forall n \in N; a_n = 1$ *si n es impar o 0 si n es par*

Series

Una **serie** es la suma de algunos términos de una sucesión.

Supongamos que $u(n)$ es una función en n definida para todos los valores de n. Al sumar los valores $u(i)$ $(1 <= i <= n)$, se obtiene otra función de n:

$S(n) = u(1) + u(2) + \ldots + u(n) = u_1 + u_2 + \ldots + u_n$.

$$S(n) = S_n = \sum_{i=1}^{n} u_i$$

Se lee: "*la suma de u sub i desde i igual a 1 hasta i igual a n*"

Cuando esta suma tiende a un límite cuando n tiende a infinito, escribimos:

$$S_n = \lim_{n \to \infty} \sum_{i=1}^{n} u_i = \sum_{i=1}^{\infty} u_i = u_1 + u_2 + \ldots$$

La serie es **divergente** si la suma tiende a +∞ ó a -∞; si la suma tiende a un valor límite, entonces la serie es **convergente**; si la suma no tiende a un valor límite, ni a +∞ ó a -∞, entonces diremos que la serie **oscila** (de forma finita o infinita).

Sucesiones (progresiones) y series aritméticas

La diferencia entre términos sucesivos es constante, por tanto se pueden representar los *n* primeros términos de la serie en la forma:

a, a + d, a + 2d, a + 3d, …, a + (n - 1)d

En donde *a* es el primer término de la sucesión y *d* es la diferencia entre términos sucesivos, ambas constantes adecuadas.

Para encontrar la suma de los primeros *n* términos de esta sucesión, indicada por la sumatoria de a_i, *1 <= i <= n*:

$$S_n = \sum_{i=0}^{n-1} a_i = \frac{n(a+a_{n-1})}{2}, \; donde \; a_0 = a$$

Si *d* es la diferencia, podemos determinar el término a_i de la sucesión usando la expresión: $a_i = a_0 + di$ aplicada adecuadamente.

Ejemplo 0.38
Sea la sucesión *2, 4, 6, 8, 10, 12, 14, ...* Determinar la suma de los primeros *7* términos.

$a = a_0 = 2$
$d = 2$
$n = 7$
$a_6 = 2 + 2 * 6 = 14$
$S_n = S_6 = 7 * (2 + 14) / 2 = 56$
En efecto: 2 + 4 + 6 + 8 + 10 + 12 + 14 = 56

Sucesión geométrica

La razón entre términos sucesivos es constante, por tanto se pueden representar los *n* primeros términos de la serie en la forma:

$a, ar, ar^2, ar^3, \ldots, ar^{n-1}$

En donde *a* es el primer término de la sucesión y *r* es la razón entre términos sucesivos, ambas constantes adecuadas.

Proposición (serie geométrica infinita)
La serie geométrica infinita $a, ar, ar^2, ar^3, \ldots,$ converge si y sólo si $-1 <= r <= 1$, y su suma es $S_n = a / (1 - r)$.

Ejemplo 0.39
Sea la sucesión *1, 2, 4, 8, 16, 32, ...*; esta es una sucesión geométrica con $r = 2$ (observe que la razón entre cada dos términos es constante y equivale a *2*).

Serie armónica

Es una serie basada en sucesiones de la forma a^{-r}, donde r es un entero positivo; si $a = r = 1$, obtenemos la serie: 1 + ½ + ⅓ + ¼ + … Dicha serie es divergente.

En los ejercicios al final del capítulo se propone resolver esta serie algorítmicamente para un *n* dado, así como otras, como las series de potencias.

Suma de Riemann

La integral definida de una función continua f en el intervalo $[a, b]$, denotada por $\int_a^b f(x)dx$ es igual al límite de una **suma de Riemann** conforme el número de subdivisiones (rectángulos) tiende al infinito:

$$\int_a^b f(x)dx = \lim_{n\to\infty} \sum_{i=1}^{n} \Delta x \cdot f(x_i),\ donde\ \Delta x = \frac{b-a}{n}\ y\ x_i = a + i\Delta x$$

Esta suma nos permite fácilmente crear un algoritmo para resolver integrales definidas. Veamos en Python la implementación de la suma de Riemann.

Ejemplo 0.40

Implementación de la suma de Riemann en Python para solucionar la integral definida $\int_2^6 \frac{1}{5}x^2 dx$

Solución

Primero solucionemos la integral con los métodos conocidos del cálculo:

$$\int_2^6 \frac{1}{5}x^2 dx = \frac{1}{5\times 3}x^3 = \frac{x^3}{15}\Big|_2^6 = \frac{6^3-2^3}{15} = \frac{216-8}{15} = \frac{208}{15} = 13.86667$$

Ahora creemos la función que realiza la suma de Riemann, la función que calcula $f(x)$ y hacemos el respectivo llamado a ésta:

```
# Suma de Riemann
def riemann_sum(n, a, b) -> float:
    dx = (b - a) / n
    sr = 0
    for i in range(1, n + 1 , 1):
        sr += dx * f(a + dx * i)
    return sr

# Función f(x) = (1/5)x**2
```

```
def f(x) -> float:
    return (1 / 5) * x ** 2

print(f"Integral definida por suma de Riemman: {riemann_sum(1000, 2, 6)}")
```

Con un valor de $n = 100$, se obtiene un suma de 13.994880000000002.
Con un valor de $n = 1000$, se obtiene un suma de 13.879468800000009.
Con un valor de $n = 10000$, se obtiene un suma de 13.867946688000005.

Observe cómo a medida que $n \to \infty$ el valor obtenido en la suma es más preciso y más cercano al valor dado por la resolución directa de la integral definida.

Preguntas

1. ¿Un lenguaje de programación es una máquina?
2. ¿Qué tipos de lenguajes de programación existen?
3. ¿Cuál es la diferencia entre un lenguaje compilado y uno interpretado?
4. ¿Cuál es la mínima unidad de almacenamiento computacional?
5. ¿Qué es un campo, una variable y una constante? ¿En qué se diferencian?
6. ¿Qué es un tipo de dato primitivo y cuáles son?
7. ¿Por qué es importante tener en cuenta el tamaño de los tipos de datos al diseñar un programa?
8. ¿Qué es un número pseudoaleatorio? ¿Para qué sirven?
9. ¿Qué es un redondeo? ¿Qué tipos de redondeo existen?
10. ¿Por qué son necesarias las matemáticas para el análisis de algoritmos?
11. ¿Qué tipos de notaciones existen en programación y cómo se usan?
12. ¿Qué es una semilla pseudoaleatoria?
13. ¿Qué es un algoritmo?
14. ¿Qué es un programa?
15. ¿Cuál es la diferencia entre código y pseudocódigo?

Ejercicios

Ejercicios de programación

1. A un grupo de hombres y mujeres les realizan una prueba de conocimientos. Se desea conocer cuántas mujeres y cuántos hombres presentaron la prueba y el total de personas que se procesaron.
2. En una fábrica contratan personal que cumpla con los siguientes requisitos para laborar medio tiempo: Mayor de 18 años y menor de 50, estado civil soltero y que actualmente se encuentre estudiando. A la fábrica se presentaron M aspirantes.

Encuentre cuántos cumplen con los requisitos que se piden y qué porcentaje sobre el total de personas representan.

3. Leer el nombre y la nota de un grupo de T estudiantes. Encuentre quien sacó la mejor nota y de cuanto fue ésta.
4. Leer el nombre y salario de un grupo de trabajadores. Muestre cuál es el empleado de más bajo salario y a cuánto equivale éste.
5. Una empresa tiene 5 sucursales y en cada sucursal hay M empleados. De cada empleado se conoce su cédula y salario. Encuentre: El salario promedio de cada sucursal y de toda la empresa. El empleado que gana mayor salario en cada sucursal y en toda la empresa. El empleado que gana menor salario en cada sucursal y en toda la empresa. El monto pagado por concepto de salarios en cada sucursal y en toda la empresa.
6. Se tiene el nombre, número de horas diurnas (nhd), número de horas nocturnas (nhn), número de horas festivas (nhf) y salario básico hora de un grupo de empleados. Calcular el salario básico teniendo en cuenta que la hora nocturna tiene un recargo del 35% y la hora festiva un recargo del 75%. Terminar la lectura de datos cuando se ingrese el nombre "***". Informar cuántos empleados se ingresaron y el promedio de salarios básicos que devengan éstos.
7. Cree un juego de azar similar al tragamonedas que muestra tres o cuatro imágenes usando caracteres en lugar de imágenes. Determine los premios a entregar según acierte el usuario y dependiendo de la imagen (carácter).
8. Realizar el cálculo de una devuelta. Debe contar con denominaciones de monedas y billetes para especificar cuántos de cada uno debe entregar. Por ejemplo, si la devuelta son $450, podría devolver 2 monedas de $200 y una de $50.
9. Convierta un número a letras. Por ejemplo: *1584 → mil quinientos ochenta y cuatro*.
10. Leer una cantidad indeterminada de números e informar si se ingresó ordenadamente.
11. Ingrese un número entero por teclado. Cree una función que determine si dicho número pertenece a la serie de **Fibonacci**.
12. Determine cuántos **números primos** hay entre *1* y *n*.
13. Encuentre los **números perfectos** entre *1* y *n*. Para *n = 10000*, mida los tiempos de ejecución que entregan Java y Python para ejecutar este programa, respectivamente.
14. Cree un programa que solicite cuatro datos numéricos para **construir un tipo de dato para fechas**: día de la semana entre 1 y 7 (el domingo es el día 1, el lunes el 2, etc.), día del mes entre 1 y 31, número del mes entre 1 y 12 (el 1 es el mes enero, el 2 febrero, etc.) y el año. Cree una función que devuelva una fecha en un formato similar a este: *Jueves, 26 de Octubre de 2023*, si los datos son: 5, 26, 10, 2023. Añada otras funciones para devolver y cambiar el día, la hora, el año, etc.
15. Crear un programa para **conversión de coordenadas**. Debe tener un menú para especificar el tipo de conversión y los datos a ingresar, así como un procedimiento que convierta coordenadas polares (r, θ) a coordenadas cartesianas: x = rcos(θ), y = rsen(θ), y otro procedimiento que convierta de coordenadas cartesianas a polares: $r = \sqrt{x^2 + y^2}$; $\theta = tan^{-1}\frac{y}{x}$
16. Realizar **operaciones lógicas y simular las compuertas** usando Java y Python

17. Utilice arreglos para realizar **operaciones entre conjuntos** en Java y en Python: defina tres conjuntos U, A, B y realice las operaciones de unión, intersección, diferencia, diferencia simétrica, complemento, producto cartesiano y conjunto potencia.
18. Dados dos números enteros positivos; cree una función para **multiplicar** éstos como una **suma sucesiva**. Ejemplo: `2 * 3 = 2 + 2 + 2 = 3 + 3`.
19. Dados dos números enteros positivos, cree una función para elevar el primero al segundo, es decir, para calcular la **potencia** como una **multiplicación sucesiva**. Ejemplo: `2 ^ 3 = 2 * 2 * 2`.
20. Dados dos enteros, calcule la **división entera** y el **módulo** entre éstos sin usar los operadores aritméticos respectivos, ni operaciones de redondeo; la solución debe ser de forma algorítmica usando **restas sucesivas**.
21. Encuentre el **logaritmo** o logaritmo aproximado de un número entero usando **divisiones sucesivas**.
22. Determine si un número entero es **par o impar** de forma algorítmica sin utilizar el operador de módulo. Sugerencia: tenga en cuenta que: $Un\ par\ se\ define\ como\ 2k\ y\ un\ impar\ como\ 2k\ +\ 1,\ k\ \in\ Z$
23. Mostrar el equivalente de un número **decimal** en binario, octal y hexadecimal
24. Ingrese un número en **binario, octal o hexadecimal** y conviértalo a decimal
25. Calcule la suma de los primeros m términos de la **serie Armónica**[11] `1 + 1/2 + 1/3 + 1/4 + ... + 1/m`
26. Calcule el valor de **π** (**pi**) usando la **serie de Leibniz**[12] para un n grande: π/4 = `1 - 1/3 + 1/5 - 1/7 + 1/9 - ... + 1/n`
27. Dado un valor real x, cree una función para calcular su **exponencial** usando la serie de Taylor para 100 o más términos: $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$
28. Dado un valor real x, cree una función para calcular su **logaritmo** usando la serie de Maclaurin para 100 o más términos: $ln(1 - x) = -x - \frac{x^2}{2} - \frac{x^3}{3} - \ldots$
29. Dado un valor real x, cree una función para calcular su **logaritmo** usando la serie de Maclaurin para 100 o más términos: $ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \ldots$
30. Dado un valor real entre `0` y 2π, cree dos funciones para calcular el **seno y coseno** trigonométricos usando la serie de Taylor para 100 o más términos: $sen(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$; $cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$
31. Usando un lenguaje de programación, cree un programa para calcular el **logaritmo en base 2 de x** (`lg(x)`). Generalice el problema para que se pueda calcular el logaritmo en cualquier base dada
32. Sean $f(x)$ y $g(x)$ dos funciones polinómicas cualesquiera. Cree un programa para sumar y restar dos **polinomios** utilizando listas ligadas. Calcule $f(a)$ y derive a $f(x)$ y a $g(x)$.

---

[11] Se llama así porque la longitud de onda de los sucesivos armónicos de una cuerda que vibra es proporcional a la longitud de onda del modo de oscilación fundamental a través de los factores de proporcionalidad dados por los correspondientes términos de la serie: 1, 1/2, 1/3, 1/4, 1/5, 1/6, 1/7... El primer término representa por tanto al modo fundamental. Serie armónica (matemática) - Wikipedia, la enciclopedia libre

[12] Nombrada así en honor a Gottfried Wilhelm Leibniz, filósofo y matemático alemán (1646 - 1716). Ver más en Serie de Leibniz - Wikipedia, la enciclopedia libre

33. La **función lineal** está expresada por y = mx + b, con m, b números reales. Dados m, b, cree un programa usando funciones para generar una tabla de valores para distintos valores de x especificados por el usuario y/o generados por el sistema, y graficar esta función por pantalla.
34. Calcule las raíces de la **ecuación cuadrática** $ax^2 + bx + c = 0$, cuya fórmula general de solución es: $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$
35. Cree un programa que permita ingresar una **función** matemática arbitraria $f(x)$ y calcular su valor en $x = a$, esto es, hallar $f(a)$.
36. Calcule la longitud del segmento $\overline{AP}$ de la figura 0.1. Dicha longitud es conocida como **Número áureo**[13] con respecto al segmento $\overline{AB}$ y que matemáticamente equivale a $\varphi = \frac{1+\sqrt{5}}{2}$. Verifique el valor hallado del segmento con el valor dado por esta expresión y que realmente coinciden ¿qué porcentaje de error hay en los cálculos?

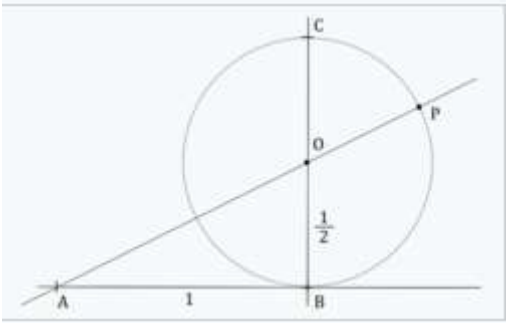


Figura 0.1. Número áureo[14]

Ejercicios de matemáticas[15]

44. Escribir la expresión exponencial dada como una expresión logarítmica equivalente.
    a. $4^{\frac{-1}{2}} = \frac{1}{2}$
    b. $9^0 = 1$
    c. $10^4 = 10000$
    d. $10^{0.3010} = 2$
45. Escribir la expresión logarítmica dada como una expresión exponencial equivalente.
    a. $\log_2 128 = 7$
    b. $\log_5 \left(\frac{1}{25}\right) = -2$
    c. $\log_{\sqrt{3}} 81 = 8$
    d. $\log_{16} 2 = \frac{1}{4}$

[13] Ver más acerca de este número en: Número áureo - Wikipedia, la enciclopedia libre
[14] Imagen tomada de: Número áureo - Wikipedia, la enciclopedia libre
[15] Ejercicios tomados de: EJERCICIOS DE FUNCIONES

48. Use las leyes de los logaritmos para volver escribir la expresión dada con un solo logaritmo.
    a. $\ln(x^4 - 4) - \ln(x^2 + 2)$
    b. $\ln\left(\frac{x}{y}\right) - 2\ln(x^3) - 4\ln(y)$
    c. $\ln 5 + \ln 5^2 + \ln 5^3 - \ln 5^6$
    d. $5\ln 2 + 2\ln 3 - 3\ln 4$

50. Use el logaritmo natural para despejar *x*
    a. $2^{x+5} = 9$
    b. $4 * 7^{2x} = 9$
    c. $5^x = 2e^{x+1}$
    d. $3^{2(x-1)} = 2^{x-3}$

51. Despeje *x*
    a. $\ln x + \ln(x - 2) = \ln 3$
    b. $\ln 3 + \ln(2x - 1) = \ln 4 + \ln(x + 1)$

Ejercicios sobre límites[16]

En los problemas del 1 al 6 determine el límite que se indica.

1. $\lim_{x\to 3}(x - 5)$
2. $\lim_{t\to -1}(1 - 2t)$
3. $\lim_{x\to -2}(x^2 + 2x - 1)$
4. $\lim_{x\to -2}(x^2 + 2t - 1)$
5. $\lim_{t\to -1}(t^2 - 1)$
6. $\lim_{t\to -1}(t^2 - x^2)$

En los problemas del 7 al 18 determine el límite que se indica. En la mayoría de los casos, es buena idea usar primero un poco de álgebra (véase el ejemplo 2).

7. $\lim_{x\to 2}\frac{x^2 - 4}{x - 2}$
8. $\lim_{t\to -7}\frac{t^2 + 4t - 21}{t + 7}$
9. $\lim_{x\to -1}\frac{x^3 - 4x^2 + x + 6}{x + 1}$
10. $\lim_{x\to 0}\frac{x^4 + 2x^3 - x^2}{x^2}$
11. $\lim_{x\to -t}\frac{x^2 - t^2}{x + t}$
12. $\lim_{x\to 3}\frac{x^2 - 9}{x - 3}$
13. $\lim_{t\to 2}\frac{\sqrt{(t + 4)(t - 2)^4}}{(3t - 6)^2}$
14. $\lim_{t\to 7^+}\frac{\sqrt{(t - 7)^3}}{t - 7}$
15. $\lim_{x\to 3}\frac{x^4 - 18x^2 + 81}{(x - 3)^2}$
16. $\lim_{u\to 1}\frac{(3u + 4)(2u - 2)^3}{(u - 1)^2}$
17. $\lim_{h\to 0}\frac{(2 + h)^2 - 4}{h}$
18. $\lim_{h\to 0}\frac{(x + h)^2 - x^2}{h}$

GC *En los problemas del 19 al 28 utilice una calculadora para encontrar el límite indicado. Utilice una calculadora gráfica para trazar la función cerca del punto límite.*

19. $\lim_{x\to 0}\frac{\text{sen } x}{2x}$
20. $\lim_{t\to 0}\frac{1 - \cos t}{2t}$
21. $\lim_{x\to 0}\frac{(x - \text{sen } x)^2}{x^2}$
22. $\lim_{x\to 0}\frac{(1 - \cos x)^2}{x^2}$
23. $\lim_{t\to 1}\frac{t^2 - 1}{\text{sen}(t - 1)}$
24. $\lim_{x\to 3}\frac{x - \text{sen}(x - 3) - 3}{x - 3}$
25. $\lim_{x\to \pi}\frac{1 + \text{sen}(x - 3\pi/2)}{x - \pi}$
26. $\lim_{t\to 0}\frac{1 - \cot t}{1/t}$
27. $\lim_{x\to \pi/4}\frac{(x - \pi/4)^2}{(\tan x - 1)^2}$
28. $\lim_{u\to \pi/2}\frac{2 - 2\,\text{sen } u}{3u}$

Ejercicios sobre aritmética y álgebra

1. Demuestre o compruebe que el cero es un número par
2. Demuestre las propiedades del valor absoluto
3. Probar que la suma de los cuadrados de los `n` primeros números naturales es `n * (n + 1) * (2 * n + 1) / 6`
4. Probar que $1^2 + 3^2 + 5^2 + \ldots + (2n - 1)^2 = n(2n - 1)(2n + 1)/3$
5. Probar que la suma de los cubos de los `n` primeros números naturales es $[n(n + 1)/2]^2$.
6. Probar que $1 + x + x^2 + \ldots + x^n = (x^{n+1} - 1)/(x - 1)$

[16] Ejercicios tomados de: Límites

Capítulo 1. Algoritmia elemental. Notación asintótica

Problemas y ejemplares

Por ejemplo, si tenemos dos enteros positivos como 56 y 84, y queremos multiplicarlos, podemos usar alguno de los métodos o algoritmos disponibles para hacerlo; sin embargo, el método usado debería funcionar para cualquier par de enteros positivos y no solo para los casos en particular. Decimos entonces que multiplicar enteros positivos (sin importar qué método se use) es el **problema**, mientras que `(56, 84)` es un **ejemplar** de este problema[17]. Otro ejemplar para este problema es `(632, 1584)`, sin embargo `(-3, 7.65)`, no lo es, ya que `-3` no es un entero positivo y `7.65` no es un número entero; es claro que `(-3, 7.65)` es un ejemplar de otro problema, correspondiente a una multiplicación más general.

Un algoritmo debe funcionar correctamente con todos los ejemplares del problema y que corresponden al ámbito de éste. De forma similar a las demostraciones en matemáticas, donde basta con encontrar un contraejemplo para invalidar un teorema, determinar si un algoritmo es incorrecto consiste en encontrar si para un ejemplar, el algoritmo no es capaz de entregar una respuesta correcta. Por tanto, basta con un resultado incorrecto que arroje el algoritmo para descartar éste.

Cuando se especifica un problema, es muy importante indicar el **dominio de definición**, esto es, el conjunto de casos que deben considerarse. Por ejemplo, el algoritmo que multiplica dos números enteros positivos no tiene porqué funcionar para números negativos y fraccionarios, ya que éstos no se encuentran en el dominio de definición del problema.

Las máquinas de cálculo tienen un límite que afectan los resultados de las operaciones. Por ejemplo, puede suceder que no haya espacio disponible para realizar el cálculo o que el resultado supere la capacidad definida para el tipo de dato dado. Los algoritmos que se desarrollarán serán correctos en abstracto, sin preocuparnos por estas limitantes, pero al realizar laboratorios con los lenguajes Java y Python debemos tener los cuidados pertinentes según las reglas que éstos imponen, incluso habría que tener otros aspectos en cuenta, como el sistema operativo, el hardware disponible, etc., pero solo nos concentraremos por ahora en el tamaño de los tipos de datos y la memoria disponible.

Operaciones elementales

Son instrucciones simples que incluyen declaraciones, asignaciones, lecturas, impresiones, operaciones aritméticas y lógicas, así como comparaciones, entre otras. El número de operaciones elementales que se requieren para obtener la solución de un caso ayuda a determinar la complejidad del algoritmo.

[17] Los ejemplares también pueden ser llamados **instancias** o **casos** del problema.

Es de anotar que en ocasiones estas “operaciones elementales” no son totalmente sencillas, por ejemplo la suma y la multiplicación se hacen lentas a medida que aumenta el tamaño de los ejemplares (operandos). Otro problema a considerar en la práctica, es que al realizar operaciones aritméticas con operandos muy grandes, se excederá la capacidad de la máquina para el resultado, por lo cual estas operaciones pueden resultar ser “no elementales” y requerir un tratamiento especial para resolver el caso.

Es claro que las operaciones elementales incluirán a los ejemplares del problema.

Eficiencia de los algoritmos

Para un mismo problema, es posible que existan varias soluciones a éste, es decir, se dispone de varios algoritmos. Surge la pregunta *¿cuál es el mejor*? Al desarrollar programas, muchas veces se escoge la solución más fácil de programar, pero ¿realmente es la mejor?

El enfoque **empírico** (o **a posteriori**) para seleccionar un algoritmo consiste en programar las técnicas e ir probándolas en distintos casos con ayuda de un computador. El enfoque **teórico** (o **a priori** -*independiente de la experiencia*-) consiste en determinar matemáticamente la cantidad de recursos necesarios para cada uno de los algoritmos *como función del tamaño de los casos considerados*. Los recursos más interesantes son el tiempo de computación y el espacio de almacenamiento, siendo el primero el más relevante en general. Por tanto, cuando hablemos de la **eficiencia** de un algoritmo, nos estaremos refiriendo a qué tan rápido se ejecuta. En ocasiones se tendrá en cuenta almacenamiento y otros recursos como el procesador.

El *tamaño* de un ejemplar se corresponde formalmente con el número de *bits* que se necesitan para representar el ejemplar en un computador.

Evaluación de algoritmos

Es el análisis que consiste en medir la eficiencia de los algoritmos en cuanto a consumo de memoria y tiempo de ejecución.

Anteriormente era crucial medir el consumo de memoria debido a las restricciones que existían en el Hardware. Por ejemplo, el computador con microprocesador 8086 de finales de la década de 1970 tenía capacidad para acceder a una memoria RAM de `1MB = 1048576B`. Un vector de enteros de dimensión `10000` para el tipo de dato entero típico de un tamaño de `2B`, ocupa en memoria `20000B = 0.0191MB`. Esto equivalía al 1.91% de la memoria para esta máquina cuyos recursos eran limitados, un consumo relativamente alto para un caso de un vector pequeño, y eso sin considerar las demás variables que pudiese tener el programa, junto con los demás programas que se estuvieran ejecutando en la máquina, razones que hacían que tener presente el diseño de los programas fuera un aspecto muy importante.

Actualmente y gracias a los avances significativos del hardware, esto ya es no considerado un problema en la mayoría de los casos, por lo que ha pasado a un segundo plano, y ahora la preocupación del análisis de algoritmos está enfocada en mejorar los tiempos de ejecución, en particular cuando se trata de entradas grandes.

Los sistemas operativos y los lenguajes suministran funciones para medir los tiempos de ejecución de los programas, pero esto en ocasiones no nos sirve de mucho, ya que cuando ejecutamos el mismo algoritmos en distintas máquinas, en realidad estamos midiendo otros factores como el lenguaje de programación, el sistema operativo y el hardware entre otros aspectos.

Este tipo de evaluación del algoritmo es la que llamamos **a posteriori** o **empírica**. Pero nos interesa poder analizar la eficiencia de un algoritmo independiente de estos aspectos, esto es, **a priori**, para lo cual requerimos de herramientas matemáticas que permitan argumentar cuál algoritmo es mejor que otro para un problema dado.

Medir la cantidad de espacio que utiliza un algoritmo

Es posible medir la cantidad de espacio que utiliza un algoritmo en función del tamaño de los ejemplares. La unidad natural en informática utilizada para calcular el espacio utilizado, independiente de la máquina, es el **bit**, por lo que es fácil establecer cuánta memoria requiere un algoritmo para solucionar un caso.

En la actualidad el consumo de espacio no es un problema que preocupe en el análisis de algoritmos, tal y como sucedía hace varios años donde era preciso tener cuidado con el abuso del uso de variables y estructuras de datos debido a que los recursos de máquina eran muy limitados.

Ejemplo 1.1

Determinar el espacio ocupado por el siguiente programa y encontrar el número de operaciones elementales.

Programa Java:

```
public int sumNaturals(int n)
{
    int i, s;
    i = 1;
    s = 0;
    while (i <= n) {
        s += i;
        i++;
    }
    return s;
}
```

El programa cuenta con tres variables de tipo entero (int), el cual ocupa en Java y los lenguajes comunes 2B = 16b. Por tanto el programa ocupa en memoria al menos 3 * 2B = 6B = 48b, lo que realmente es un consumo de espacio muy bajo.

El programa cuenta con diez operaciones elementales, entre las que se cuentan declaraciones, asignaciones y comparaciones.

Medir la cantidad de tiempo que utiliza un algoritmo

Medir el tiempo que utiliza un algoritmo no tiene opciones tan obvias como en el caso de medir el espacio que ocupa éste. Pensar en una unidad de tiempo específica, como el segundo por ejemplo, no proporciona necesariamente una medida confiable, porque dependería del hardware donde se ejecute el programa, o el lenguaje de programación u otros factores, en otras palabras, no contamos con un computador estándar que permita establecer estas mediciones usando las unidades convencionales.

Una respuesta a este problema viene dado en el **principio de invariancia** que se enuncia a continuación, junto con otros conceptos que también se describen más adelante.

Principio de invariancia

Este principio afirma que dos implementaciones distintas de un mismo algoritmo no diferirán en su eficiencia en de más de alguna constante multiplicativa, esto es, si dos implementaciones requieren $t_1(n)$ y $t_2(n)$ segundos, respectivamente, para un resolver un caso de tamaño n, entonces existen dos constantes positivas, conocidas como **constantes ocultas**, tales que $t_1(n) \leq at_2(n)$, $t_2(n) \leq bt_1(n)$, siempre que n sea suficientemente grande. Es decir, el tiempo de ejecución de una implementación está acotado por el tiempo de ejecución de otra implementación multiplicado por una constante positiva. Así por ejemplo, si una de las constantes fuese 5, entonces sabríamos que una implementación en una máquina (o lenguaje de programación) tardaría un segundo, mientras que en otra máquina tardaría a lo sumo 5 segundos más.

Este principio no es demostrable, es un hecho confirmado por la observación y se cumple sea cual sea el computador (siempre y cuando sea de diseño convencional -clásico-), el lenguaje de programación y el compilador utilizado, o el sistema operativo donde se implemente la solución, entre otros aspectos. De esta forma, un cambio de lenguaje o de máquina puede significar aumentar la velocidad de ejecución de un algoritmo en el doble, 10 veces, 20 veces o más veces, siempre en un factor constante.

Contador de Frecuencias (CF)

Es una expresión algebraica que indica el número de veces que se ejecutan las instrucciones de un algoritmo (programa). Para determinar el CF es importante identificar las operaciones elementales. Es una herramienta muy útil que permite determinar el tiempo de ejecución de un algoritmo en “el orden de” en las notaciones asintóticas.

Ejemplo 1.2
Establecer el contador de frecuencias de cada uno de los siguientes algoritmos.

Al lado de cada operación elemental (instrucción) indicaremos el número de veces que se ejecuta ésta; al final sumamos estas cantidades para obtener el CF. La sentencia de inicio puede omitirse en el conteo, a no ser que sea una función, la cual generalmente involucra parámetros.

Pseudo programa Java a)

```
public static void main(String args[])
{
    int s, a, b; _____________________________ 1
    a = 10;      _____________________________ 1
    b = 15;      _____________________________ 1
    s = (a + b) / 2; _________________________ 1
    print(s);        _________________________ 1
}   ___________________________________________ 1

                                              _____
Contador de frecuencias CF =                    6
```

Pseudo programa Java b)

```
public static void main(String args[])
{
    int i, s, n;_____________________________ 1
    i = 1;    _____________________________ 1
    s = 0;    _____________________________ 1
    read(n);  _____________________________ 1
    while (i <= n) { _____________________ n + 1
        s += i;      _____________________ n
        i++;         _____________________ n
    }   _____________________________________ n
    print(s); _____________________________ 1
}   ___________________________________________ 1

                                            ________
Contador de frecuencias CF =              4n + 7
```

Pseudo programa Java c)

```
public static void main(String args[])
{
    int i, m, n;_____________________________ 1
    i = 1;    _____________________________ 1
    read(m);  _____________________________ 1
    while (i <= m) { _____________________ m + 1
        print(i * i); ____________________ m
        print(1 / i); ____________________ m
        i++;         _____________________ m
```

```
    }   _____________________________ m
    i = 1;    ____________________________ 1
    read(n);  ____________________________ 1
    while (i <= n) { ____________________ n + 1
        print(2 * i); ___________________ n
        i++;         ____________________ n
    }   _____________________________ n
}   _____________________________ 1

                                  ___________
Contador de frecuencias CF =       5m + 4n + 8
```

Pseudo programa Java d)

```
public static void main(String args[])
{
    int i, j, n, s; _____________________ 1
    i = 1;    ____________________________ 1
    read(n);  ____________________________ 1
    while (i <= n) {  ___________________ n + 1
        s = 0;        ___________________ n
        j = 1;        ___________________ n
        while (j <= n) { _______________ n * n + n
            s = s + i + j; _____________ n * n
            j++;     ___________________ n * n
        }   _____________________________ n * n
        i++;          ___________________ n
        print(s);     ___________________ n
    }   _____________________________ n
    print(s);         ___________________ 1
}   _____________________________ 1

                                  ___________
Contador de frecuencias CF =       4n² + 7n + 6
```

Pseudo programa Java e)

```
public static void main(String args[])
{
    int i, j, n, s; _____________________ 1
    i = 1;    ____________________________ 1
    read(m);  ____________________________ 1
    while (i <= m) {  ___________________ m + 1
        read(n);      ___________________ m
        s = 0;        ___________________ m
        j = 1;        ___________________ m
        while (j <= n) { _______________ n * m + m
            s = s + i + j; _____________ n * m
            j++;     ___________________ n * m
        }   _____________________________ n * m
```

```
        i++;          _____________________ m
        print(s);     _____________________ m
    }   ___________________________________ m
    print(s);         _____________________ 1
}   ___________________________________ 1

                                   ___________
Contador de frecuencias CF =        4mn+ 8m + 6
```

Pseudo programa Java f)

```
public static void main(String args[])
{
    int i, n;   ____________________________ 1
    read(n);  ____________________________ 1
    i = n;    ____________________________ 1
    while (i > 1) {  ____________________ lg(n) + 1
        print(i);    ____________________ lg(n)
        i /= 2;      ____________________ lg(n)
    }   ___________________________________ lg(n)
}   ___________________________________ 1

                                   ___________
Contador de frecuencias CF =        4lg(n) + 5
```

Pseudo programa Java g)

```
public static void main(String args[])
{
    int i, n;   ____________________________ 1
    read(n);  ____________________________ 1
    i = 1;    ____________________________ 1
    while (i <= n) { ____________________ log5(n) + 1
        print(i);    ____________________ log5(n)
        i *= 5;      ____________________ log5(n)
    }   ___________________________________ log5(n)
}   ___________________________________ 1

                                   ___________
Contador de frecuencias CF =        4log5(n) + 5
```

Pseudo programa Java h)

```
public static void main(String args[])
{
    int i, j, n; ____________________________ 1
    read(n);  ____________________________ 1
    i = 1;    ____________________________ 1
    while (i < = n) { ___________________ n + 1
        j = n;       ____________________ n
        while (j > 1) {  _______________ nlg(n) + n
            print(i, j); _______________ nlg(n)
            j /= 2;      _______________ nlg(n)
```

```
        }   _____________________________ nlg(n)
        i++;  _____________________________ n
    } _____________________________________ n
}   _____________________________________ 1

                                          _________________
Contador de frecuencias CF =               5n + 4nlg(n) + 5
```

Notación Asintótica

Es una forma de representar matemáticamente el comportamiento de un algoritmo y analizar su eficiencia a medida que aumenta el tamaño de la entrada.

Aplicaciones de la notación asintótica

En computación científica tenemos entre otras aplicaciones, las siguientes:

- Analizar el tiempo de ejecución de un algoritmo
- Entender cómo crecen los tiempos de ejecución y el uso de memoria
- Comprender la eficiencia de un algoritmo cuando la entrada es muy grande
- Aproximar la complejidad temporal o espacial de un algoritmo

Funcionamiento básico

La notación asintótica es relativamente fácil de aplicar y realiza lo siguiente con el CF y los recursos a analizar en detalle:

- Se descartan los coeficientes constantes y los términos menos significativos
- Se enfoca en la parte importante del tiempo de ejecución de un algoritmo y su tasa de crecimiento

Notaciones asintóticas comunes

En el análisis de algoritmos se cuenta con las siguientes notaciones asintóticas, de las cuales revisaremos las tres primeras de la lista que son las más implementadas:

- Big O (O Grande)
- Big Omega (Ω Grande)
- Big Theta (Θ Grande)
- Condicional
- Little o (o pequeña)
- Little omega (ω pequeña)
- Landau Notation (notación Landau)

Notación Asintótica O Grande (*Big O*)

Representada con la letra **O** mayúscula, define la eficiencia del algoritmo a partir del contador de frecuencias. Para esto, se toma la expresión dada por el `CF` y se eliminan los coeficientes, las constantes y los términos negativos; de los términos resultantes, si son dependientes entre sí, se toma el mayor de ellos y este será el orden de magnitud del algoritmo; en otro caso, el orden de magnitud estará determinado por la suma de los términos restantes.

De manera más formal, la notación **O Grande** proporciona una **cota superior** e indica que si el tiempo de ejecución de un algoritmo es $O(f(n))$, entonces para n suficientemente grande, el tiempo de ejecución es a lo sumo $cf(n)$: $t \leq cf(n)$ para alguna constante $c \in \mathbf{R}^{\geq 0}$. La condición de un n grande no es realmente necesaria, como se verá en la "regla del umbral", pero permite hallar constantes más pequeñas de las que se encontrarían en caso contrario, lo cual es útil cuando se buscan buenas cotas sobre el tiempo de ejecución de una implementación cuando se conoce el tiempo de otra.

Gráficamente se muestra como la notación O Grande proporciona una cota superior, esto es, un tiempo máximo de ejecución del algoritmo:

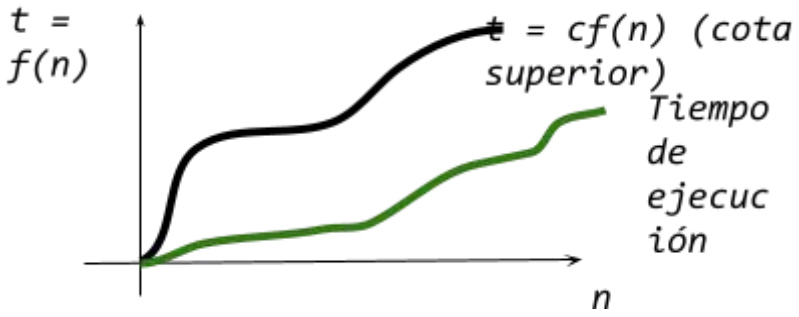


Figura 1.1. Comparación del tiempo de ejecución "normal" con el máximo posible determinado por la cota superior

Órdenes de magnitud en notación asintótica O Grande

Los órdenes de magnitud más frecuentes en informática, expresados en notación asintótica O Grande, se ilustran a continuación.

Orden de magnitud constante `O(1)`

Si el `CF` de un algoritmo es una constante, entonces su orden de magnitud es constante y lo denotamos como `O(1)`. Este tipo de algoritmos carece de ciclos o procesos repetitivos. Los algoritmos de magnitud constante son los ideales, ya que su eficiencia es la mejor, pero no siempre es fácil o posible obtener algoritmos con un `CF` constante.

Orden de magnitud lineal O(n)

El CF se obtiene en términos del valor de un n dado como entrada del algoritmo; generalmente interesan casos de entradas grandes, pero en casos de un n pequeño o "normal" también se cumple la definición (este valor de n puede representar el número de elementos de un arreglo, los registros de un archivo, datos arbitrarios ingresados por el usuario, etc.). Denotamos este orden de magnitud como O(n). Este tipo de algoritmos posee uno o más ciclos independientes.

Orden de magnitud lineal O(m + n)

El CF se obtiene en términos de los valores de m, n, dados como entradas del algoritmo, siendo independientes entre sí. Denotamos este orden de magnitud como O(m + n). Este tipo de algoritmos posee uno o más ciclos independientes y se reduce al caso anterior cuando m = n, por lo que O(m + n) = O(n + n) = O(2n) = O(n). En realidad, este es un caso derivado del orden de magnitud lineal O(n).

Orden de magnitud cuadrático $O(n^2)$

Este tipo de algoritmos posee un ciclo que se ejecuta n veces, y en su interior se encuentra un ciclo (anidado) que también se ejecuta n veces; dichos ciclos son dependientes y como veremos, los términos cuadrático y lineal del CF también lo serán. Denotamos este orden de magnitud como $O(n^2)$.

Orden de magnitud cuadrático O(m * n)

El CF se obtiene en términos de los valores de m, n, dados como entradas del algoritmo, siendo dependientes entre sí. Denotamos este orden de magnitud como O(m * n). Este tipo de algoritmos posee uno o más ciclos dependientes y se reduce al caso anterior cuando m = n, por lo que O(m 2 n) = O(n * n) = $O(n^2)$. En realidad, este es un caso derivado del orden de magnitud cuadrático $O(n^2)$.

Orden de magnitud cúbico $O(n^3)$

Similar al orden de magnitud cuadrático, pero en su interior se encuentran dos ciclos, uno anidado dentro de otro, y el tercero anidado en el segundo, todos ejecutándose n veces; dichos ciclos son dependientes entre sí, y como veremos, el término cúbico del CF predominará al medir la eficiencia del algoritmo. Denotamos este orden de magnitud como $O(n^3)$.

Orden de magnitud logarítmico O(log(n))

Este orden de magnitud se obtiene cuando en un ciclo cuya entrada final es n, se avanza en cada iteración en función de un cociente que afecta la variable controladora del ciclo. Denotamos este orden de magnitud como O(log(n)).

Orden de magnitud semilogarítmico O(nlog(n))

Se obtiene este orden de magnitud cuando en un ciclo de magnitud lineal O(n) se encuentra un ciclo de magnitud O(log(n)).

Orden de magnitud exponencial $O(x^n)$

Aunque no es tan recurrente, hay casos de algoritmos con una eficiencia muy deficiente, tal y como se presenta en los programas con orden de magnitud exponencial y que se denota por $O(x^n)$, $x \in \mathbf{Z}$, $x > 1$.

Clasificación de los algoritmos según su eficiencia

En la siguiente tabla se listan los principales órdenes de magnitud que se encuentran en informática según su eficiencia en orden ascendente, siendo 1 el más eficiente y 7 el más ineficiente.

Eficiencia	Orden de magnitud	Representación	Ejemplo
1	Constante	O(1)	Ver ejemplo 1.2 a)
2	Logarítmico	O(log(n))	Ver ejemplo 1.2 f) y g)
3	Lineal	O(n)	Ver ejemplo 1.2 b)
4	Semilogarítmico	O(nlog(n))	Ver ejemplo 1.2 h)
5	Cuadrático	$O(n^2)$	Ver ejemplo 1.2 d)
6	Cúbico	$O(n^3)$	
7	Exponencial	$O(x^n)$ $x \in \mathbf{Z}$, $x > 1$	

Tabla 1.1. Clasificación de los algoritmos según su eficiencia

Regla del umbral

Sean $f, t: \mathbf{N} \rightarrow \mathbf{R}^+$ dos funciones arbitrarias de los naturales en los reales estrictamente positivos, entonces $t(n) \in O(f(n))$ si y sólo si existe una constante real positiva tal que $t(n) \in O(f(n))$ para cada número natural n.

Regla del máximo

Esta regla permite demostrar que una función es del orden de otra. Sean $f, g: \mathbf{N} \rightarrow \mathbf{R}^+$ dos funciones arbitrarias de los naturales en los reales no negativos. La regla del máximo dice que $O(f(n) + g(n)) = \max(f(n), g(n))$, o si se quiere igual a $\max(O(f(n)), O(g(n))$.

Operaciones sobre notación asintótica

Para simplificar cálculos, es posible manipular la notación asintótica mediante el uso de operadores aritméticos, así por ejemplo $O(f(n)) + O(g(n))$ representa el conjunto de operaciones obtenidas sumando punto a punto en las funciones $O(f(n))$ y $O(g(n))$ para cada $n \in N$. Intuitivamente se puede observar que $f(n)$ es el tiempo que requiere una primera fase del algoritmo para ejecutarse y $g(n)$ el tiempo que requiere una segunda parte para ejecutarse después de haber ejecutado la primera.

Proposición

El tiempo de ejecución de un algoritmo está dado por $f(n) + g(n)$, es decir : $t(n) = O(f(n) + g(n))$. Es válida la expresión $O(f(n) + g(n)) = O(f(n)) + O(g(n))$.

Demostración

Sea $f(n) = m$, $g(n) = n$ los tiempos de ejecución que toman dos partes de un algoritmo, entonces:

$O(f(n) + g(n)) = O(m + n)$ (1).

Por la regla del máximo esto es idéntico a:

$O(max(f(n), g(n))) = O(max(m, n)) = O(m + n)$ (2).

Ahora $O(f(n)) + O(g(n)) = O(m) + O(n)$ (3). Calculando el máximo de esta última igualdad: $max(O(m), O(n)) = O(max(m, n))$.

Pero por la ecuación (2): $O(max(m, n)) = O(m + n)$ (4). Por tanto tenemos por transitividad que $O(f(n) + g(n)) = O(f(n)) + O(g(n))$.

Esta demostración justifica porqué $O(m + n)$ es lineal usando la regla del máximo que dice que esta expresión es equivalente a $O(máx(m, n))$.

Ejemplo 1.3

Supongamos que un algoritmo tiene dos partes de las cuales a la primera le toma un tiempo de $O(n)$ para ejecutarse y a la segunda le toma un tiempo de $O(m)$, por tanto el algoritmo requiere un tiempo de $O(O(n) + O(m)) = max(O(n), O(m))$.

Ejemplo 1.4

Supongamos que un algoritmo ejecuta tres partes, con tiempos de $O(n^2)$, $O(n^3)$ y $O(nlog(n))$, respectivamente. Es claro que el algoritmo completo requiere un tiempo de $O(n^2 + n^3 + nlog(n)) = max(O(n^2), O(n^3), O(nlog(n))) = O(n^3)$.

Aun cuando el tiempo requerido por el algoritmo es lógicamente la suma de los tiempos requeridos por cada parte separada, dicho tiempo es del orden del tiempo requerido por la parte que más consuma tiempo, siempre y cuando el número de partes sea una constante, independientemente del tamaño de la entrada.

La regla del máximo nos dice que si $t(n)$ es una función complicada tal como $t(n) = 15n^3\log(n) - 5n^2 + \log^2(n) + 27$ y si $f(n)$ es el término más significativo de $t(n)$, entonces $O(t(n)) = O(f(n))$, lo cual permite una simplificación significativa y casi que inmediata en la notación asintótica. Para el caso del ejemplo, es fácil ver que $f(n) = 15n^3\log(n)$ es el término más significativo, y descartando el coeficiente, se tiene que $O(t(n)) = O(f(n)) = O(n^3\log(n))$.

En otras palabras, se pueden eliminar los términos de orden inferior porque son despreciables en comparación con el término más significativo para entradas de n suficientemente grandes.

Regla del límite

Es considerada la herramienta más potente y versátil para demostrar que algunas funciones están en el orden de otras.

Sean $f, g: \mathbf{N} \rightarrow \mathbf{R}^{\geq 0}$ dos funciones arbitrarias de los naturales en los reales no negativos, se cumple lo siguiente:

1. Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in R^+$ entonces $f(n) \in O(g(n))$ y $g(n) \in O(f(n))$.
2. Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ entonces $f(n) \in O(g(n))$ pero $g(n) \notin O(f(n))$.
3. Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ entonces $f(n) \notin O(g(n))$ pero $g(n) \in O(f(n))$.

Ejemplo 1.5

Sea $f(n) = \log(n)$, $g(n) = n^{1/2}$. Queremos determinar el orden relativo de estas funciones. Ambas funciones tienden al infinito cuando n tiende al infinito, por tanto utilizaremos la regla de L'Hopital para hallar el límite:

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{log(n)}{\sqrt{n}} = \lim_{n\to\infty} \frac{1/n}{1/(2\sqrt{n})} = \lim_{n\to\infty} \frac{2}{\sqrt{n}} = 0$ Esto significa que $f(n) \in O(g(n))$ pero $g(n) \notin O(f(n))$, entras palabras, $g(n) = n^{1/2}$ crece asintóticamente más rápido que $f(n) = \log(n)$.

Ejemplo 1.6

Tomemos la función arbitraria cuadrática $f(n) = 2n^2 + 10n + 50$. Veamos cómo un simple análisis de los términos de esta función, nos permite ilustrar cómo el término cuadrático predomina en los resultados de la salida final para entradas de n grandes.

Si tomamos $g(n) = 2n^2$, $h(n) = 10n + 50$, también es fácil ver que el término más significativo de $f(n)$ es $2n^2$, y si aplicamos la regla del límite, encontramos igual resultado:

$\lim_{n \to \infty} \frac{h(n)}{g(n)} = \lim_{n \to \infty} \frac{10n+50}{2n^2} = 0$ Por tanto `O(h(n)) ∈ O(g(n))`.

La siguiente tabla ilustra algunas entradas para `n`; la gráfica muestra el comportamiento de las funciones a medida que la entrada se hace más grande.

`f(n) = 2 * n ** 2 + 10 * n + 50` `f(n) = f1(n) + f2(n)`		
`n`	`f1(n) = 2 * n ** 2`	`f2(n) = 10 * n + 50`
0	0	50
20	800	250
40	3200	450
60	7200	650
80	12800	850
100	20000	1050
120	28800	1250
140	39200	1450
160	51200	1650
180	64800	1850
200	80000	2050

Tabla 1.2. Tabla de valores para n, $f_1(n)$ y $f_2(n)$

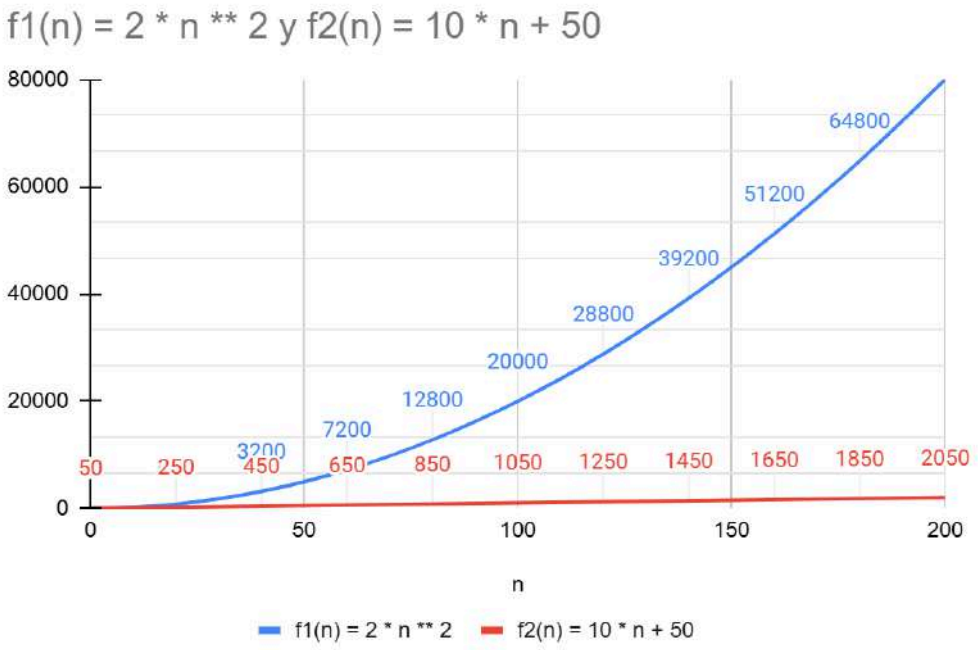


Figura 1.2. Comparación del comportamiento algorítmico entre el orden de magnitud lineal y cuadrático

Notación Asintótica Omega (Ω) Grande (*Big Omega*)

La notación O Grande solo proporciona **cotas superiores** sobre la cantidad de recursos requeridos; mediante la notación asintótica **Omega Grande** (o simplemente **Omega**) es posible obtener una notación que proporciona **cotas inferiores**. La notación Omega permite establecer la cantidad mínima de tiempo que puede tomar un algoritmo en ejecutarse.

De manera más formal, la notación **Ω** Grande proporciona una **cota inferior** e indica que si el tiempo de ejecución de un algoritmo es $\Omega(f(n))$, entonces para n suficientemente, el tiempo de ejecución es a lo menos $cf(n)$: $t \geq cf(n)$ para alguna constante $c \in \mathbf{R}^{\geq 0}$.

Gráficamente se muestra como la notación **Ω** Grande proporciona una cota inferior, esto es, un tiempo mínimo de ejecución del algoritmo:

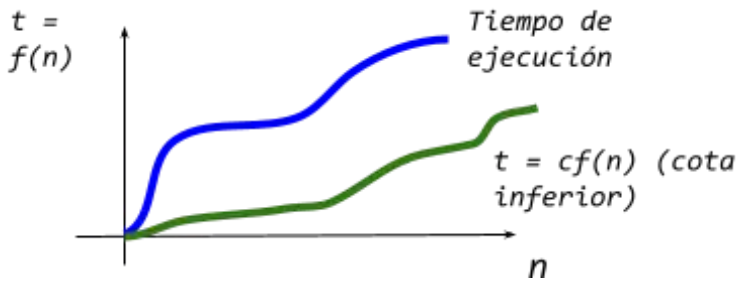


Figura 1.3. Comparación del tiempo de ejecución “normal” con el mínimo posible determinado por la cota inferior

Notación Asintótica Theta (Θ) Grande (*Big Theta*)

La notación **Theta Grande** proporciona una **cota inferior** y otra **cota superior** como límites del tiempo de ejecución de un algoritmo; esto significa que la ejecución del algoritmo no tardará menos de cierto tiempo ni más que otro cierto tiempo. Para ser más precisos, para dos constantes positivas, se tiene que $c_1 f(n) \leq t \leq c_2 f(n)$ para constantes arbitrarias c_1, $c_2 \in \mathbf{R}^{\geq 0}$. Esto es, el tiempo de ejecución está acotado por arriba y por abajo.

La notación Θ Grande indica que se cuenta con una **cota asintóticamente ajustada** sobre el tiempo de ejecución. "Asintóticamente" porque importa en general para valores grandes de n. "Cota ajustada" porque se ajusta el tiempo de ejecución dentro de un rango determinado por un par de constantes positivas hacia arriba y hacia abajo.

Gráficamente se muestra como la notación Θ Grande proporciona una cota inferior y otra superior, esto es, suministra un tiempo mínimo y un tiempo máximo de ejecución del algoritmo:

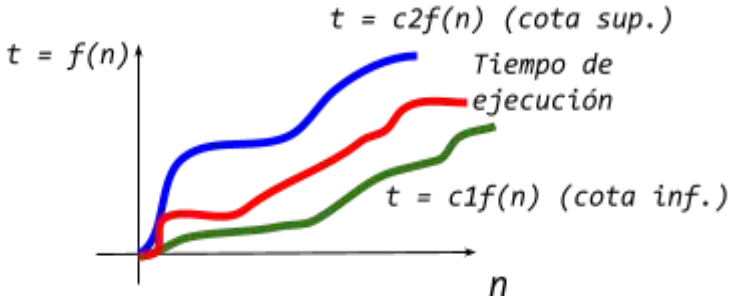


Figura 1.4. Comparación del tiempo de ejecución “normal” con el mínimo y máximo posible determinado por las cotas inferior y superior respectivamente

Preguntas

Ejercicios

1. Utilice la regla del límite para encontrar en qué orden de magnitud relativo está cada función. En donde sea necesario utilice la suma de funciones para separar éstas y hacer el análisis
   a. $f(n) = 10n + 7$
   b. $f(n) = 7n^2 + 2n + 7$
   c. $f(n) = 8n^3 + 5n + 10$
   d. $f(n) = 3n^3 + 4n^2$
   e. $f(n) = 8n^3 + 4n^2 + 5n + 1$
   f. $f(n) = 8n^3 + 2n\lg(n) + 3n + 6$
   g. $f(n) = 4m^2 + 10mn + 20$
   h. $f(n) = m^2 + 2mn + n^2$
   i. $f(n) = 2^{3n} + 3n^2 + 12$
   j. $f(n) = n^4 + 4n^3 + 5n + 10$
   k. $f(n) = 2\log_4(n^2) + n + 3$
   l. $f(n) = n\log(n) + 23n + 12$
   m. $f(n) = 2^{3n} + 3n^2$; $g(n) = \log(n) + 3n$
   n. $f(n) = 5n^3 + n^2$; $g(n) = n^4 + 1$
   o. $f(n) = \lg(n^2)$; $g(n) = 2\lg(n)$
2. Grafique los pares de funciones correspondientes del punto 1) a partir de una tabla de valores; compare los rendimientos de cada parte del algoritmo y verifique que coincidan con lo encontrado anteriormente
3. Encuentre el contador de frecuencias y el orden de magnitud de los siguientes algoritmos. Indique además la memoria que consume cada uno y si contienen operaciones no elementales:

   Algoritmo a)

```
1. Inicio
```

```
2. Hacer el pedido del producto al encargado de la tienda
3. Si el producto está disponible, entonces pagarlo y esperar la
   devuelta; en caso contrario, retirarse de la tienda
4. Fin
```

b. Algoritmo b)

```
1. Inicio
2. Escoger el número a determinar si es primo
3. Si el número es un entero positivo (número >= 0) entonces
4. Asigne 2 a divisor
5. Si número / divisor es división exacta entonces
6. El número no es primo y voy al paso 14
7. Si no cumple 5., entonces
8. Incremento en 1 a divisor
9. Si divisor <= número / 2 entonces
10.   Vuelvo al paso 5.
11.   En caso contrario (divisor > número / 2)
12.   El número es primo y voy al paso 14
13.   Si el paso 3 no se cumple, entonces la entrada no es válida y voy
   al paso 14
14.   Fin
```

Algoritmo c):

```
Inicio
Imprimir “Hoy es martes”
Escribir “Hola mundo”
Fin
```

Algoritmo d)

```
Inicio
Leer nombre
Leer a, b, c
Imprimir “Nombre ingresado: “, nombre
Imprimir a, b, c
Fin
```

Algoritmo e)

```
Inicio
Constante pi <- 3.141592
Cadena: nombre
//También puede indicarse: Carácter: nombre
Enteros: a, b, c
nombre <- “Pepe”
Leer a, b, c
Imprimir nombre
Imprimir a, b, c
Fin
```

Algoritmo f)

```
Inicio
a <- 4
b <- 5
c <- -2
r <- a * b + 3 * a * c ^ 3
Imprimir "a = ", a
Imprimir "b = ", b
Imprimir "c = ", c
Imprimir "Resultado operación: ", r
r = b ^ (2 / 3) + a / 3
Imprimir "Resultado operación 2: ", r
Fin
```

Algoritmo g)

```
Inicio
Carácter: nombre
Leer nombre
Si nombre = “Pedro” Entonces
    Imprimir “Reciba un cordial saludo, señor “, nombre
FinSi
Fin
```

Algoritmo h)

```
Inicio
Enteros: numero
Leer numero
Si numero > 0 Entonces
    Imprimir numero, “ es positivo“
SiNo
    Si numero < 0 Entonces
        Imprimir numero, “ es negativo“
    SiNo
        Imprimir “El número es cero”
    FinSi
FinSi
Fin
```

Algoritmo i)

```
Inicio
//El estado civil será guardado en la variable de tipo carácter ec
//ec puede tener los valores: “C” para casado, “S” para soltero
//”V” para viudo, “U” para unión libre y “D” para divorciado
Carácter: nombre, ec
Leer nombre, ec
EnCasoDe ec Hacer
```

```
    Caso “S”:
        Imprimir nombre, “ usted es soltero(a)“
    Caso “C”:
        Imprimir nombre, “ usted es casado(a)“
    Caso “D”:
        Imprimir nombre, “ usted es divorciado(a)“
    Caso “U”:
        Imprimir nombre, “ usted está en unión libre“
    Caso “V”:
        Imprimir nombre, “ usted es viudo(a)“
    EnOtroCaso:
        Imprimir “Estado civil incorrecto“
FinCaso
Fin
```

Algoritmo j)

```
Inicio
Enteros: i, n
i = 1
Leer n
Mientras i <= n Hacer
    Imprimir i
    i = i + 1
FinMientras
Fin
```

Algoritmo k.1)

```
Inicio
Enteros: i, n, suma
Leer n
i = 1
suma = 0
Mientras i <= n Hacer
    suma = suma + i
    i = i + 1
FinMientras
Imprimir “Suma de 1 a “, n, “: “, suma
Fin
```

Algoritmo k.2)

```
Inicio
Enteros: n, suma
Leer n
suma = n * (n + 1) / 2
Imprimir suma
Fin
```

¿Qué hacen este par de algoritmos? ¿Cuál es más eficiente?

Algoritmo l)

```
Inicio
Enteros: totalPersonas
Reales: salario, mayorSalario, sumaSalario, promedioSalario
Carácter: nombre, nombreMayor
totalPersonas = 0
sumaSalario = 0
mayorSalario = 0
Leer nombre
Mientras nombre <> “***” Hacer
    Leer salario
    totalPersonas = totalPersonas  + 1
    sumaSalario = sumaSalario + salario
    Si salario > mayorSalario Entonces
        mayorSalario = salario
        nombreMayor = nombre
    FinSi
    Leer nombre
FinMientras
Si totalPersonas > 0 Entonces
    promedioSalario = sumaSalario / totalPersonas
    Imprimir totalPersonas, promedioSalario, mayorSalario, nombreMayor
SiNo
    Imprimir “No se procesó información”
FinSi
Fin
```

Algoritmo m)

```
Inicio
Enteros: n, i
Reales: nota, suma, promedio
Lógicos: sw
Carácter: codigo
i = 1
sw = Falso //Supuesto: nadie sacó 5.0
Leer n
Mientras i <= n Hacer
    Leer codigo, nota
    suma = suma + nota
    Si nota = 5 Entonces
        sw = Verdadero
    FinSi
    i = i + 1
FinMientras
promedio = suma / n
```

```
Si sw Entonces
    Imprimir “Al menos un estudiante obtuvo una nota de 5.0”
SiNo
    Imprimir “No hubo estudiantes que obtuvieran una nota de 5.0”
FinSi
Fin
```

Algoritmo n)

```
Inicio
Definir tp Como Entero
Definir nota, sumaNota, menorNota, promedio Como Real
Definir nombre, menorNombre Como Caracter
tp = 0 //Total personas: contador
sumaNota = 0 //Suma de notas para hallar promedio
menorNota = 6
Repetir
    Imprimir "Nombre: " Sin Saltar
    Leer nombre
    Imprimir "nota: " Sin Saltar
    Leer nota
    Si nombre <> '*' Y nota >= 0 Y nota <= 5
        tp = tp + 1
        sumaNota = sumaNota + nota
        Si nota < menorNota Entonces
            menorNota = nota
            menorNombre = nombre
        FinSi
    SiNo
        Imprimir "----- Último registro no procesado -----"
    FinSi
Hasta Que nombre = '*'
Imprimir " "
Imprimir "***** Resumen *****"
Imprimir "Total estudiantes: ", tp
Si tp > 0
    Imprimir "Peor nota --> ", menorNombre, ": ", menorNota
    promedio = sumaNota / tp
    Imprimir "promedio notas: ", promedio
FinSi
Fin
```

Algoritmo ñ)

```
Inicio
Enteros: n, i
Reales: cuadrado, raizCuadrada
Leer n
Para i = 1 Hasta n Paso 1
```

```
    cuadrado = i ^ 2
    raizCuadrada = i ^ 0.5
    Imprimir i, cuadrado, raizCuadrada
FinPara
Fin
```

Algoritmo o)

```
Inicio
Enteros: N, i, fact
Leer N
fact = 1
Para i = N, 1, -1
    fact = fact * i
FinPara
Imprimir N, “! = ”, fact
Fin
```

Algoritmo p)

```
Algoritmo CiclosAnidados
    Definir cpe, cps, cp_si, cp_no Como Entero
    Definir respuesta, nombre Como Caracter
    cpe = 0 //Contador personas empresa
    cp_si = 0 //Contador personas participan
    cp_no = 0 //Contador personas no participan
    Para i = 1 Hasta 3 Con Paso 1
     cps = 0 //Contador personas sucursal
     Imprimir "--- Sucursal: ", i, " ---"
     Imprimir("Ingrese su nombre (* para terminar): ") Sin Saltar
     Leer nombre
     Mientras nombre <> '*' Hacer
           cps = cps + 1
           Imprimir("¿Participa en la rifa? (s/n): ") Sin Saltar
           Leer respuesta
           Si respuesta == 's'
                 cp_si = cp_si + 1
           SiNo
                 cp_no = cp_no + 1
           FinSi
           Imprimir("Ingrese su nombre (* para terminar): ") Sin Saltar
           Leer nombre
     FinMientras
     Imprimir "Total empleados sucursal: ", i, ": ", cps
     cpe = cpe + cps
    FinPara
    Imprimir "Total empleados empresa: ", cpe
    Imprimir "Total empleados que participan: ", cp_si
    Imprimir "Total empleados que no participan: ", cp_no
```

```
FinAlgoritmo
```

Algoritmo q)

```
Inicio
Enteros: c
Carácter: nombre
c = 0
Mientras Verdadero Hacer
    Imprimir "Ingrese un el nombre (* para terminar): "
    Leer nombre
    Si nombre = "*" Entonces
        Interrumpir
    FinSi
    c = c + 1
FinMientras
Imprimir "Total personas: ", c
Fin
```

Algoritmo r)

```
// Función sumaNumeros para sumar dos números enteros
Funcion sum <- sumaNumeros(Enteros x, Enteros z)
      sum <- x + z
Fin Funcion

// Procedimiento imprimirInfo para mostrar un dato. No devuelve valores
SubAlgoritmo imprimirInfo(Carácter dato)
      Imprimir dato
FinSubAlgoritmo

// Procedimiento duplicar; recibe un parámetro por valor y otro por
referencia
SubAlgoritmo duplicar(Enteros num1 Por Valor, Enteros num2 Por Referencia)
      num1 <- num1 * 2
      num2 <- num2 * 2
      Imprimir "Número 1 duplicado: ", num1
      Imprimir "Número 2 duplicado: ", num2
FinSubAlgoritmo

// Programa principal desde donde se llaman los subprogramas
Algoritmo ProgramaPrincipal
      Definir a, b Como Caracter
      Definir n1, n2, s Como Entero
      Imprimir "Ingrese a: " Sin Saltar
      Leer a
      si a no es numero Entonces
            a <- "0"
      FinSi
```

```
      n1 <- ConvertirANumero(a)
      Imprimir "Ingrese b: " Sin Saltar
      Leer b
      si b no es numero Entonces
            b <- "0"
      FinSi
      n2 <- ConvertirANumero(b)
      s <- sumaNumeros(n1, n2)
      imprimirInfo("Resultado suma: " + ConvertirATexto(s))
      duplicar(n1, n2)
      imprimirInfo("Valor de a: " + ConvertirATexto(n1))
      imprimirInfo("Valor de b: " + ConvertirATexto(n2))
FinAlgoritmo
```

Algoritmo s)

```
Inicio
Enteros: c, i, n, num
Leer n
c = 0
i = 1
Mientras i <= n Hacer
    Imprimir "Ingrese un número: "
    Leer num
    Si num == 0 Entonces
        c = c + 1
    FinSi
    i = i * 2
FinMientras
Imprimir "Total ceros: ", c
Fin
```

¿Qué hace este algoritmo?

Algoritmo t)

```
Inicio
Enteros: c, i, n, num
Leer n
c = 0
i = n
Mientras i > 1 Hacer
    c = c + i
    i = i / 4
FinMientras
Imprimir "Valor de c: ", c
Fin
```

¿Qué hace este algoritmo?

Algoritmo u)

```
Inicio
Enteros: c, i, j, n, num
Leer n
i = 1
Mientras i <= n
    j = n
    c = 0
    Mientras j > 1 Hacer
        leer num
        c = c + num * j ^ 2
        j = j / 2
    FinMientras
    Imprimir "Valor de c: ", c
FinMientras
Fin
```

¿Qué hace este algoritmo?

Algoritmo v)

```
Inicio
Enteros: c, i, j, n, num
Leer n
i = 1
Mientras i <= n
    j = i
    c = 0
    Mientras j > 1 Hacer
        leer num
        c = c + num * j ^ 2
        j = j / 2
    FinMientras
    Imprimir "Valor de c: ", c
FinMientras
Fin
```

¿Qué hace este algoritmo?

Algoritmo w)

```
Inicio
Enteros: c, i, j, n, m, num
Leer n
i = 1
Mientras i <= m
    j = 1
    Leer m
```

```
    c = 1
    Mientras j <= m Hacer
        leer num
        c = c * num
        j = j + 1
    FinMientras
    Imprimir "Valor de c: ", c
FinMientras
Fin
```

¿Qué hace este algoritmo?

Algoritmo x)

```
Inicio
Enteros: h, m, s
h = 0
Mientras h < 24
    m = 0
    Mientras m < 60 Hacer
        s = 0
        Mientras s < 60 Hacer
            Imprimir h + “:” + m + “:” + s + “\n”
            s = s + 1
        FinMientras
        m = m + 1
    FinMientras
    h = h + 1
FinMientras
Fin
```

¿Qué hace este algoritmo?

Capítulo 2. Búsqueda y ordenamiento

Búsqueda

La búsqueda es una de las operaciones más importantes y comunes en vectores, otras estructuras de datos y otros objetos computacionales, por lo que se han desarrollado distintas técnicas, unas más complejas que otras para la localización de información; los tipos de búsqueda sobre vectores y otras estructuras de datos más conocidos, son:

- Búsqueda secuencial
- Búsqueda binaria
- Búsqueda por transformación de claves
- Árboles de búsqueda

A continuación se analizan los algoritmos que permiten realizar dichas búsquedas midiendo y comparando su eficiencia, tanto interna como externa.

Búsqueda interna

Búsqueda secuencial

Es el tipo de búsqueda más sencilla y de las más utilizadas; consiste en tomar un dato y compararlo elemento por elemento del vector hasta encontrarlo o hasta terminar de recorrer el vector. El tipo de dato que devuelve el método (función) puede ser un valor *lógico* (*booleano*) para indicar que el dato está o no en el arreglo; también puede devolver un valor *entero* con la posición donde el dato se encuentra, en cuyo caso se inicializa en un valor por fuera de los posibles valores que tome el índice del vector para indicar que el dato no se encuentra.

Ejemplo 1.
Búsqueda secuencial de un dato en un vector. Este tipo de búsqueda también puede implementarse sobre una lista ligada devolviendo preferiblemente un valor booleano.

Programa Java:

```
public int secuencialSearchVector(int datum)
{
    int i, pos;
    i = 0;
    pos = -1;
    while (i < this.n && pos == -1) {
        if (this.vec[i] == datum) {
            pos = i;
        } else {
```

```
            i++;
        }
    }
    return pos;
}
```

Búsqueda binaria

Para implementar este tipo de búsqueda, el requisito es que **el vector debe estar ordenado**. El método consiste en establecer un **límite inferior** y un **límite superior**, y a partir de estos datos, se toma el elemento central calculando la posición "**promedio**"; si el dato se encuentra en dicha posición, la búsqueda finaliza, de lo contrario, se evalúa si el dato es mayor o menor que el elemento en la posición y se redefinen los límites. El proceso finaliza si el dato es encontrado o si no hay un intervalo de búsqueda. Esta búsqueda reduce significativamente el número de comparaciones, lo cual puede verse en vectores grandes.

Ejemplo 1.
Búsqueda binaria de un dato en un vector.

Programa Java:

```
public int binarySearchVector(int datum)
{
    int lowerLimit, upperLimit, pos, centralPos;
    lowerLimit = 0;
    upperLimit = this.n;
    pos = -1;
    while (lowerLimit <= upperLimit && pos == -1) {
        centralPos = (upperLimit + lowerLimit) / 2;
        if (this.vec[centralPos] == datum) {
            pos = centralPos;
        } else {
            if (this.vec[centralPos] > datum) {
                upperLimit = centralPos - 1;
            } else {
                lowerLimit = centralPos + 1;
            }
        }
    }
    return pos;
}
```

Búsqueda externa

Concepto

Ordenamiento

La ordenación de datos es una operación importante y requerida en distintas situaciones al operar con vectores y otras estructuras de datos; consiste en clasificar los elementos en un orden determinado, facilitando así las tareas de búsqueda. Al igual que con la búsqueda de datos, con la ordenación se han desarrollado distintas técnicas, unas más complejas que otras que permiten la clasificación de la información. Los tipos de ordenación más conocidos sobre vectores y otras estructuras de datos, son:

- Ordenación por intercambio directo o burbuja
- Ordenación por inserción directa o método de la baraja
- Ordenación por selección directa
- Ordenación por el método de *Shell*
- Ordenación por el método *quicksort*
- Ordenación por el método del montículo (*heapsort*)

A continuación se analizan los algoritmos que permiten realizar dichas ordenaciones midiendo y comparando su eficiencia interna y externa.

Ordenación interna

Concepto

Ordenación por intercambio directo o burbuja

Es el tipo de ordenación más sencillo de todos, de los más utilizados, pero también el más ineficiente. Este método consiste en tomar cada elemento del vector y compararlo con los siguientes $n - i - 1$ elementos; si alguno es mayor (o menor), se realiza un *intercambio directo* y se continúa comparando. Este método es lento comparado con otros, pero garantiza que deja ordenado cada elemento antes de pasar al siguiente.

Ejemplo 1.
Ordenación de un vector por el método de **intercambio directo o burbuja**.

Programa Java:

```
public void sortBubbleVector()
{
    int i, j, aux;

    for (i = 0; i < this.n - 1; i++) {
        for (j = i + 1; j < this.n; j++) {
            if (this.vec[i] > this.vec[j]) {
                aux = this.vec[i];
                this.vec[i] = this.vec[j];
                 this.vec[j] = aux;
            }
        }
```

```
    }
}
```

Ordenación por inserción directa o método de la baraja

Este método es tomado de los juegos de cartas y cómo los jugadores ordenan éstas ubicándolas de menor a mayor de izquierda a derecha; de ahí que se conozca con el nombre de método de la baraja.

Se trata de insertar un elemento del arreglo en la parte izquierda del mismo, que ya se encuentra ordenada; el proceso se repite hasta el n-ésimo elemento.

Ejemplo 1.
Ordenación de un vector por el método de **inserción directa o método de la baraja**.

Supongamos que se tiene el siguiente vector:

vec

15	*67*	*8*	*16*	*44*	*27*	*12*	*35*

Se deben realizar las siguientes comparaciones:

Primera pasada
vec[1] < vec[0] → 67 < 15: no hay intercambio y va a la siguiente pasada
vec

15	*67*	*8*	*16*	*44*	*27*	*12*	*35*

Segunda pasada
vec[2] < vec[1] → 8 < 67: sí hay intercambio
vec[1] < vec[0] → 8 < 15: sí hay intercambio
vec

8	*15*	*67*	*16*	*44*	*27*	*12*	*35*

Tercera pasada
vec[3] < vec[2] → 16 < 67: sí hay intercambio
vec[2] < vec[1] → 16 < 15: no hay intercambio y va a la siguiente pasada
vec

8	*15*	*16*	*67*	*44*	*27*	*12*	*35*

Una vez se determina la posición del elemento, se terminan las comparaciones, como en el caso de la tercera pasada, pues los elementos van quedando organizados en cada pasada a la izquierda del arreglo de menor a mayor.

Se deja como ejercicio verificar las demás pasadas que realiza el algoritmo y hacer el análisis de eficiencia de éste.

Ordenación por el método de selección directa

En este método se localiza el menor elemento y se intercambia con la primera posición; luego se busca el segundo en elemento menor en los n - 1 elementos restantes y se intercambia con la segunda, y así sucesivamente con los demás elementos, es decir, se busca el menor en los primeros n elementos, luego en los n - 1, luego en los n - 2, hasta llegar al penúltimo elemento.

Ejemplo 1.
Ordenación de un vector por el método de **selección directa**.

Programa Java:

```
public void sortSelectionVector()
{
    int i, j, k, min;
    for (i = 0; i < this.n - 1; i++) {
        min = this.vec[i];
        k = i;
        for (j = i + 1; j < this.n; j++) {
            if (this.vec[j] < min) {
                min = this.vec[j];
                k = j;
            }
        }
        this.vec[k] = this.vec[i];
        this.vec[i] = min;
    }
}
```

Ordenación por fusión (merge sort)

El enfoque para la solución de este problema pertenece a la técnica **Divide y Vencerás (VyC)**, de la cual se hablará más adelante. Para ordenar una lista dada de n números, se divide en dos listas de aproximadamente n/2 números cada una, se ordenan cada una por turno y se intercalan ambos resultados adecuadamente para obtener la versión ordenada de la lista dada (ver figura). Este enfoque se conoce como algoritmo de **ordenación por fusión** (***Merge Sort***)[18].

[18] El algoritmo de ordenación por fusión se debe al científico húngaro John von Neumann, quien lo propuso en 1945 y a quien le debemos también la llamada arquitectura “von Neumann”, base de las máquinas que utilizamos actualmente.

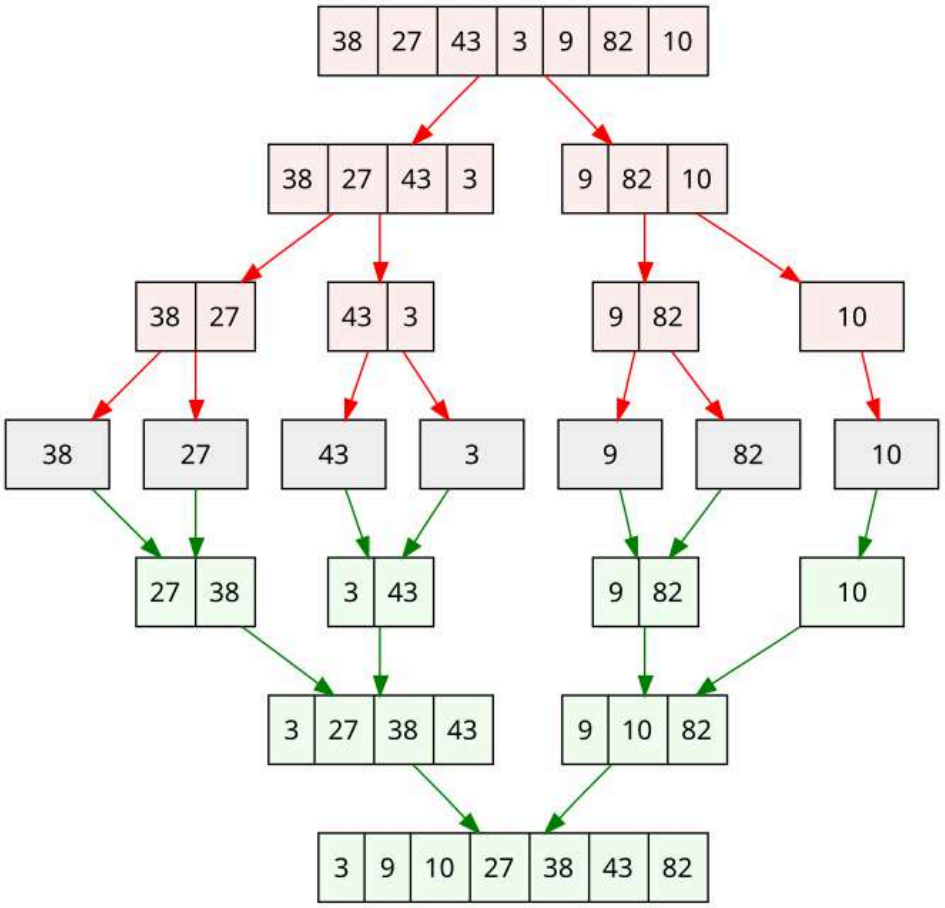


Figura. Método de divide y vencerás para ordenar la lista (38, 27, 43, 3, 9, 82, 10) en orden creciente. Mitad superior: división en sublistas; mitad media: ordenación trivial de una lista de un elemento; mitad inferior: composición de sublistas ordenadas.[19]

Merge Sort es un algoritmo de ordenación altamente eficiente y fiable, ideal para manejar grandes volúmenes de datos debido a su complejidad temporal consistente de $O(nlg(n))$. Aunque su implementación puede ser ligeramente más compleja que otros algoritmos más simples de ordenación, los beneficios de su rendimiento y fiabilidad justifican su uso en entornos que demandan alta eficiencia y precisión. Merge Sort proporciona una solución robusta y eficaz en el campo de los algoritmos de ordenación.

En esencia, Merge Sort funciona descomponiendo recursivamente un vector sin ordenar en vectores más pequeños, ordenándolos y luego fusionándolos de nuevo de forma ordenada. Este proceso se basa en el paradigma de "divide y vencerás", una potente técnica algorítmica que consta de tres pasos fundamentales:

- Dividir: divide el vector original en dos mitades.
- Conquistar: ordena recursivamente cada mitad.
- Combinar: fusiona las dos mitades ordenadas para crear un vector ordenado.

[19] Imagen tomada de https://en-m-wikipedia-org.translate.goog/wiki/Divide-and-conquer_algorithm?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=sge#:~:text=A%20divide%2Dand%2Dconquer%20algorithm,solution%20to%20the%20original%20problem.

Esta descomposición recursiva continúa hasta alcanzar el caso base: vectores con un solo elemento, que están ordenados de forma inherente. El proceso de fusión reconstruye entonces el vector ordenado paso a paso.

Ejemplo
Algoritmo “mergesort” (ordenación por fusión).

Este algoritmo se implementa sobre la clase Vector del proyecto

Programa Java

```
   public void merge(int arr[], int l, int m, int r)
    {
        int n1 = m - l + 1;
        int n2 = r - m;

        int L[] = new int[n1];
        int R[] = new int[n2];

        for (int i = 0; i < n1; ++i) {
            L[i] = arr[l + i];
        }

        for (int j = 0; j < n2; ++j) {
            R[j] = arr[m + 1 + j];
        }

        int i = 0;
        j = 0;
        int k = l;

        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            } else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }
```

```
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }

    public void sort(int arr[], int l, int r)
    {
        if (l < r) {
            int m = l + (r - l) / 2;

            sort(arr, l, m);
            sort(arr, m + 1, r);

            merge(arr, l, m, r);
        }
    }

// Para hacer uso de la ordenación por fusión:
objVec.sort(objVec.getVec(), 0 , objVec.getN() - 1);
```

Ordenación externa

Concepto

Preguntas

Ejercicios

Capítulo 3. Complejidad algorítmica en la recursión y en estructuras de datos

La complejidad de algoritmos que utilizan estructuras de datos o recursividad también es estudiada en la computación científica en donde se analiza tanto su complejidad temporal como espacial.

Complejidad algorítmica en arreglos

La complejidad de los algoritmos en arreglos, tanto en tiempo como en espacio, varía según la operación que se realice y las dimensiones del arreglo.

Arreglos unidimensionales (vectores)

Ya se vio como los vectores tienen una complejidad temporal en operaciones como la búsqueda secuencial de O(n), así como la inserción o eliminación de elementos, mientras que la búsqueda binaria tiene un orden de magnitud O(lg(n)). Agregar elementos por el final del vector tiene un tiempo de O(1) y recorrerlo de O(n). La complejidad espacial también es del mismo orden.

Arreglos bidimensionales (matrices)

Los algoritmos sobre matrices más comunes como llenarla o recorrerla, tienen en general un orden de magnitud de O(m*n) o de $O(n^2)$ en caso de matrices cuadradas. La complejidad espacial también es del mismo orden.

Arreglos multidimensionales (n > 2)

En general, para realizar operaciones sobre arreglos, se requerirá un orden de magnitud proporcional al producto de sus dimensiones, esto es, un orden de magnitud de $O(n_1*n_2*n_3*...*n_m)$ o de $O(n^m)$ (n_1, n_2, n_3, ..., n_m, m, $\in$ **N**) en caso de dimensiones del mismo orden; por ejemplo, para recorrer una estructura en forma de cubo (tridimensional), se tiene una complejidad algorítmica de $O(n^3)$. La complejidad espacial también es del mismo orden.

Complejidad algorítmica en listas ligadas

La complejidad de los algoritmos en listas ligadas[20], tanto en tiempo como en espacio, varía según la operación que se realice y es similar al tratamiento con vectores. La complejidad temporal de operaciones como la búsqueda lineal es O(n), mientras que la inserción o eliminación en la cabeza o final es O(1), sin embargo, eliminar el último también puede tener un orden de magnitud O(n),dependiendo de cómo se esté tratando la lista, si es simplemente ligada, doblemente ligada o circular. La complejidad espacial es generalmente O(n) para almacenar los nodos y sus referencias, aunque también puede ser O(1) si cada nodo se agrega uno a uno.

Complejidad Temporal

Se analizan las operaciones fundamentales sobre listas para determinar el tiempo que requiere la ejecución de cada una de estas.

Búsqueda

La búsqueda de un elemento en una lista ligada requiere, en el peor caso, recorrer todos los nodos (O(n)). Si la lista está ordenada, la búsqueda binaria (O(log n)) podría ser una alternativa en la cual pensar, pero por las características del algoritmo, se hace casi que inaplicable en este caso.

Inserción

La inserción en la cabeza o final es de complejidad O(1) porque no requiere recorrer la lista; al final es aplicable esto si se tiene un puntero que apunta al último nodo o es circular, ya que de lo contrario sería necesario ir hasta dicho nodo para hacer la inserción. Insertar en una posición cualquiera puede requerir en general tener que recorrer la lista hasta esa posición (O(n)), dado que implica realizar una búsqueda de la referencia donde se va a insertar. En realidad, la inserción requiere de un búsqueda previa, que hace que su orden de magnitud sea O(n), aunque si se analiza desde la operación como tal suponiendo que ya se realizó la búsqueda, el orden de magnitud es O(1).

Eliminación

Eliminar un elemento de la lista puede requerir recorrerla para encontrar la posición del elemento, esto es, buscarlo, por lo que requiere en general un tiempo de O(n). Esta operación es similar a la inserción y aplica la misma observación hecha allí.

Nota: Inserción/Eliminación en una lista doblemente ligada:
Las operaciones de inserción y eliminación son de complejidad O(1) si se conoce la posición o el elemento a eliminar/insertar. De igual manera, si no hay un puntero apuntando al nodo requerido, será necesario realizar la búsqueda, que requiere un tiempo O(n).

[20] Al hablar de listas ligadas, se consideran todos los tipos: simplemente ligadas (LSL), simplemente ligada circular (LSLC), doblemente ligada (LDL) y doblemente ligada circular (LDLC).

Modificación

Esta operación es más simple, ya que solo es cambiar la información del nodo (no se considera la complejidad del registro que compone el nodo como tal), por lo que el orden de magnitud es el mismo que en la búsqueda, es decir, O(n). Si se cuenta con la referencia del nodo, el algoritmo solo requiere un tiempo de O(1).

Recorrido

Recorrer la lista (visitar cada nodo) tiene una complejidad de O(n).

Complejidad Espacial

La complejidad espacial es O(n), esto es, es proporcional al número de nodos de la lista. Cada nodo requiere un espacio para almacenar la información de acuerdo a su tipo de dato y una referencia al siguiente nodo (y anterior de ser doblemente ligada). El espacio requerido por cada nodo es proporcional al tamaño de los datos almacenados en el nodo y al tamaño de la referencia/puntero; en el caso más simple, el nodo puede estar compuesto de un registro con un dato primitivo y uno o dos datos tipo puntero (referencia). El tratamiento de listas ligadas requerirá también de un puntero que apunte al primer nodo (cabeza) de la lista.

Complejidad algorítmica en pilas y colas

Las pilas y colas son estructuras de datos abstractas que no tienen implementación directa en los lenguajes de programación, pero que pueden simularse mediante vectores o listas ligadas.

Complejidad temporal

Se podría decir que las operaciones fundamentales sobre pilas y colas tendrían una complejidad algorítmica O(1), sin embargo, por el tratamiento mismo de cada estructura de datos con que se implementan las pilas y colas, es necesario considerar cada caso.

Pilas

Las pilas obedecen a la metodología conocida como **LIFO** (*Last In, First Out*), por lo que las operaciones se hacen por un extremo. Las operaciones más importantes sobre pilas son conocidas como **apilar** y **desapilar**, las cuales consisten en agregar y quitar elementos respectivamente; ambas tienen un orden de magnitud O(1) tanto si se trata con vectores o con listas ligadas. Otras operaciones, como buscar un elemento, insertar o eliminar, requieren, en el peor de los casos, mover todos los elementos (o n-1 elementos) a una

nueva pila, insertar o quitar el elemento, y luego regresar los elementos a la pila original, lo que representa 2n movimientos, obteniendo así una complejidad algorítmica de O(n).

Colas

Las colas obedecen a la metodología conocida como **FIFO** (*First In, First Out*), por lo que las operaciones se hacen por los extremos. Las operaciones más importantes sobre colas son conocidas como **encolar** y **desencolar**, las cuales consisten en agregar y quitar elementos respectivamente.

Agregar un elemento a la cola (encolar) en un vector tiene una complejidad de O(1); en una LSL o LSLC es O(1) si se tiene un puntero en el último nodo, de lo contrario, será necesario recorrer la lista hasta éste, lo que implica una complejidad de O(n). En una LDL o LDLC la complejidad es O(1).

Eliminar un elemento de la cola (desencolar) en un vector tendrá un orden de magnitud O(n), ya que implica mover los n-1 elementos que le siguen hacia la derecha; si se trata con listas ligadas, su complejidad es de O(1).

Otras operaciones, como buscar un elemento, insertar o eliminar, requieren, en el peor de los casos, mover todos los elementos (o n-1 elementos) a una nueva cola, insertar o quitar el elemento, y luego regresar los elementos a la cola original, lo que representa 2n movimientos, obteniendo así una complejidad algorítmica de O(n).

Complejidad espacial

La complejidad espacial tanto para pilas como para colas es O(n), ya sean tratadas con arreglos unidimensionales o con listas ligadas.

Complejidad algorítmica en la recursión

El análisis temporal de un algoritmo iterativo es simple con base en la operación básica de este, para los algoritmos recursivos hay una dificultad añadida, pues la función que establece su tiempo de ejecución viene dada por una ecuación en recurrencia, es decir, $\boldsymbol{T}(\boldsymbol{n}) = \boldsymbol{E}(\boldsymbol{n})$, en donde en la expresión E aparece la propia función $\boldsymbol{T}$.

Ecuaciones de recurrencia

Cuando se quiere calcular la demanda de recursos de un algoritmo definido recursivamente, la función complejidad que resulta no está definida sólo en términos del tamaño del problema y algunas constantes, sino en términos de la función de complejidad misma.

Además no es una sola ecuación, dado que existen otras (al menos una) que determinan la cantidad de recursos para los casos base de los algoritmos recursivos. Dada esta situación,

para poder obtener el comportamiento del algoritmo, es necesario resolver el sistema recurrente obtenido.

Ejemplo
Calcular el factorial de un número n.

Programa Java
```
public int factorial(int n)
{
    if (n == 0) {
        return 1;
    } else {
         return n * factorial(n - 1);
    }
}
```

Se considera el producto `n * factorial(n - 1)` como operación básica y el costo del caso base como 1, esto es `T(0) = 1`; así, se puede construir la ecuación de recurrencia para calcular la complejidad del algoritmo:

`T(n) = 1 + T(n - 1).`
Donde 1 es el costo de la multiplicación y `T(n - 1)` es el costo de la llamada a `factorial(n - 1)`.
`T(0) = 1`. Costo del caso base.

Ejemplo
Encontrar el n-ésimo término de la serie de Fibonacci.

Programa Java
```
public static int fibonacci(int n)
{
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 0;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Consideremos la suma `fibonacci(n - 1) + fibonacci(n - 2)` como operación básica y el costo 1 de los dos casos base, así se puede construir la ecuación de recurrencia para calcular la complejidad del algoritmo:

`T(n) = T(n - 1) + 1 + T(n - 2).`
Donde 1 es el costo de la suma, `T(n - 1)` es el costo de la llamada a `fibonacci(n - 1)` y `T(n - 2)` es el costo de la llamada a `fibonacci(n - 2)`.

`T(0) = 1`, `T(1) = 1`. Costo del caso base para `n = 0` y `n = 1`.

Ejemplo

Encontrar el MCD de dos números naturales (este ejemplo también se ilustra más adelante en la sección de problemas NP).

Programa Python

```
def mcdRecursive(self, m, n) -> int:
    if n == 0:
        return m
    else:
        return self.mcdRecursive(n, m % n)
```

Consideremos el caso base (n = 0) con un coste de 1; `mcdRecursive(n, m % n)` como operación básica y el costo 1 de los dos casos base, así se puede construir la ecuación de recurrencia para calcular la complejidad del algoritmo:

`T(n) = T(m % n) + 1`.

Donde 1 es el costo del caso base y `T(m % n)` es el costo de la llamada a `mcdRecursive(n, m % n)`.

`T(0) = m`, costo del caso base para `n = 0`.

Complejidad algorítmica en árboles y grafos

En matemáticas y en ciencias de la computación, la teoría de grafos (también llamada teoría de las gráficas) estudia las propiedades de los **grafos** (también llamadas gráficas). Un grafo es un conjunto, no vacío, de objetos llamados vértices (o nodos) y una selección de pares de vértices, llamados aristas (*edges* en inglés) que pueden ser orientados o no. Típicamente, un grafo se representa mediante una serie de puntos (los vértices) conectados por líneas (las aristas).

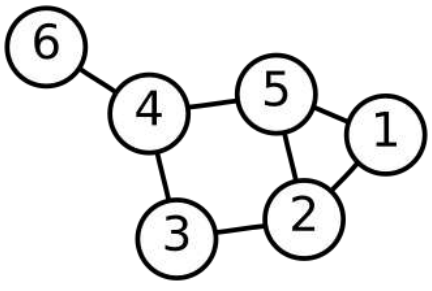


Figura. Grafo con seis vértices y siete aristas

Preguntas

1.

Ejercicios

1.

Capítulo 4. Técnicas de diseño de algoritmos

Problemas P, NP y NP-completos

En la teoría de la complejidad computacional, los problemas P, NP y NP-completos se refieren a la dificultad de resolver problemas computacionales. P (problemas polinomiales) son problemas deterministas que pueden resolverse en tiempo polinomial. NP (problemas no deterministas polinomiales) son problemas cuya solución se puede verificar en tiempo polinomial. Los problemas NP-completos son los más difíciles en NP, y si se encuentra un algoritmo eficiente para uno de ellos, se puede encontrar uno eficiente para todos los problemas en NP.

La relación entre las clases de complejidad NP y P la formuló el científico computacional Stephen Cook y que está aún sin resolver por la teoría de la complejidad computacional. En esencia, la pregunta "**¿es P = NP completo?**" significa: "Si es posible verificar rápidamente las soluciones de un problema de tipo NP ¿eso implica que también es posible 'obtener' las respuestas con la misma rapidez? (es decir, es un problema de tipo P)", donde "rápidamente" significa "en tiempo polinómico".

Los recursos comúnmente estudiados en complejidad computacional son:

- **Tiempo**: mediante una aproximación al número de pasos de ejecución que un algoritmo emplea para resolver el problema.

- **Espacio**: mediante una aproximación a la cantidad de memoria utilizada para resolver el problema.

Los problemas se clasifican en conjuntos o clases de complejidad (L, NL, P, P-Completo, NP, NP-Completo, NP Duro, etc.)[21].

Ejemplo
Suma de subconjuntos. Este problema consiste en hallar un subconjunto no vacío a partir de un conjunto de enteros, cuya suma de elementos sea cero.

Sea A = {−2, −3, 15, 14, 7, −10} un conjunto de números enteros. ¿Existe algún subconjunto de A cuyos elementos sumen 0?

Podemos ver que el subconjunto B = {−2, −3, 15, −10} cumple con esta condición. Sin embargo, encontrar el subconjunto tarda más que encontrar la suma de sus elementos.

La información necesaria para verificar un resultado positivo/afirmativo es llamada **certificado**. Podemos concluir entonces que dado los certificados apropiados, es posible

[21] El Clay Mathematics Institute ha ofrecido un premio de un millón de dólares estadounidenses para la persona que demuestre la solución de este problema (fuente: Wikipedia).

verificar rápidamente las respuestas afirmativas de nuestro problema (en tiempo polinomial) y es esta la razón por la que el problema se encuentra en NP.

Una respuesta a la pregunta P = NP sería determinar si en problemas del tipo SUMA-SUBCONJUNTO es tan fácil hallar la solución como verificarla. Si se encuentra que P no es igual a NP, significa que algunos problemas NP serían significativamente más difíciles de hallar su solución que verificar la misma. La respuesta es aplicable a todo este tipo de problemas y no solo al ejemplo específico citado.

La restricción a problemas de tipo SÍ/NO realmente no es importante; aún si se permiten respuestas más complicadas, el problema resultante es equivalente.

En este tipo de análisis, se requiere un modelo de computador para la que desea estudiar el requerimiento en términos de tiempo. Típicamente, dichos modelos suponen que la computadora es determinista (dado el estado actual de la máquina y las variables de entrada, existe una única acción posible que ésta puede tomar) y secuencial (realiza las acciones una después de la otra). Estas suposiciones son adecuadas para representar el comportamiento de todas las computadoras existentes, incluyendo a las máquinas con computación en paralelo.

En esta teoría, la clase P consiste de todos aquellos problemas de decisión que pueden ser resueltos en una máquina determinista secuencial en un período de tiempo polinomial en función a los datos de entrada; en la teoría de complejidad computacional, la clase P es una de las más importantes.

La clase NP consiste de todos aquellos problemas de decisión cuyas soluciones positivas/afirmativas pueden ser verificadas en tiempo polinómico a partir de ser alimentadas con la información apropiada, o en forma equivalente, cuya solución puede ser hallada en tiempo polinómico en una máquina no determinista. Por lo tanto, la principal pregunta aún sin respuesta en la teoría de la computación está referida a la relación entre estas dos clases:

¿Es **P** igual a **NP**?[22]

[22] En una encuesta realizada en el 2002 entre 100 investigadores, 61 creían que la respuesta era NO, 9 creían que la respuesta era SI, 22 no estaban seguros, y 8 creían que la pregunta podía ser independiente de los axiomas actualmente aceptados, y por lo tanto imposible de demostrar por el SI o por el NO (fuente: Wikipedia)

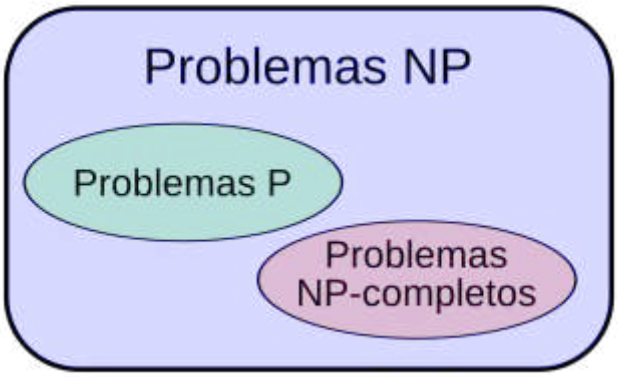


Figura. Diagrama de clases de complejidad para el caso en que P ≠ NP. La existencia de problemas fuera tanto de P como de NP-completos, fue determinada por Pichard T. Ledner.[23]

Problema de decisión

Es un problema que especifica una cadena de caracteres de datos de entrada y requiere como solución una respuesta por el SI o por el NO. Si existe un algoritmo (por ejemplo una máquina de Turing, o un programa en lenguajes Lisp o Pascal con memoria irrestricta) que es capaz de entregar la respuesta correcta para toda cadena de datos de longitud n en a lo sumo cn^k pasos, donde k y c son constantes independientes del conjunto de datos, entonces se dice que el problema puede ser resuelto en tiempo polinómico y lo clasificamos como perteneciente a la clase P. En forma intuitiva, consideramos que los problemas contenidos en P son aquellos que pueden ser resueltos en forma razonablemente rápida.

P es la clase de problemas computacionales que son “eficientemente resolubles” o “tratables”, aunque también existen problemas en P que no son tratables en términos prácticos; por ejemplo, unos requieren al menos $n^{1000000}$ operaciones.

NP es un problema de decisión que es difícil de resolver si no se posee ningún otro dato o información adicional. Sin embargo, si se recibe la información adicional llamada “certificado”, entonces el problema puede ser resuelto fácilmente. Por ejemplo, si se tiene el número 323 y se pregunta si 323 es un número factorizable, sin ningún dato o información adicional, podemos hallar la raíz cuadrada de 323 (≈17.9722), redondear al entero menor en (17) y dividir desde 17 a 2 hasta encontrar una división exacta. Sin embargo, si se da el número 17, podemos dividir 323 por 17 y rápidamente verificar que 323 es factorizable. El número 17 es llamado “un certificado”. El proceso de división para verificar que 323 es factorizable es en esencia una máquina Turing y en este caso es denominado el verificador para 323.

Técnicamente se dice que el problema es fácil si se puede resolver en tiempo polinómico y que es difícil si se resuelve en tiempo exponencial.

[23] Tomado de: Clases de complejidad P y NP - Wikipedia, la enciclopedia libre

La clase P (Problemas Polinomiales)

Son problemas que pueden ser resueltos por un algoritmo en tiempo polinomial, es decir, el tiempo que tarda en ejecutarse el algoritmo aumenta de manera polinomial con respecto al tamaño de la entrada. Por ejemplo ordenar un vector, buscar un elemento en una lista ligada, etc.

P es conocido por contener muchos problemas naturales, incluyendo las versiones de decisión de programa lineal, cálculo del máximo común divisor (MCD), y encontrar una correspondencia máxima.

Problemas notables en P

Algunos problemas naturales son completos para P, incluyendo la conectividad (o la accesibilidad) en grafos no dirigidos.

Una generalización de P es NP, que es la clase de lenguajes decidibles en tiempo polinómico sobre una máquina de Turing no determinista. De forma trivial, tenemos que P es un subconjunto de NP, aunque este hecho no está demostrado, pero la mayor parte de la comunidad científica creen que es un subconjunto estricto.

Nota
Los problemas más difíciles en **P** son los problemas **P-completos**.

Propiedades

Los algoritmos de tiempo polinómico son cerrados respecto a la composición. Intuitivamente, esto quiere decir que si uno escribe una función con un determinado tiempo polinómico y consideramos que las llamadas a esa misma función son constantes y, de tiempo polinómico, entonces el algoritmo completo es de tiempo polinómico. Esto es uno de los motivos principales por los que **P** se considera una máquina independiente; algunos rasgos de esta máquina, como el acceso aleatorio, es que puede calcular en tiempo polinómico el tiempo polinómico del algoritmo principal reduciéndolo a una máquina más básica.

Nota: pruebas existenciales de algoritmos de tiempo polinómico
Se conoce que algunos problemas son resolubles en tiempo polinómico, pero no se conoce ningún algoritmo concreto para solucionarlos. Por ejemplo, el teorema Robertson-Seymour garantiza que hay una lista finita de los menores permitidos que compone (por ejemplo) el conjunto de los grafos que pueden ser integrados sobre un toroide; además, Robertson y Seymour demostraron que hay una complejidad $O(n^3)$ en el algoritmo para determinar si un grafo tiene un grafo incluido. Esto nos da una prueba no constructiva de que hay un algoritmo de tiempo polinómico para determinar si dado un grafo puede ser integrado sobre un toroide, a pesar de no conocerse ningún algoritmo concreto para este problema.

Ejemplos

- Camino Mínimo: encontrar el camino mínimo desde un vértice origen al resto de los vértices.

- Ciclo Euleriano: Encontrar un ciclo que pase por cada arco de un grafo una única vez.

La clase NP (Problemas No Deterministas Polinomiales):

Son problemas para los cuales, dada una solución candidata, existe un algoritmo que puede verificar si esa solución es correcta en tiempo polinomial. Sin embargo, encontrar la solución en sí puede no ser posible en tiempo polinomial. Ejemplos: el problema del viajero, el problema de la mochila, etc.

La clase NP está compuesta por los problemas que tienen un certificado sucinto (también llamado testigo polinómico) para todas las instancias cuya respuesta es un SÍ. La única forma de que tengan un tiempo polinomial es realizando una etapa aleatoria, incluyendo el azar de alguna manera para elegir una posible solución, y entonces en etapas posteriores comprueba si esa solución es correcta.

Completitud de NP

Para analizar la pregunta **P = NP**, resulta muy útil el concepto de completitud **NP**. De manera informal, los problemas de completitud **NP** son los problemas más "difíciles" en **P** en el sentido de que ellos son los que son más probables no se encuentren en P. Problemas **P**-difíciles son aquellos para los cuales cualquier problema en **P** puede ser reducido en tiempo polinómico. Los problemas de completitud **P** son aquellos problemas **P**-difícil que se encuentran en **P**. Si cualquier problema P-completo se encuentra contenido en **NP**, entonces se verificaría que **P = NP**. Desafortunadamente, se ha demostrado que muchos problemas importantes son **P**-completos y no se conoce la existencia de ningún algoritmo rápido para ellos.

Ejemplos

- Camino Máximo: Dados dos vértices de un grafo encontrar el camino (simple) máximo.

- Ciclo Hamiltoniano: Ciclo simple que contiene cada vértice del grafo.

NP-Completo

Son problemas en **NP** que son "los más difíciles" en **NP**. Si se encuentra un algoritmo polinomial para cualquier problema **NP**-completo, se puede encontrar un algoritmo polinomial para todos los problemas en **NP**.

Para abordar la pregunta de si P=NP, el concepto de la completitud de NP es muy útil. Informalmente, los problemas de NP-completos son los problemas más difíciles de NP, en el sentido de que son los más probables de no encontrarse en P. Los problemas de

NP-completos son esos problemas NP-duros que están contenidos en NP, donde los problemas NP-duros son estos que cualquier problema en P puede ser reducido a complejidad polinomial. Por ejemplo, la decisión del Problema del viajero de comercio es NP-completo, así que cualquier caso de cualquier problema en NP puede ser transformado mecánicamente en un caso del Problema del viajero de comercio, de complejidad polinomial. El Problema del viajero de comercio es de los muchos problemas NP-completos existentes.

Si cualquier problema NP-completo estuviera en P, entonces indicaría que P=NP. Desafortunadamente, se sabe que muchos problemas importantes son NP-completos y a fecha de hoy, no se conoce ningún algoritmo rápido para ninguno de ellos. Según esto, no es obvio que exista un problema NP-completo.

Un problema NP-completo trivial e ideado, se puede formular como: Dada una descripción de una máquina de Turing M que se detiene en tiempo polinómico, ¿existe una entrada de tamaño polinómico que M acepte? Es NP porque, dada una entrada, es simple comprobar si M acepta o no la entrada simulando M, es NP-duros porque el verificador para cualquier caso particular de un problema en NP puede ser codificado como una máquina M de tiempo polinomial que toma la solución para ser verificada como entrada. Entonces la pregunta de si el caso es o no un caso, está determinado por la existencia de una entrada válida. El primer problema natural que se demostró ser NP-completo fue el Problema booleano de satisfacibilidad. Este resultado es conocido como el teorema de Cook-Levin; su prueba de que la satisfacibilidad es NP-completo contiene los detalles técnicos sobre máquinas de Turing y como se relacionan con la definición de NP. Sin embargo, después se demostró que el problema era NP-completo, la prueba por reducción proporcionó una manera más simple de demostrar que muchos otros problemas están en esta clase. Así, una clase extensa de problemas aparentemente sin relación es reducible a otra, y son en este sentido el mismo problema.

El Problema P vs NP

Es uno de los problemas abiertos más importantes en matemáticas y computación. Se pregunta si P es igual a NP, es decir, si todos los problemas cuya solución puede verificarse rápidamente (NP) también pueden resolverse rápidamente (P).

Ejemplo

Algoritmo de Euclides para hallar el Máximo Común Divisor (MCD) de dos números naturales.

Matemáticamente, se buscan los divisores comunes de ambos números y se escoge el mayor de éstos.

Recursivamente, el MCD se define así:

$$MCD(m, n) = m,\ si\ n = 0$$
$$MCD(m, n) = MCD(n, m\ MOD\ n),\ si\ n > 0$$

Programa Python

```
class MaxCommonDivisor:

    # Constructor
    def __init__(self) -> None:
        pass

    def mcd(self, m, n) -> int:
        i = m if m < n else n
        while m % i != 0 or n % i != 0:
            i -= 1
        return i

    def mcdEuclides(self, m, n) -> int:
        while n > 0:
            aux = m
            m = n
            n = aux % n
        return m

    def mcdRecursive(self, m, n) -> int:
        if n == 0:
            return m
        else:
            return self.mcdRecursive(n, m % n)
```

Algoritmos de fuerza bruta

También conocidos como **algoritmos exhaustivos**, es una estrategia de resolución de problemas que intenta todas las posibles soluciones hasta encontrar la correcta o demostrar que no hay solución. Es simple de implementar, pero puede ser ineficiente para problemas grandes debido a su alta complejidad temporal.

Es una técnica trivial pero a menudo usada, que consiste en enumerar sistemáticamente todos los posibles candidatos para la solución de un problema, con el fin de chequear si dicho candidato satisface la solución al mismo.

Por ejemplo, un algoritmo de fuerza bruta para encontrar el divisor de un número natural n consistiría en enumerar todos los enteros desde 1 hasta n, chequeando si cada uno de ellos divide n exactamente. Otro ejemplo de búsqueda por fuerza bruta, en este caso para solucionar el problema de las ocho reinas (posicionar ocho reinas en el tablero de ajedrez de forma que ninguna de ellas ataque al resto), consistiría en examinar todas las combinaciones de posición para las 8 reinas (en total 64!/8!(64-8)! = 4.426.165.368 posiciones diferentes), comprobando en cada una de ellas si las reinas se atacan mutuamente.

La búsqueda por fuerza bruta es sencilla de implementar y, siempre que exista, encuentra una solución. Sin embargo, su coste de ejecución es proporcional al número de soluciones candidatas, el cual es exponencialmente proporcional al tamaño del problema.

Es un método utilizado también cuando es más importante una implementación sencilla que una mayor rapidez. Este puede ser el caso en aplicaciones críticas donde cualquier error en el algoritmo puede acarrear serias consecuencias; también es útil como método "base" cuando se desea comparar el desempeño de otros algoritmos metaheurísticos. La búsqueda de fuerza bruta puede ser vista como el método metaheurístico más simple.

Algoritmo básico de fuerza bruta

Para poder utilizar la fuerza bruta a un tipo específico de problema, se deben implementar las funciones **primero**, **siguiente**, **válido**, y **mostrar**. Todas recogerán el parámetro P indicando una instancia en particular del problema:

1. **primero (P)**: genera la primera solución candidata para P.
2. **siguiente (P, c)**: genera la siguiente solución candidata para P después de una solución candidata c.
3. **válido (P, c)**: chequea si una solución candidata c es una solución correcta de P.
4. **mostrar (P, c)**: informa que la solución c es una solución correcta de P.

La función **siguiente** debe indicar de alguna forma cuándo no existen más soluciones candidatas para el problema P después de la última. Una forma de realizar esto consiste en devolver un valor "nulo". De esta misma forma, la función **primero** devolverá un valor "nulo" cuando no exista ninguna solución candidata al problema P.

Pseudocódigo

```
c = primero(P)
Mientras c != nulo
    Si valido(P,c) Entonces
        mostrar(P, c)
        c = siguiente(P,c)
    FinSi
FinMientras
```

Por ejemplo, para buscar los divisores de un entero n, la instancia del problema P es el propio número n. La llamada **primero(n)** devolverá 1 siempre y cuando n ≥ 1, y "nulo" en otro caso; la función **siguiente(n, c)** debe devolver c + 1 si c < n, y "nulo" en caso contrario; **válido(n, c)** devolverá verdadero si y sólo si c es un divisor de n.

Nota

El algoritmo descrito anteriormente llama a la función **mostrar** para cada solución al problema. Este puede ser fácilmente modificado de forma que termine una vez encuentre la primera solución, o bien después de encontrar un determinado número de soluciones, después de probar con un número específico de soluciones candidatas, o después de haber consumido una cantidad fija de tiempo de CPU.

Explosión combinacional

La principal desventaja del método de fuerza bruta es que, para la mayoría de problemas reales, el número de soluciones candidatas es altamente elevado.

Por ejemplo, para buscar los divisores de un número n tal y como se describe anteriormente, el número de soluciones candidatas a probar será de n. Por tanto, si n consta de, digamos, 16 dígitos, la búsqueda requerirá de al menos 1015 comparaciones computacionales, tarea que puede tardar varios días en un ordenador personal. Si n es un bit de 64 dígitos, que aproximadamente puede tener hasta 19 dígitos decimales, la búsqueda puede tardar del orden de 10 años.

Este crecimiento exponencial en el número de candidatos, cuando crece el tamaño del problema ocurre en todo tipo de problemas. Por ejemplo, si buscamos una combinación particular de 10 elementos entonces tendremos que considerar 10! = 3,628,800 candidatos diferentes, lo cual en un PC habitual puede ser generado y probado en menos de un segundo. Sin embargo, añadir un único elemento más —lo cual supone solo un 10% más en el tamaño del problema— multiplicará el número de candidatos 11 veces — lo que supone un 1000% de incremento. Para 20 elementos el número de candidatos diferentes es 20!, es decir, aproximadamente 2.4×1018 o 2.4 millones de millones de millones; la búsqueda podría tardar unos 10000 años. A este fenómeno no deseado se le denomina **explosión combinacional**.

Fuerza bruta lógica

La fuerza bruta lógica, consiste básicamente en lo mismo, pero evitando casos que por razones demasiado obvias se sabe que quedan fuera de la solución buscada.

Este método sólo se usa cuando el número de posibilidades a evitar es lo suficientemente grande y que contando con el tiempo de ejecución del código necesario para implementarlo, reduzca ampliamente el tiempo necesario para encontrar el resultado esperado.

Cuando se habla de **fuerza bruta lógica**, por contraste, la contrapuesta es llamada **fuerza bruta ciega**.

Con el ejemplo de la factorización, cuando tratamos por fuerza bruta de encontrar el divisor de un número natural n enumeraríamos todos los enteros desde 1 hasta n, chequeando si cada uno de ellos divide a n sin generar resto. La fuerza bruta lógica haría lo mismo pero solo con los números primos, dada una tabla de primos, o solo con los impares y el 2 si no poseemos una tabla de primos. Dado que sabemos que cualquier número está compuesto de primos en cualquier cantidad y esto es demasiado obvio, no es absolutamente necesario revisar el 4,6,8,10,12,14,15,16 si ya hemos mirado el 2,3,5,7...

La fuerza bruta lógica sigue siendo fuerza bruta, ya que recorre sin estrategia el espacio de posibilidades pero descartando posibilidades muy obvias y relativamente fáciles de implementar.

En *criptografía*, se denomina **ataque de fuerza bruta** a la forma de recuperar una clave probando todas las combinaciones posibles hasta encontrar aquella que permite el acceso.

Dicho de otro modo, define al procedimiento por el cual a partir del conocimiento del algoritmo de cifrado empleado y de un par texto claro/texto cifrado, se realiza el cifrado (respectivamente, descifrado) de uno de los miembros del par con cada una de las posibles combinaciones de clave, hasta obtener el otro miembro del par. El esfuerzo requerido para que la búsqueda sea exitosa con probabilidad mejor que la par será 10^n-1 operaciones, donde n es la longitud de la clave (también conocido como el *espacio de claves*). Este enfoque no depende de tácticas intelectuales, se basa en hacer muchos intentos.

Otro factor determinante en el coste de realizar un ataque de fuerza bruta es el juego de caracteres que se pueden utilizar en la clave. Contraseñas que sólo utilicen dígitos numéricos serán más fáciles de descifrar que aquellas que incluyen otros caracteres como letras, así como las que están compuestas por menos caracteres serán también más fáciles de descifrar. La complejidad impuesta por la cantidad de caracteres en una contraseña es logarítmica.

La fuerza bruta suele combinarse con un ataque de diccionario, en el que se encuentran diferentes palabras para ir probando con ellas.

Estos tipos de ataques, no son rápidos, para una contraseña compleja, puede tardar siglos (aunque también depende de la capacidad de operación del ordenador que lo ejecute).

En la actualidad este tipo de ataques son usados para hackear Facebook, correos electrónicos, auditar redes WiFi y otras redes sociales.

Se puede calcular la cantidad de posibles contraseñas. De esto depende la cantidad de caracteres de la contraseña en cuestión, y el conjunto de caracteres.

Divide y vencerás

En informática , "**divide y vencerás**" (**DyV**) es un paradigma de diseño de algoritmos . Un algoritmo de "divide y vencerás" descompone recursivamente un problema en dos o más subproblemas del mismo tipo o de tipo relacionado, hasta que estos se vuelven lo suficientemente simples como para resolverse directamente. Las soluciones de los subproblemas se combinan para obtener una solución al problema original.

La técnica de divide y vencerás es la base de algoritmos eficientes para muchos problemas, como la ordenación (por ejemplo, quicksort, mergesort), la multiplicación de números grandes (por ejemplo, el algoritmo Karatsuba), la búsqueda del par de puntos más cercano, el análisis sintáctico (por ejemplo, los analizadores de arriba hacia abajo) y el cálculo de la

transformada de Fourier discreta (FFT), así como el máximo común divisor y la búsqueda binaria[24], entre otros ejemplos.

Diseñar algoritmos eficientes de divide y vencerás puede ser difícil. Al igual que en la inducción matemática, a menudo es necesario generalizar el problema para que sea susceptible de una solución recursiva. La exactitud de un algoritmo de divide y vencerás suele demostrarse mediante inducción matemática, y su coste computacional suele determinarse resolviendo relaciones de recurrencia.

El término "divide y vencerás" se aplica a veces a algoritmos que reducen cada problema a un solo subproblema, como el algoritmo de búsqueda binaria para encontrar un registro en una lista ordenada (o su análogo en computación numérica , el algoritmo de bisección para la búsqueda de raíces). Estos algoritmos pueden implementarse de forma más eficiente que los algoritmos generales de divide y vencerás; en particular, si utilizan recursión de cola, pueden convertirse en bucles simples. Sin embargo, bajo esta definición amplia, todo algoritmo que utilice recursión o bucles podría considerarse un "algoritmo de divide y vencerás". Por lo tanto, algunos autores consideran que el término "divide y vencerás" debería utilizarse sólo cuando cada problema pueda generar dos o más subproblemas. Se ha propuesto el término "decrecimiento y vencerás" para la clase de un solo subproblema.

Una aplicación importante de dividir y vencer es en la optimización, donde si el espacio de búsqueda se reduce ("poda") por un factor constante en cada paso, el algoritmo general tiene la misma complejidad asintótica que el paso de poda, y la constante depende del factor de poda (sumando las series geométricas); esto se conoce como podar y buscar.

El paradigma de divide y vencerás se utiliza a menudo para encontrar la solución óptima de un problema. Su idea básica es descomponer un problema dado en dos o más subproblemas similares, pero más simples, resolverlos uno por uno y componer sus soluciones para resolver el problema dado. Los problemas con suficiente simplicidad se resuelven directamente.

El término "divide y vencerás" se aplica a veces a algoritmos que reducen cada problema a un solo subproblema, como el algoritmo de búsqueda binaria para encontrar un registro en una lista ordenada (o su análogo en computación numérica , el algoritmo de bisección para la búsqueda de raíces). Estos algoritmos pueden implementarse de forma más eficiente que los algoritmos generales de divide y vencerás; en particular, si utilizan recursión de cola, pueden convertirse en bucles simples. Sin embargo, bajo esta definición amplia, todo algoritmo que utilice recursión o bucles podría considerarse un "algoritmo de divide y vencerás". Algunos autores consideran que el término "divide y vencerás" debería utilizarse sólo cuando cada problema pueda generar dos o más subproblemas. Se ha propuesto el término "decrecimiento y vencerás" para la clase de un solo subproblema.

[24] La búsqueda binaria es un algoritmo de decrecimiento y conquista donde los subproblemas tienen aproximadamente la mitad del tamaño original, no es precisamente nuevo. La descripción clara del algoritmo en computación la da John Mauchly en un artículo de 1946, sin embargo, la idea de usar una lista ordenada de elementos para facilitar la búsqueda se remonta al menos a Babilonia en el año 200 a. C. Otro antiguo algoritmo de decrecimiento y conquista es el algoritmo euclidiano para calcular el máximo común divisor de dos números reduciendo los números a subproblemas equivalentes cada vez más pequeños, y que data de varios siglos antes de Cristo.

Diseño e implementación

La resolución de un problema mediante esta técnica consta fundamentalmente de los siguientes pasos:

- Se plantea el problema de forma que pueda ser descompuesto en k subproblemas del mismo tipo, pero de menor tamaño. Es decir, si el tamaño de la entrada es n, hemos de conseguir dividir el problema en k subproblemas (donde $1 \leq k \leq n$), cada uno con una entrada de tamaño n / k, y donde $0 \leq n / k < n$. A esta tarea se le conoce como división.
- Luego se resuelven independientemente todos los subproblemas, bien directamente si son elementales o bien de forma recursiva. El hecho de que el tamaño de los subproblemas sea estrictamente menor que el tamaño original del problema garantiza la convergencia hacia los casos elementales, también denominados casos base.
- Al final se combinan las soluciones obtenidas en el paso anterior para construir la solución del problema original.

Los algoritmos divide y vencerás (o *divide and conquer*, en inglés), se diseñan como procedimientos generalmente recursivos.

La corrección de un algoritmo de “divide y vencerás”, está habitualmente probada por inducción matemática, y su coste computacional se determina resolviendo relaciones de recurrencia.

Un algoritmo genérico de implementación de DyV puede ser el siguiente:

```
AlgoritmoDyV (p: TipoProblema): TipoSolucion
   si esCasoBase(p)
      retorne resuelve(p)
   sino
      subproblemas: arreglo de TipoProblema
      subproblemas = divideEnSubproblemas(p)
      solucionesParciales: arreglo de TipoSolucion

      para cada sp en subproblemas
         solucionesParciales.agregar(AlgoritmoDYV(sp))
      finPara

      retorne mezcla(solucionesParciales)
   finSi
finAlgoritmoDyV
```

Por el hecho de usar un diseño recursivo, los algoritmos diseñados mediante la técnica de Divide y Vencerás van a heredar las ventajas e inconvenientes que la recursión plantea:

- El diseño que se obtiene suele ser simple, claro, robusto y elegante, lo que da lugar a una mayor legibilidad y facilidad de depuración y mantenimiento del código obtenido.
- Los diseños recursivos conllevan normalmente un mayor tiempo de ejecución que los iterativos, además de la complejidad espacial que puede representar el uso de la pila de recursión.

Sin embargo, este tipo de algoritmos también se pueden implementar como un algoritmo no recursivo que almacene las soluciones parciales en una estructura de datos explícita, como puede ser una pila, cola, o cola de prioridad. Esta aproximación da mayor libertad al diseñador, de forma que se pueda escoger qué subproblema es el que se va a resolver a continuación, lo que puede ser importante en el caso de usar técnicas como ramificación y acotación o de optimización.

Recursión

Los algoritmos de “divide y vencerás” están naturalmente implementados, como procesos recursivos. En ese caso, los subproblemas parciales encabezados por aquel que ya ha sido resuelto se almacenan en la pila de llamadas de procedimientos.

Pila explícita

Los algoritmos de divide y vencerás también pueden ser implementados por un programa no recursivo que almacena los subproblemas parciales en alguna estructura de datos explícita, tales como una pila, una cola, o una cola de prioridad. Este enfoque permite más libertad a la hora de elegir los subproblemas a resolver después, una característica que es importante en algunas aplicaciones, por ejemplo en la búsqueda en anchura y en el método de ramificación y poda para optimización de subproblemas. Este enfoque es también la solución estándar en lenguajes de programación que no permiten procedimientos recursivos.

Elección de los casos base

En los algoritmos recursivos hay una libertad considerable para elegir los casos bases, los subproblemas pequeños que son resueltos directamente para acabar con la recursión.

Elegir los casos base más pequeños y simples posibles es más elegante y normalmente nos da lugar a programas más simples, porque hay menos casos a considerar y son más fáciles de resolver. Por ejemplo, el algoritmo quicksort de ordenación podría parar cuando la entrada es una lista vacía. Sólo hay que considerar un caso base, que no se define en términos de si mismo.

Por otra parte, la eficiencia normalmente mejora si la recursión se para en casos relativamente grandes, y estos son resueltos no recursivamente. Esta estrategia evita la sobrecarga de llamadas recursivas que hacen poco o ningún trabajo, y pueden también permitir el uso de algoritmos especializados no recursivos que, para esos casos base, son

más eficientes que la recursión explícita. Ya que un algoritmo de DyV reduce cada instancia del problema o subproblema a un gran número de instancias base, éstas habitualmente dominan el coste general del algoritmo, especialmente cuando la sobrecarga de separación/unión es baja. Es de observar que estas consideraciones no dependen de si la recursión está implementada por compilador o por pila explícita.

Programación dinámica

Concepto

Memorización y tabulación

Concepto

Algoritmos de programación dinámica

Concepto

Algoritmos probabilistas

Un **algoritmo probabilístico** utiliza decisiones aleatorias en su funcionamiento para alcanzar un objetivo, obteniendo resultados correctos en promedio para una amplia variedad de entradas. A diferencia de los algoritmos determinísticos, que siempre producen el mismo resultado para una entrada dada, los algoritmos probabilísticos pueden variar su comportamiento y resultados debido a la introducción de aleatoriedad en su proceso.

Repasemos algunos conceptos básicos de la teoría de la probabilidad.

Combinatoria

Definición (permutación)
Una **permutación** de `n` objetos es una disposición ordenada de los objetos.

Esta definición nos habla del número de formas en que se pueden ordenar `n` objetos. Por ejemplo, si tenemos tres objetos *a, b c*, los podemos ordenar de la siguiente forma: `abc`, *acb, bac, bca, cab, cba*.

La experiencia muestra que el número de permutaciones de *n* objetos es *n! = n * (n - 1) * (n - 2) * ... * 3 * 2 * 1*. Para el caso de los tres objetos `a, b c`, vemos que el número de permutaciones es *3! = 3 * 2 * 1 = 6*.

Definición (combinación)
Una **combinación** de *r* objetos de entre *n* objetos es una selección de *r* objetos sin tener en cuenta el orden.

Por ejemplo, si tenemos los objetos *a, b, c, d*, podemos seleccionar dos objetos de *6* maneras diferentes, siempre y cuando no se tenga en cuenta el orden: *ab, ac, ad, bc, bd, cd*.

Si se tiene en cuenta el orden, el número de formas de seleccionar *r* objetos de *n* es: *n (n - 1) (n - 2) ... (n - r + 1)*.

Si no se tiene en cuenta el orden, el número de formas de seleccionar *r* objetos de entre *n* es[25]:

$C_r^n = \frac{n!}{r!(n-r)!}$; $si\ 0 \leq r \leq n$.

Si *r = 0*, sólo tenemos una forma de no seleccionar ningún objeto, por tanto tomamos: $C_0^n = 1$.

Si *r < 0*, lo cual no tiene sentido en términos combinatorios, se define $C_r^n = 0$.

Si *r > n*, se está diciendo que van a elegir más de n objetos de n disponibles, lo cual es imposible, por tanto $C_r^n = 0$, ya que hay cero formas de hacer dicha elección.

Probabilidad

La teoría de la **probabilidad** se encarga del estudio de fenómenos **aleatorios**, esto es, de analizar eventos en los que su futuro no se puede conocer con certeza. Para aplicar esta teoría, se consideran estos fenómenos como **experimentos aleatorios**, cuyo resultado no se conoce con certeza. Los experimentos pueden ser de muchos tipos, como lanzar una moneda o dado, hasta esperar los resultados que pueda arrojar un sistema informático a partir de unas entradas. Cada dato obtenido es anotado y se denomina **resultado**.

El **espacio muestral** S es el conjunto de todos los resultados posibles de un experimento aleatorio. Cada resultado individual se conoce como **punto muestral** o **suceso elemental**. Un espacio muestral puede ser finito o infinito. El espacio muestral es *discreto* si el número de puntos muestrales (elementos del conjunto) es finito o si se pueden rotular éstos utilizando los número enteros; de lo contrario, diremos que es *continuo*.

Si el experimento consiste en lanzar un dado, el espacio muestral será *S = {1, 2, 3, 4, 5, 6}*, el cual es finito y discreto. Si el experimento aleatorio consiste en medir los tiempos de respuesta de un sistema informático, el espacio muestral es *S = {t | t > 0}*, el cual

[25] En los textos de estadística se encuentran otras notaciones para expresar el número de combinaciones de r elementos tomados de n disponibles.

es continuo. Si el experimento aleatorio consiste en contar las aves que pasan por un cierto lugar en un tiempo determinado, el espacio muestral es $S = \{0, 1, 2, 3, 4, 5, ...\}$, el cual es infinito, pero discreto, ya que puede asignarse un número entero a cada ave.

Un **suceso** es un subconjunto de un espacio muestral. Un suceso A *sucede (ocurre)* si el experimento aleatorio arroja un resultado que se hace parte de A. Por ejemplo un suceso A al lanzar un dado puede ser obtener un número par, por tanto $A = \{2, 4, 6\}$. Decimos que el conjunto vacío $\varnothing$ es un **suceso imposible** y el espacio muestral es un **suceso universal**.

Dado que un espacio muestral es un conjunto y los sucesos subconjuntos, son válidas las operaciones entre ellos, como unión, intersección, etc., por lo que es posible formar nuevos sucesos a partir de éstas. Por ejemplo, decir que el suceso "A no sucede", corresponde a decir "todos los sucesos ocurren excepto el suceso A", que en conjuntos equivale al complemento de A y que representamos así A'[26]. Decir que "sucede A ó B, o ambos", representa la unión entre los sucesos A y B: $A \cup B$. De manera análoga, la frase "sucede A y B" corresponde a la intersección de los sucesos A y B: $A \cap B$. Dos sucesos son **mutuamente excluyentes** si $A \cap B = \varnothing$.

Una **medida de probabilidad** es una función que asigna un valor numérico $Pr[A]$ a todo suceso A del espacio muestral. Ésta debe cumplir estos tres axiomas:

1. Para todo suceso A, $Pr[A] >= 0$
2. $Pr[S] = 1$
3. Si los sucesos A y B son mutuamente excluyentes ($A \cap B = \varnothing$), entonces $Pr[A \cup B] = Pr[A] + Pr[B]$. Este axioma puede extenderse por medio de inducción matemática para n sucesos.

De estos axiomas se deduce:

1. $Pr[A'] = 1 - Pr[A]$, para todo suceso A.
2. $Pr[A \cup B] = Pr[A] + Pr[B] - Pr[A \cap B]$, para todo suceso A y B.

Enfoque básico para resolver problemas de probabilidad

Aunque en la práctica y en los casos reales es posible tomar distintos caminos, los siguientes pasos muestran una guía para abordar problemas relacionados con probabilidad:

1. Identificar el espacio muestral
2. Asignar probabilidades a los elementos del espacio muestral
3. Identificar los sucesos de interés
4. Calcular las probabilidades deseadas

[26] En los textos de matemáticas es posible que encuentre otras notaciones para el complemento de un conjunto, como poner una barra encima del nombre de éste .

Tipos de algoritmos de probabilidad

Los algoritmos probabilísticos se dividen en varias categorías según su método y aplicación. Estos incluyen algoritmos de inferencia bayesianos, que se basan en teoremas bayesianos para actualizar las creencias a la luz de nueva evidencia, y son fundamentales para la clasificación y la predicción. Los métodos Monte Carlo de cadena de Markov (MCMC) son otra categoría utilizada para muestrear distribuciones complejas generando secuencias de números aleatorios. Los algoritmos de regresión logística también son importantes en problemas de clasificación binaria, donde estiman la probabilidad de que una observación caiga en una categoría determinada. Los algoritmos de árboles de decisión, por otro lado, utilizan probabilidades condicionales para dividir los datos en subconjuntos homogéneos. Estos diferentes tipos de algoritmos probabilísticos tienen una variedad de aplicaciones, desde inteligencia artificial hasta ciencia de datos y toma de decisiones.

Algoritmos de regresión logística

La **regresión logística** es una técnica de análisis de datos que utiliza las matemáticas para encontrar las relaciones entre dos factores de datos. Luego, utiliza esta relación para predecir el valor de uno de esos factores basándose en el otro. Normalmente, la predicción tiene un número finito de resultados, como un sí o un no.

Por ejemplo, supongamos que desea adivinar si el visitante de su sitio web va a hacer clic en el botón de pago de su carrito de compras o no. El análisis de regresión logística analiza el comportamiento de los visitantes anteriores, como el tiempo que permanecen en el sitio web y la cantidad de artículos que hay en el carrito. Determina que, si anteriormente los visitantes pasaban más de cinco minutos en el sitio y agregaban más de tres artículos al carrito, hacían clic en el botón de pago. Con esta información, la función de regresión logística puede predecir el comportamiento de un nuevo visitante en el sitio web.

La **regresión logística** implementa un algoritmo de clasificación, específicamente para problemas binarios, que predice la probabilidad de que un evento pertenezca a una de dos categorías. Es un algoritmo estadístico que, a través de una función logística (conocida como *sigmoide*), modela la relación entre las variables independientes (o predictores) y la variable dependiente binaria.

La regresión logística es una técnica importante en el campo de la inteligencia artificial y el machine learning (AI/ML). Los modelos de ML son programas de software que se pueden entrenar para realizar tareas complejas de procesamiento de datos sin intervención humana. Los modelos de ML creados mediante regresión logística ayudan a las organizaciones a obtener información procesable a partir de sus datos empresariales. Pueden usar esta información para el análisis predictivo a fin de reducir los costos operativos, aumentar la eficiencia y escalar más rápido. Por ejemplo, las empresas pueden descubrir patrones que mejoran la retención de los empleados o conducen a un diseño de productos más rentable.

Proceso general de la regresión logística

1. Definición del problema: se identifica la variable dependiente binaria que se quiere predecir (por ejemplo: sí/no, 0/1) y las variables independientes que se creen que influyen en ella.
2. Recopilación y preparación de datos: se recopilan datos históricos y se preparan para el análisis, incluyendo limpieza, preprocesamiento y transformación de variables.
3. Modelo de regresión: se construye el modelo de regresión logística, que utiliza la función logística para modelar la relación entre las variables independientes y la probabilidad de la variable dependiente.
4. Ajuste del modelo: se ajustan los parámetros del modelo (pesos o coeficientes) mediante un proceso de optimización, generalmente utilizando métodos de máxima verosimilitud, para minimizar la diferencia entre los valores predichos y los reales.
5. Evaluación del modelo: se evalúa la precisión del modelo utilizando métricas como la precisión, la sensibilidad, la especificidad, el área bajo la curva ROC (AUC) y otras.
6. Predicción: una vez ajustado y evaluado, el modelo puede utilizarse para predecir la probabilidad de que un nuevo evento pertenezca a una de las dos categorías.

Ejemplos de aplicaciones

Hay una variada gama de aplicaciones de la regresión logística, algunas de ellas son:

- Predicción de spam: determinar si un correo electrónico es spam o no spam.
- Diagnóstico médico: predecir si un paciente tiene o no una enfermedad específica.
- Detección de fraude: identificar transacciones fraudulentas.
- Marketing: predecir si un cliente va a responder a una campaña de marketing.
- Fabricación: estimar la probabilidad de fallo de las piezas en la maquinaria.
- Finanzas: analizar las transacciones financieras en busca de fraudes y evaluar las solicitudes de préstamos y seguros en busca de riesgos.
- Transporte: analizar las distintas situaciones relacionadas con el transporte, tales como accidentalidad, velocidad de flujo en horas pico, etc.

Algoritmos de árboles de decisión

Un **árbol de decisión** es un algoritmo de aprendizaje supervisado no paramétrico, que se utiliza tanto para tareas de clasificación como de regresión. Tiene una estructura jerárquica de árbol, que consta de un nodo raíz, ramas, nodos internos y nodos terminales u hojas.

Un árbol de decisión tiene la estructura de un árbol binario y el aprendizaje en ellos emplea una estrategia de divide y vencerás realizando una búsqueda codiciosa para identificar los puntos de división óptimos dentro de un árbol. Este proceso de división se repite de forma descendente y recursiva hasta que todos o la mayoría de los registros se hayan clasificado con etiquetas de clase específicas.

El hecho de que todos los puntos de datos se clasifiquen como conjuntos homogéneos depende en gran medida de la complejidad del árbol de decisión. Los árboles más pequeños tienen más facilidad para alcanzar nodos de hojas puras, es decir, puntos de datos en una sola clase. Sin embargo, a medida que un árbol crece en tamaño, se vuelve

cada vez más difícil mantener esta pureza y, por lo general, da como resultado que caigan muy pocos datos dentro de un subárbol determinado. Cuando esto ocurre, se conoce como fragmentación de datos y, a menudo, puede conducir a un sobreajuste.

Como resultado, los árboles de decisión tienen preferencia por los árboles pequeños, lo que es coherente con el principio de parsimonia de la Navaja de Occam; es decir, "las entidades no deben multiplicarse más allá de lo necesario". Dicho de otra manera, los árboles de decisión deben agregar complejidad solo si es necesario, ya que la explicación más simple suele ser la mejor. Para reducir la complejidad y evitar el sobreajuste, se suele emplear la poda; se trata de un proceso que elimina las ramas que se dividen en características con poca importancia. A continuación, se puede evaluar el ajuste del modelo mediante el proceso de validación cruzada.

Otra forma en que los árboles de decisión pueden mantener su precisión es formando un conjunto mediante un algoritmo bosque aleatorio; este clasificador predice resultados más precisos, sobre todo cuando los árboles individuales no están correlacionados entre sí.

Tipos de árboles de decisión

El algoritmo de Hunt, desarrollado en la década de 1960 para modelar el aprendizaje humano en Psicología, es la base de muchos algoritmos populares de árboles de decisión, como los siguientes:

- ID3: a Ross Quinlan se le atribuye el desarrollo de ID3, que es la abreviatura de "Iterative Dichotomiser 3". Este algoritmo aprovecha la entropía y la ganancia de información como métricas para evaluar las divisiones de los candidatos.

- C4.5: este algoritmo se considera una iteración posterior de ID3, que también fue desarrollado por Quinlan. Puede utilizar la ganancia de información o los ratios de ganancia para evaluar los puntos de división dentro de los árboles de decisión.

- CART: el término CART es una abreviatura de "árboles de clasificación y regresión" y fue introducido por Leo Breiman. Este algoritmo suele utilizar la impureza de Gini para identificar el atributo ideal para dividir. La impureza de Gini mide la frecuencia con la que se clasifica erróneamente un atributo elegido al azar. Al evaluar utilizando la impureza de Gini, un valor más bajo es más ideal.

Ventajas y desventajas de los árboles de decisión

Aunque los árboles de decisión se pueden utilizar en una variedad de casos de uso, otros algoritmos generalmente superan a los algoritmos de árboles de decisión. Los árboles de decisión son particularmente útiles para las tareas de minería de datos y descubrimiento de conocimiento.

Ventajas

- Fácil de interpretar: la lógica booleana y las representaciones visuales de los árboles de decisión facilitan su comprensión y consumo. La naturaleza jerárquica de un árbol de decisión también hace que sea fácil ver qué atributos son los más importantes, lo que no siempre está claro con otros algoritmos, como las redes neuronales.

- Requiere poca o ninguna preparación de datos: los árboles de decisión tienen una serie de características que los hacen más flexibles que otros clasificadores. Puede manejar varios tipos de datos, es decir, valores discretos o continuos, y los valores continuos se pueden convertir en valores categóricos mediante el uso de umbrales. Además, también puede manejar valores con valores faltantes, lo que puede ser problemático para otros clasificadores, como Naive Bayes.

- Más flexibles: los árboles de decisión pueden utilizarse tanto para tareas de clasificación como de regresión, lo que los hace más flexibles que otros algoritmos. También es insensible a las relaciones subyacentes entre atributos; esto significa que si dos variables están muy correlacionadas, el algoritmo sólo elegirá una de las características para dividir.

Desventajas

- Propenso al sobreajuste: los árboles de decisión complejos tienden a sobreajustarse y no se generalizan bien a los nuevos datos. Este escenario se puede evitar mediante los procesos de poda previa o posterior a la poda. La poda previa detiene el crecimiento de los árboles cuando no hay datos suficientes, mientras que la poda elimina los subárboles con datos inadecuados tras la construcción del árbol.

- Estimadores de alta varianza: pequeñas variaciones dentro de los datos pueden producir un árbol de decisión muy diferente. Embolsado, o el promedio de estimaciones, puede ser un método para reducir la varianza de árboles de decisión. Sin embargo, este enfoque es limitado, ya que puede conducir a predictores altamente correlacionados. Es fácil que el árbol se degenere con cierto datos ingresados.

- Más costoso: dado que los árboles de decisión adoptan un enfoque de búsqueda codicioso durante la construcción, su entrenamiento puede resultar más costoso en comparación con otros algoritmos.

Tratamiento de la probabilidad en los lenguajes de programación

Los lenguajes de programación disponen a su vez de funciones para generar números **pseudoaleatorios**; esto es realizado por un algoritmo basado en **reglas matemáticas**, además de una **semilla**, la cual es utilizada por dichas reglas para generar los números. Actualmente los lenguajes tienen la semilla implícita en la funciones de generación, generalmente asociada al tiempo del sistema, por lo que no es necesario especificarla, aunque en algunos lenguajes se puede establecer de forma explícita. El nombre de pseudoaleatorios proviene de que son números que se generan a partir de un algoritmo con unas reglas matemáticas, pero que en realidad no son aleatorios, y es ahí donde entra en juego y se manifiesta la importancia de la semilla, ya que ésta se encarga de inicializar el algoritmo generador. Una semilla constante, siempre generará la misma secuencia de números pseudoaleatorios, por lo que tomar el tiempo como semilla es una opción apropiada, ya que éste cambia siempre, y así el algoritmo basará sus cálculos en valores iniciales diferentes.

Algunos lenguajes tienen funciones que generan un número pseudoaleatorio entero comprendido en un intervalo (`[límite_inferior, límite_superior]`); algunos de estos lenguajes permiten que el segundo parámetro sea opcional, siendo el límite superior igual en este caso al mayor entero disponible en el lenguaje. Otros lenguajes entregan un número real en el intervalo `[0, 1]`.

La clase Math de Java dispone de un método para generar números pseudo aleatorios: **`random`**, el cual devuelve un real entre 0 y 1, con una aproximación de 17 cifras significativas. Por ejemplo: *`Math.random()`* puede devolver algo como `0.28737539213034247` ó `0.03293366594495917`. Si queremos obtener un entero comprendido en un intervalo, por ejemplo `[1, 10]`, podemos hacer lo siguiente, tal y como se ilustra en el siguiente ejemplo.

Ejemplo

Si queremos obtener en Java un entero comprendido en un intervalo, por ejemplo entre 1 y 10 `[1, 10]`, podemos hacer la siguiente operación aritmética usando `Math.random()`.

```
class Maths
{
    public static void main(String[] args)
    {
        double randNumb = Math.random() * 10 + 1;
        System.out.println("Number random:" + randNumb);
        System.out.println("Number random int:" + (int) randNumb);
    }
}
```

Nota

Observe que la fórmula general para obtener números enteros en Java en un intervalo cualquiera se establece así:

```
(int) (Math.random() * (límSuperior - límInferior + 1) + límInferior).
```

Por tanto, podemos reescribir nuestro programa anterior para generar un número aleatorio usando una función parametrizada, así:

```
public class Maths
{
    public int randomNumber(int minLim, int maxLim)
    {
        double randNumb = Math.random() * (maxLim - minLim + 1) + minLim;
        return (int) randNumb;
    }
}
```

Podemos llamar la función anterior generando `n` (en este caso `n = 10`) números aleatorios, así:

```
public static void main(String[] args)
{
    Maths mt = new Maths();
    for (int i = 1; i <= 10; i++) {
        System.out.println("Number random int:" + mt.randomNumber(1, 10));
    }
}
```

Java también dispone de la clase **Random** para trabajar con números pseudoaleatorios generados en un rango determinado, como lo veremos en un ejemplo más adelante.

Asignar una probabilidad mayor a un dato de un conjunto

Es posible que en un fenómeno distintos eventos tengan distintas probabilidades de suceder, como por ejemplo que en un dado cargado uno de los lados caiga con mayor probabilidad que los demás. En los lenguajes de programación se puede simular esto aplicado tanto a valores individuales como a listas, que serán las que definen el universo de eventos o sucesos.

Los pasos para realizar esto pueden ser los siguientes:

- Generar un número pseudoaleatorio: usando alguna de las funciones disponibles en los lenguajes o usando un generador congruencial.
- Definir un rango de probabilidades: determinar el rango de números a usar para representar las probabilidades. Por ejemplo, en el rango de 1 a 10, el 100% equivale a 10.
- Asignar probabilidades a los datos: asignar a cada dato un rango de números que representan su probabilidad. Por ejemplo, un dato que tenga un 60% de probabilidad, se le puede asignar el rango del 0 al 7.
- Comparar con el número aleatorio: si el número aleatorio cae dentro del rango de probabilidad asignado a un dato, entonces ese dato es elegido.

Ejemplo

Sea U = {A, B}; la probabilidad de que suceda A es del 70% y la de B es del 30%. Simular esta situación en Java generando un número aleatorio entre uno y diez usando la clase Random.

El método `nextInt` de la clase `Random` puede ser invocado sin parámetros (`Random.nextInt()`), con lo cual se genera un número pseudoaleatorio comprendido en el rango de los enteros, incluidos tanto valores negativos como positivos. Si se especifica un argumento entero (`Random.nextInt(n)`), se genera entonces un número pseudoaleatorio entre cero y el número dado menos uno (`[0, n - 1]`); por ejemplo, si `n = 10`, entonces el método `nextInt` generará pseudoaleatorios entre 0 y 9.

```
import java.util.Random;
import java.util.Scanner;

class Main
{
    public static void main(String[] args)
    {
        Random random = new Random();
        Scanner sc = new Scanner(System.in);

        // Definir las probabilidades de cada dato
        int probA = 7; // 70% de probabilidad
        int probB = 3; // 30% de probabilidad

        // Generar eventos con distinta probabilidad de ocurrencia
        String op;
        do {
            int numRandom = random.nextInt(10);

            // Determinar qué dato se elige
            if (numRandom < probA) {
                    System.out.println("Dato seleccionado al azar:
A");
            } else {
                    System.out.println("Dato seleccionado al azar:
B");
            }
            System.out.println("Seguir [s/?]");
            op = sc.next().toLowerCase();
        } while (op.equals("s"));
    }
}
```

Python por su lado incluye una librería llamada **`random`** que dispone de una serie de métodos para trabajar con números pseudo aleatorios[27]. Por ejemplo, para generar un aleatorio (entero) entre 1 y 10, indicamos el método randint de la clase random especificando el intervalo respectivo por medio de dos parámetros: `random.randint(1, 10)`.

Ejemplo

Otros métodos interesantes de esta clase son los que operan sobre secuencias: **`choice`**, el cual elige aleatoriamente un elemento de una secuencia; **`shuffle`**, que devuelve una lista con los elementos distribuidos aleatoriamente:

```
import random
```

[27] Puede consultar más acerca de esta librería en: Generate pseudo-random numbers — Python 3.13.1 documentation y en Python Random Module

```
print(f"Número aleatorio entre 1 y 10: {random.randint(1, 10)}")
data_list = [0, 5, 4, 8]
print(f"Lista: {data_list}")
print(f"choice: {random.choice(data_list)}")
random.shuffle(data_list)
print(f"Lista shuffle: {data_list}")
```

Generación de números aleatorios: generadores congruenciales

Un número pseudoaleatorio representa la probabilidad de un suceso cualquiera y son la base de la simulación. Los números son generados suponiendo que las variables aleatorias son independientes e idénticamente distribuidas (i.i.d.) $U(0, 1)$.

Los principales generadores de números pseudo-aleatorios utilizados hoy en día son los llamados **generadores congruenciales lineales**, introducidos por Lehmer en 1951. Un método congruencial comienza con un valor inicial (**semilla**) x_0, y los sucesivos valores x_n, $n >= 1$ se obtienen recursivamente así:

$$x_n = (ax_{n-1} + b) \% m$$

Donde a, b, m son enteros positivos conocidos como el *multiplicador*, el *incremento* y el *módulo*, respectivamente.

La sucesión de números pseudoaleatorios U_n $(n >= 1)$ se obtienen así:

$$U_i = x_i / m$$

Ejemplo

Generadores congruenciales lineales. Este ejemplo muestra una forma de generar números pseudoaleatorios en Python con un método congruencial simple.

```
def pseudo_random(n, x0, a, b, m) -> None:
    u = []
    x = []
    seed = x0
    for i in range(1, n, 1):
        seed = (a * seed + b) % m
        x.append(seed)
        u.append(seed / m)
    print(x)
    print(u)

pseudo_random(5, 50, 7 ** 5, 0, 2 ** 31 -1)
```

Al ser la semilla constante, la secuencia de números se repite cada ejecutamos el programa, pero si indicamos una semilla apropiada que varíe, podemos obtener una

secuencia distinta cada vez en cada ejecución, simulando adecuadamente los números pseudoaleatorios. Es claro que un dato que siempre cambia es el tiempo, por lo que podemos usarlo como semilla. Veamos cómo queda la solución indicando como semilla al **timestamp** del sistema, el cual devuelve un entero que indica el tiempo del sistema.

```
from datetime import datetime

def pseudo_random(n, x0, a, b, m) -> None:
    u = []
    x = []
    seed = x0
    for i in range(1, n + 1, 1):
        seed= (a * seed + b) % m
        x.append(seed)
        u.append(x[i - 1] / m)
    print(x)
    print(u)

now = datetime.now()
pseudo_random(5, datetime.timestamp(now), 7 ** 5, 0, 2 ** 31 - 1)
```

Una adecuada elección de los parámetros permite obtener números aparentemente pseudoaleatorios de manera eficiente. Park y Miller (1988) propusieron en el ***minimal standard*** los siguientes valores para las constantes: $a = 7^5, b = 0, m = 2^{31} - 1$.

Algoritmos voraces

En informática, un **algoritmo voraz** (también llamado ávido, codicioso, devorador) es aquel que encuentra la solución a un problema en el menor tiempo posible. Elige la ruta que parece óptima en ese momento, sin tener en cuenta la optimización global de la solución resultante.

Edsger Dijkstra, informático y matemático que deseaba calcular un árbol de expansión mínima, introdujo el término "algoritmo voraz". Prim y Kruskal desarrollaron técnicas de optimización para minimizar el coste de los grafos.

Algoritmos codiciosos frente a algoritmos no codiciosos

Un algoritmo es voraz cuando la ruta elegida se considera la mejor opción según un criterio específico, sin tener en cuenta las consecuencias futuras. Sin embargo, normalmente evalúa la viabilidad antes de tomar una decisión final. La corrección de la solución depende del problema y de los criterios utilizados.

Por ejemplo, en un grafo que tiene distintos valores, se debe determinar el valor máximo en éste. Se empezaría buscando en cada nodo y comprobando su peso para ver si es el valor más alto.

Existen dos enfoques para resolver este problema: un enfoque voraz y un enfoque no voraz.

En el enfoque codicioso el algoritmo se detiene una vez que obtiene una solución óptima, sin ser necesariamente la mejor, y al ejecutarse una vez, no tiene opción de realizar correcciones; mientras que en el enfoque no codicioso se revisan todas las opciones antes de concluir el resultado final.

Características de un algoritmo voraz

- El algoritmo resuelve el problema encontrando una solución óptima. Esta solución puede ser un valor máximo o mínimo. Toma decisiones basándose en la mejor opción disponible.
- El algoritmo es rápido y eficiente, con una complejidad temporal de O(nlog n) u O(n). Por lo tanto, se aplica en la resolución de problemas a gran escala.
- La búsqueda de la solución óptima se realiza sin repetición: el algoritmo se ejecuta una sola vez.
- Es sencillo y fácil de implementar.

Los algoritmos voraces emplean un procedimiento de resolución de problemas para construir progresivamente soluciones candidatas, aproximando el óptimo global mediante la obtención de soluciones óptimas locales cada vez mejores en cada etapa. En general, los algoritmos voraces no pueden generar una solución óptima global , pero sí pueden producir buenas soluciones óptimas locales en un tiempo razonable y con menor esfuerzo computacional . GRASP (Feo y Resende, 1989) es un conocido algoritmo voraz iterativo basado en búsqueda local que implica varias iteraciones para construir soluciones voraces aleatorias y mejorarlas sucesivamente. El algoritmo consta de dos etapas principales: construcción y búsqueda local. La primera etapa consiste en construir una solución y la segunda en mejorarla para lograr la factibilidad. El algoritmo genera soluciones voraces aleatorias seleccionando nuevos elementos de un conjunto de soluciones candidatas ya construidas, basándose en el grado de mejora de la solución parcial en construcción, mediante una función de evaluación voraz. Los elementos seleccionados se incorporan a la solución parcial actual sin comprometer la factibilidad (si esta se ve comprometida, se selecciona un nuevo elemento), hasta obtener una solución factible completa , cuyo entorno se explora hasta encontrar un óptimo local mediante el procedimiento de búsqueda local. Se genera una lista, denominada lista de candidatos restringidos, con los mejores elementos, mediante la evaluación de los elementos con la función de evaluación voraz. Finalmente, se elige un elemento aleatoriamente de la lista restringida. Se crea una lista de candidatos para incorporarlo a la solución parcial, y una vez que el nuevo elemento se incluye en la solución parcial, el conjunto de candidatos de soluciones voraces se actualiza junto con la función de evaluación voraz.

¿Cómo funcionan los algoritmos voraces?

Los algoritmos voraces resuelven problemas siguiendo un enfoque simple, paso a paso:

1. Identificar el problema: El primer paso es comprender el problema y determinar si se puede resolver mediante un enfoque voraz. Esto suele implicar reconocer que el problema se puede descomponer en una serie de decisiones.
2. Tomar la decisión óptima: En cada paso del problema, el algoritmo toma la mejor decisión que parece óptima en ese momento. Esta decisión se basa en la propiedad de elección óptima, lo que significa seleccionar la opción que ofrece el beneficio más inmediato.
3. Resolver los subproblemas: Tras realizar una elección voraz, el problema se reduce a un subproblema más pequeño. El algoritmo repite entonces el proceso, realizando otra elección voraz para el problema más pequeño.
4. Combinar las soluciones: El algoritmo continúa tomando decisiones ávidas y resolviendo subproblemas hasta que se resuelve el problema completo. La solución al problema global es simplemente la combinación de todas las decisiones ávidas tomadas en cada paso.

Algoritmos voraces comunes

Los algoritmos ávidos más conocidos son:

- El problema de selección de actividades consiste en seleccionar el número máximo de actividades que no se solapen, dados sus horarios de inicio y finalización. El objetivo es maximizar el número de actividades a las que se puede asistir.
- El árbol de expansión mínima (MST) utiliza el algoritmo de Prim; no contiene ciclos y tiene el peso total mínimo posible en sus aristas. Este árbol se deriva de un grafo no dirigido conexo con pesos asignados.
- El algoritmo de Dijkstra para encontrar el camino más corto es un algoritmo de búsqueda que encuentra el camino más corto entre un vértice y otros vértices en un grafo ponderado.
- El problema del viajante consiste en encontrar la ruta más corta que visite cada lugar solo una vez y regrese al punto de partida.
- La codificación Huffman asigna un código más corto a los símbolos que aparecen con frecuencia y un código más largo a los que aparecen con menos frecuencia. Se utiliza para codificar datos de forma eficiente.

Tipos de algoritmos voraces

Algoritmos voraces fraccionarios

Estos algoritmos funcionan tomando la mejor decisión posible en cada paso basándose en fracciones. Se suelen utilizar en problemas donde los elementos se pueden dividir en partes más pequeñas.

Ejemplo: Problema de la mochila fraccionada, cuyo objetivo es maximizar el valor total de la mochila tomando fracciones de los artículos si es necesario.

Algoritmos puramente voraces

Estos algoritmos toman la mejor decisión posible en cada paso sin considerar las consecuencias futuras ni la posibilidad de revisar decisiones anteriores. Una vez tomada la decisión, es definitiva.

Ejemplo: Problema de selección de actividades, en donde las actividades se seleccionan en función de sus tiempos de finalización para maximizar el número de actividades que no se superponen.

Algoritmos voraces recursivos

Estos algoritmos utilizan la recursión para realizar una serie de elecciones voraces. El problema se divide en subproblemas, y el algoritmo realiza una elección voraz y luego resuelve recursivamente el subproblema.

Ejemplo: Algoritmo de Prim, que construye un árbol de expansión mínima tomando la decisión voraz de agregar la arista más pequeña en cada paso y continuando recursivamente este proceso.

Algoritmos de elección voraces

Estos algoritmos se basan en la propiedad de elección voraz, que garantiza que se puede alcanzar una solución óptima global tomando decisiones óptimas locales.

Ejemplo: Codificación Huffman, donde a los caracteres se les asignan códigos de longitud variable en función de sus frecuencias, con el objetivo de minimizar la longitud total del mensaje codificado.

Algoritmos voraces adaptativos

Estos algoritmos adaptan sus estrategias en función del estado actual del problema. Pueden reconsiderar decisiones anteriores si es necesario para mejorar la solución global.

Ejemplo: Coloreado de grafos mediante un algoritmo voraz; aquí el algoritmo asigna colores a los vértices de un grafo, ajustando potencialmente las elecciones a medida que se colorean nuevos vértices.

Algoritmos voraces constructivos

Estos algoritmos construyen la solución paso a paso, añadiendo elementos al conjunto de soluciones en función de un criterio específico.

Ejemplo: Algoritmo de Kruskal, que construye un árbol de expansión mínima añadiendo las aristas más cortas que no forman un ciclo.

Algoritmos voraces no adaptativos

Estos algoritmos toman decisiones definitivas que no se adaptan ni cambian una vez tomada la decisión.

Ejemplo: Algoritmo de Dijkstra – que encuentra el camino más corto desde un origen a todos los demás vértices de un grafo tomando decisiones localmente óptimas en cada paso.

Complejidad temporal de los algoritmos voraces

La complejidad temporal de los algoritmos voraces comunes es mayoritariamente O(nlog n) u O(n).

Algoritmo	Complejidad temporal	Complejidad espacial
Problema de selección de actividades	O(nlog(n))	O(1)
Codificación Huffman	O(nlog(n))	O(n)
Problema de la mochila fraccionaria	O(nlog(n))	O(1)
Problema con el cambio de monedas	O(n)	O(1)
Secuenciación de tareas con plazos de entrega	O(nlog(n))	O(n)

Aplicaciones de los algoritmos voraces

- **Enrutamiento de red**: Se utilizan algoritmos voraces como el algoritmo de Dijkstra para encontrar la ruta más corta en el enrutamiento de red, optimizando la entrega de paquetes de datos a través de Internet.
- **Planificación de tareas**: Se utiliza en problemas de secuenciación de trabajos para maximizar las ganancias o minimizar los plazos de entrega mediante la selección de trabajos en orden de prioridad.
- **Asignación de recursos**: Se aplica en escenarios como el problema de la mochila fraccionada para asignar recursos de manera eficiente en función de las relaciones valor-peso.
- **Compresión de datos**: La codificación Huffman utiliza un enfoque voraz para comprimir los datos de manera eficiente mediante la codificación de los caracteres de uso frecuente con códigos más cortos.
- **Cambio de monedas**: A menudo se utiliza un enfoque codicioso para proporcionar el número mínimo de monedas por una cantidad determinada de dinero,

especialmente cuando las denominaciones son estándar (como en la mayoría de las monedas).
- **Coloración voraz**: Se aplica en teoría de grafos para problemas como la coloración de grafos, donde el objetivo es minimizar el número de colores necesarios para colorear un grafo asegurando al mismo tiempo que ningún par de vértices adyacentes compartan el mismo color.
- **Heurística para el Problema del Viajante (TSP)**: Se utilizan heurísticas voraces para aproximar las soluciones al TSP seleccionando la ciudad no visitada más cercana en cada paso.

Ventajas de los algoritmos voraces

- **Sencillez**: Los algoritmos voraces son fáciles de entender e implementar porque siguen un enfoque directo para tomar la mejor decisión en cada paso.
- **Eficacia**: Suelen tener una complejidad temporal menor en comparación con otros algoritmos como la programación dinámica, lo que los hace más rápidos para ciertos problemas.
- **Decisiones óptimas locales**: Al tomar decisiones óptimas locales, los algoritmos voraces pueden llegar rápidamente a una solución sin necesidad de un retroceso complejo.
- **Funciona bien para ciertos problemas**: Los algoritmos voraces son particularmente efectivos para problemas que exhiben la propiedad de elección voraz y subestructura óptima, como el árbol de expansión mínima y la codificación Huffman.

Desventajas de los algoritmos voraces

- **Puede que no siempre produzcan soluciones óptimas**: Los algoritmos voraces se centran en los beneficios inmediatos, lo que a veces puede conducir a soluciones subóptimas para problemas en los que se necesita una perspectiva global.
- **Aplicabilidad limitada**: Los algoritmos voraces solo funcionan bien para problemas que satisfacen la propiedad de elección voraz y la subestructura óptima; de lo contrario, pueden fallar o dar resultados incorrectos.
- **Sin posibilidad de rectificar**: Una vez tomada una decisión, no se puede deshacer, lo que puede conducir a resultados incorrectos si se toma la decisión equivocada al principio.
- **Falta de flexibilidad**: Los algoritmos voraces son rígidos en su enfoque, y modificarlos para tener en cuenta diferentes restricciones del problema puede resultar complicado.

Preguntas

1. ¿Qué es un árbol?
2. ¿Qué es un árbol binario?

3. ¿Qué es un árbol completo?
4. ¿Qué es un árbol AVL?
5. ¿Qué es un grafo?

Ejercicios

1. Convertir a notación polaca prefija y postfija las siguientes expresiones escritas en notación infija.
    a. x ^ y + z * w
    b. x - (w / z) * y
    c. a / (a - b) * (c + b)
    d. m / (n - q) ^ p
    e. a * (b + (x * y)) ^ c
    f. z ^ (x + y + z) / w
    g. (a + b) - c * (d + e ^ x)
    h. a + b + c + d + e
    i. (x + y + z) ^ e / a + (b * c)
    j. ((m - n) * (p + q)) ^ r / s
2. Escriba funciones para convertir expresiones escritas en notación infija a notación polaca prefija y postfija, respectivamente.
3. Problema sobre Colas. Una institución educativa universitaria requiere para el departamento de Admisiones y Registro un control básico de las personas que atiende en el día por ventanilla. Las personas se atienden una a una a medida que van llegando, y se guarda de ellas el documento, el nombre y por defecto se establece que no ha sido atendida. A medida que llegan personas, se ubican en la fila y van saliendo de ésta una vez son atendidas.
    a. Cree un menú de opciones para gestionar la atención de personas de acuerdo con los siguientes literales. Adicione una opción para finalizar.
    b. Registrar el ingreso a la fila de una persona
    c. Mostrar la fila de personas a atender
    d. Cuántas personas hay en espera
    e. Atender personas
4. Problema sobre Pilas. Una tienda vende dos tipos de productos (ratón -mouse- y teclados) que llegan en cajas y los organizan en estanterías; de cada producto se conoce su código, nombre y precio, y ambos tipos de productos se organizan de forma independiente de acuerdo con su tipo.
    a. Cree un menú de opciones para gestionar el movimiento del inventario de acuerdo con los siguientes literales. Adicione una opción para finalizar.
    b. Acomode productos en las estanterías de acuerdo con su tipo registrando su ingreso al inventario
    c. Liste los productos disponibles de acuerdo con el tipo seleccionado con todos sus datos e indique cuántos hay en dicha estantería
    d. Muestre el producto de mayor valor de acuerdo a su tipo
    e. Aliste productos para la venta. Los productos que se alistan para la venta salen del inventario; el usuario debe tener la posibilidad de indicar el tipo de producto a alistar y cuántos productos va a alistar

5. Recorrer listas LSL, LSLC, LDL y LDLC de forma recursiva
6. La *función de Ackermann*[28] se define como:
   $A(m,n) = n + 1,\ si\ m = 0$
   $A(m,n) = A(m-1,1),\ si\ n = 0$
   $A(m,n) = A(m-1, A(m,n-1)),\ en\ otro\ caso$
   Esta función toma dos números enteros positivos como argumentos y devuelve un único entero positivo. Escriba una función recursiva para calcular la función de Ackermann $A(M,N)\ para\ M,N \geq 0$
7. El *algoritmo de Euclides* para el cálculo del *Máximo Común Divisor* (*MCD*) se define de la siguiente forma:
   $MCD(m,n) = m,\ si\ n = 0$
   $MCD(m,n) = MCD(n, m\ MOD\ n),\ si\ n > 0$
   Escriba una función recursiva para calcular $MCD(m,n)\ para\ m,n \geq 0$
8. Invierta una palabra de forma recursiva
9. Escriba funciones no recursivas para solucionar los problemas 6, 7 y 8
10. Muestre el cuadrado de los primeros n números naturales
11. Muestre la suma de los cuadrados de los primeros n números naturales
12. Escriba un programa para simular el problema de *Las Torres de Hanoi*[29]. Resuelva de forma iterativa y recursiva. Este problema fue ideado por el matemático francés Edouard Lucas en el año de 1883 inspirado en una leyenda hindú. El problema consiste en resolver lo siguiente: se tienen tres torres a las que llamamos *origen*, *destino* y *auxiliar*; en la torre de origen se encuentran n discos todos de distintos tamaños, ordenados de mayor a menor tamaño desde la base de la torre; se deben pasar los discos a la torre de destino usando como apoyo la torre auxiliar con las siguientes condiciones: 1) solo puede pasarse un disco a la vez; 2) no puede quedar un disco de mayor tamaño sobre uno de menor tamaño. Después de muchas pruebas y de analizar este problema, se ha encontrado que el número de movimientos a efectuar es $2^n - 1$.

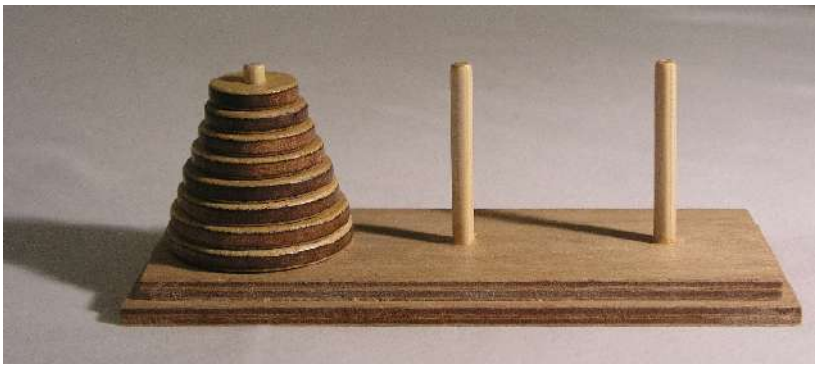


Figura 3.4. Las Torres de Hanoi como juego infantil[30]

[28] Esta función se debe al matemático alemán Wilhelm Ackermann y tiene interés en las ciencias de la computación. En Función de Ackermann - Wikipedia, la enciclopedia libre se habla más sobre esta función, así como en otras fuentes de literatura escrita y digital.
[29] Puede encontrar más detalles sobre el curioso problema de Las torres de Hanoi en distinta literatura, tanto de matemáticas como de computación. La siguiente fuente habla un poco acerca de la leyenda hindú y una solución usando Python
4.10. Las torres de Hanoi — Solución de problemas con algoritmos y estructuras de datos
[30] Imagen tomada de: Torres de Hanói - Wikipedia, la enciclopedia libre

Capítulo 5. Algoritmos heurísticos y metaheurísticos

Concepto

Problemas de optimización

Concepto

Problemas de programación lineal entera

Concepto

Herramientas de optimización

Concepto

Solver

Concepto

OR-Tools

Concepto

Algoritmos heurísticos y metaheurísticos o aproximados

Concepto

Métricas de eficacia de algoritmos aproximados

Concepto

Preguntas

Ejercicios

Fuentes y referencias adicionales

Bibliografía

Algoritmia y programación

Aho, A. (1995). Foundations of computer science. Computer science press.

Cormen, T.; Lieserson, C.; Rivest, R. (2009). Introduction to algorithms. Ed. the mit press

Flórez, R. (2005) Algoritmos, estructuras de datos y programación orientada a objetos. Bogotá, ecoe ediciones.

Skiena, S. (2008). The algorithm design manual. Springer; 2nd edition.

Weiss, M. (1992). Estructuras de datos y algoritmos. Addison – Wesley Iberoamericana. wilmington, delaware, e.u.a.

JOYANES AGUILAR, LUIS. Fundamentos de Programación, Algoritmos y Estructuras de Datos. Segunda edición. McGraw-Hill/Interamericana de España. Madrid, 1996.

CAIRÓ, OSVALDO; GUARDATI BUEMO, SILVIA. Estructuras de Datos. McGraw-Hill. México DF, 1996.

AHO, ALFRED V., HOPCROFT, JOHN E., ULLMAN, JEFFREY D. Estructuras de datos y algoritmos. Addison-Wesley Iberoamericana. Wilmington, Delaware, E.U.A., 1988.

FLÓREZ R., ROBERTO. Algoritmos y Estructuras de Datos. Universidad de Antioquia. Medellín, 1996.

BRASSARD, G., BRATLEY, T. Fundamentos de Algoritmia. Prentice Hall. Madrid, 1996.

Matemáticas

Leithold, Louis. El Cálculo con Geometría Analítica. Sexta edición. Harla. México D.F., 1992.

Referencias en Internet

Algoritmia y programación

Árboles Balanceados AVL. Universidad Don Bosco, Facultad de Ingeniería, Escuela de Computación, Asignatura Programación con Estructuras de Datos
En Internet (en pdf)
Tema: “Árboles Balanceados AVL”

Generación de números aleatorios
Tema 1 Generación de números aleatorios

Generadores congruenciales lineales. Técnicas de Simulación y Remuestreo
2.1 Generadores congruenciales lineales | Técnicas de Simulación y Remuestreo

Ciencias de la computación
Ciencias de la computación | Khan Academy

Notación asintótica
Notación asintótica (artículo) | Algoritmos | Khan Academy

Clases de complejidad P y NP
Clases de complejidad P y NP - Wikipedia, la enciclopedia libre

Búsqueda de fuerza bruta
Búsqueda de fuerza bruta - Wikipedia, la enciclopedia libre

Ataque de fuerza bruta
Ataque de fuerza bruta - Wikipedia, la enciclopedia libre

Algoritmo de divide y vencerás
https://en-m-wikipedia-org.translate.goog/wiki/Divide-and-conquer_algorithm?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=sge#:~:text=A%20divide%2Dand%2Dconquer%20algorithm,solution%20to%20the%20original%20problem.

Introducción al Algoritmo de Ordenación Merge Sort
Introducción al Algoritmo de Ordenación Merge Sort - Blog de SW Hosting

Algoritmo divide y vencerás
Algoritmo divide y vencerás - Wikipedia, la enciclopedia libre

Análisis de algoritmos recursivos.
Tema 06 "Análisis de algoritmos recursivos"

Teoría de grafos
Teoría de grafos

¿Qué es la regresión logística?
¿Qué es la regresión logística? - Explicación del modelo de regresión logística - AWS

¿Qué es un árbol de decisión?
¿Qué es un árbol de decisión? | IBM.

What is a Greedy Algorithm? Examples of Greedy Algorithms
What is a Greedy Algorithm? Examples of Greedy Algorithms

Greedy Algorithms: Examples, Types, Complexity
Greedy Algorithms: Examples, Types, Complexity

Matemáticas

El método de la inducción matemática
El método de la inducción matemática.

La integral definida como el límite de una suma de Riemann
La integral definida como el límite de una suma de Riemann (artículo) | Khan Academy