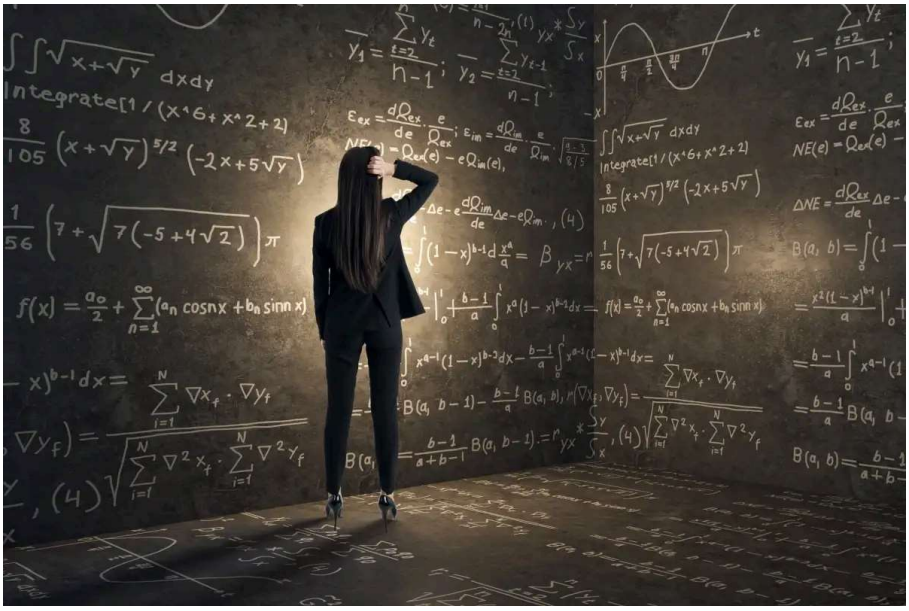
3. ALGORITMOS



Un **algoritmo** é un **conxunto ordenado e finito de operacións sinxelas que conducen á resolución dun problema**, como por exemplo a formulación programática paso a paso para producir unha serie de resultados nun programa en informática.

A palabra algoritmo ten orixe no alcume Al-Khwarizmi, do matemático persa do século IX, Abu Yafar Mohámmed Abenmusa, cuxas obras foron traducidas no occidente cristián no século XII, recibindo unha delas o nome "Algorithmi de numero indorum", sobre os algoritmos usando o sistema de numeración decimal (indiano). Outros autores, con todo, defenden a orixe da palabra en Al-goreten (raíz - concepto que se pode aplicar aos cálculos).

O concepto de algoritmo **é frecuentemente ilustrado co exemplo dunha receita**, aínda que moitos algoritmos son máis complexos. Eles poden repetir pasos (facer interaccións) ou necesitar decisións (tales como comparacións ou lóxica) ata que a tarefa sexa completada. Un algoritmo correctamente executado non resolverá un problema se o algoritmo fose incorrecto ou non fose apropiado ao problema.

3.1. CARACTERÍSTICAS

Aquí están as principais características dun algoritmo:

- **Claro e preciso**: Cada paso do algoritmo debe estar claramente definido e sen ambigüidades. As instrucións deben ser precisas e non deixar lugar a interpretacións distintas.
- **Finito**: Un algoritmo debe ter un número finito de pasos. Debe terminar despois dun número determinado de pasos, garantindo que non entre en ciclos infinitos.
- **Entrada**: Un algoritmo pode recibir datos de entrada, que son os valores ou parámetros cos que comeza a súa execución.
- **Saída**: Un algoritmo debe producir resultados ou saídas a partir das entradas proporcionadas. A saída é a solución ao problema ou o resultado da tarefa.
- **Xeral**: Un algoritmo debe ser xeral suficiente para resolver un conxunto de problemas similares e non só un caso específico. A súa aplicación debe ser ampla e adaptable a diferentes situacións ou datos.
- **Determinismo**: Un algoritmo debe ser determinista, o que significa que, para un conxunto dado de entradas, debe producir a mesma saída ou resultado cada vez que se execute.

3.1.1. Exemplo de Algoritmos

Para ilustrar estas características, aquí tes un exemplo simple de algoritmo para calcular a suma de números enteiros desde 1 ata (n):

Algoritmo para a suma dos primeiros N números enteiros

1. **Entrada**: Un número enteiro positivo (n).
2. **Inicialización**: Establecer a variable `suma` a 0.
3. **Proceso**:
    - Para cada número `i` desde 1 ata (n):
        - Engadir `i` á variable `suma`.
4. **Saída**: Devolver o valor de `suma`.

Este algoritmo:

- É **claro e preciso** porque cada paso está ben definido.
- É **finito** porque a suma de números ten un número finito de pasos.
- Require **entrada** (o valor de (n)) e produce **saída** (a suma).
- Pode ser **eficiente** para valores pequenos e medianos de (n).
- É **xeral** porque pode ser aplicado a calquera número enteiro positivo.
- É **determinista** porque para cada valor de (n), a saída será sempre a mesma.

3.2. LINGUAXES DE DESCRICIÓN DE ALGORITMOS

As linguaxes de descrición de algoritmos son **ferramentas** utilizadas para **expresar e documentar a lóxica e a estrutura de algoritmos** de maneira que sexan fáciles de entender e comunicar. Estas linguaxes están deseñadas para permitir unha representación clara e precisa dos pasos necesarios para resolver un problema ou realizar unha tarefa.

Algunhas destas linguaxes son:

- **Pseudocódigo**.
- **Diagramas de fluxo**: Os diagramas de fluxo son representacións gráficas dos algoritmos que usan símbolos e frechas para mostrar o fluxo de control e as operacións realizadas.
- **Notación algorítmica**: A notación algorítmica usa unha forma máis formalizada e estruturada para describir algoritmos, frecuentemente empregando símbolos matemáticos e operadores lóxicos.
- **Linguaxes de programación especializados para algoritmos**: Algunhas linguaxes de programación están deseñadas especificamente para a descrición e análise de algoritmos. Estas linguaxes proporcionan unha sintaxe e semántica especializadas para facilitar a definición e a probas de algoritmos. Un exemplo é o *Algorithmic Language* (Algol).
- **Linguaxes de modelado e diagramas**: Utilizan diagramas e linguaxes gráficas para representar algoritmos e procesos. Estes métodos son útiles para describir procesos complexos e sistemas. Un exemplo desta linguaxe é **Unified Modeling Language (UML)**. Esta usa diagramas como diagramas de secuencia e diagramas de actividades para modelar sistemas e algoritmos.

3.2.1. PSEUDOCÓDIGO

O **pseudocódigo** é unha **forma de representación de algoritmos** que usa unha combinación de **linguaxe natural e notación matemática** para describir a lóxica e os **pasos necesarios para resolver un problema**.

A principal vantaxe do pseudocódigo é a súa simplicidade e claridade, permitindo aos programadores e analistas describir algúns algoritmos de forma que sexan comprensibles independentemente da linguaxe de programación que se utilizará para implantalos.

3.2.1.1. Características

- **Simplicidade**: Está deseñado para ser **lido e comprendido sen coñecer unha linguaxe de programación particular**.
- **Flexibilidade**: **Non segue unha sintaxe estrita**, polo que pode ser adaptado para adecuarse ás necesidades do problema ou do desenvolvedor. Isto permite que se enfoque na lóxica do algoritmo máis que na sintaxe específica dunha linguaxe de programación.
- **Claridade**: A finalidade do pseudocódigo é **expresar a lóxica do algoritmo de forma clara e precisa**. Utiliza estruturas simples e termos familiares que axudan a visualizar a solución ao problema.
- **Independencia da linguaxe**:O pseudocódigo **non está ligado a ningunha linguaxe de programación** en particular. A idea é que o pseudocódigo **pode ser convertido en código fonte en diferentes linguaxes de programación**, dependendo das necesidades do proxecto.

3.2.1.2. Estrutura e convencións comúns

- **Operacións Básicas**: Usa termos como `INICIO`, `FINAL`, `SE`, `SENÓN`, `PARA`, `MENTRES` para representar estruturas de control de fluxo.
- **Variables e Datos**: As variables e os datos son representados de maneira intuitiva, co obxectivo de facer a lóxica do algoritmo máis clara.
- **Entrada e Saída**: A entrada e a saída dos datos son especificadas de maneira simple. Por exemplo, `LER dato` para ler unha entrada e `ESCRIBIR resultado` para imprimir un resultado.

3.2.1.3. Exemplos de pseudocódigo

Pseudocódigo dun algoritmo que calcula a suma dos números do 1 ao (n)

```
INICIO
   LER n
   suma = 0
   PARA i DESDE 1 ATA n
       suma = suma + i
   FIN PARA
   MOSTRAR suma
FIN
```

- **INICIO e FIN**: Marcan o inicio e o final do algoritmo.
- **LEER n**: Solicita ao usuario que ingrese un número (n).
- **suma = 0**: Inicializa a variable `suma` a 0.

- **PARA i DESDE 1 ATA n**: Unha bucle que vai desde 1 ata (n).
- **suma = suma + i**: Engade o valor de `i` á `suma` en cada iteración.
- **MOSTRAR suma**: Mostra o resultado final da suma ao usuario.

Pseudocódigo dun algoritmo que determina se un número é positivo, negativo ou cero

```
INICIO
    LER numero

    SE numero > 0 ENTÓN
        MOSTRAR "O número é positivo."
    SENON SE numero < 0 ENTÓN
        MOSTRAR "O número é negativo."
    SENON
        MOSTRAR "O número é cero."
    FIN SE
FIN
```

- **INICIO e FIN**: Marcan o inicio e o final do algoritmo.
- **LEER numero**: Solicita ao usuario que ingrese un número.
- **SE numero > 0 ENTÓN**: Comproba se o número é maior que 0.
- **SENON SE numero < 0 ENTÓN**: Senón é maior que 0, comproba se o número é menor que 0.
- **SENON**: Senón é maior que 0 nin menor que 0.
- **MOSTRAR “O número é positivo.”**, **MOSTRAR “O número é negativo.”** e **MOSTRAR “O número é cero.”**: Mostra o tipo de número que se introduciu.

3.3. CLASIFICACIÓN

A clasificación de algoritmos informáticos pode facerse de varias formas, dependendo dos criterios que se consideren. Aquí está unha clasificación baseada en diferentes enfoques:

3.3.1. Clasificación por Estrutura de Datos

- **Algoritmos de Ordenación** Organizar un conxunto de elementos en orde ascendente ou descendente. Exemplos:
    - **Ordenación Rápida** (*Quick Sort*)
- **Algoritmos de Busca**: Localizar un elemento específico dentro de unha estrutura de datos. Exemplos:
    - **Búsqueda Lineal** (*Linear Search*)
    - **Búsqueda Binaria** (*Binary Search*)

3.3.2. Clasificación por Estratexia de Resolución

- **Algoritmos de divide e vencerás**: Resolver un problema dividindo en problemas máis pequenos e resolvéndoos de forma recursiva.
    - **Algoritmo de Karatsuba** (para multiplicación de números grandes)
- **Algoritmos de programación dinámica** Resolver problemas almacenando resultados de subproblemas para evitar cálculos redundantes.
    - **Cálculo do Número de Fibonacci** (Fibonacci Sequence)
- **Algoritmos de voraces** Construír solucións paso a paso, elixindo a mellor opción dispoñible en cada paso.
- **Algoritmos de Backtracking**: Explorar todas as posibles solucións e retroceder cando se detecta que unha opción non é viable.
    - **Problema das Reinas** (N-Queens Problem)
- **Algoritmos de busca e optimización** Buscar solucións óptimas ou buscar na espazo de solucións para problemas complexos.

3.3.2. Clasificación por paradigma

- **Algoritmos Iterativos**: Utilizan bucles para repetir un conxunto de operacións ata que se cumpre unha condición.
- **Algoritmos Recursivos**: Resolven problemas chamando á mesma función dentro de si mesma, normalmente para dividir o problema en problemas máis pequenos.
- **Algoritmos Concorrentes e Paralelos**: Realizan múltiples tarefas simultaneamente para mellorar a eficiencia e o tempo de execución.
    - **Algoritmos de Map-Reduce** (para procesamento paralelo de grandes conxuntos de datos)

3.4. COMPLEXIDADE

A complexidade dos algoritmos é unha **medida que nos axuda a entender canto tempo ou espazo (memoria) necesita un algoritmo para resolver un problema en función do tamaño da entrada**. A complexidade pode ser avaliada principalmente en termos de tempo e espazo, e axuda a comparar a eficiencia de diferentes algoritmos. A continuación, explico os conceptos principais de forma sinxela:

A notación ***Big-O*** é unha forma de **describir a complexidade de algoritmos** en termos de como o tempo ou o espazo necesario para executar un algoritmo crece co tamaño da entrada, (n). É unha ferramenta esencial na análise de algoritmos, permitindo comparar a eficiencia de diferentes algoritmos de forma abstracta e independente das características específicas dos sistemas nos que se executan.

3.4.1. Complexidade Temporal

A complexidade temporal **mide o tempo que leva a execución dun algoritmo**. A idea é ver **como varía o tempo de execución cando o tamaño da entrada cambia**.

- **O(1) - Complexidade constante**: O tempo de execución non cambia co tamaño da entrada. Por exemplo, acceder a un elemento específico nun array.
- **O(n) - Complexidade lineal**: O tempo de execución aumenta linealmente co tamaño da entrada. Por exemplo, percorrer unha lista de (n) elementos.
- **O(n^2) - Complexidade cuadrática**: O tempo de execución aumenta proporcionalmente ao cadrado do tamaño da entrada. Por exemplo, o algoritmo de ordenación por inserción.
- **O(log n) - Complexidade logarítmica**: O tempo de execución aumenta logarítmicamente co tamaño da entrada. Por exemplo, a busca binaria en listas ordenadas.
- **O(2^n) - Complexidade exponencial**: O tempo de execución aumenta exponencialmente co tamaño da entrada. Por exemplo, a solución de forza bruta para o problema do subconxunto.

3.4.2. Complexidade Espacial

A complexidade espacial mide a **cantidade de memoria** adicional que necesita un algoritmo para executar. Aínda que normalmente estamos máis interesados no tempo de execución, o uso de memoria tamén é importante, especialmente para grandes conxuntos de datos.

- **O(1) - Espazo Constante**: O algoritmo usa unha cantidade fixa de memoria, independentemente do tamaño da entrada.
- **O(n) - Espazo Lineal**: O algoritmo usa memoria proporcional ao tamaño da entrada.
