

PONTIFICIA UNIVERSIDAD JAVERIANA



Análisis de algoritmos

Taller fuerza bruta

Gabriela Cruz, Angelica Piedrahita,

Catalina Lizarazo, Juan Rozo

Maria Paula Rodríguez

Agosto, 2025

Problema 1: Búsqueda lineal

Origen

La búsqueda lineal es uno de los algoritmos más simples y antiguos en informática.

Aunque no se atribuye a un único inventor, se considera una técnica fundamental en la

teoría de algoritmos. Su origen se remonta a los primeros días de la computación, cuando se requería una forma sencilla de localizar elementos en listas no ordenadas. Este algoritmo se basa en la idea de recorrer secuencialmente una lista hasta encontrar el elemento deseado o llegar al final de la misma.

Explicación del Problema: Objetivo

El objetivo de la búsqueda lineal es encontrar la posición de un elemento específico dentro de una lista. Dado que no se asume ningún orden en la lista, el algoritmo revisa cada elemento de manera secuencial hasta encontrar una coincidencia

Complejidad Computacional

Tiempo: La complejidad temporal en el peor caso es $O(n)$, donde n es el número de elementos en la lista. Esto ocurre cuando el elemento buscado está al final de la lista o no está presente en absoluto.

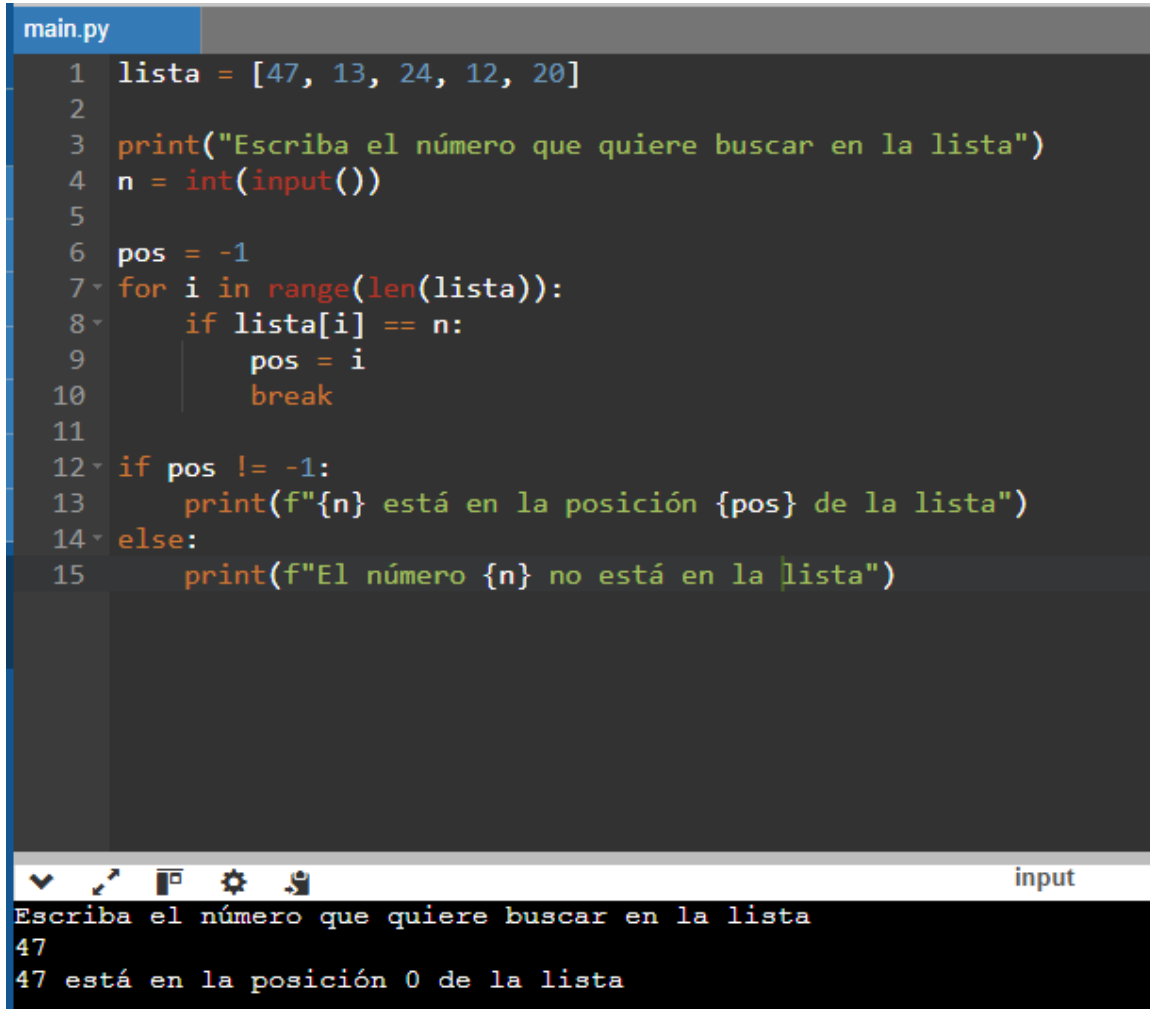
Espacio: La complejidad espacial es $O(1)$, ya que el algoritmo utiliza una cantidad constante de memoria adicional independientemente del tamaño de la lista.

Este comportamiento lo convierte en una opción adecuada para listas pequeñas o cuando la simplicidad es más importante que la eficiencia.

Fuerza Bruta en la Búsqueda Lineal

La búsqueda lineal puede considerarse un algoritmo de *fuerza bruta*, ya que no hace ninguna optimización ni toma decisiones inteligentes sobre el orden de los elementos en la lista. El algoritmo simplemente prueba todos los elementos de manera secuencial hasta encontrar el que busca. Esta aproximación garantiza que se encontrará el elemento, pero no es eficiente para listas grandes, ya que se requieren comparaciones repetitivas sin ninguna heurística o estrategia para reducir el número de búsquedas. La fuerza bruta en este contexto es su mayor debilidad, ya que, en el peor de los casos, el algoritmo realiza n comparaciones, lo que lo hace ineficiente comparado con otros métodos más sofisticados como la búsqueda binaria, que requiere una lista ordenada.

Código y pruebas

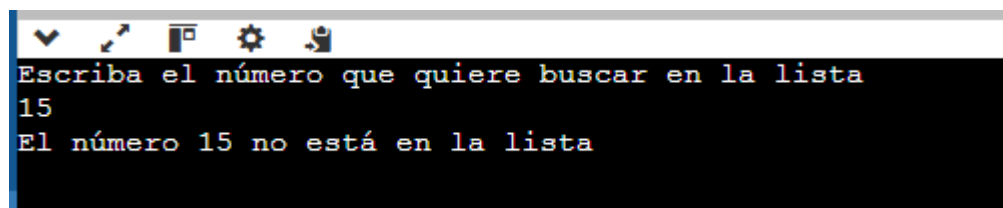


```
main.py
1 lista = [47, 13, 24, 12, 20]
2
3 print("Escriba el número que quiere buscar en la lista")
4 n = int(input())
5
6 pos = -1
7 for i in range(len(lista)):
8     if lista[i] == n:
9         pos = i
10        break
11
12 if pos != -1:
13     print(f"{n} está en la posición {pos} de la lista")
14 else:
15     print(f"El número {n} no está en la lista")
```

input

Escriba el número que quiere buscar en la lista
47
47 está en la posición 0 de la lista

Imagen 1. Código del algoritmo búsqueda lineal



```
Escriba el número que quiere buscar en la lista  
15  
El número 15 no está en la lista
```

Imagen 2. Prueba del algoritmo búsqueda lineal

Problema 4: TSP por Fuerza Bruta

1. Origen:

El Problema del Viajante de Comercio (TSP) aparece de manera natural en logística y planificación de rutas: un agente debe visitar un conjunto de ubicaciones exactamente una vez y volver al inicio minimizando la distancia (o el costo) total. Pronto se volvió un caso emblemático de la optimización combinatoria porque su espacio de soluciones explota factorialmente con el número de ciudades.

2. ¿Quién lo formalizó?

Aunque la idea es muy antigua, la formulación académica del TSP se atribuye a los años 1930 con el matemático Karl Menger, quien lo usó como ejemplo teórico de rutas óptimas y geometría. En los años 50, investigadores como Hassler Whitney y equipos en RAND Corporation lo popularizaron en el ámbito computacional y lo conectaron con la teoría de grafos y la programación entera. Desde entonces es un “benchmark” clásico: NP-difícil, central en investigación y aplicaciones.

3. Objetivo

Dadas n ciudades y un costo/distancia $d(i, j)$ para cada par, hallar un ciclo Hamiltoniano (ruta cerrada) que:

- Empieza en una ciudad (origen), visita cada ciudad exactamente una vez, regresa al origen, y minimiza la suma de costos a lo largo de la ruta.

4. Explicación del algoritmo:

La formulación más directa es considerar el problema sobre un grafo completo con pesos. Una ruta factible es una permutación de las ciudades; su costo es la suma de los pesos de aristas consecutivas, incluyendo el regreso al origen.

Ideas clave:

- Eliminar simetrías: fijar la ciudad 0 como punto de partida/retorno para no contar rutas equivalentes que solo rotan el inicio.
- Evaluar el costo de una permutación π formando el ciclo $0 \rightarrow \pi_1 \rightarrow \pi_2 \rightarrow \dots \rightarrow \pi_{(n-1)} \rightarrow 0$.
- Mantener el mejor costo encontrado y su ruta.

5. Evaluación con fuerza bruta

La fuerza bruta recorre todas las permutaciones de las $n-1$ ciudades restantes (al fijar la 0). Para cada una:

1. Construye la ruta cerrada.
2. Suma los costos de aristas consecutivas.
3. Actualiza el mínimo global si mejora.

6. Complejidad

- Tiempo: $O(n!)$. Más precisamente, se evalúan $(n-1)!$ permutaciones y el cálculo de cada ruta es $O(n)$, por lo que el tiempo total es $\Theta(n \cdot (n-1)!)$ dominado por el factorial.
- Espacio: $O(n)$ para la ruta actual y la mejor ($O(n^2)$ si guardas matriz de distancias).

7. Código

```

1 from itertools import permutations
2 from math import hypot
3 from typing import List, Tuple
4
5 Point = Tuple[float, float]
6
7 def distancia(a: Point, b: Point) -> float:
8     return hypot(a[0] - b[0], a[1] - b[1])
9
10 def tsp_bruteforce(coords: List[Point]) -> Tuple[List[int], float]:
11     """
12     Recorre todas las permutaciones de las ciudades (fijando 0 como origen)
13     y devuelve la mejor ruta cerrada y su longitud total.
14     Tiempo: O(n!) | Espacio: O(n)
15     """
16     n = len(coords)
17     if n <= 1:
18         return [0], 0.0
19
20     # Precompute matriz de distancias para acelerar
21     dist = [[0.0]*n for _ in range(n)]
22     for i in range(n):
23         for j in range(i+1, n):
24             d = distancia(coords[i], coords[j])
25             dist[i][j] = dist[j][i] = d
26
27     mejor_ruta = []
28     mejor_long = float("inf")
29
30     for perm in permutations(range(1, n)):        # ciudades 1..n-1
31         ruta = (0,) + perm + (0,)                # circuito que inicia/termina en 0
32         # costo de la ruta
33         costo = sum(dist[ruta[i]][ruta[i+1]] for i in range(n))
34         if costo < mejor_long:
35             mejor_long = costo
36             mejor_ruta = list(ruta)
37
38     return mejor_ruta, mejor_long
39
40 # --- Ejemplo del enunciado ---
41 coords = [(0,0), (1,0), (1,1), (0,1)]
42 ruta, longitud = tsp_bruteforce(coords)
43 print("Ruta óptima:", ruta)          # p.ej. [0, 1, 2, 3, 0] o equivalente
44 print("Longitud:", longitud)         # ≈ 4.0
45

```

Imagen 3. Código del algoritmo de TSP por fuerza bruta

Problema 1: N-Reinas (backtracking)

1. Origen:

- El problema de las N-reinas tiene sus raíces en el siglo XIX, específicamente en el año 1848, cuando el ajedrecista alemán Max Bezzel propuso el problema de colocar 8 reinas en un tablero de ajedrez con 8 filas y 8 columnas sin que se atacaran entre sí. Décadas después, el matemático Franz Nauck generalizó el problema para un tablero de tamaño $N \times N$, surgiendo así el problema de las N-Reinas, fue entonces cuando surgió el backtracking que básicamente es una técnica que se basa en probar y retroceder cuando algo no funciona. Funciona bien para problemas como el de las N reinas, donde se exploran diferentes combinaciones y se descartan las que no cumplen con las reglas, de manera rápida y sistemática.

2. Objetivo

- El objetivo del problema de las N-Reinas es colocar N reinas en un tablero de ajedrez de $N \times N$ casillas de tal manera que ninguna pueda atacar a otra. Esto significa que no puede haber dos reinas en la misma fila, columna o diagonal. La solución implica encontrar una o más configuraciones válidas que cumplan esta restricción.

3. Complejidad computacional

- Tiempo: La complejidad temporal en el peor caso es $O(N!)$, ya que en cada fila hay que probar hasta N posiciones y retroceder si no se encuentra solución. Con poda (verificación previa de columnas y diagonales) el tiempo se reduce pero sigue siendo exponencial.
- Espacio:
La complejidad espacial es $O(N)$ para almacenar la posición de las reinas (una por fila) más las estructuras auxiliares para verificar columnas y diagonales ocupadas. Si se guardan todas las soluciones, el espacio requerido crece proporcionalmente al número de soluciones.

4. Fuerza bruta

- Usar fuerza bruta para resolver las N-Reinas significa probar **todas** las formas posibles de colocar las reinas en el tablero hasta encontrar las que cumplen las reglas. El problema es que, aunque este método asegura que al final vas a

encontrar las soluciones correctas, se vuelve muy lento cuando el tablero es grande, porque hay millones de formas posibles de acomodar las reinas.

5. Código y pruebas

```
def solve_one_n_queens(n: int) -> List[int]:
    """Devuelve UNA solución al problema de las N-Reinas como lista de columnas."""
    cols = [-1] * n
    usadas_col = set()
    diag1 = set()
    diag2 = set()
    encontrada = [False]

    def backtrack(filas: int):
        if filas == n:
            encontrada[0] = True
            return True
        for c in range(n):
            if (c in usadas_col) or ((filas - c) in diag1) or ((filas + c) in diag2):
                continue
            cols[filas] = c
            usadas_col.add(c)
            diag1.add(filas - c)
            diag2.add(filas + c)

            if backtrack(filas + 1):
                return True

            usadas_col.remove(c)
            diag1.remove(filas - c)
            diag2.remove(filas + c)
            cols[filas] = -1
        return False

    backtrack(0)
    return cols

def print_board(sol: List[int]):
    n = len(sol)
    print("Fila->", *range(n))
    for f in range(n):
        fila = ["." for _ in range(n)]
        fila[sol[f]] = "Q"
        print(" ", " ".join(fila))

def main():
    n = int(input("Ingresa N (tamaño del tablero): "))
    sol = solve_one_n_queens(n)
    print(f"\nN = {n} | Una solución encontrada (columnas por fila): {sol}\n")
    print_board(sol)

if __name__ == "__main__":
    main()
```

Imagen 4. Código del algoritmo de N-Reinas (backtracking)


```

Ingresa N (tamaño del tablero): 8

N = 8 | Una solución encontrada (columnas por fila): [0, 4, 7, 5, 2, 6, 1, 3]

Fila-> 0 1 2 3 4 5 6 7
Q . . . . . . .
. . . . Q . . .
. . . . . . Q
. . . . Q . .
. . Q . . . .
. . . . . Q .
. Q . . . . .
. . . Q . . .

```

Imagen 5. Prueba del algoritmo de N-Reinas (backtracking)

Problema 10: Multiplicacion de matrices (naive)

Origen

La multiplicación de matrices es una operación fundamental en álgebra lineal y en múltiples áreas de la informática, desde gráficos por computadora hasta aprendizaje automático. Su definición y uso formal provienen de las matemáticas puras, pero fue adoptada tempranamente en la computación debido a su capacidad para representar y combinar transformaciones, resolver sistemas de ecuaciones y modelar datos. El método naive (o clásico) es el más básico y se basa directamente en la definición matemática del producto de matrices.

Explicación del Problema: Objetivo

El objetivo de la multiplicación de matrices es obtener una nueva matriz donde cada elemento $C[i][j]$ se calcula sumando el producto de los elementos correspondientes de la fila i de la matriz A y la columna j de la matriz B .

Complejidad Computacional

- Tiempo: La complejidad temporal es $O(n^3)$ para matrices cuadradas de tamaño $n \times n$ ya que se realizan tres bucles anidados: uno para las filas de A, otro para las columnas de B y otro para el índice intermedio k.
- Espacio: La complejidad espacial extra es $O(1)$ si se almacena el resultado directamente en una matriz de salida de tamaño fijo y no se utilizan estructuras auxiliares significativas.

Fuerza Bruta en la Multiplicación de Matrices

El método naive puede considerarse un algoritmo de fuerza bruta porque sigue la definición matemática al pie de la letra, sin aplicar optimizaciones como la reducción del número de multiplicaciones o el aprovechamiento de la estructura de las matrices. En el peor de los casos, se ejecutan n^3 multiplicaciones y casi el mismo número de sumas, lo que lo vuelve costoso computacionalmente para n grandes.

Código y pruebas

```
main.py
1 def multiplicar_matrices(A, B):
2     C = [[0 for _ in range(3)] for _ in range(3)]
3
4     for i in range(3):
5         for j in range(3):
6             for k in range(3):
7                 C[i][j] += A[i][k] * B[k][j]
8     return C
9
10
11 A = [
12     [1, 2, 3],
13     [4, 5, 6],
14     [7, 8, 9]
15 ]
16
17 B = [
18     [9, 8, 7],
19     [6, 5, 4],
20     [3, 2, 1]
21 ]
22
23 resultado = multiplicar_matrices(A, B)
24 for fila in resultado:
25     print(fila)
26
27
```

```
[30, 24, 18]
[84, 69, 54]
[138, 114, 90]

...Program finished with exit code 0
Press ENTER to exit console.
```

Imagen 6. Código y prueba del algoritmo de multiplicación de matrices

Problema 16: Sudoku Solver

Origen:

El concepto fundamental del sudoku proviene de los cuadrados latinos, una idea matemática estudiada por Leonhard Euler en el siglo XVIII los cuales son una cuadrícula donde cada símbolo aparece solo una vez por fila y columna. Basado en esta idea, el primer juego similar al sudoku moderno fue creado en 1979 en Estados Unidos por Howard Garns,

y se publicó con el nombre “Number Place” en la revista “Dell Pencil Puzzles and Word Games”.

Explicación del problema: Objetivo

El sudoku es un rompecabezas de lógica muy popular en todo el mundo. Su versión más común se presenta como una cuadrícula de 9x9 casillas, dividida en nueve bloques de 3x3. El objetivo del juego es llenar todas las casillas con números del 1 al 9, de manera que no se repitan dentro de cada fila, cada columna y cada uno de los nueve bloques. A pesar de usar números, el sudoku no requiere operaciones matemáticas, ya que se basa únicamente en la deducción lógica.

Aunque el sudoku es un juego de entretenimiento, también tiene conexiones profundas con la matemática y la informática teórica. Resolver un sudoku generalizado (tableros más grandes que 9x9) se considera un problema NP-completo, lo cual significa que no existe (hasta el momento) una forma eficiente de resolver todos los posibles casos, y está relacionado con problemas fundamentales en la teoría de la computación. Además, desde un punto de vista matemático, el sudoku se relaciona con la combinatoria, la teoría de grafos y la teoría de matrices.

Complejidad computacional:

- **Tiempo:** Su complejidad algorítmica es exponencial en el peor de los casos. Para el desarrollo de este taller se ha desarrollado una matriz de 9X9 por lo cual su complejidad algorítmica suponiendo un tablero vacío sería (cosa que no es común en estos juegos):

$$O(n^{n \cdot n}) = O(9^{81})$$

Por esto, se ha decidido hacer el código en base a un tablero clásico de sudoku con algunos unos números ya predeterminados como es este rompecabezas usualmente.

- **Espacio:** La complejidad de espacio del algoritmo es $O(1)$ en términos de memoria adicional, ya que solo utiliza una matriz fija de 9x9 (el tablero) y unas pocas variables auxiliares.

Fuerza bruta:

La función de backtracking implementa una estrategia de fuerza bruta para resolver problemas. Consiste en intentar todas las posibles soluciones (fuerza bruta), pero retrocediendo (backtrack) tan pronto como se detecta que una opción no puede llevar a una solución válida. En este caso del Sudoku, el algoritmo coloca un número en una celda vacía, verifica si es válido, y continúa con la siguiente; si más adelante se encuentra un conflicto, deshace (retrocede) la elección y prueba con otro número. De esta forma, evita explorar combinaciones inútiles, reduciendo significativamente el número de intentos respecto a una fuerza bruta pura.

Código y pruebas:

```

#include <iostream>
using namespace std;

const int N = 9;

// Función para imprimir el tablero
void imprimirTablero(int tablero[N][N]) {
    for (int fila = 0; fila < N; fila++) {
        for (int col = 0; col < N; col++) {
            cout << tablero[fila][col] << " ";
        }
        cout << endl;
    }
}

// Verifica si es válido colocar un número en la celda
bool esValido(int tablero[N][N], int fila, int col, int num) {
    // Revisar fila
    for (int x = 0; x < N; x++)
        if (tablero[fila][x] == num)
            return false;

    // Revisar columna
    for (int x = 0; x < N; x++)
        if (tablero[x][col] == num)
            return false;

    // Revisar subcuadro 3x3
    int startRow = fila - fila % 3;
    int startCol = col - col % 3;

    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            if (tablero[i + startRow][j + startCol] == num)
                return false;

    return true;
}

// Algoritmo de fuerza bruta (backtracking)
bool resolverSudoku(int tablero[N][N]) {
    for (int fila = 0; fila < N; fila++) {
        for (int col = 0; col < N; col++) {
            if (tablero[fila][col] == 0) { // Casilla vacía
                for (int num = 1; num <= 9; num++) {
                    if (esValido(tablero, fila, col, num)) {
                        tablero[fila][col] = num;

                        if (resolverSudoku(tablero))
                            return true;

                        tablero[fila][col] = 0; // Backtrack
                    }
                }
                return false; // No hay número válido
            }
        }
    }
    return true; // Sudoku resuelto
}

```

Imagen 7. Código del algoritmo de Sudoku solver

```

62 // Función principal
63 int main() {
64     int sudoku[N][N] = {
65         {5, 3, 0, 0, 7, 0, 0, 0, 0},
66         {6, 0, 0, 1, 9, 5, 0, 0, 0},
67         {0, 9, 8, 0, 0, 0, 0, 6, 0},
68         {8, 0, 0, 0, 6, 0, 0, 0, 3},
69         {4, 0, 0, 8, 0, 3, 0, 0, 1},
70         {7, 0, 0, 0, 2, 0, 0, 0, 6},
71         {0, 6, 0, 0, 0, 0, 2, 8, 0},
72         {0, 0, 0, 4, 1, 9, 0, 0, 5},
73         {0, 0, 0, 0, 8, 0, 0, 7, 9}
74     };
75
76     if (resolverSudoku(sudoku)) {
77         cout << "Sudoku resuelto:\n";
78         imprimirTablero(sudoku);
79     } else {
80         cout << "No tiene solución." << endl;
81     }
82
83     return 0;
84 }

```

Imagen 8. Código del algoritmo de Sudoku solver

```

Sudoku resuelto:
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9

```

Imagen 9. Prueba del algoritmo de Sudoku solver

REFERENCIAS:

- Applegate, D. L., Bixby, R. E., Chvátal, V., & Cook, W. J. (2006). *The traveling salesman problem: A computational study*. Princeton University Press.
<https://doi.org/10.1515/9781400841103>
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. *Introduction to Algorithms* (3rd ed.). MIT Press, 2009.
- Strang, G. *Introduction to Linear Algebra* (5th ed.). Wellesley-Cambridge Press, 2016.
- Knuth, D. E. *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley, 1998.
- Logic puzzles, (2025). *Sudoku History*. Conceptis puzzles. Recuperado de:
<https://www.conceptispuzzles.com/index.aspx?uri=puzzle/sudoku/history>
- Sudoku.com (2018). *¿Cómo jugar Sudoku?*. EasyBrain. Recuperado de:
<https://sudoku.com/es>
- M. Rodríguez, “El problema matemático de las reinas del ajedrez que un científico de Harvard resolvió tras 150 años sin solución,” *BBC News Mundo*, Feb. 13, 2022. <https://www.bbc.com/mundo/noticias-60337860>