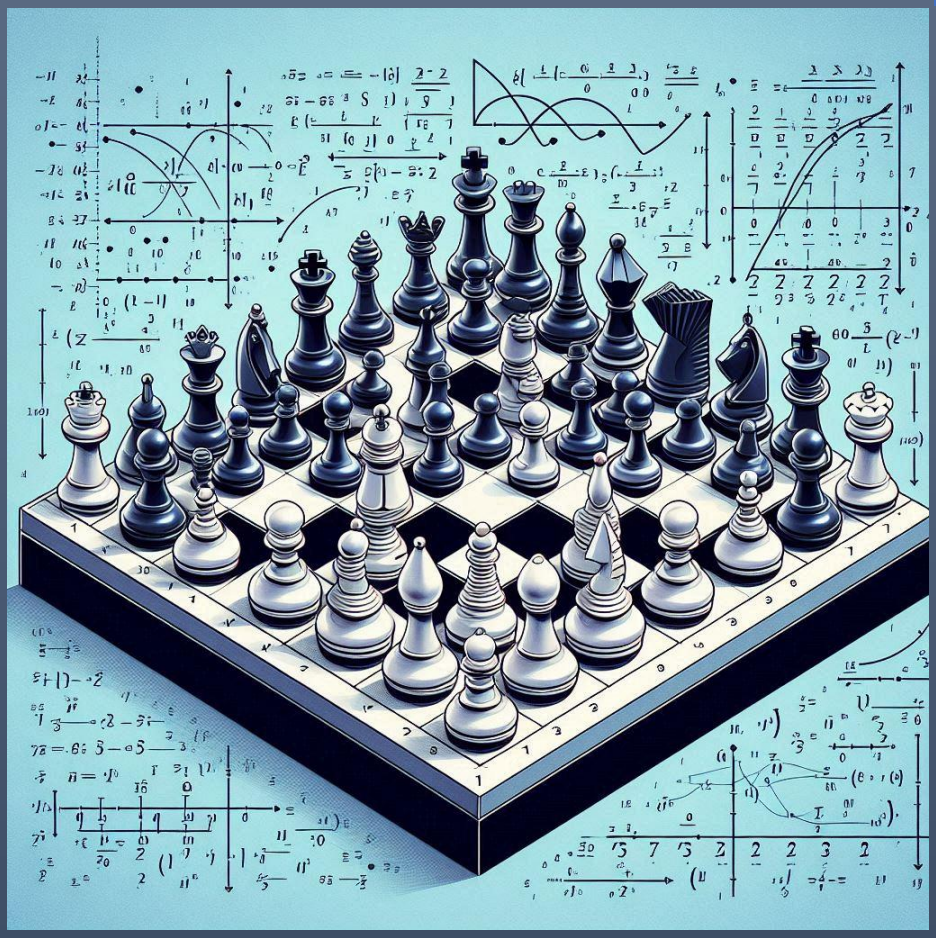
2024



# Modelos de Inteligencia Artificial

Profesor/a: Águeda María López Moreno

PRÁCTICA 1.3.- PROBLEMA DE LAS N REINAS CON ALEATORIEDAD

21/10/2024

PEDRO MANUEL GARCÍA ÁLVAREZ | int.marisma@gmail.com | CEIA | I.E.S. La Marisma

Índice

# 1.- Introducción Teórica

El Problema de las N-Reinas es un clásico en la teoría de algoritmos y optimización, que desafía al programador a colocar N reinas en un tablero de ajedrez de N×N de tal forma que ninguna reina pueda atacar a otra. Este problema, que parece simple, se convierte en un reto complejo conforme aumenta el tamaño del tablero, siendo un ejercicio excelente para explorar distintas técnicas de resolución de problemas y optimización.

En esta práctica, se abordará la resolución del Problema de las N-Reinas utilizando una técnica aleatoria de búsqueda, que introduce una perspectiva innovadora en comparación con métodos tradicionales, como el backtracking y la búsqueda en anchura (BFS).

## 1.1.- Problema de las N-Reinas

El Problema de las N-Reinas requiere colocar N reinas en un tablero de ajedrez de tamaño N×N de manera que no haya dos reinas en la misma fila, columna o diagonal. La solución al problema no solo es una aplicación matemática interesante, sino que también ilustra conceptos importantes en la teoría de algoritmos, inteligencia artificial y optimización.

La dificultad del problema aumenta exponencialmente a medida que crece el valor de N, lo que lo convierte en un candidato ideal para estudiar y aplicar diversas estrategias de búsqueda y optimización. Las soluciones deben garantizar que cada reina esté colocada en una posición segura, evitando conflictos horizontales, verticales y diagonales.

## 1.2.- Técnicas de Búsqueda y Resolución Aleatoria

La resolución aleatoria de problemas, también conocida como búsqueda aleatoria, es una técnica que selecciona posibles soluciones de manera no determinista, es decir, introduce un componente de aleatoriedad en el proceso. En el contexto del Problema de las N-Reinas, una estrategia aleatoria implica seleccionar columnas al azar para colocar cada reina y continuar verificando la validez de la configuración hasta encontrar una solución completa.

En este enfoque, el programa comienza colocando una reina en una columna aleatoria de la primera fila. Si al avanzar por las filas no

hay una posición válida para colocar la siguiente reina, el proceso se reinicia desde el principio. Esta técnica puede considerarse ineficiente para valores de N grandes, pero permite obtener una solución viable para tamaños de tablero pequeños a medianos, ofreciendo una alternativa a métodos tradicionales.

## 1.3.- Definición de Algoritmos con Aleatoriedad

Los algoritmos aleatorios son aquellos que incorporan algún grado de incertidumbre en su ejecución, seleccionando elementos de forma no predecible y basándose en decisiones probabilísticas para avanzar en la búsqueda de soluciones. Los algoritmos aleatorios pueden ser clasificados como:

- **Aleatorios Las Vegas**: Donde el algoritmo garantiza una solución correcta, pero el tiempo de ejecución es variable.
- **Aleatorios Monte Carlo**: Donde el algoritmo da una respuesta rápidamente, pero la respuesta puede ser incorrecta con una pequeña probabilidad.

Para el Problema de las N-Reinas, el enfoque aleatorio adoptado pertenece a la categoría Las Vegas, ya que no se detendrá hasta que se encuentre una solución válida. El uso de aleatoriedad puede evitar la exploración exhaustiva de todas las combinaciones, lo que en algunos casos reduce el tiempo de búsqueda en comparación con algoritmos deterministas.

## 1.4.- Comparación con otras Técnicas (Búsqueda en Anchura y Backtracking)

El enfoque aleatorio en el Problema de las N-Reinas ofrece una alternativa interesante frente a técnicas de búsqueda en anchura (BFS) y backtracking, que son métodos deterministas ampliamente utilizados.

### Ventajas del Algoritmo Aleatorio frente a BFS y Backtracking

- **Menor estructuración**: El algoritmo aleatorio no necesita una estructura de datos compleja para administrar el estado del tablero, ya que simplemente reinicia el proceso si se encuentra un conflicto.

- **Exploración no sistemática**: La aleatoriedad evita la trampa de ciertos patrones repetitivos, abriendo la posibilidad de hallar soluciones que los métodos deterministas pasarían por alto o que requerirían más tiempo.
- **Simplicidad en la implementación**: Este método es relativamente sencillo de implementar y es útil para encontrar soluciones rápidamente para valores pequeños de N, sin necesidad de una planificación compleja de la búsqueda.

### Ventajas del BFS y Backtracking frente al Algoritmo Aleatorio

- **Garantía de solución eficiente**: Tanto BFS como backtracking garantizan encontrar una solución en menos pasos que el método aleatorio, ya que exploran de forma organizada cada posibilidad en el tablero.
- **Optimización del espacio de búsqueda**: El backtracking descarta rápidamente caminos no válidos, lo que hace que este método sea más eficiente en términos de tiempo y espacio que la aleatoriedad, especialmente para valores grandes de N.
- **Determinismo y reproducibilidad**: Al ser deterministas, BFS y backtracking producen soluciones reproducibles y más adecuadas para tableros grandes, donde la aleatoriedad podría no ser eficiente debido a la alta probabilidad de reinicio.

## 2.- Desarrollo del Problema

Para abordar el Problema de las N-Reinas con un enfoque aleatorio, se emplea una estrategia en la que cada reina se coloca en una columna seleccionada al azar dentro de cada fila. Este enfoque se basa en intentar múltiples configuraciones, lo que añade variabilidad y, eventualmente, permite encontrar una solución sin conflictos en el tablero.

### 2.1.- Explicación del Enfoque Aleatorio

El enfoque aleatorio consiste en ubicar cada reina en una columna seleccionada al azar para cada fila del tablero de N×N, sin seguir un patrón o secuencia fija. Este método permite explorar configuraciones menos estructuradas, evitando el orden sistemático

que utilizan las estrategias deterministas como el backtracking. En este proceso:

1. **Selección aleatoria de columnas**: La reina de cada fila se coloca en una de las columnas elegidas aleatoriamente. Si una reina puede ocupar la columna sin entrar en conflicto con otras reinas ya ubicadas en filas anteriores, el algoritmo procede a la siguiente fila.
2. **Detección de conflictos**: En cada intento, el algoritmo verifica si la posición de una reina generará un conflicto con las demás reinas en filas anteriores. Si se detecta un conflicto y no hay posiciones válidas en una fila, se borra el tablero y se reinicia el proceso desde la primera fila.
3. **Reinicio del proceso**: Dado que este enfoque es probabilístico, el proceso se repetirá desde el inicio hasta que se encuentre una configuración en la que las N reinas estén correctamente ubicadas.

Este método es menos eficiente en términos de pasos para encontrar una solución cuando N es grande, ya que el algoritmo puede reiniciar muchas veces antes de dar con una solución válida. Sin embargo, es un enfoque simple y adecuado para obtener soluciones en un tiempo aceptable para valores pequeños y medianos de N.

## 2.2.- Implementación en Python

En este apartado se presenta la implementación del algoritmo en Python, que incluye la definición del problema, sus restricciones y una explicación detallada del código. Este código utiliza funciones para verificar la seguridad de cada posición en el tablero y aprovechar la aleatoriedad en la selección de columnas.

### 2.2.1.- Definición del Problema y Restricciones

1. **Objetivo**: Colocar N reinas en un tablero de N×N de forma que ninguna reina pueda atacar a otra.
2. **Restricciones**:

    - Cada reina debe ocupar una fila y una columna única.
    - No puede haber reinas en la misma columna.
    - Ninguna reina debe estar en la misma diagonal (ascendente o descendente) que otra reina.

3. **Criterio de éxito**: El algoritmo debe encontrar una configuración en la que todas las reinas estén posicionadas sin conflictos, es decir, que cumpla con las restricciones anteriores.
4. **Reinicio**: Si en algún punto no es posible colocar una reina sin conflictos, el algoritmo reinicia la búsqueda desde la primera fila.

### 2.2.2.- Explicación Detallada del Código

```
import random
from collections import deque
```

El código importa el módulo random para generar números y selecciones aleatorias, y la clase deque del módulo collections para crear una cola de doble extremo que permite agregar y eliminar elementos eficientemente desde ambos extremos.

```
# Función para comprobar si es seguro colocar una reina en la posición dada
def es_seguro(tablero, fila, columna):
    for i in range(fila):
        if tablero[i] == columna or tablero[i] == columna - (fila - i) or tablero[i] == columna + (fila - i):
            return False
    return True
```

La función es_seguro comprueba si es seguro colocar una reina en una posición específica en el tablero verificando si hay reinas en la misma columna o en las diagonales que podrían atacarla, devolviendo False si hay un conflicto y True si es seguro.

```
# Función para resolver el problema de las N-Reinas usando búsqueda en anchura con aleatoriedad
def resolver_n_reinas_aleatorio(n):
    while True:  # Seguimos buscando hasta encontrar una solución
        tablero = []  # Estado actual del tablero
        fila_actual = 0  # Empezamos en la primera fila
        cola = deque([(fila_actual, tablero)])  # Cola para la BFS

        while cola:
            fila, estado_actual = cola.popleft()

            # Si todas las reinas se han colocado, hemos encontrado una solución
            if fila == n:
                return estado_actual

            # Generar columnas de manera aleatoria para intentar colocar la reina
            columnas_disponibles = list(range(n))
            random.shuffle(columnas_disponibles)  # Aleatorizar las columnas

            # Intentamos colocar la reina en cada columna disponible de forma aleatoria
            for columna in columnas_disponibles:
                if es_seguro(estado_actual, fila, columna):
                    nuevo_estado = estado_actual + [columna]
                    cola.append((fila + 1, nuevo_estado))  # Avanzamos a la siguiente fila
                    break  # Pasamos a la siguiente fila si colocamos una reina correctamente
            else:
                # Si no fue posible colocar una reina en ninguna columna, reiniciar búsqueda
                break
```

La función resolver_n_reinas_aleatorio implementa una búsqueda en anchura aleatoria para resolver el problema de las N reinas, intentando colocar reinas en filas del tablero de manera aleatoria hasta encontrar

una configuración válida que coloque todas las reinas sin conflictos, reiniciando la búsqueda si no se puede colocar una reina en ninguna columna disponible.

```
# Función para mostrar el tablero de forma gráfica con numeración de soluciones
def imprimir_solucion(solucion, n):
    print("Solución encontrada:\n")
    for fila in range(n):
        linea = ""
        for columna in range(n):
            if solucion[fila] == columna:
                linea += "Q "  # Representamos la reina con 'Q'
            else:
                linea += ". "  # Representamos espacios vacíos con '.'
        print(linea)
    print("\n" + "-" * (2 * n - 1) + "\n")
```

La función imprimir_solucion imprime de manera gráfica una solución al problema de las N reinas, representando cada reina con una 'Q' y los espacios vacíos con un '.', organizando las filas y columnas del tablero según la solución proporcionada.

```
# Solicitar el tamaño del tablero al usuario
try:
    n = int(input("Introduce el tamaño del tablero (n para n x n): "))
    if n <= 0:
        print("El tamaño del tablero debe ser un número positivo.")
    else:
        # Resolver el problema de las N-Reinas con el tamaño dado y aleatoriedad
        solucion = resolver_n_reinas_aleatorio(n)

        # Imprimir la solución encontrada
        imprimir_solucion(solucion, n)

except ValueError:
    print("Por favor, introduce un número entero válido.")
```

El código solicita al usuario el tamaño del tablero para el problema de las N reinas, valida la entrada asegurándose de que sea un número entero positivo, y si es válido, llama a la función para resolver el problema de manera aleatoria e imprime la solución encontrada; si la entrada no es válida, muestra un mensaje de error.

## 3.1.- Ejecución del Algoritmo con Diferentes Valores de N

El algoritmo se probó para diferentes valores de N, observando tanto el tiempo que tomó encontrar una solución válida como el número de intentos o reinicios necesarios en algunos casos para lograr la configuración correcta. Los valores de N usados incluyen:

- **N = 4**: Con este tamaño pequeño de tablero, el algoritmo encontró soluciones en pocos intentos y con alta rapidez, logrando una solución sin conflictos en prácticamente todos los casos de ejecución en un corto número de intentos.

```
Introduce el tamaño del tablero (n para n x n):  4
Solución encontrada:

. Q . .
. . . Q
Q . . .
. . Q .

-------
```

- **N = 8**: Este valor de N es más complejo, y aunque se encontraron soluciones, el número de intentos antes de encontrar una solución satisfactoria aumentó considerablemente. Los tiempos de ejecución permanecieron aceptables, pero en algunas ejecuciones fueron necesarios varios reinicios.

```
Introduce el tamaño del tablero (n para n x n):  8
Solución encontrada:

. . . . . Q
. . Q . . . .
Q . . . . . .
. . . . Q . .
. Q . . . . .
. . . Q . . .
. . . . . Q .
. . Q . . . .

---------------
```

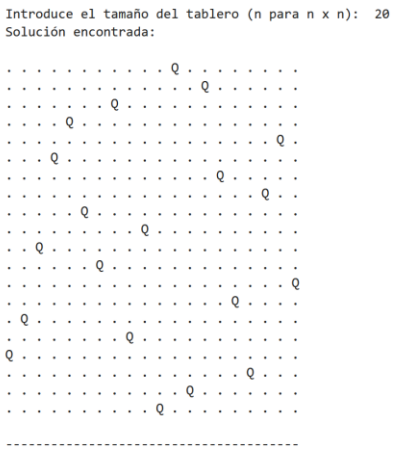
- **N = 16**: Para este tamaño, el algoritmo aleatorio empezó a mostrar limitaciones en eficiencia, con un número de reinicios que aumentó de manera notable. Los tiempos de ejecución fueron mayores y se observó que la variabilidad aleatoria produce en algunos casos tiempos significativamente más largos para hallar una solución sin conflictos.

```
Introduce el tamaño del tablero (n para n x n):  16
Solución encontrada:

. . . . . Q . . . . . .
. . . . . Q . . . . . .
. . . . . Q
. . . . . Q . . . . . .
Q . . . . . .
. . . . . Q . . . . . .
. . . . . Q . .
. . . . Q . . . . . .
. . . . . Q . . . . .
. . . Q . . . . . .
. . . . . Q . . . . . .
. . . . . Q . . .
. . Q . . . . . .
. . . . . Q . . . .
. . . . . Q .
. Q . . . . . .

------------------------------
```

- **N = 20** y superiores: A medida que N crece, la complejidad del problema aumenta considerablemente, y el enfoque aleatorio tiende a tener dificultades para encontrar soluciones en un tiempo razonable. En estos casos, el algoritmo tuvo que reiniciar muchas veces, generando un rendimiento menos confiable y requiriendo un tiempo considerable para alcanzar configuraciones válidas.




## 3.2.- Interpretación de los Resultados Obtenidos

El análisis de los resultados muestra cómo el algoritmo aleatorio proporciona soluciones viables para valores pequeños de N, mientras que su rendimiento se degrada con valores más altos. Esto se debe a la naturaleza probabilística del algoritmo: a medida que el número de reinas aumenta, el espacio de posibles configuraciones válidas se reduce, haciendo que el enfoque aleatorio sea menos eficiente.

1. **Eficiencia en el Tiempo**: Para valores bajos de N, el algoritmo aleatorio es rápido y encuentra soluciones de manera eficiente. Sin embargo, al incrementar el valor de N, el tiempo promedio para encontrar una solución aumenta exponencialmente, ya que el algoritmo debe reiniciar muchas veces en busca de una configuración libre de conflictos.
2. **Variabilidad en los Intentos**: Otro aspecto observado es la gran variabilidad en el número de intentos. En algunos casos, el algoritmo logra una solución en pocos reinicios; en otros, necesita un número considerable de intentos. Esto demuestra la naturaleza impredecible del enfoque aleatorio, lo que puede ser una limitación significativa en aplicaciones que requieren tiempos de ejecución consistentes.
3. **Comparación con Otros Algoritmos**: Comparado con el backtracking y la búsqueda en anchura, el enfoque aleatorio muestra ventajas en rapidez para tableros pequeños, donde ofrece una solución rápida y menos estructurada. No obstante, para valores de N más grandes, algoritmos como el backtracking, que siguen un proceso sistemático, suelen superar en eficiencia al método aleatorio al reducir la cantidad de intentos innecesarios.

### Conclusiones sobre los Resultados

A partir de los resultados obtenidos, se pueden concluir los siguientes puntos sobre el enfoque aleatorio:

- **Escalabilidad Limitada**: El método es práctico y eficiente para valores pequeños y medianos de N, donde puede encontrar soluciones sin conflictos en un tiempo razonable. Sin embargo, no es ideal para valores de N altos debido a la alta probabilidad de reinicios.
- **Utilidad en Soluciones Rápidas**: Para aplicaciones que no requieren una solución en un tiempo fijo o cuando el valor de N es pequeño, el enfoque aleatorio puede ser una opción rápida y fácil de implementar, proporcionando resultados con mínima complejidad.
- **Viabilidad contra Consistencia**: La aleatoriedad introduce variabilidad en el tiempo de ejecución. Aunque el algoritmo puede resolver el problema de las N-Reinas, la falta de consistencia en el tiempo para valores altos de N sugiere que el enfoque es menos viable para aplicaciones que requieran precisión en tiempos de ejecución o estabilidad en la cantidad de intentos.

Estos hallazgos demuestran que, si bien el enfoque aleatorio ofrece una alternativa innovadora, en ciertos casos la precisión y eficiencia de algoritmos sistemáticos como el backtracking o BFS son preferibles para asegurar un rendimiento confiable en el tiempo.

# 4.- Conclusión

La conclusión final de este trabajo examina el valor del enfoque aleatorio para resolver el problema de las N-Reinas, evaluando su desempeño, así como posibles mejoras y optimizaciones futuras.

## 4.1.- Reflexión sobre el Enfoque Aleatorio

El uso de aleatoriedad en el problema de las N-Reinas permite explorar configuraciones de soluciones sin seguir un patrón rígido, lo que lo convierte en un método atractivo para encontrar resultados en ciertos casos de manera rápida y con menor esfuerzo computacional. Este enfoque también introduce variabilidad, lo que significa que los

resultados no son siempre consistentes ni garantizan tiempos de ejecución predecibles.

El método aleatorio es eficaz para valores pequeños de N, ya que encuentra configuraciones válidas de manera más rápida. Sin embargo, al incrementar N, se evidencia su limitación para escalar, pues requiere cada vez más reinicios debido al espacio reducido de soluciones válidas. Esto sugiere que, aunque el enfoque es innovador y ofrece una alternativa distinta, no es la solución ideal para valores grandes de N donde la eficiencia y el tiempo de ejecución son críticos.

## 4.2.- Desempeño del Algoritmo

El desempeño del algoritmo aleatorio es considerablemente eficiente en el caso de tableros pequeños, donde rápidamente encuentra una configuración libre de conflictos sin necesidad de numerosos reinicios. Para valores de N más altos, sin embargo, el número de intentos y el tiempo para alcanzar una solución incrementan de manera no lineal, lo cual reduce su efectividad.

En comparación con algoritmos más estructurados, como el backtracking, el enfoque aleatorio carece de un método sistemático de reducción de opciones, lo que lo lleva a reiniciar el proceso con frecuencia al enfrentarse a situaciones en las que no puede continuar de manera viable. Por lo tanto, el algoritmo aleatorio es una opción interesante y sencilla de implementar, pero su rendimiento es inconstante en el contexto de grandes tableros.

## 4.3.- Posibles Mejoras y Optimización Futura

Para mejorar el rendimiento del algoritmo aleatorio, algunas posibles optimizaciones podrían incluir:

- **Selección Prioritaria de Columnas Válidas**: Implementar una evaluación preliminar de las columnas antes de realizar selecciones aleatorias podría reducir el número de intentos innecesarios y ayudar al algoritmo a evitar caminos que lleven a un reinicio.
- **Memorización de Configuraciones Inválidas**: Guardar configuraciones que resultaron en conflictos durante ejecuciones anteriores ayudaría al algoritmo a aprender de sus intentos previos, evitando repetir combinaciones problemáticas y reduciendo la cantidad de reinicios.
- **Algoritmo Híbrido**: Integrar pasos de backtracking en el enfoque aleatorio permitiría aprovechar la flexibilidad de ambos

métodos. Por ejemplo, el algoritmo podría utilizar aleatoriedad en las primeras filas y recurrir al backtracking cuando la búsqueda de configuraciones válidas se vuelva compleja, optimizando el tiempo de ejecución y la consistencia.

Estas mejoras y optimizaciones podrían hacer que el algoritmo sea más efectivo, especialmente para valores altos de N.

### Conclusión Final

El enfoque aleatorio para el problema de las N-Reinas resulta ser una técnica interesante y potencialmente útil para tableros pequeños, permitiendo hallar soluciones de forma rápida sin la estructura rígida de métodos tradicionales. Sin embargo, al crecer el tamaño del tablero, el número de reinicios aumenta de manera significativa, lo que limita la viabilidad del método para aplicaciones que requieren un rendimiento confiable y consistente.

A pesar de sus limitaciones, el enfoque aleatorio ofrece una perspectiva valiosa sobre la variabilidad y las alternativas en el diseño de algoritmos, mostrando que, en ciertos contextos, un método menos convencional puede proporcionar resultados satisfactorios. Con las mejoras y optimizaciones adecuadas, este método podría ser una opción complementaria interesante para resolver problemas similares en programación y optimización.

## 5.- Opinión Personal

El problema de las N-Reinas no solo representa un desafío interesante en el campo de la teoría de algoritmos, sino que también ofrece perspectivas útiles para problemas de optimización en programación. Resolver el problema de manera aleatoria aporta una alternativa creativa, aunque no necesariamente eficiente, a métodos como el backtracking o la búsqueda en anchura. En el desarrollo de este proyecto, he podido apreciar cómo la aleatoriedad introduce flexibilidad en la resolución de problemas y cómo se pueden aplicar estos conceptos en contextos donde la solución exacta no es necesariamente la meta principal.

## 5.1.- Impacto del Problema en la Programación y Optimización

El problema de las N-Reinas ha servido como ejemplo fundamental en programación, enseñando principios de resolución de problemas complejos mediante algoritmos específicos y enfoques como el backtracking y la búsqueda aleatoria. El impacto de este problema en la programación reside en su capacidad para ilustrar cómo abordar espacios de búsqueda amplios y resolver problemas de restricción.

Además, el uso de enfoques aleatorios en problemas de optimización permite desarrollar soluciones innovadoras en contextos donde los métodos tradicionales podrían ser poco viables. Esto se observa en aplicaciones modernas donde los algoritmos aleatorios se emplean para encontrar soluciones aproximadas en problemas de alto costo computacional, como el ajuste de parámetros en inteligencia artificial y la optimización de redes.

## 5.2.- Aplicaciones del Problema de las N-Reinas en el Mundo Real

Aunque el problema de las N-Reinas es, en su esencia, una cuestión académica, su estructura ha inspirado la resolución de problemas reales en diversos campos. Ejemplos incluyen:

- **Optimización de Recursos**: La lógica de colocar elementos sin conflicto es utilizada en el diseño de circuitos electrónicos, donde componentes como los transistores deben ser ubicados de manera que no interfieran entre ellos.
- **Programación de Horarios**: Se utiliza una lógica similar para la asignación de horarios en los que se eviten conflictos, como la organización de turnos de trabajo en empresas, en hospitales o la planificación de asignaturas en instituciones educativas.
- **Inteligencia Artificial**: En robótica y videojuegos, el concepto de encontrar ubicaciones óptimas sin conflictos es fundamental, especialmente en la toma de decisiones de agentes autónomos en ambientes con obstáculos.

En conjunto, el problema de las N-Reinas ha proporcionado un marco para explorar aplicaciones prácticas de optimización y diseño sin conflicto, siendo una referencia para problemas de distribución y asignación en múltiples disciplinas.

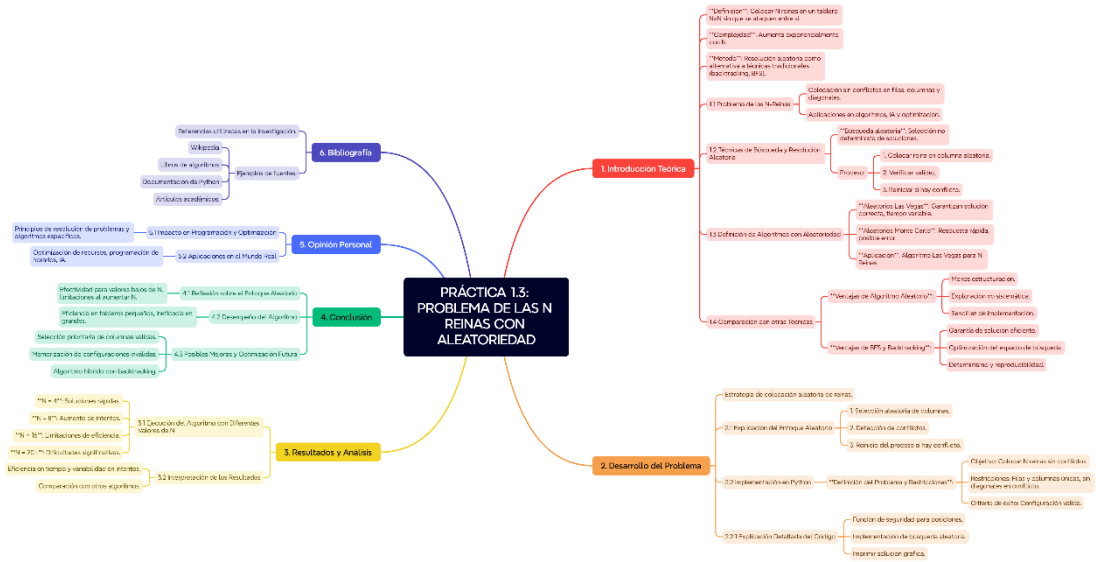
# 6.- Bibliografía

Para la elaboración de este trabajo, se ha hecho uso de diversas fuentes que ayudaron a entender la complejidad del problema de las N-Reinas y las distintas aproximaciones para resolverlo. Las referencias incluyen libros de algoritmos, documentación de Python y artículos sobre el uso de técnicas aleatorias en programación.

## 6.1.- Referencias Bibliográficas y Recursos Utilizados


- Algoritmo de Las Vegas. En Wikipedia. Recuperado de https://es.wikipedia.org/wiki/Algoritmo_de_Las_Vegas
- Método de Montecarlo. En Wikipedia. Recuperado de https://es.wikipedia.org/wiki/Método_de_Montecarlo
- Algoritmo determinista. En Wikipedia. Recuperado de https://es.wikipedia.org/wiki/Algoritmo_determinista
- Búsqueda en anchura. En Wikipedia. Recuperado de https://es.wikipedia.org/wiki/B%C3%BAsqueda_en_anchura
- Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
- Sedgewick, R., & Wayne, K. (2011). *Algorithms*. Addison-Wesley Professional.
- Python Programming Documentation. Python Software Foundation. Disponible en: https://docs.python.org
- Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach*. Pearson Education.
- Artículos sobre técnicas de algoritmos aleatorios en repositorios de programación y sitios de divulgación académica.

# 7. Esquema Resumido




Este mapa mental resume la estructura del Problema de las N-Reinas, abarcando la introducción teórica, técnicas de búsqueda aleatoria, comparaciones con métodos deterministas, desarrollo de un algoritmo en Python y su aplicabilidad en programación y optimización.

# Índice Alfabético

## *S*



## *T*



## *U*



## *V*



## *W*