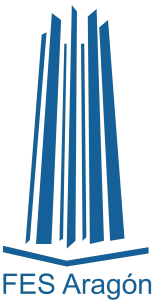

Universidad Nacional Autónoma de México

Facultad de Estudios Superiores Aragón

Diseño y Análisis de Algoritmos

Semestre 2026-1

Documentación del Proyecto Final

Algoritmos de Fuerza Bruta y Geometría Computacional

Presenta:

Enrique Emiliano Cano García

Fecha:

26 de noviembre de 2025

Índice

1. Introducción

El documento constituye la memoria técnica del Proyecto Final de la asignatura *Diseño y Análisis de Algoritmos*. El objetivo central de este trabajo no se limita a la resolución funcional de problemas computacionales, sino que se enfoca en el análisis crítico de la eficiencia algorítmica y la aplicación de paradigmas avanzados de programación para la optimización de recursos.

1.1. Metodología de Trabajo

Para el desarrollo de las soluciones aquí presentadas, se siguió una metodología rigurosa dividida en cuatro fases estratégicas, garantizando así la robustez y optimalidad del software entregado:

1. **Análisis y Modelado Matemático:** En esta etapa inicial, se estudiaron las restricciones de cada problema. Para el *Problema del Robot*, se modeló el espacio de búsqueda como un grafo implícito sobre una matriz de $N \times M$. Para el *Problema de los Postes*, se identificaron las propiedades geométricas fundamentales, específicamente la relación entre el área máxima de un polígono y su frontera convexa.

2. **Implementación Base (Benchmarking):** Se desarrollaron primeras versiones utilizando enfoques de *Fuerza Bruta* y recursividad simple. El propósito de esta fase fue establecer una "línea base"de corrección para validar los resultados posteriores y evidenciar las limitaciones de rendimiento (complejidades de orden cúbico $O(n^3)$ o exponencial) inherentes a los algoritmos ingenuos.

3. **Refinamiento y Optimización:** Identificados los cuellos de botella, se procedió a la reintegración de los algoritmos utilizando técnicas aprendidas a lo largo del curso e investigaciones realizadas con el fin de la correcta implementación:

    - **Programación Dinámica (Memoización):** Se aplicó para eliminar el recálculo de subproblemas superpuestos en la navegación del robot, transformando una complejidad exponencial en lineal. Visto y utilizado en *programación dinámica*
    - **Geometría Computacional (Graham Scan):** Se implementó el cálculo del Casco Convexo para reducir drásticamente el espacio de búsqueda en el problema de triangulación, pasando de un enfoque combinatorio puro a uno geométrico eficiente.

4. **Validación Asintótica:** Finalmente, se realizó el análisis teórico de la complejidad temporal (*Notación Big-O*) para demostrar matemáticamente la superioridad de las versiones optimizadas frente a las implementaciones base.

A lo largo de este documento, se detallan las implementaciones en lenguaje Python, acompañadas de sus respectivos análisis de complejidad y las justificaciones teóricas que avalan las decisiones de diseño tomadas durante el desarrollo del proyecto.

2. Problema 1: Navegación del Robot

2.1. Descripción del Problema

Se plantea un escenario donde un robot debe desplazarse en una cuadrícula de dimensiones $N \times M$, partiendo de la coordenada $(0, 0)$ hasta el objetivo $(N-1, M-1)$. El robot tiene movimientos restringidos: solo puede avanzar mediante los vectores de traslación $(+2, 0)$ (hacia abajo) y $(0, +3)$ (hacia la derecha).

2.2. Enfoque de Fuerza Bruta (Backtracking)

La primera aproximación para resolver este problema consiste en explorar exhaustivamente todas las rutas posibles mediante recursividad simple. Este enfoque genera un árbol de decisión donde se calculan repetidamente los mismos estados.

A continuación, se muestra la implementación que busca todos los caminos posibles sin almacenar estados previos:

```python
def todos_caminos_robot(n, m):
    target = (n - 1, m - 1)
    soluciones = []

    def buscar(r, c, actual_path):
        # 1. Caso base: Fuera de limites
        if r >= n or c >= m:
            return

        actual_path = actual_path + [(r, c)]

        # 2. Caso base: Exito
        if r == target[0] and c == target[1]:
            soluciones.append(actual_path)
            return

        # 3. Paso recursivo: explorar opciones
        buscar(r + 2, c, actual_path) # Abajo
        buscar(r, c + 3, actual_path) # Derecha

    buscar(0, 0, [])
    return soluciones
```

2.3. Justificación y Optimización (Memoización)

Análisis de Ineficiencia: El enfoque de fuerza bruta posee una complejidad exponencial debido a la superposición de subproblemas. El mismo punto (r, c) puede ser alcanzado por múltiples rutas, lo que provoca que el algoritmo recalcular su viabilidad innecesariamente.

Solución Optimizada: Se implementó *Programación Dinámica* con la técnica de **Memoización**. Se utiliza un diccionario (`memo`) para almacenar si una coordenada ya fue visitada y si conduce al objetivo. Esto reduce la complejidad a $O(N \times M)$, ya que cada celda se procesa una única vez.

```python
def todos_caminos_robot(n, m):
    """
    Encuentra todos los caminos posibles desde (0,0) hasta (n-1, m-1) usando backtracking.
    :param n: Numero de filas de la cuadricula
    :param m: Numero de columnas de la cuadricula
    :return: Lista de listas, donde cada lista interna representa un camino como una secuencia
             de coordenadas (tuples) desde el inicio hasta el destino.
    """
    target = (n - 1, m - 1)
    soluciones = []

    def buscar(r, c, actual_path):
        #1. Caso base: Fracaso(fuera de limites)
        if r >= n or c >= m:
            return # Esta rama no lleva a una solucion

        #2. A adir la casilla actual al camino que estamos explorando
        actual_path = actual_path + [(r, c)]

        #3. Caso base: Exito
        if r == target[0] and c == target[1]:
            soluciones.append(actual_path) #Se encontro el camino
            return #Termina esta rama

        #4. Paso recursivo: explorar las dos opciones

        # Opcion 1: moverse hacia abajo
        buscar(r + 2, c, actual_path)

        # Opcion 2: moverse hacia la derecha
        buscar(r, c + 3, actual_path)

    buscar(0, 0, [])
    return soluciones
```

3. Problema 2: Triángulo de Área Máxima

3.1. Descripción del Problema

Dado un conjunto de puntos P (postes) en un plano cartesiano, se busca encontrar el triplete de puntos (p_1, p_2, p_3) que forme el triángulo con la mayor área posible.

3.2. Enfoque de Fuerza Bruta

La solución intuitiva consiste en iterar sobre todas las combinaciones posibles de tres puntos, calcular su área y conservar el máximo. Aunque garantiza la solución óptima, es computacionalmente costosa.

```python
def resolver_fuerza_bruta(postes):
    """
    Resuelve el problema de encontrar el triangulo de area maxima
       usando fuerza bruta.
    :param postes: Lista de tuplas (x, y) que representan los postes
    :return: Lista de tuplas (x, y) que forman el triangulo de area
       maxima
    """
    n = len(postes)
    if n < 3:
        print("No hay suficientes postes para formar un triangulo.")
        return None

    max_area_doble = 0
    postes_max = []

    #Tres bucles anidados para considerar todas las combinaciones de
       3 postes
    for i in range (n):
        for j in range(i + 1, n):
            for k in range(j + 1, n):
                p1 = postes[i]
                p2 = postes[j]
                p3 = postes[k]

                area_actual_doble = calcular_doble_area(p1, p2, p3)

                if area_actual_doble > max_area_doble:
                    max_area_doble = area_actual_doble
                    postes_max = [p1, p2, p3]
    return postes_max
```

3.3. Justificación y Optimización (Graham Scan)

Análisis de Ineficiencia: La complejidad $O(n^3)$ hace que el algoritmo sea inviable para grandes volúmenes de datos. Por ejemplo, con 1,000 puntos, se realizarían aproximadamente 166 millones de operaciones.

Solución Optimizada: Se utiliza el algoritmo de **Graham Scan** para calcular el *Casco Convexo* (Convex Hull). La justificación teórica se basa en que los vértices del triángulo de área máxima deben pertenecer a la frontera convexa del conjunto. Esto reduce el espacio de

búsqueda de n puntos a h (puntos en el casco), donde generalmente $h \ll n$. La complejidad dominante pasa a ser la de ordenamiento: $O(n \log n)$.

```
def resolver_optimizado(postes):
    """
    Encuentra el tri ngulo de  rea  m xima usando el casco convexo

    Algoritmos:
    1. Calcular el hull: O(n log n)
    2. Probar todas las combinaciones de 3 puntos que pertenezcan al
       hull: O(h^3), donde 'h' es el n mero de puntos en el hull

    Complejidad total: O(n log n + h^3) donde h <= n tipicamente h <<
        n para 500 puntos, 'h' puede ser,
    ~20-50 puntos -> mucho mas rapido y eficiente que O(n^3)
    :param postes: Lista de tuplas (x, y) que representan los postes
    :return: Lista de tuplas (x, y) que forman el triangulo de area
       maxima
    """

    n = len(postes)
    if n < 3:
        print("No hay suficientes postes para formar un triangulo.")
        return None
```

```
##Paso 1: Calcular el casco convexo
    hull = convex_hull(postes)
    h = len(hull)

    print(f"Puntos totales: {n}")
    print(f"Puntos en el hull convexo: {h}")
    print(f"Reduccion: {n} -> {h} ({100 * h / n:.1f}%)")

    #Paso 2: Encontrar el triangulo de area m xima dentro del hull
    max_area_doble = 0
    postes_max = []
    for i in range(h):
        for j in range(i + 1, h):
            for k in range(j + 1, h):
                p1 = hull[i]
                p2 = hull[j]
                p3 = hull[k]

                area_actual_doble = calcular_doble_area(p1, p2, p3)

                if area_actual_doble > max_area_doble:
                    max_area_doble = area_actual_doble
                    postes_max = [p1, p2, p3]
    return postes_max
```

4. Justificación de las Soluciones

En este proyecto se resolvieron dos problemas independientes utilizando paradigmas algorítmicos distintos: Programación Dinámica/Backtracking para la navegación de un robot y Geometría Computacional para optimización de áreas.

4.1. Problema 1: Caminos del Robot

Para el problema del desplazamiento del robot en una cuadrícula de $N \times M$ con movimientos restringidos (çaballo de ajedrez modificado"), se desarrollaron dos variantes:

- **Verificación de Existencia (Memoización):** Para determinar si existe un camino viable sin recorrer rutas repetidas innecesariamente, se utilizó **Programación Dinámica** con la técnica de *Memoización*. **Justificación:** La recursión simple recalcula los

mismos estados (r, c) múltiples veces, llevando a una complejidad exponencial. Al almacenar los resultados en un diccionario (`memo`), reducimos la complejidad temporal a $O(N \times M)$, ya que cada celda se procesa una única vez.

- **Búsqueda de Todos los Caminos (Backtracking):** Para listar todas las rutas posibles, se empleó un algoritmo de **Backtracking** puro. **Justificación:** A diferencia de la verificación, aquí es necesario explorar exhaustivamente todas las ramas del árbol de decisión para construir las rutas completas. Se utiliza una lista temporal para almacenar el camino actual, haciendo "backtrack"(retroceso) al encontrar un límite o un destino.

4.2. Problema 2: Triángulo de Área Máxima

Para encontrar el triángulo de mayor área en una nube de puntos (postes), se contrastaron dos métodos:

- **Fuerza Bruta:** Se evaluaron todas las combinaciones de tres puntos (p_1, p_2, p_3), con una complejidad de $O(n^3)$. Este método asegura la optimalidad pero es ineficiente para n grande.
- **Optimización (Graham Scan):** Se calculó primero el Casco Convexo (*Convex Hull*). La justificación teórica reside en que los vértices del triángulo máximo siempre pertenecen al perímetro del casco convexo. Esto reduce el espacio de búsqueda drásticamente, bajando la complejidad global a $O(n \log n)$.

5. Conclusión General y Aprendizajes

El desarrollo integral de este proyecto me ha permitido extraer conclusiones fundamentales sobre la creación de algoritmos, yendo más allá de la simple codificación para adentrarse en el diseño eficiente de software.

5.1. Uso Estratégico de Estructuras de Datos

Uno de los aprendizajes clave fue cómo la elección de la estructura de datos correcta puede transformar la viabilidad de una solución:

- **Diccionarios (Hash Maps) en Programación Dinámica:** En el problema del robot, el uso de un diccionario para la memoización fue crítico. Permitió transformar

un algoritmo de tiempo exponencial en uno de tiempo lineal, demostrando que el intercambio de memoria (espacio) por velocidad (tiempo) es una estrategia poderosa.

- **Pilas y Listas en Geometría:** Para el algoritmo de Graham Scan, la manipulación eficiente de listas y pilas para construir el casco convexo evidenció la importancia de preprocesar los datos (ordenamiento previo) para simplificar los cálculos geométricos subsiguientes.

5.2. Lecciones del Proyecto

Este trabajo consolida la idea de que la **Fuerza Bruta** es útil únicamente como herramienta de validación o para instancias triviales del problema. Sin embargo, para aplicaciones reales y escalables, es imperativo realizar un análisis asintótico previo.

Se aprendió que la optimización no siempre proviene de escribir "mejor código"línea por línea, sino de cambiar fundamentalmente el enfoque matemático del problema: ya sea evitando el re-trabajo (Programación Dinámica) o descartando datos irrelevantes mediante propiedades teóricas (Geometría Computacional). En conclusión, la eficiencia del software se decide en la etapa de diseño, mucho antes de escribir la primera línea de código.

6. Referencias Bibliográficas

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3a ed.). MIT Press. [Capítulos 15 y 33: Dynamic Programming & Computational Geometry].
- Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. (2008). *Algorithms*. McGraw-Hill Higher Education. [Sección 6.2: Memoization].
- Graham, R. L. (1972). An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4), 132-133.
- Preparata, F. P., & Shamos, M. I. (1985). *Computational Geometry: An Introduction*. Springer-Verlag.