



**Tecnológico
de Monterrey**

Campus Monterrey

Actividad Complementaria Party Rock: Equipo Greedy

Juan Eduardo Cibrián Loera A01198418

Rodrigo López Murguía A01742780

Víctor Misael Escalante Alvarado A01741176

María Valentina Narváez Terán

TC2038 Análisis y diseño de algoritmos avanzados

Grupo 605

24 de noviembre del 2024

Enlace a aplicación :

[\[Enlace Party-rock\]](#)

Fase 1: Investigación asistida por IA.

Usando ChatGPT con el siguiente prompt:

“Dame información del tema de los algoritmos greedy, de conceptos, personas relevantes, ejemplos y partes claves para que un algoritmo se convierta en greedy?”

Los resultados fueron los siguientes:

Los **algoritmos greedy** (o voraces) son una técnica de diseño de algoritmos que toma decisiones paso a paso, eligiendo en cada paso la opción que parece más prometedora o beneficiosa en ese momento, con la esperanza de que esta estrategia lleve a una solución óptima global. Aunque no siempre garantizan la solución óptima, son útiles cuando la elección local conduce de manera natural a la mejor solución global.

Conceptos clave

1. **Estrategia localmente óptima:** Cada paso del algoritmo toma la mejor decisión posible según el criterio definido, sin preocuparse por las implicaciones futuras.
2. **Eficiencia:** Los algoritmos greedy suelen ser más simples y rápidos que otros enfoques como la programación dinámica, ya que no necesitan retroceder o explorar todas las posibilidades.
3. **Optimalidad global:** Los algoritmos greedy funcionan correctamente solo si el problema satisface ciertas propiedades que garantizan que una solución construida paso a paso también sea la mejor solución global.



Partes clave de un algoritmo greedy

Para que un algoritmo greedy funcione, el problema debe cumplir dos propiedades fundamentales:

1. **Propiedad greedy (o de elección local):** La solución global óptima puede ser construida eligiendo decisiones óptimas locales en cada paso.
2. **Propiedad de subestructura óptima:** La solución óptima del problema puede ser construida a partir de soluciones óptimas de subproblemas más pequeños.

Personas relevantes

1. **Joseph Kruskal y Robert Prim:** Conocidos por sus contribuciones al diseño de algoritmos greedy para árboles de expansión mínima.
2. **George Dantzig:** Uno de los pioneros en el diseño de algoritmos greedy, y creador del método simplex para programación lineal.
3. **Jon Bentley:** Autor del libro clásico *Programming Pearls*, que aborda problemas resueltos con técnicas greedy y sus principios.

Ventajas y desventajas

Ventajas:

- **Simplicidad:** Son fáciles de implementar y entender.
- **Velocidad:** Tienden a ser rápidos debido a la ausencia de retroceso o exploración exhaustiva.

Desventajas:

- **Optimalidad no garantizada:** No siempre encuentran la mejor solución global.
- **Limitaciones:** Solo funcionan en problemas que cumplen las propiedades greedy y de subestructura óptima.

Comparativa con otros paradigmas

1. **Programación dinámica:** Resuelve problemas dividiéndolos en subproblemas y almacenando soluciones intermedias. Es más poderosa que la técnica greedy pero menos eficiente.
2. **Backtracking:** Explora todas las posibilidades, retrocediendo si una rama no lleva a una solución.

Un algoritmo greedy es ideal si el problema tiene propiedades que permitan una solución global óptima mediante decisiones locales. Si no, otras técnicas como programación dinámica podrían ser mejores.

Fase 2: Diseño de la aplicación.

Prompts utilizados:

Chatbot:

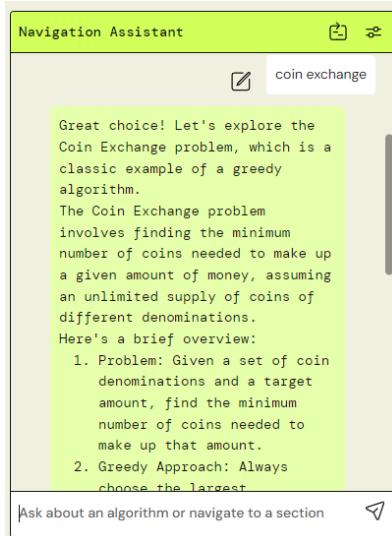
- Hello! I'm here to learn about greedy algorithms. Can you guide me through the app?

Input: Coin Exchange

TextGeneration:

- Provide a detailed explanation of the greedy algorithm mentioned in [Navigation Assistant]. Include its purpose, how it works, and a step-by-step breakdown of the algorithm. If no specific algorithm is mentioned, give an overview of greedy algorithms in general.

Resultado:



- Generate a simple coding problem which its solution is findable by the greedy algorithm mentioned in [Algorithm Explanation] . Make sure to include an example input and an example output

Resultado:

Try it out

Certainly! Here's a simple coding problem that can be solved using a greedy algorithm:

Problem: Coin Change

You are given an infinite supply of coins of different denominations and a target amount of money. Your task is to find the minimum number of coins needed to make up the target amount.

Input:

- An array of integers representing the available coin denominations (sorted in descending order)
- A target amount of money

Output:

- The minimum number of coins needed to make up the target amount

Example:

```
Input: Coin denominations: [25, 10, 5, 1] Target amount: 63
```

- Evaluate the code solution provided in [Code Solution Input] for the greedy algorithm problem. If no specific problem is mentioned, assume it's for the algorithm explained in [Algorithm Explanation]. Provide feedback on correctness, efficiency, and potential improvements.

Resultado:

Solution Evaluation (Generating)

Thank you for providing the code for the coin exchange greedy algorithm. Let's evaluate it in terms of correctness, efficiency, and potential improvements:

Correctness: The algorithm is correctly implemented for the greedy approach to the coin exchange problem. It:

- Sorts the coins in descending order.
- Iterates through the coins, using the largest possible coin at each step.
- Returns the list of coins used if an exact change is possible, or None otherwise.

Efficiency: The algorithm is relatively efficient:

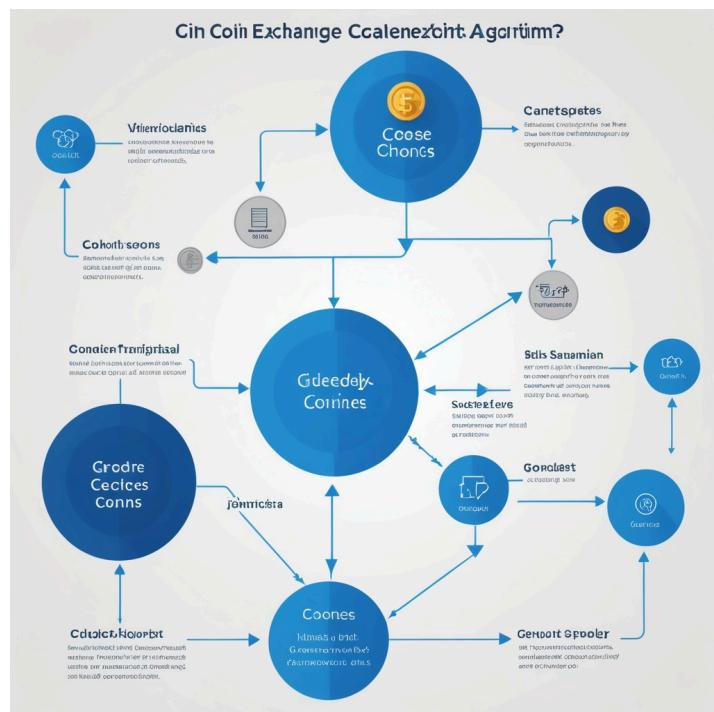
- Time complexity: $O(n \log n + m)$, where n is the number of coin denominations (due to sorting) and m is the input amount.
- Space complexity: $O(m)$ in the worst case, where m is the input amount (if all coins are of value 1).

Image Generation:

- Create a visual representation or flowchart of the greedy algorithm explained in [Algorithm Explanation] in english. If no specific algorithm

is mentioned, create a general flowchart of how greedy algorithms work in english.

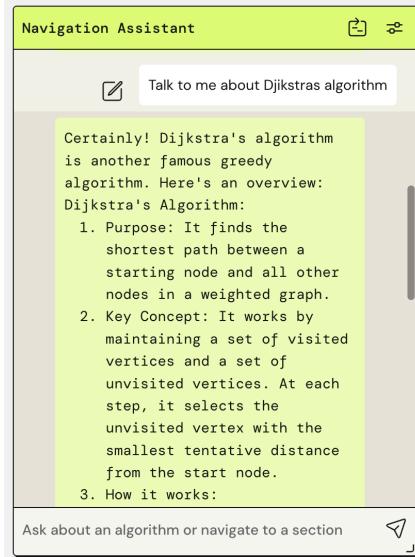
Resultado:



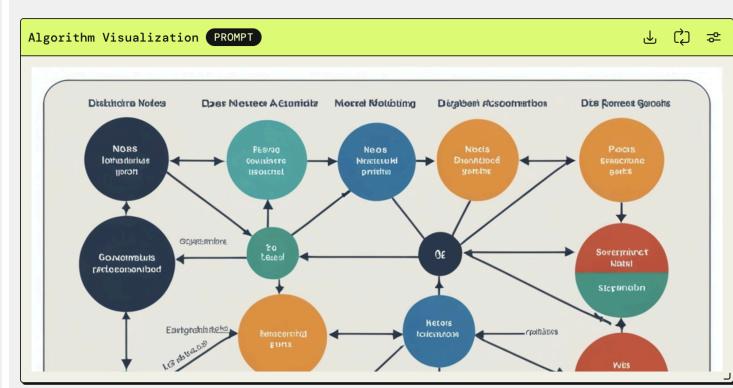
Fase 3: Evaluación de la aplicación creada.

Primer caso : *Talk to me about Dijkstras algorithm*

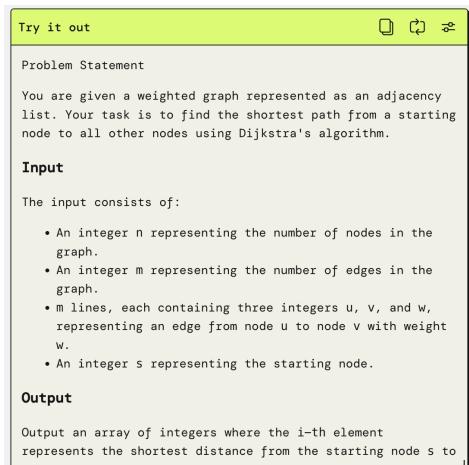
Navigation assistant



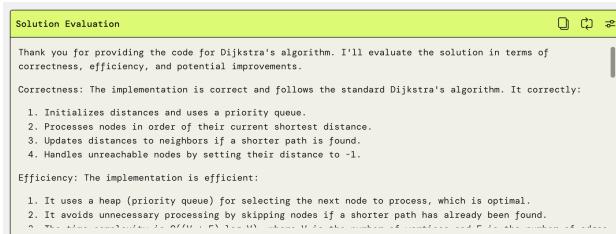
Algorithm visualization



Try it out



Solution Evaluation



Segundo caso: *Talk to me about min spanning tree*

Navigation Assistant

Navigation Assistant

Certainly! Minimum Spanning Tree (MST) is an important concept in graph theory and is often solved using greedy algorithms. Here's an overview:

1. Definition: A minimum spanning tree is a subset of edges in an undirected, weighted graph that connects all vertices together with the minimum total edge weight, without creating any cycles.
2. Properties:
 - It spans all vertices.
 - It has $(n-1)$ edges.

Algorithm Explanation

Algorithm Explanation

Since no specific algorithm was mentioned in the original query, I'll provide a detailed explanation of Kruskal's algorithm, which is a greedy algorithm commonly used for finding the Minimum Spanning Tree.

Kruskal's Algorithm:

Purpose: Kruskal's algorithm is used to find the Minimum Spanning Tree (MST) of a weighted, undirected graph. It aims to connect all vertices with the minimum total edge weight, without creating cycles.

How it works: The algorithm builds the MST by iteratively adding the smallest edge that doesn't create a cycle with the edges already included in the MST.

Step-by-step breakdown:

1. Sort all edges of the graph in non-decreasing order of their weight.
2. Initialize an empty set (or forest) to store the MST edges.

Algorithm Explanation



3. For each edge in the sorted list, starting from the smallest weight:
 - a. Check if adding this edge creates a cycle with the edges already in the MST.
 - b. If no cycle is formed, add the edge to the MST.
 - c. If a cycle would be formed, skip this edge and move to the next one.
4. Repeat step 3 until the MST has $(n-1)$ edges, where n is the number of vertices.
5. The resulting set of edges forms the Minimum Spanning Tree.

To efficiently check for cycles, Kruskal's algorithm typically uses a disjoint-set data structure (also known as a union-find data structure).

Example: Consider a graph with vertices A, B, C, D and edges: (A-B, weight 4), (B-C, weight 2), (C-D, weight 3), (D-A, weight 1), (B-D, weight 5)

Steps:

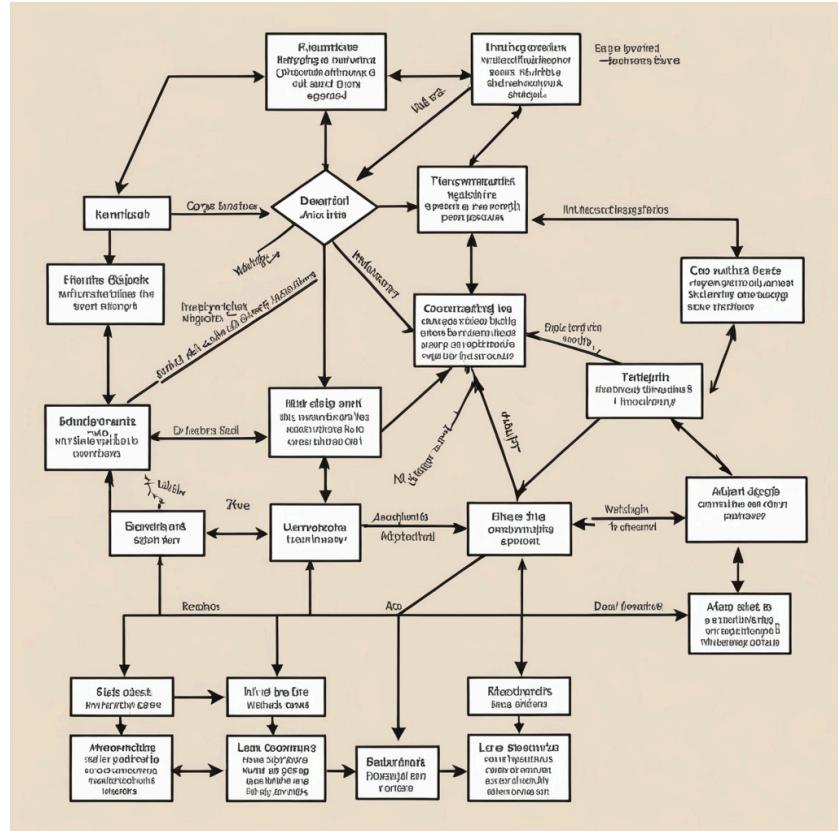
Steps:

1. Sort edges: D-A(1), B-C(2), C-D(3), A-B(4), B-D(5)
2. Start with an empty MST
3. Add D-A(1) to MST
4. Add B-C(2) to MST
5. Add C-D(3) to MST
6. Skip A-B(4) and B-D(5) as they would create cycles

The resulting MST consists of edges: D-A, B-C, and C-D, with a total weight of 6.

Kruskal's algorithm is efficient and easy to implement, with a time complexity of $O(E \log E)$ or $O(E \log V)$, where E is the number of edges and V is the number of vertices.

Algorithm visualization



Try it out:

Try it out



! Here's a simple coding problem that can be solved using a greedy algorithm, specifically Kruskal's algorithm for finding the Minimum Spanning Tree (MST):

Problem Statement

You are given an undirected, weighted graph with n vertices and m edges. Your task is to find the Minimum Spanning Tree (MST) of the graph. The MST is a subset of the edges that connects all vertices together, without any cycles, and with the minimum possible total edge weight.

Input

The first line of input contains two integers n and m , representing the number of vertices and the number of edges, respectively.

The next m lines each contain three integers u , v , and w , representing an edge between vertices u and v with weight w .

Output

Output the total weight of the Minimum Spanning Tree.

Example

Input

```
1 4 5
2 0 1 4
3 1 2 2
4 2 3 3
5 3 0 1
6 1 3 5
```

Output

1 6

**Explanation**

The graph has 4 vertices and 5 edges. The edges are:

- (0, 1) with weight 4
- (1, 2) with weight 2
- (2, 3) with weight 3
- (3, 0) with weight 1
- (1, 3) with weight 5

Using Kruskal's algorithm, the MST will include the edges (3, 0) with weight 1, (1, 2) with weight 2, and (2, 3) with weight 3. The total weight of the MST is $1 + 2 + 3 = 6$.

Solution using Kruskal's Algorithm

Here's a Python implementation of Kruskal's algorithm to solve the problem:

```
1 class UnionFind:
2     def __init__(self, n):
3         self.parent = list(range(n))
4         self.rank = [0] * n
5
6     def find(self, u):
7         if self.parent[u] != u:
8             self.parent[u] = self.find(self.parent[u])
9         return self.parent[u]
10
11    def union(self, u, v):
12        root_u = self.find(u)
13        root_v = self.find(v)
14        if root_u != root_v:
15            if self.rank[root_u] > self.rank[root_v]:
16                self.parent[root_v] = root_u
22
23    def kruskal(self, edges):
24        edges.sort(key=lambda x: x[2]) # Sort edges
25        uf = UnionFind(n)
26        mst_weight = 0
27        mst_edges = 0
28
29        for u, v, w in edges:
30            if uf.find(u) != uf.find(v):
31                uf.union(u, v)
32                mst_weight += w
33                mst_edges += 1
34            if mst_edges == n - 1:
35                break
36
37        return mst_weight
38
```

```

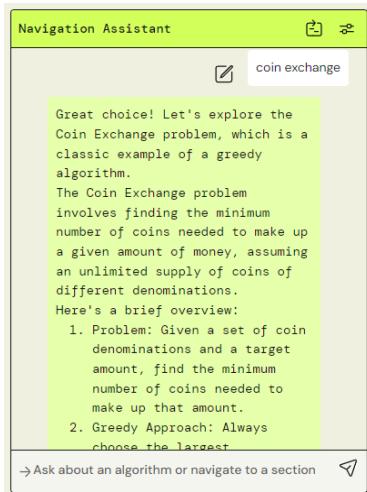
39 # Example usage
40 n, m = 4, 5
41 edges = [
42     (0, 1, 4),
43     (1, 2, 2),
44     (2, 3, 3),
45     (3, 0, 1),
46     (1, 3, 5)
47 ]
48
49 print(kruskal(n, edges)) # Output: 6

```

This implementation uses the Union-Find data structure to efficiently check for cycles and merge sets. The edges are sorted by weight, and the algorithm iteratively adds the smallest edge that doesn't form a cycle to the MST. The total weight of the MST is then returned.

Tercer caso: *Coin Exchange*

Navigation Assistant



Algorithm Explanation

Algorithm Explanation



The algorithm terminates with a total of 6 coins used: 2x25c, 1x10c, 1x5c, 2x1c.

Advantages of the Greedy Approach:

1. Simple to implement and understand.
2. Generally fast, with a time complexity of $O(n)$ where n is the number of coin denominations.
3. Produces optimal results for standard coin systems (like US coins).

Limitations:

1. May not always produce the optimal solution for all coin systems.
2. Assumes an unlimited supply of each coin denomination.

In summary, the coin exchange greedy algorithm provides a straightforward and efficient method for solving the coin change problem in many practical scenarios, making it a popular choice for this type of problem.

Code revision

Solution Evaluation



Thank you for providing the code for the coin exchange greedy algorithm. Let's evaluate it in terms of correctness, efficiency, and potential improvements:

Correctness: The algorithm is correctly implemented for the greedy approach to the coin exchange problem. It:

1. Sorts the coins in descending order.
2. Iterates through the coins, using the largest possible coin at each step.
3. Returns the list of coins used if an exact change is possible, or None otherwise.

Efficiency: The algorithm is relatively efficient:

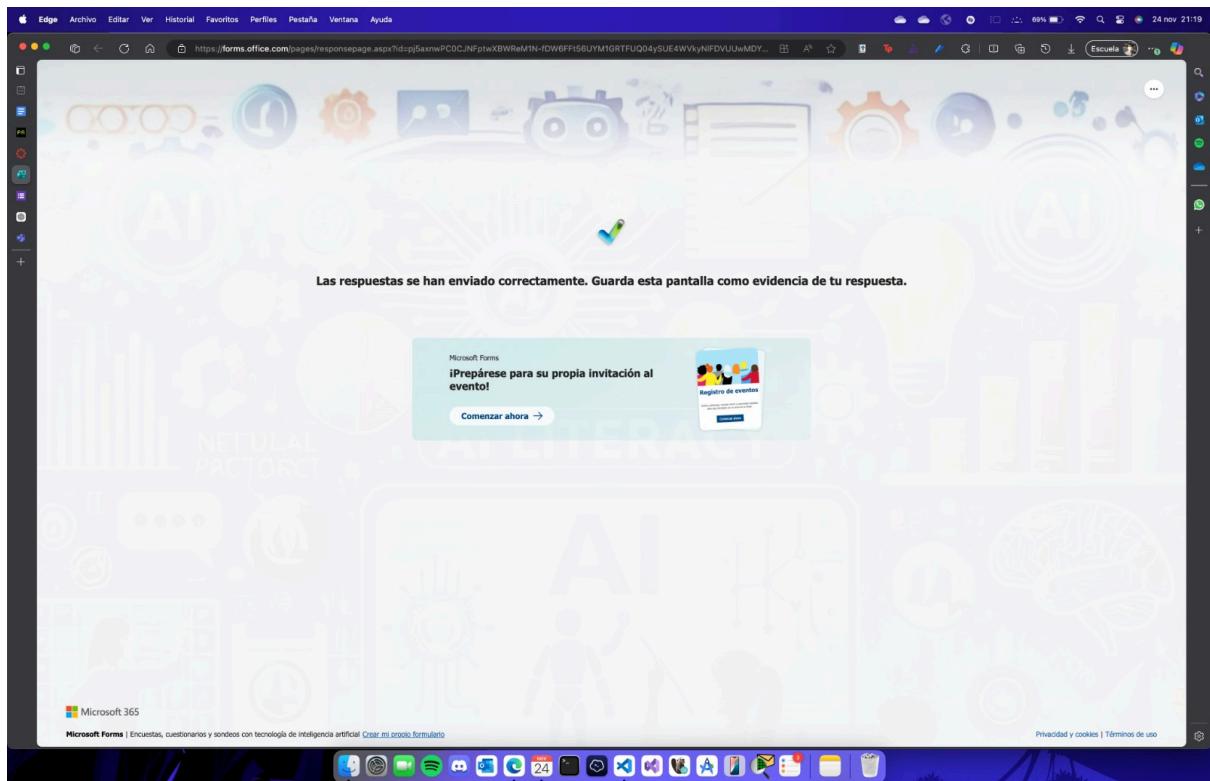
1. Time complexity: $O(n \log n + m)$, where n is the number of coin denominations (due to sorting) and m is the input amount.
2. Space complexity: $O(m)$ in the worst case, where m is the input amount (if all coins are of value 1).

Generated Image

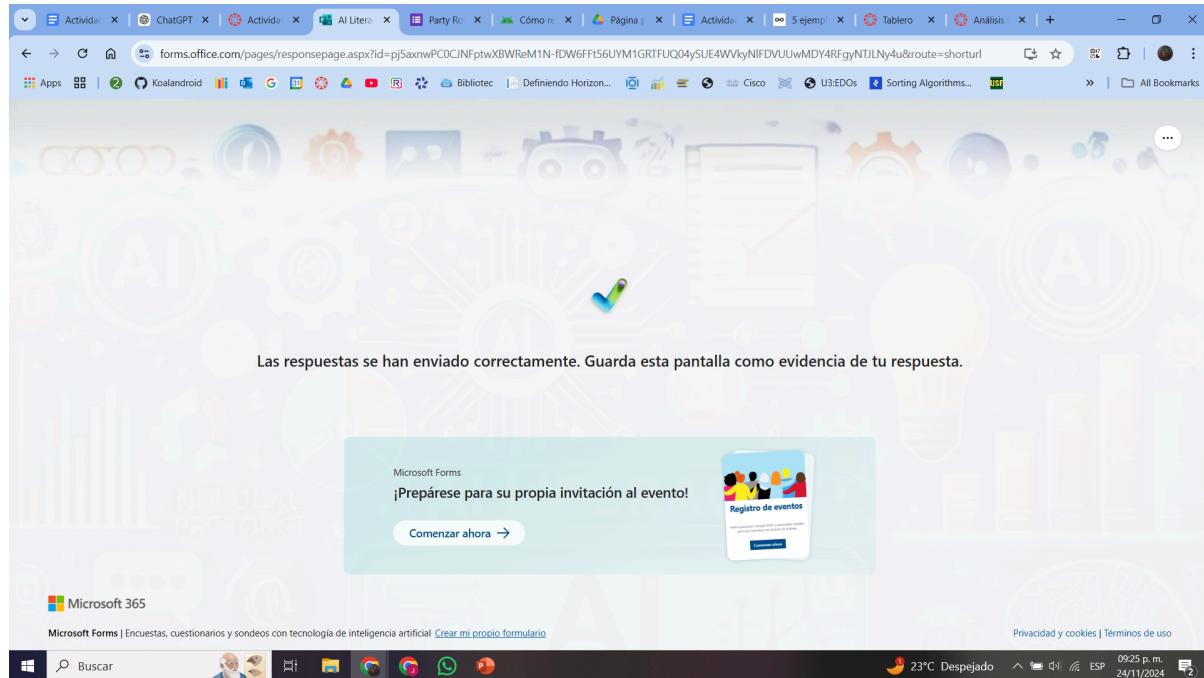


Experiencia de aprendizaje

Víctor Misael Escalante Alvarado



Juan Eduardo Cibrián Loera



Rodrigo López Murguía

