



TEORÍA DE ALGORITMOS  
(TB024) CURSO BUCHWALD - GENENDER

# Trabajo Práctico 3

## Problemas NP-Completo para la defensa de la Tribu del Agua



13 de noviembre de 2025

Alen Davies  
107.084

Hugo Huzan  
67.910

Joaquin Vedoya  
110.282

Franco Altieri Lamas  
105.400

## Trabajo Práctico 3: Problemas NP-Completo para la defensa de la Tribu del Agua

### 1. Introducción

Es el año 95 DG. La Nación del Fuego sigue su ataque, esta vez hacia la Tribu del Agua, luego de una humillante derrota a manos del Reino de la Tierra, gracias a nuestra ayuda. La tribu debe defenderse del ataque.

El maestro Pakku ha recomendado hacer lo siguiente: Separar a todos los Maestros Agua en  $k$  grupos  $(S_1, S_2, \dots, S_k)$ . Primero atacará el primer grupo. A medida que el primer grupo se vaya cansando entrará el segundo grupo. Luego entrará el tercero, y de esta manera se busca generar un ataque constante, que sumado a la ventaja del agua por sobre el fuego, buscará lograr la victoria.

En función de esto, lo más conveniente es que los grupos estén parejos para que, justamente, ese ataque se mantenga constante.

Conocemos la fuerza/maestría/habilidad de cada uno de los maestros agua, la cual podemos cuantificar diciendo que para el maestro  $i$  ese valor es  $x_i$ , y tenemos todos los valores  $x_1, x_2, \dots, x_n$  (todos valores positivos).

Para que los grupos estén parejos, lo que buscaremos es minimizar la adición de los cuadrados de las sumas de las fuerzas de los grupos. Es decir:

$$\min \sum_{i=1}^k \left( \sum_{x_j \in S_i} x_j \right)^2$$

El Maestro Pakku nos dice que esta es una tarea difícil, pero que con tiempo y paciencia podemos obtener el resultado ideal.

## Resolución

### 2. El problema es NP

Se demostrará que el problema pertenece a la clase de complejidad **NP**, presentando un *certificador eficiente* capaz de verificar un *certificado* en tiempo polinomial.

#### Certificado

Un vector  $C = (S_1, \dots, S_k)$ , donde cada  $S_j$  es un subconjunto de índices que indica a qué grupo pertenece cada elemento  $x_i$ .

#### Algoritmo del certificador

El certificador debe verificar las siguientes condiciones:

- Que la cantidad de subconjuntos  $S_j$  sea exactamente  $k$ .
- Que cada elemento  $x_i$  pertenezca a algún subconjunto  $S_j$ .
- Que ningún elemento  $x_i$  aparezca en más de un subconjunto.
- Que la suma de los cuadrados de las sumas de los elementos de cada  $S_j$  sea menor o igual a  $B$ .

La implementación del certificador se encuentra en `certificador.py`

```
1 def es_particion_valida(maestros, k, B, particion):
2     if len(particion) != k:
3         return False
4
5     vistos = set()
6     for grupo in particion:
7         for m in grupo:
8             if m in vistos:
9                 return False
10            vistos.add(m)
11
12     if len(vistos) != len(maestros):
13         return False
14
15     suma_total = 0
16     for grupo in particion:
17         suma_grupo = 0
18         for maestro in grupo:
19             suma_grupo += maestros[maestro]
20         suma_total += suma_grupo**2
21
22     if suma_total > B:
23         return False
24
25     return True
```

Listing 1: Implementación del certificador

## Justificación de la complejidad

Sean:

- $n$ : la cantidad de elementos (maestros) a agrupar,
- $k$ : la cantidad de grupos permitidos,
- $B$ : el valor máximo permitido para la suma de los cuadrados.

El certificador realiza las siguientes verificaciones:

1. Que la cantidad de grupos sea exactamente  $k$ .
2. Que ningún elemento se repita en más de un grupo.
3. Que todos los elementos estén asignados a algún grupo.
4. Que la suma total de los cuadrados de las sumas de cada grupo se calcule correctamente.
5. Que el resultado obtenido se compare con el valor límite  $B$ .

El análisis de complejidad de cada paso es el siguiente:

Paso	Descripción	Complejidad
1	Comparar la cantidad de grupos con $k$	$O(1)$
2	Recorrer todos los elementos para verificar repeticiones	$O(n)$
3	Verificar que no falte ningún elemento	$O(1)$
4	Calcular la suma total de los cuadrados	$O(n)$
5	Comparar el resultado con $B$	$O(1)$

Por lo tanto, la complejidad total del certificador es:

$$O(1) + O(n) + O(1) + O(n) + O(1) = O(n)$$

Es decir, el certificador se ejecuta en tiempo lineal con respecto a la cantidad de elementos  $n$ . Dado que la verificación de una solución candidata puede realizarse en tiempo polinomial respecto del tamaño de la entrada, el **Problema de la Tribu del Agua pertenece a la clase NP**.

### 3. Backtracking

```
1 import math
2
3
4 def suma_cuadrados(sumas):
5     return sum(s * s for s in sumas)
6
7
8 def backtracking(i, habilidades, k, sumas, particion, mejor_valor, mejor_particion)
9     :
10
11     sum_cuad_actual = suma_cuadrados(sumas)
12
13     if sum_cuad_actual >= mejor_valor:
14         return mejor_valor, mejor_particion
15
16     if i == len(habilidades):
17         if sum_cuad_actual < mejor_valor:
18             mejor_valor = sum_cuad_actual
19             mejor_particion = [list(g) for g in particion]
20         return mejor_valor, mejor_particion
21
22     valor, indice = habilidades[i]
23
24     ya_uso_vacio = False # para evitar permutaciones de los conjuntos
25     for g in range(k):
26         if sumas[g] == 0:
27             if ya_uso_vacio:
28                 continue
29             ya_uso_vacio = True
30
31         particion[g].append(indice)
32         sumas[g] += valor
33
34         mejor_valor, mejor_particion = backtracking(
35             i + 1, habilidades, k, sumas, particion, mejor_valor, mejor_particion)
36
37         sumas[g] -= valor
38         particion[g].pop()
39
40     return mejor_valor, mejor_particion
41
42 def resolver(k, habilidades, ordenar=True):
43
44     tuplas = sorted([(v, i)
45                     for i, v in enumerate(habilidades)], reverse=True) if ordenar
46     else [(v, i) for i, v in enumerate(habilidades)]
47
48     mejor_valor, mejor_particion = math.inf, None
49
50     sumas = [0] * k
51     particion = [[] for _ in range(k)]
52
53     return backtracking(0, tuplas, k, sumas, particion, mejor_valor,
54                         mejor_particion)
```

## 4. Greedy - Pakku

```
1 def greedy_pakku(k, habilidades):  
2     ordenadas = sorted([(v, i)  
3                          for i, v in enumerate(habilidades)], reverse=True)  
4     sumas = [0]*k  
5     particion = [[] for _ in range(k)]  
6     for valor, indice in ordenadas:  
7         j = min(range(k), key=lambda x: sumas[x])  
8         particion[j].append(indice)  
9         sumas[j] += valor  
10    return sum(s*s for s in sumas), particion
```

## 5. Programación Lineal

## 6. Ejemplo de Ejecuciones

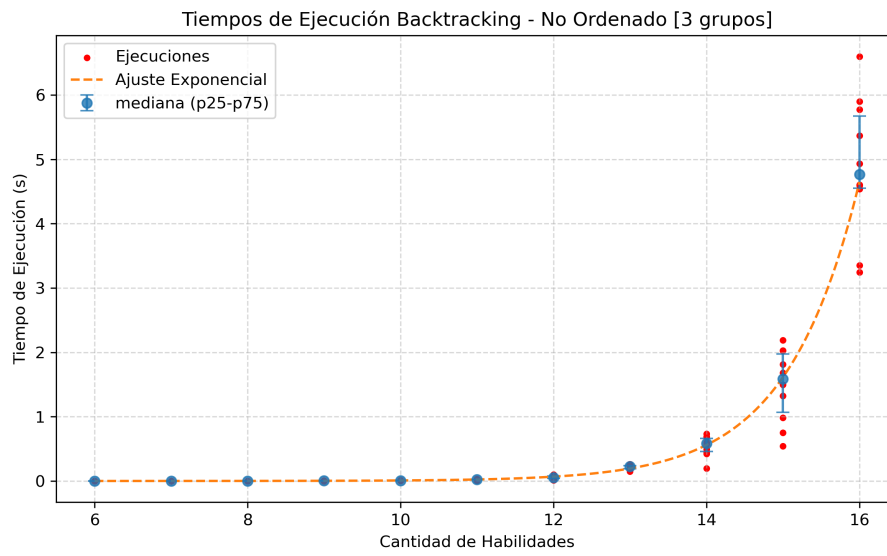


## 7. Mediciones

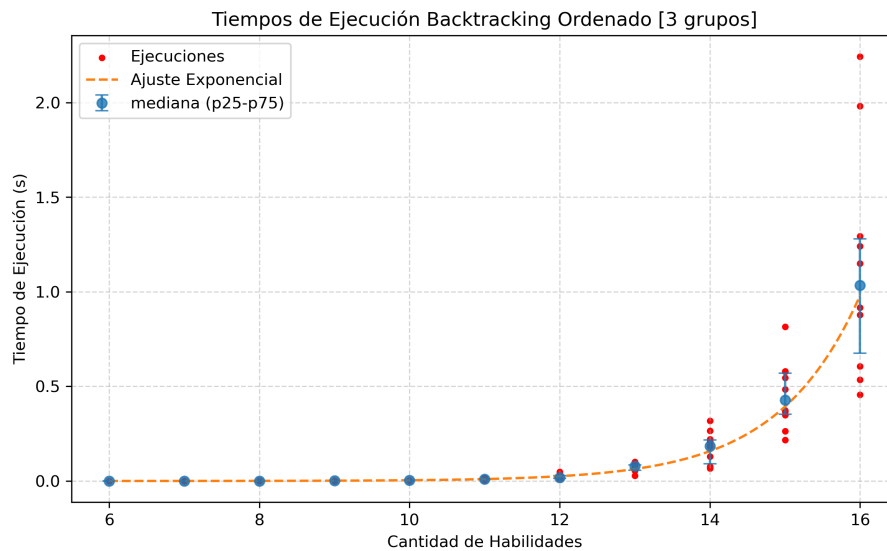
### 7.1. Backtracking

TODO: Agregar contexto

Con o Sin Ordenamiento Previo



(a) Sin Ordenar



(b) Datos Ordenados

Figura 1: Comparación de Tiempos de Ejecución Con o Sin Ordenamiento Previo

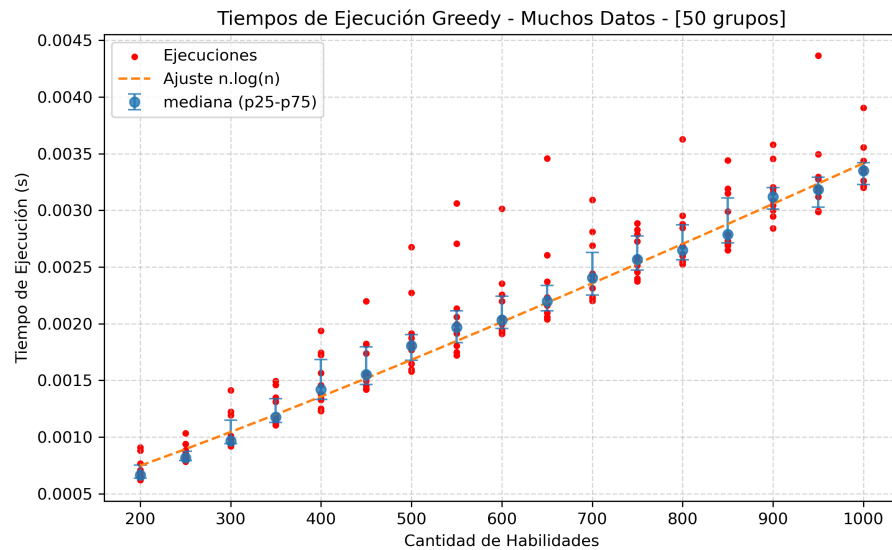
Se observa que ordenando en forma descendente los datos antes del realizar el backtracking, los tiempos de ejecución se reducen a aproximadamente un tercio.

TODO: Ampliar

## 7.2. Greedy (Algoritmo de Pakku)

TODO: Agregar Contexto

### Dataset Grande

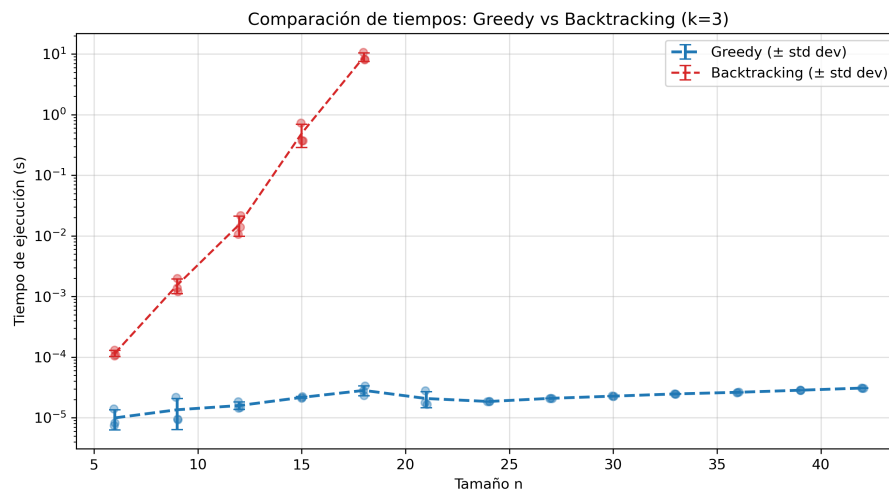
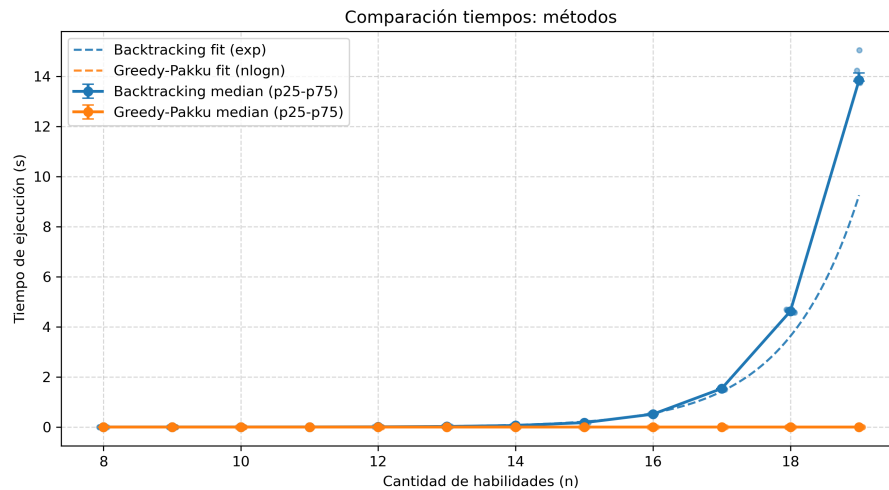


TODO: Agregar Interpretación

### 7.3. Greedy Pakku vs. Backtracking

TODO: Agregar contexto

#### Tiempos de Ejecución

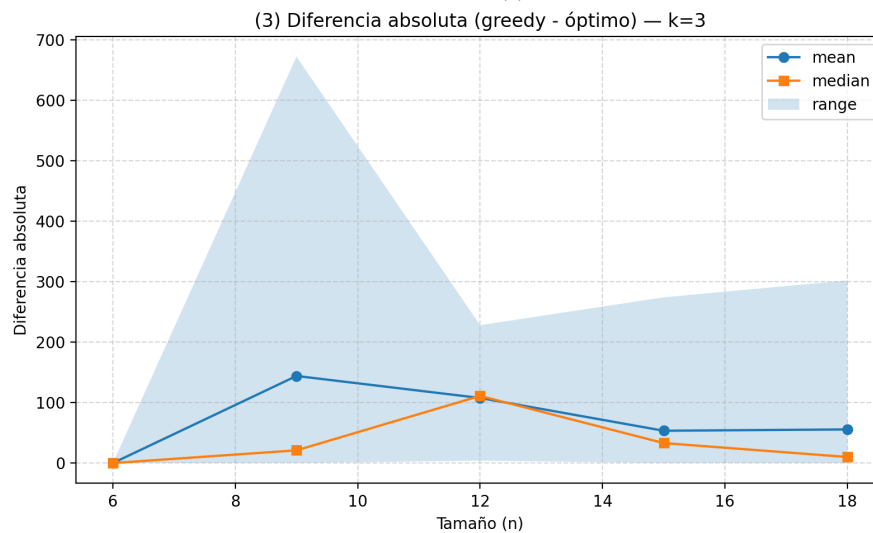
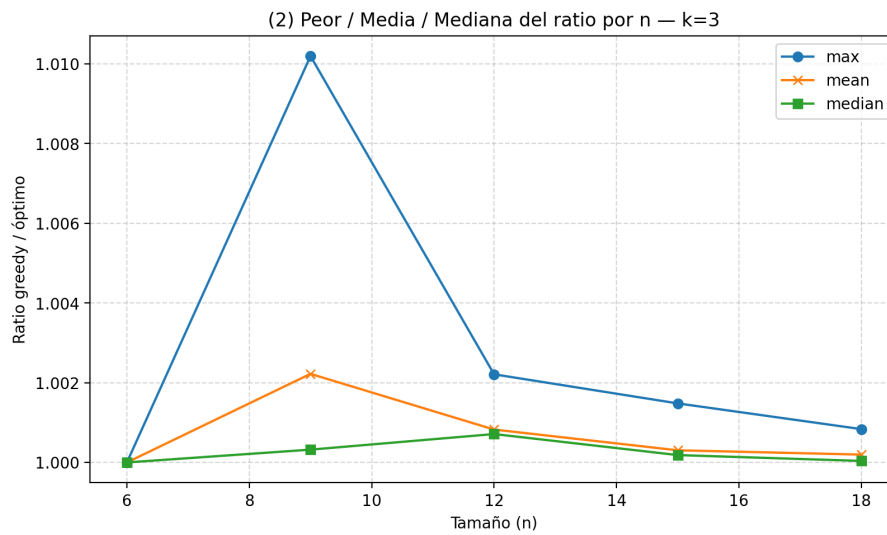
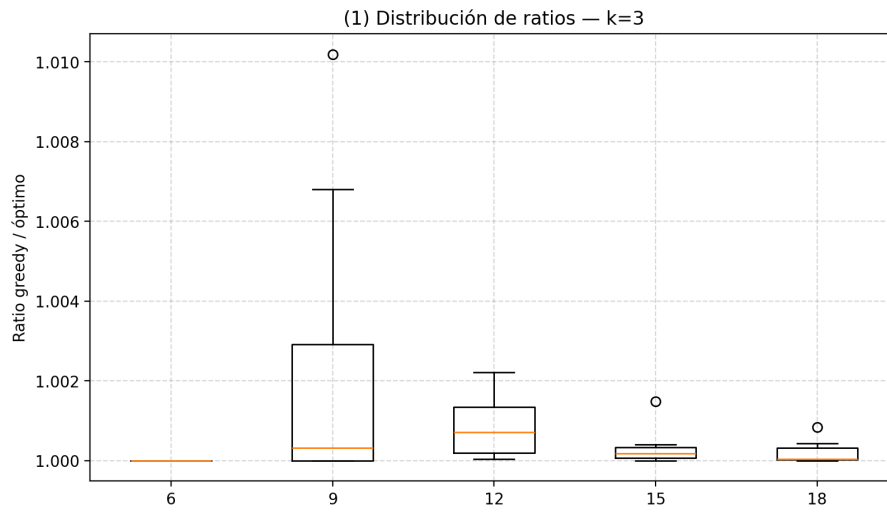


TODO: Agregar Interpretación

#### Ratio de la Aproximación

TODO: Agregar contexto

TODO: EL GRÁFICO ES MUY REDUNDANTE



TODO: Agregar Interpretación

## 8. Conclusiones