.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(TB024) Curso Echeverria

Trabajo Práctico 1
Problema 1, 2 y 3

5 de mayo de 2025

Grupo Nº 1

Los Desviados

Integrantes:

Céspedes, Brian - 108219

Ledesma, Cristian - 111426

Juarez Lezama, Juan Ernesto - 110418

Janampa Salazar, Mario Rafael - 108344

Índice

1. Problema 1 **4**
1.1. Consigna . . . 4
1.2. Restricciones del algoritmo . . . 4
1.3. Diseño y Pseudocodigo . . . 4
1.3.1. Pseudocodigo . . . 4
1.3.2. Implementación . . . 5
1.4. Seguimiento del Algoritmo . . . 5
1.5. Análisis de Complejidad . . . 6
1.6. Set de Datos . . . 7
1.7. Graficos . . . 8
1.7.1. Tiempos de Ejecucion . . . 8
1.8. Conclusión . . . 8

2. Problema 2 **9**
2.1. Consigna . . . 9
2.2. Restricciones del algoritmo . . . 9
2.3. Diseño y Pseudocodigo . . . 9
2.3.1. Pseudocodigo . . . 9
2.3.2. Implementación . . . 10
2.3.3. Justificación de la estrategia Greedy . . . 10
2.4. Seguimiento del Algoritmo . . . 11
2.5. Análisis de Complejidad . . . 12
2.6. Set de Datos . . . 13
2.7. Graficos . . . 13
2.7.1. Tiempos de Ejecucion . . . 13
2.8. Conclusión . . . 14

3. Problema 3 **15**
3.1. Consigna . . . 15
3.2. Restricciones del algoritmo . . . 15
3.3. Diseño y Pseudocodigo . . . 15
3.3.1. Pseudocodigo . . . 15
3.3.2. Implementación . . . 17
3.3.3. Justificación de la estrategia Backtracking . . . 17
3.3.4. Justificación de la optimalidad Backtracking . . . 17
3.4. Seguimiento del Algoritmo . . . 18
3.5. Análisis de Complejidad . . . 22
3.6. Conjunto de Datos . . . 23
3.7. Graficos . . . 23
3.7.1. Tiempos de Ejecucion . . . 23

3.8. Conclusión 24

1. Problema 1

1.1. Consigna

Tenemos una serie de n partículas cargadas muy pequeñas, ubicadas a intervalos regulares a lo largo de una línea recta en las ubicaciones $\{1, 2, \ldots, n\}$. En cada uno de estos puntos, tenemos una partícula cargada q_i, que puede ser positiva o negativa. Queremos estudiar la fuerza total sobre cada partícula mediante la Ley de Coulomb:

$$F_j = \sum_{i<j} C\frac{q_i q_j}{(j-i)^2} - \sum_{i>j} C\frac{q_i q_j}{(j-i)^2}$$

Desarrollar un algoritmo que calcule F_j mediante División y Conquista.

1.2. Restricciones del algoritmo

El algoritmo del Problema 1 recibe una lista de particulas:

- Se toma la constante de Coulomb como 1, para evaluar los resultados de forma más ágil.
- Debe contener números (que pueden ser con parte decimal).
- Los números pueden ser positivos, negativos o cero.
- La lista no debe estar vacía.
- Las partículas tienen que estar ordenadas según su ubicación.

1.3. Diseño y Pseudocodigo

1.3.1. Pseudocodigo

Funcion fuerza_coulomb(q, j, i)

```
Funcion fuerza_coulomb(q, j, i)
    distancia := j - i
    SI distancia = 0 ENTONCES:
        RETORNAR 0
    RETORNAR COULOMB * (q_j * q_i) / (distancia ^ 2)
Fin Funcion
```

Funcion calcular_fuerzas(particulas, izquierda, derecha)

```
Funcion calcular_fuerzas(particulas, izquierda, derecha)
    SI izquierda = derecha ENTONCES:
        RETORNAR lista_con_un_0

    mitad := (izquierda + derecha) DIV 2

    fuerza_izquierda := calcular_fuerzas(particulas, izquierda,
   mitad)
    fuerza_derecha := calcular_fuerzas(particulas, mitad + 1,
   derecha)

    fuerzas := CONCATENAR(fuerza_izquierda, fuerza_derecha)

    DESDE j := izquierda HASTA mitad:
```

```
        DESDE i := mitad + 1 HASTA derecha:
            fuerza := fuerza_coulomb(particulas, j, i)
            fuerzas_i := i - izquierda
            fuerzas_j := j - izquierda
            fuerzas[fuerzas_i] := fuerzas[fuerzas_i] + fuerza
            fuerzas[fuerzas_j] := fuerzas[fuerzas_j] - fuerza

    RETORNAR fuerzas
Fin Funcion
```

Estructuras de datos utilizadas:

- **Lista de partículas:** Un arreglo unidimensional que representa las cargas de cada partícula, donde se asume que la posición de cada partícula está dada por su índice en la lista.
- **Lista de fuerzas:** Una lista del mismo tamaño que el de partículas, que almacena la fuerza resultante sobre cada partícula.

1.3.2. Implementación

```python
def fuerza_coulomb(q, j, i):
    distancia = j - i
    if distancia == 0:
        return 0
    return COULOMB * (q[j] * q[i]) / (distancia ** 2)

def calcular_fuerzas(particulas, izquierda, derecha):
    if izquierda == derecha:
        return [0]

    mitad = (izquierda + derecha) // 2
    fuerza_izquierda = calcular_fuerzas(particulas, izquierda, mitad)
    fuerza_derecha = calcular_fuerzas(particulas, mitad + 1, derecha)

    fuerzas = fuerza_izquierda + fuerza_derecha

    for j in range(izquierda, mitad + 1):
        for i in range(mitad + 1, derecha + 1):
            fuerza = fuerza_coulomb(particulas, j, i)
            fuerzas[i - izquierda] += fuerza
            fuerzas[j - izquierda] -= fuerza

    return fuerzas

def calcular_fuerzas_coulomb(particulas):
    return calcular_fuerzas(particulas, 0, len(particulas) - 1)
```

1.4. Seguimiento del Algoritmo

Ejemplo de ejecución:

- A fines prácticos, la constante de Coulomb se toma igual a 1.
- Para un set de datos: `particulas = [1, -1, 2, -2]`
- Se llama a `calcular_fuerzas_coulomb(particulas)`
- Retorna el llamado a `calcular_fuerzas([1, -1, 2, -2], 0, 3)`
- `mitad = 1`
- Se divide en:

(A) `calcular_fuerzas(particulas, 0, 1)`:

- `mitad = 0`
- Se divide en:
  - `calcular_fuerzas(particulas, 0, 0)` → Retorna [0]
  - `calcular_fuerzas(particulas, 1, 1)` → Retorna [0]
- Luego se combina lo obtenido: `fuerzas = [0, 0]`
- Se calcula la interacción entre `j = 0` y `i = 1` usando `fuerza_coulomb(q, j, i)`:
  - `fuerza_coulomb(particulas, 0, 1)`:
    - `distancia = j - i = 0 - 1 = -1`
    - `fuerzas = (q[0] * q[1]) / (-1)`^{2} `= (1 * -1) / 1 = -1`
    - `fuerzas[1 - 0] += -1 → fuerzas[1] = -1`
    - `fuerzas[0 - 0] -= -1 → fuerzas[0] = 1`
- Resultado parcial: `[1, -1]`

(B) `calcular_fuerzas(particulas, 2, 3)`:

- `mitad = 2`
- Se divide en:
  - `calcular_fuerzas(particulas, 2, 2)` → Retorna [0]
  - `calcular_fuerzas(particulas, 3, 3)` → Retorna [0]
- Se combina en: `fuerzas = [0, 0]`
- Se calcula la interacción entre `j = 2` y `i = 3` usando `fuerza_coulomb(q, j, i)`:
  - `distancia = 2 - 3 = -1`
  - `fuerzas = (2 * -2) / 1 = -4`
  - `fuerzas[3 - 2] += -4 → fuerzas[1] = -4`
  - `fuerzas[2 - 2] -= -4 → fuerzas[0] = 4`
- Resultado parcial: `[4, -4]`

- Volvemos a `calcular_fuerzas(particulas, 0, 3)`:
  - `fuerza_izquierda = [1, -1]` (A)
  - `fuerza_derecha = [4, -4]` (B)
  - Unificamos ambas mitades en: `fuerzas = [1, -1, 4, -4]`
  - Se calcula la interacción entre las partículas de la izquierda (`j = 0, 1`) y las de la derecha (`i = 2, 3`):
    - `j = 0, i = 2: fuerzas[2] += 0.5 → 4.5; fuerzas[0] -= 0.5 → 0.5`
    - `j = 0, i = 3: fuerzas[3] += -0.222 → -4.222; fuerzas[0] -= -0.222 → 0.722`
    - `j = 1, i = 2: fuerzas[2] += -2 → 2.5; fuerzas[1] -= -2 → 1.0`
    - `j = 1, i = 3: fuerzas[3] += 0.5 → -3.722; fuerzas[1] -= 0.5 → 0.5`
  - Finalmente obtenemos: `fuerzas = [0.722, 0.5, 2.5, -3.722]`

1.5. Análisis de Complejidad

- Analizaremos la función recursiva `calcular_fuerzas`.
- Para el caso base, tenemos una ejecución en tiempo constante:

$$\mathcal{O}(1)$$

- En las llamadas recursivas, cada una se encarga de la mitad del intervalo, lo cual implica un tamaño de $n/2$.

- Para el cálculo cruzado de las fuerzas, tenemos una doble iteración sobre las dos mitades (izquierda de tamaño $n/2$ y derecha, del mismo tamaño), obteniendo:

$$T_{\text{cruzado}}(n) = \frac{n}{2} \cdot \frac{n}{2} = \frac{n^2}{4} \Rightarrow \mathcal{O}(n^2)$$

- Entonces, podemos expresar la complejidad total como:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \mathcal{O}(n^2)$$

- Aplicando el **Teorema Maestro**:
  - $a = 2$ (número de llamadas recursivas)
  - $b = 2$ (proporción de los subproblemas: $n/2$)
  - $c = 2$ (costo de combinación)
- Evaluando: $\log_2(2) = 1$ y como $1 < c = 2$, se trata del **caso 1** del teorema maestro.
- Por lo tanto, la complejidad temporal es:

$$\boxed{T(n) = \mathcal{O}(n^2)}$$

1.6. Set de Datos

En el código fuente se incluye la función aleatoria `generar_particulas(n)` que se encarga de generar una lista de números enteros entre -100 y 100 de tamaño `n` a partir de una semilla fija que arbitrariamente se eligió con el valor 6. Se utilizó para `n = 1.000`, `10.000` y `100.000`. Los resultados de cada ejecución se encuentran guardados dentro de la carpeta `resultados`, donde cada uno cuenta con su tiempo de ejecución, la lista de números aleatorios y la lista de fuerzas que actúan sobre cada una de estas.

Los resultados son generados por el archivo `generador_resultados.py` que cuenta con una función para almacenar lo obtenido dentro de un `.json` para cada valor de `n`.

1.7. Graficos

1.7.1. Tiempos de Ejecucion

Figura 1: Grafico que relaciona el valor de **n** con el tiempo de ejecucion

A continuación, se expone la tabla con los datos obtenidos:

n	**Tiempo de Ejecucion (s)**
2000	$0,393$
4000	$1,603$
6000	$3,655$
8000	$6,469$
10000	$10,181$

1.8. Conclusión

El algoritmo desarrollado implementa División y Conquista al descomponer el problema original en dos subproblemas más pequeños: se divide el arreglo de partículas en dos mitades, se calcula recursivamente la fuerza en cada mitad y finalmente se combinan los resultados sumando las fuerzas entre las partículas de distintas mitades.

Los sets de datos analizados arrojaron tiempos de ejecución que se corresponden con la complejidad planteada inicialmente, ya que dentro del gráfico se puede observar la tendencia cuadrática. Además, dividiendo el tiempo de ejecución para `n`, con el valor de n^2, obtenemos un factor que se mantiene constante para los distintos valores observados.

En conclusión, los resultados concuerdan con la complejidad cuadrática determinada.

2. Problema 2

2.1. Consigna

Juan Curuchet tiene planeando un rally por el Camino de las Altas Cumbres. Puede llevar dos litros de agua y rodar 7 kilómetros antes de que se le agote. Tiene un mapa con lugares donde puede repostar agua, y conoce la distancia entre cada uno. El objetivo de Juan es detenerse la menor cantidad de veces que sea posible. Desarrollar un algoritmo Greedy que determine en qué lugares detenerse a cargar agua, y mostrar si siempre encuentra el óptimo o no.

2.2. Restricciones del algoritmo

El algoritmo del Problema 2 recibe una lista de distancias:

- Debe contener números (que pueden ser con parte decimal).
- Los números deben ser mayores o iguales a cero.
- La lista no debe estar vacía.
- Admite duplicados.
- Admite una lista desordenada.

2.3. Diseño y Pseudocodigo

2.3.1. Pseudocodigo

Funcion planificar_paradas(distancias)

```
Funcion planificar_paradas(distancias)
    ELIMINAR DUPLICADOS DE distancias
    ORDENAR distancias

    paradas := lista_vacia

    SI distancias[ultimo_elemento] <= DISTANCIA_MAXIMA ENTONCES:
        AGREGAR distancias[ultimo_elemento] a paradas
        RETORNAR paradas

    SI distancias[0] > DISTANCIA_MAXIMA ENTONCES:
        RETORNAR paradas

    numero_de_elementos := LONGITUD(distancias)
    ultima_parada := 0

    DESDE i := 0 HASTA numero_de_elementos:
        SI distancias[i] - ultima_parada > DISTANCIA_MAXIMA
   ENTONCES:
            SI distancias[i] - distancias[i - 1] > DISTANCIA_MAXIMA
    ENTONCES:
                RETORNAR lista_vacia
            AGREGAR distancias[i - 1] a paradas
            ultima_parada := distancias[i - 1]
    RETORNAR paradas

Fin Funcion
```

Estructuras de datos utilizadas:

- **Lista** `distancias`: Representa el mapa inicial que posee el deportista, contiene los kilometrajes en los cuales se puede repostar agua.
- **Lista** `paradas`: Muy similar a la lista anterior, esta es retornada al finalizar el algoritmo y representa también kilometrajes, pero esta vez en los que debe detenerse el deportista para repostar agua de manera óptima.

2.3.2. Implementación

```python
DISTANCIA_MAXIMA: int = 7


def planificar_paradas(distancias: list) -> list:

    distancias: list = list(set(distancias)) # Elimina duplicados
    distancias.sort()

    paradas: list = []

    if distancias[-1] <= DISTANCIA_MAXIMA:
        paradas.append(distancias[-1])
        return paradas

    if distancias[0] > DISTANCIA_MAXIMA:
        return paradas

    numero_de_elementos: int = len(distancias)
    ultima_parada: int = 0

    for i in range(numero_de_elementos):
        if distancias[i] - ultima_parada > DISTANCIA_MAXIMA:
            if distancias[i] - distancias[i - 1] > DISTANCIA_MAXIMA:

                return []

            paradas.append(distancias[i - 1])
            ultima_parada = distancias[i - 1]

    return paradas
```

2.3.3. Justificación de la estrategia Greedy

El algoritmo es claramente greedy, ya que siempre buscaremos aplazar la carga de agua lo máximo posible, parando solamente cuando sea necesario y cuando sepamos que de otra forma no podremos completar el recorrido. Esto se hace buscando los puntos de carga más lejanos, hasta que lleguemos a un punto el cual no es alcanzable desde el punto actual. Si estamos en esta situación, entonces debemos parar a cargar agua en el punto inmediatamente anterior. El algoritmo simplemente se fija en la mejor opción que se posea en ese momento, o sea, busca siempre el óptimo local. Se puede demostrar que siguiendo la regla greedy, se puede llegar al óptimo global.

- **Justificación de la optimalidad del algoritmo:**

  Supongamos que para una lista de distancias D obtenemos dos listas de paradas
  A = [a1, a2, a3, . . . , an] y B = [b1, b2, b3, . . . , bm], la lista A es devuelta al ejecutar nuestro algoritmo, y la lista B es devuelta por otro algoritmo (que supongamos que devuelve una solución óptima).

  - SI los puntos de B estuviesen a menor distancia que los puntos de A, entonces, tendríamos que realizar potencialmente más paradas y no estaríamos aprovechando al máximo la distancia máxima que se puede recorrer (7 km).

- SI los puntos de B estuviesen a mayor distancia que los puntos de A, entonces no se podría completar el recorrido ya que tendríamos que recorrer más distancia de la que somos capaces.

La única alternativa que resta es que A = B, por lo tanto se demuestra que si B era una solución óptima, entonces A también lo es, pues son la misma solución.

- Este problema es muy similar al visto en la primera clase de greedy del curso, denominado "Carga de Combustible".

2.4. Seguimiento del Algoritmo

Para empezar, debemos notar dos casos, uno en el que todos los elementos son menores o iguales a 7 (la distancia máxima) y otro en el que todos los elementos son mayores a ella.

- Por ejemplo, si tenemos la lista [3, 2, 4, 1, 5.10, 1, 6.99], luego de eliminar duplicados y ordenarla obtendremos la lista [1, 2, 3, 4, 5.10, 6.99]. El último elemento de la lista (6.99) es menor o igual que 7, entonces sabremos que todos los anteriores son alcanzables desde esta distancia.
- Un caso similar es teniendo una lista en la que todos los elementos son mayores a 7. Por ejemplo, la lista [14, 13.20, 66, 7.01]. Luego de eliminar duplicados y ordenarla obtendremos la lista [7.01, 13.20, 14, 66]. El primer elemento de la lista (7.01) es mayor que 7, entonces jamás podremos alcanzarlo, ya que la distancia máxima nos lo impide.

Haremos un seguimiento para dos listas de distancias, la primera tiene un resultado válido, la segunda no.

- **Primera lista:** [7.01, 6, 8, 7.01, 15, 14, 6].
  1. Inicialmente, se eliminan los duplicados, la lista queda de la siguiente manera: [7.01, 6, 8, 15, 14].
  2. Luego, se ordena la lista y queda de la siguiente manera: [6, 7.01, 8, 14, 15].
  3. En la primera iteración, `ultima_parada` vale 0 y `distancias[i]` vale 6, por lo que al hacer la resta 6 - 0 = 6 ¡7, por lo que no se cumple la condición del primer `if`.
  4. En la segunda iteración, `ultima_parada` sigue valiendo 0, pero `distancia[i]` ahora vale 7.01. 7.01 - 0 = 7.01 ¿7, por lo que se cumple la condición del primer `if`. Se agrega a la lista de paradas el valor anterior a 7.01, que en este caso es 6, además este será el nuevo valor de `ultima_parada`.
  5. En la tercera iteración, `ultima_parada` vale 6, `distancia[i]` vale 8. 8 - 6 = 2 ¡7, no se cumple la condición del primer `if`.
  6. En la cuarta iteración, `ultima_parada` continúa valiendo 6, `distancia[i]` = 14 y 14 - 6 = 8, que es mayor a 7, entonces se agrega el valor anterior, que es 8. El nuevo valor de `ultima_parada` ahora es 8.
  7. En la quinta y última iteración, `ultima_parada` vale 8 y `distancia[i]` vale 15. 15 - 8 = 7, pero como no es mayor a 7 entonces tampoco se ingresa en el `if`.

  La lista de paradas resultante es [6, 8]
- **Segunda lista:** [20, 6.99, 27.01, 14, 6.99]
  1. Inicialmente, se eliminan los duplicados: [20, 6.99, 27.01, 13.99].
  2. Luego se ordena la lista: [6.99, 13.99, 20, 27.01].

3. En la primera iteración `ultima_parada` = 0 y `distancias[i]` = 6.99, 6.99 - 0 = 6.99 ¡7, entonces no se cumple la condición.
4. En la segunda iteración `ultima_parada` = 0 y `distancias[i]` = 13.99, 13.99 - 0 = 13.99 ¿7, se cumple la condición del `if`, entonces se agrega el valor 6.99 a la lista de paradas y se lo asigna como `ultima_parada`.
5. En la tercera iteración `ultima_parada` = 6.99 y `distancias[i]` = 20, 20 - 6.99 = 13.01 ¿7, como se cumple la condición del `if`, se agrega el valor 13.99 a la lista de paradas y se lo asigna como `ultima_parada`.
6. En la cuarta iteración `ultima_parada` = 13.99 y `distancias[i]` = 27.01. 27.01 - 13.99 = 13.02 que es mayor que 7, pero además se cumple que 27.01 - 20 = 7.01, los anteriores son dos elementos consecutivos en la lista. Esto quiere decir que sería imposible ir de un punto al otro ya que la distancia a recorrer sería mayor a la máxima.

La lista de paradas resultante es [] (una lista vacía) que indica que no es posible el recorrido.

2.5. Análisis de Complejidad

Para analizar la complejidad temporal del algoritmo, podremos hacerlo analizando la única función `planificar_paradas()`.

Eliminar los duplicados de la lista se hace primero convirtiendo la lista original en un `set`, proceso que es **O(n)**. Luego, se la vuelve a convertir en una lista, lo cual nuevamente es **O(n)**. Podemos concluir que eliminar duplicados en una lista tiene una complejidad algorítmica de:

$$\mathcal{O}(n)$$

Ordenar la lista tiene una complejidad de:

$$\mathcal{O}(n \cdot \log(n))$$

Analizando el primer bloque `if`: acceder al último elemento de una lista en Python tiene una complejidad constante **O(1)**. Agregar un elemento a la lista también tiene una complejidad de **O(1)**.

Analizando el segundo bloque `if`: acceder al primer elemento de una lista en Python tiene una complejidad de **O(1)**.

Analizando el ciclo `for`: este ciclo se iterará, como máximo, hasta n (siendo n el número de elementos de la lista de distancias). Dentro del ciclo hay operaciones de comparación y de agregado a la lista de paradas, todas se ejecutan en tiempo constante **O(1)**.

Complejidad resultante (temporal):

$$\mathcal{O}(n \cdot \log(n))$$

Para analizar la complejidad espacial del algoritmo, podremos hacerlo analizando la única función `planificar_paradas()`.

La lista de paradas es la única estructura que se utiliza y tendrá como mucho n elementos, siendo n los elementos de la lista de distancias. Por lo tanto la complejidad espacial es:

Complejidad resultante (espacial):

$$\boxed{\mathcal{O}(n)}$$

2.6. Set de Datos

En el archivo **test_aleatorio.txt** se encuentran las pruebas realizadas sobre los sets de datos generados aleatoriamente con el programa **ejercicio2_test_aleatorio.py**. Para medir los tiempos de ejecución, se optó por utilizar otro programa **ejercicio2_test_tiempo_de_ejecucion.py** debido a la naturaleza de los datos de entrada y para evitar que se generen constantemente listas de distancias en las cuales es imposible de finalizar el recorrido. Como tercer opción, se encuentra el programa **ejercicio2_test_manual.py** el cual permite al usuario generar una lista de distancias a demanda y luego probar el algoritmo.

2.7. Graficos

2.7.1. Tiempos de Ejecucion

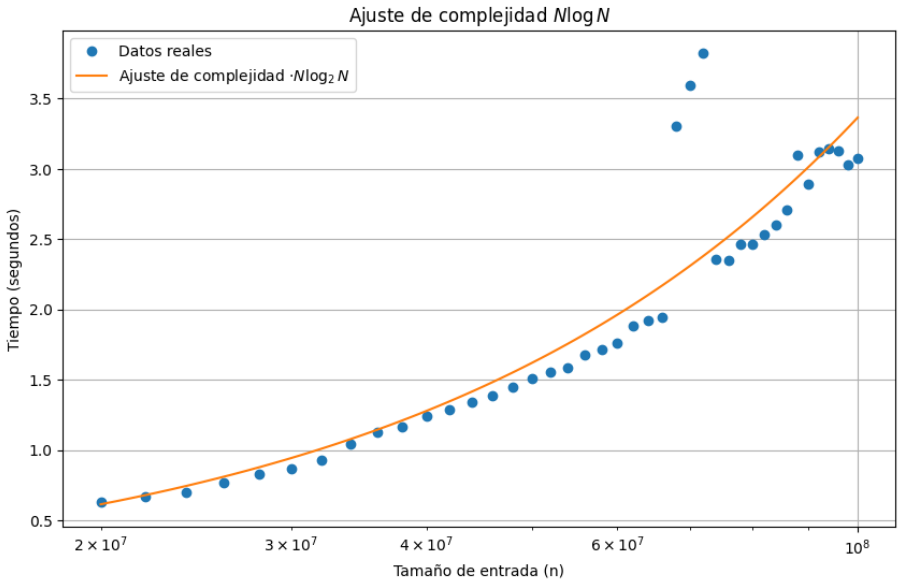

Figura 2: Grafico que relaciona el valor de **n** con el tiempo de ejecucion

Se observa en el gráfico anterior que los mismos se condicen con la complejidad temporal analizada previamente $O(n \log n)$, ya que el mismo tiene una tendencia a seguir una curva característica de crecimiento logarítmico multiplicado por el tamaño de entrada.

A continuación, se expone la tabla con los datos obtenidos:

n	Tiempo de Ejecución (s)
20000000	0,633
22000000	0,666
24000000	0,700
26000000	0,768
28000000	0,827
30000000	0,864
32000000	0,930
34000000	1,044
36000000	1,127
38000000	1,166
40000000	1,245
42000000	1,286
44000000	1,340
46000000	1,390
48000000	1,448
50000000	1,513
52000000	1,558
54000000	1,585
56000000	1,674
58000000	1,714
60000000	1,765
62000000	1,882
64000000	1,922
66000000	1,946
68000000	3,308
70000000	3,598
72000000	3,824
⋮	⋮
100000000	3,073

Las mediciones fueron realizadas en una computadora con las siguientes características:

- **Procesador:** [Intel Core i5 1235U 1.3 GHz]
- **Memoria RAM:** [24GB de RAM]
- **Sistema Operativo:** [WINDOWS 11]
- **Intérprete de Python:** [Python 3.12.9]

2.8. Conclusión

Podemos concluir que el algoritmo propuesto se corresponde con la metodología greedy. Se ha indicado la regla greedy, y además se ha demostrado la optimalidad de la solución propuesta. También se han encontrado similitudes con un problema greedy ya conocido, cuyo nombre es **"Carga de Combustible".** La complejidad temporal del algoritmo se condice con el gráfico expuesto, ya que se observa la tendencia lineal del mismo.

3. Problema 3

3.1. Consigna

Una *suma encadenada* es una secuencia de números $a_0, a_1, \ldots, a_l$ tal que $a_0 = 1$ y para cada $k > 0$: $a_k = a_i + a_j$ para algún $i, j, k < l$.

Por ejemplo, $1, 2, 3, 6$ es una suma encadenada para $n = 6$ de longitud $l = 3$ ya que:

- $2 = 1 + 1$
- $3 = 2 + 1$
- $6 = 3 + 3$

Desarrollar un algoritmo de **Backtracking** para calcular una suma encadenada de longitud mínima para un entero positivo n.

3.2. Restricciones del algoritmo

El algoritmo del Problema 3 recibe un número n como parámetro:

- $n \in \mathbb{Z}^+$, es decir, debe ser un número entero positivo.
- No se admiten valores negativos ni cero: $n > 0$.
- n debe ser de tipo `int`; no se aceptan flotantes (`float`) ni otros tipos de datos.
- La función siempre asume como punto de partida el número 1.

3.3. Diseño y Pseudocodigo

3.3.1. Pseudocodigo

Funcion suma_encadenada_minima(n)

```
Funcion suma_encadenada_minima(n)
    cadena_minima := lista_vacia
    llamar buscar_la_cadena(n, [1], cadena_minima)
    RETORNAR cadena_minima
Fin Funcion
```

Corresponde a la función principal del algoritmo. Esta función inicia la búsqueda de la suma encadenada mínima llamando a la función recursiva `buscar_la_cadena` con los parámetros iniciales.

A continuación, se detalla la función encargada de realizar la búsqueda recursiva.

Funcion buscar_la_cadena(objetivo, cadena_actual, cadena_minima)

```
1
2  Funcion buscar_la_cadena(objetivo, cadena_actual, cadena_minima)
3      ultimo_numero := ULTIMO_ELEMENTO(cadena_actual)
4
5      SI ultimo_numero = objetivo ENTONCES
6          SI cadena_minima ES VACIA O LONGITUD(cadena_actual) <
       LONGITUD(cadena_minima) ENTONCES
7              cadena_minima := COPIA(cadena_actual)
8          RETORNAR
9
10     SI cadena_minima NO ES VACIA Y LONGITUD(cadena_actual) >=
       LONGITUD(cadena_minima) ENTONCES
11         RETORNAR
12
13     SI cadena_minima NO ES VACIA Y MAXIMO(cadena_actual) * 2^(
       LONGITUD(cadena_minima) - LONGITUD(cadena_actual)) < objetivo
       ENTONCES
14         RETORNAR
15
16     PARA i := LONGITUD(cadena_actual) - 1 HASTA 0 CON PASO -1 HACER
17         PARA j EN INVERSO(cadena_actual[i:]) HACER
18             resultado := cadena_actual[i] + j
19
20             SI resultado <= ultimo_numero ENTONCES
21                 ROMPER
22
23             SI resultado > objetivo ENTONCES
24                 CONTINUAR
25
26             AGREGAR resultado A cadena_actual
27             llamar buscar_la_cadena(objetivo, cadena_actual,
       cadena_minima)
28             ELIMINAR ULTIMO_ELEMENTO DE cadena_actual
29 Fin Funcion
```

Esta función implementa la búsqueda recursiva de la suma encadenada mínima. Explora todas las combinaciones posibles a partir de la cadena actual, aplicando podas para reducir el espacio de búsqueda y evitar soluciones no óptimas o redundantes.

Estructuras de datos utilizadas:

- **Lista** `cadena_actual`: Almacena la secuencia parcial de números generados mediante sumas. Permite construir progresivamente una posible solución.
- **Lista** `cadena_minima`: Guarda la mejor solución encontrada hasta el momento. Se actualiza solo si se encuentra una secuencia más corta.

3.3.2. Implementación

```python
def buscar_la_cadena(objetivo, cadena_actual, cadena_minima):

    ultimo_numero = cadena_actual[-1]

    if ultimo_numero == objetivo:
        if not cadena_minima or len(cadena_actual) < len(cadena_minima):
            cadena_minima[:] = cadena_actual[:]
        return

    if cadena_minima and len(cadena_actual) >= len(cadena_minima):
        return

    if cadena_minima and max(cadena_actual) * (2 ** (len(cadena_minima) - len(
    cadena_actual))) < objetivo:
        return

    for i in range(len(cadena_actual) - 1, -1, -1):
        sublista_actual_invertida = reversed(cadena_actual[i:])
        for j in sublista_actual_invertida:
            resultado = cadena_actual[i] + j
            if resultado <= ultimo_numero:
                break
            if resultado > objetivo:
                continue
            cadena_actual.append(resultado)
            buscar_la_cadena(objetivo, cadena_actual, cadena_minima)
            cadena_actual.pop()

def suma_encadenada_minima(n):
    cadena_minima = []
    buscar_la_cadena(n, [1], cadena_minima)
    return cadena_minima
```

3.3.3. Justificación de la estrategia Backtracking

El algoritmo adopta claramente la estrategia backtracking debido a que nuestro problema equivale a un espacio de busquedad muy amplio y es necesario explorar las diferentes combinaciones posibles para poder construir la solucion optima que llegue al numero objetivo. Pero para poder evitar esta explosion de combinaciones, se incorporaron diferentes podas que permiten descartar los caminos que no mejoran la solucion actual. Empezamos desde el 1 e ir sumando combinaciones posibles entre los elementos de la cadena. A medida que se generan estos nuevos numeros, se decide mediante las podas si vale la pena seguir por ese camino o no. Si una cadena ya es mas larga que la mejor que encotramos hasta ahora, o si ya no tiene chances reales de llegar al obtejivo, se cortara ahi mismo lo cual optimizara los tiempos en busqueda.

3.3.4. Justificación de la optimalidad Backtracking

Supongamos que existen dos posibles soluciones al problema:

- Una cadena $A = [1, a_2, a_3, \ldots, a_n]$ encontrada por nuestro algoritmo,
- y otra cadena $B = [1, b_2, b_3, \ldots, b_m]$, que representa una solución hipotéticamente más corta u óptima.

Nuestro algoritmo explora todas las combinaciones posibles siguiendo la regla de solo sumar elementos ya existentes en la cadena, y se detiene unicamente cuando encuentra una solucion que es mas corta que cualquier otra conocida. Ademas, cada vez que se genera un nuevo número, se verifica si tiene sentido seguir con esa cadena o si ya no puede mejorar lo que se tiene.

Si la cadena B fuera realmente más corta, el algoritmo la habría encontrado, ya que recorre todas las posibilidades viables respetando las restricciones. Por otro lado, si B contiene una secuencia más larga o igual que A, entonces claramente A es tan buena o mejor.

Por lo tanto, se concluye que, al aplicar correctamente las condiciones de poda, el algoritmo no se pierde ninguna solución valida y siempre termina encontrando la de menor longitud.

3.4. Seguimiento del Algoritmo

Vamos a hacer el seguimiento con el número 15, pero saltaremos cosas que son evidentes para no ser tan descriptivos con cada línea y que el seguimiento se vuelva largo.

1. Entramos a la función de búsqueda con: **(15, [1], [])**
   - Entramos directamente al ciclo for:
     - `for i in range (1-1, -1, -1):`
       - `i = 0, cadena_actual[i] = 1`
       - `sublista_invertida = [1]`
       - `for j in sublista_invertida:`
         - `resultado = 1 + 1 =>2`
         - llamada recursiva con: **(15, [1, 2], [])**
2. Entramos primera llamada recursiva: **(15, [1, 2], [])**
   - `for i in range (2-1, -1, -1):`
     - `i = 1, cadena_actual[i] = 2`
     - `sublista_invertida = [2]`
     - `for j in sublista_invertida:`
       - `resultado = 2 + 2 =>4`
       - llamada recursiva con: **(15, [1, 2, 4], [])**
3. Entramos segunda llamada recursiva: **(15, [1, 2, 4], [])**
   - `for i in range (3-1, -1, -1):`
     - `i = 2, cadena_actual[i] = 4`
     - `sublista_invertida = [4]`
     - `for j in sublista_invertida:`
       - `resultado = 4 + 4 =>8`
       - llamada recursiva con: **(15, [1, 2, 4, 8], [])**
4. Entramos tercera llamada recursiva: **(15, [1, 2, 4, 8], [])**
   - `for i in range (4-1, -1, -1):`
     - `i = 3, cadena_actual[i] = 8`
     - `sublista_invertida = [8]`
     - `for j in sublista_invertida:`
       - `resultado = 8 + 8 =>16`
       - ¿Es mayor que n? Sí
     - `i = 2, cadena_actual[i] = 4`
     - `sublista_invertida = [8, 4]`
     - `for j in sublista_invertida:`
       - `resultado = 4 + 8 =>12`
       - llamada recursiva con: **(15, [1, 2, 4, 8, 12], [])**

5. Entramos cuarta llamada recursiva: **(15, [1, 2, 4, 8, 12], [])**
   - `for i in range (5-1, -1, -1):`
     - `i = 4, cadena_actual[i] = 12`
     - `sublista_invertida = [12]`
     - `for j in sublista_invertida:`
       - `resultado = 12 + 12 =>24`
       - ¿Es mayor que n? Sí
     - `i = 3, cadena_actual[i] = 8`
     - `sublista_invertida = [12, 8]`
     - `for j in sublista_invertida:`
       - `resultado = 8 + 12 =>20`
       - ¿Es mayor que n? Sí
       - `resultado = 8 + 8 =>16`
       - ¿Es mayor que n? Sí
     - `i = 2, cadena_actual[i] = 4`
     - `sublista_invertida = [12, 8, 4]`
     - `for j in sublista_invertida:`
       - `resultado = 4 + 12 =>16`
       - ¿Es mayor que n? Sí
       - `resultado = 4 + 8 =>12`
       - ¿El resultado es igual o menor al último número de la cadena? Sí
     - `i = 1, cadena_actual[i] = 2`
     - `sublista_invertida = [12, 8, 4, 2]`
     - `for j in sublista_invertida:`
       - `resultado = 2 + 12 =>14`
       - llamada recursiva con: **(15, [1, 2, 4, 8, 12, 14], [])**
6. Entramos quinta llamada recursiva: **(15, [1, 2, 4, 8, 12, 14], [])**
   - `for i in range (6-1, -1, -1):`
     - `i = 5, cadena_actual[i] = 14`
     - `sublista_invertida = [14]`
     - `for j in sublista_invertida:`
       - `resultado = 14 + 14 =>28`
       - ¿Es mayor que n? Sí
     - `i = 4, cadena_actual[i] = 12`
     - `sublista_invertida = [14, 12]`
     - `for j in sublista_invertida:`
       - `resultado = 12 + 14 =>26`
       - ¿Es mayor que n? Sí
       - `resultado = 12 + 12 =>24`
       - ¿Es mayor que n? Sí
     - `i = 3, cadena_actual[i] = 8`
     - `sublista_invertida = [14, 12, 8]`
     - `for j in sublista_invertida:`
       - `resultado = 8 + 14 =>22`
       - ¿Es mayor que n? Sí
       - `resultado = 8 + 12 =>20`
       - ¿Es mayor que n? Sí

    - `resultado = 8 + 8 =>16`
    - ¿Es mayor que n? Sí
- `i = 2, cadena_actual[i] = 4`
- `sublista_invertida = [14, 12, 8, 4]`
- `for j in sublista_invertida:`
    - `resultado = 4 + 14 =>18`
    - ¿Es mayor que n? Sí
    - `resultado = 4 + 12 =>16`
    - ¿Es mayor que n? Sí
    - `resultado = 4 + 8 =>12`
    - ¿El resultado es igual o menor al último número de la cadena? Sí
- `i = 1, cadena_actual[i] = 2`
- `sublista_invertida = [14, 12, 8, 4, 2]`
- `for j in sublista_invertida:`
    - `resultado = 2 + 14 =>16`
    - ¿Es mayor que n? Sí
    - `resultado = 2 + 12 =>14`
    - ¿El resultado es igual o menor al último número de la cadena? Sí
- `i = 0, cadena_actual[i] = 1`
- `sublista_invertida = [14, 12, 8, 4, 2, 1]`
- `for j in sublista_invertida:`
    - `resultado = 1 + 14 =>15`
    - llamada recursiva con: **(15, [1, 2, 4, 8, 12, 14, 15], [])**

7. Entramos sexta llamada recursiva: **(15, [1, 2, 4, 8, 12, 14, 15], [])**
    - ¿El último elemento de la cadena es igual a n? Sí
    - cadena_minima = [1, 2, 4, 8, 12, 14, 15]
8. Regresamos a la quinta llamada recursiva:
    - Tenemos: **(15, [1, 2, 4, 8, 12, 14], [1, 2, 4, 8, 12, 14, 15])**
    - `resultado = 1 + 12 =>13`
    - ¿El resultado es igual o menor al último número de la cadena? Sí
9. Regresamos a la cuarta llamada recursiva:
    - .....
    - .....

    Para no tener que calcular todo y hacerlo largo, dejo todos los resultados que se encontraron de longitud 7, después del primer resultado encontrado.
10. Se encontraron soluciones de longitud 7:
    - $[1, 2, 4, 8, 12, 13, 15]$
    - $[1, 2, 4, 8, 10, 14, 15]$
    - $[1, 2, 4, 8, 10, 11, 15]$
    - $[1, 2, 4, 8, 9, 13, 15]$
    - $[1, 2, 4, 8, 9, 11, 15]$
    - $[1, 2, 4, 6, 12, 14, 15]$
    - $[1, 2, 4, 6, 12, 13, 15]$

- $[1, 2, 4, 6, 10, 14, 15]$
- $[1, 2, 4, 6, 10, 11, 15]$
- $[1, 2, 4, 6, 8, 14, 15]$
- $[1, 2, 4, 6, 8, 9, 15]$
- $[1, 2, 4, 6, 8, 14, 15]$
- $[1, 2, 4, 6, 8, 9, 15]$
- $[1, 2, 4, 6, 7, 14, 15]$
- $[1, 2, 4, 6, 7, 13, 15]$
- $[1, 2, 4, 6, 7, 11, 15]$

11. Se encontró una mejor forma al regresar a la segunda llamada recursiva:
    - Estado actual: `(15, [1, 2, 4], [1, 2, 4, 8, 12, 14, 15])`
    - Se ejecuta: `for i in range(3 - 1, -1, -1)`
        - $i = 2$ (ya se hizo)
        - $i = 1$ (ya se hizo)
        - $i = 0$, `cadena_actual[i] = 1`
        - `sublista_invertida = [4, 2, 1]`
        - `for j in sublista_invertida:`
            - resultado $= 4 + 1 = 5$
            - Se hace llamada recursiva: `(15, [1, 2, 4, 5], [])`
    - En la llamada recursiva `(15, [1, 2, 4, 5], [1,2,4,8,12,14,15])`:
        - `for i in range(4 - 1, -1, -1):`
        - $i = 3$, `cadena_actual[i] = 5`
        - `sublista_invertida = [5]`
        - `for j in sublista_invertida:`
            - resultado $= 5 + 5 = 10$
            - Llamada recursiva: `(15, [1, 2, 4, 5, 10], [1,2,4,8,12,14,15`
    - En la llamada recursiva `(15, [1, 2, 4, 5, 10], [1,2,4,8,12,14,15)`:
        - `for i in range(5 - 1, -1, -1):`
        - $i = 4$, `cadena_actual[i] = 10`, `sublista_invertida = [10]`
        - `for j in sublista_invertida:`
            - resultado $= 10 + 10 = 20$
            - `¿Es mayor que n? Sí`
        - $i = 3$, `cadena_actual[i] = 5`, `sublista_invertida = [10, 5]`
        - `for j in sublista_invertida:`
            - $5 + 10 = 15$
            - Llamada recursiva: `(15, [1, 2, 4, 5, 10, 15], [1, 2, 4, 8, 12, 14, 15])`
    - En la llamada recursiva `(15, [1, 2, 4, 5, 10, 15], [1, 2, 4, 8, 12, 14, 15])`:
        - ¿El último número de `cadena_actual` es igual a n? Sí
        - ¿La longitud de `cadena_actual` es menor que `cadena_mínima`? Sí
        - Se actualiza: `cadena_mínima = [1, 2, 4, 5, 10, 15]`
12. Más adelante, se obtuvieron:
    - $[1, 2, 3, 6, 12, 15]$
    - $[1, 2, 3, 6, 9, 15]$
    - $[1, 2, 3, 5, 10, 15]$
13. Pero nos quedamos con la primera mínima encontrada:

    `[1, 2, 4, 5, 10, 15]`

3.5. Análisis de Complejidad

Para encontrar la complejidad computacional de nuestro algoritmo, podemos observar que todo lo que esta por fuera del ciclo `for` es:

$$\mathcal{O}(1)$$

ya que se trata de comparaciones y obtencion de elementos. La unica excepcion ocurre cuando encontramos una cadena que nos lleva al resultado n, ya que en ese caso debemos copiar todos los valores de la cadena encontrada a `cadena_minima`, lo cual implica:

$$\mathcal{O}(n)$$

Ahora que entendimos eso, el problema principal radica en los ciclos `for` y en la llamada recursiva.

El primer ciclo `for` itera sobre los elementos de la cadena actual:

$$\mathcal{O}(n)$$

El segundo ciclo `for` itera sobre una sublista inversa dependiente de la variable i, por lo tanto, tambien implica:

$$\mathcal{O}(n)$$

Combinando ambos bucles, estamos realizando n veces una operacion de orden n, resultando en:

$$\mathcal{O}(n^2)$$

Ahora nos falta analizar la llamada recursiva. Aqui estamos explorando los posibles valores que se pueden agregar a la cadena, en busca de la cadena minima. Entonces lo que hacemos aquí es agregar o no el resultado de la suma, dicha decisión se repite de manera recursiva y genera un tipo de árbol de decisión, donde cada nivel de profundidad tiene la opción de aceptar o no el número.

Dado que en cada llamado recursivo, tenemos una cantidad n de elementos en nuestra cadena actual, el algoritmos tiene que explorar las posibles combinaciones de estos elementos, dado que cada combinación es agregar o no agregar a nuestra cadena actual, esto se vuelve en la complejidad computacional:

$$\mathcal{O}(2^n)$$

Finalmente, combinando la parte recursiva con los ciclos anidados, obtenemos:

$$\boxed{\mathcal{O}(n^2 \cdot 2^n + n)}$$

Dado que el termino n es de menor orden, se puede omitir en la notacion asintotica, quedando como complejidad final:

$$\boxed{\mathcal{O}(n^2 \cdot 2^n)}$$

3.6. Conjunto de Datos

En el archivo **758Resultados.txt** se encuentran los resultados de las pruebas realizadas sobre los valores de N que van desde 1 hasta 758, utilizando el archivo **generar_txt_cadenas_minimas.py**. Para obtener las mediciones, se realizó un test de volumen, acumulando los tiempos de ejecución de cada N. Estos tiempos se dividieron en bloques de 1 a 100, de 1 a 200, y así sucesivamente hasta llegar a 758.

Además, se incluyen las mediciones unitarias, que muestran el comportamiento de cada N sin hacer la división por bloques, lo que permite ver la tendencia de los tiempos de ejecución a medida que N aumenta.

3.7. Graficos

3.7.1. Tiempos de Ejecucion

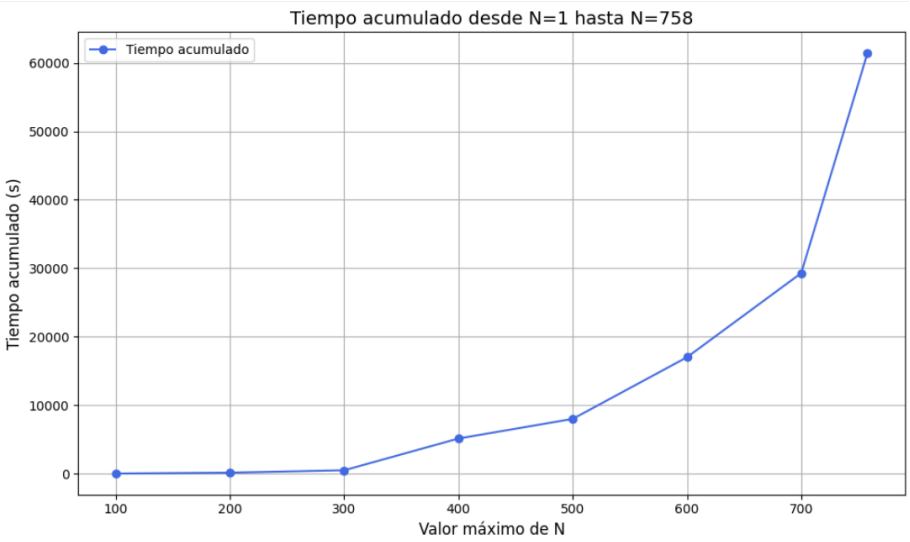

Figura 3: Gráfico que relaciona el valor de **n** con el tiempo de ejecución acumulados por bloque

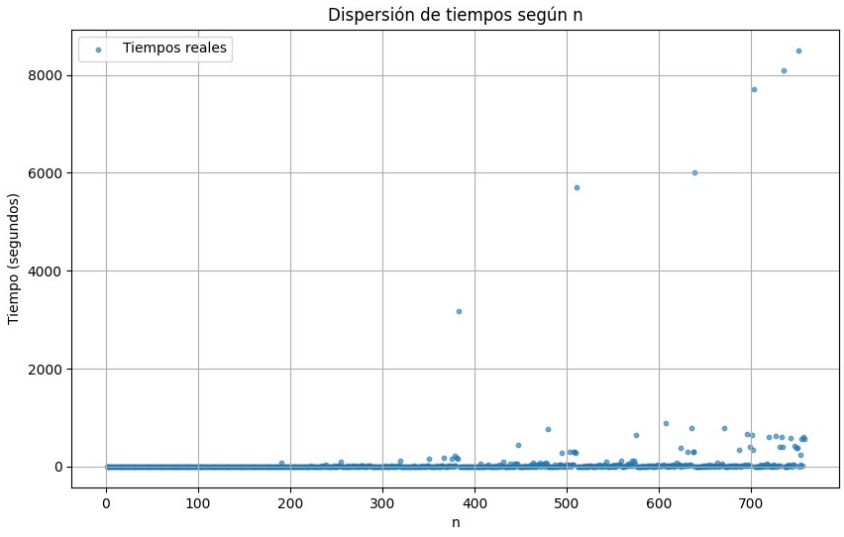

Figura 4: Gráfico que relaciona el valor de **n** con el tiempo de ejecución unitario

Se observa en el grafico Figura 4 que coindicen con la complejidad temporal analizada previamente, ya que el mismo tiene una tendencia a ser exponencial.

A continuación, se expone la tabla con los datos obtenidos:

Bloques	**Tiempo de Ejecución Acumulado (s)**
N = 1 a 100	4.3167
N = 1 a 200	133.7772
N = 1 a 300	480.5587
N = 1 a 400	5120.1612
N = 1 a 500	7987.0196
N = 1 a 600	16994.0263
N = 1 a 700	29288.0145
N = 1 a 758	61431.8092

Las mediciones fueron realizadas en una computadora con las siguientes características:

- **Procesador:** [Intel Core i5- 1135G7 2.40 GHz]
- **Memoria RAM:** [8GB de RAM]
- **Sistema Operativo:** [WINDOWS 11]
- **Intérprete de Python:** [Python 3.12.9]

3.8. Conclusión

Después de todo el laburo que hicimos con este problema, podemos decir que el algoritmo encaja dentro de la metodología de Backtracking. A lo largo del desarrollo, fuimos viendo cómo funcionan las podas y el impacto que tienen al momento de buscar soluciones óptimas. Nos costó bastante entender y estimar su complejidad, pero gracias a las mediciones que hicimos con distintos valores de N, pudimos ver que el comportamiento apunta claramente a un crecimiento exponencial.