TDA TP2

Agustin Hejeij, Renzo Jacinto

5 de mayo de 2023

Contents

1 Ejercicio 1

El objetivo de este ejercicio es demostrar que este problema es NP completo. Para ello, deben cumplirse 2 condiciones: la primera es que el problema depe ser NP (la solución debe ser validable en tiempo polinomial); la seguna es que exista otro problema NP-Completo que sea reducible al problema que estamos analizando. Para ello utilizaremos los conceptos de reducción vistos en clase, buscando un problema que NP completo que sea reducible a nuestro problema del empaquetamiento. Vale aclarar que esto corre sobre el problema de decisión de empaquetamiento: *¿Es posible almacenar los objetos en k envases de tamaño 1?*

Comenzamos primero probando la condición más sencilla, que el problema del empaquetamiento se encuentra en NP. Para eso partimos de una solución, y tenemos que poder evaluar si es una solución válida en tiempo polinomial. Digamos que queremos validar si una solución es válida para "*¿Es posible almacenar los objetos en k envases de tamaño 1?*". Es fácil ver que esto es $O(n^2)$, siendo n la cantidad de elementos, por lo siguiente:

- Que la cantidad de envases sea menor a k. Esto es $O(1)$, es validar la longitud de la solución
- Que se incluyan todos los elementos en la solución. Habría que validar que cada elemento esté en la solución, esto sería $O(n^2)$ dado que en el peor de los casos habría que recorrer toda la solución con cada uno de los n elementos.
- Que no haya elementos duplicados. Esto es $O(n^2)$, dado que habría que recorrer la solución nuevamente con cada elemento y ver que se encuentre una sola vez.
- Que ningun envase supere 1. Esto es $O(n$, dado que habría que sumar los contenidos de cada envase, que a su vez termina pasando por todos los elementos.

Entonces, como la solución se puede validar en tiempo polinomial, el problema se encuentra en NP.

Luego tenemos que encontrar un problema NP-Completo que se pueda reducir al del empaquetamiento. Para ello, partiremos del problema de 3-Partition, el cuál consiste en un conjunto de S de $3n$ números enteros positivos con suma nT, e intenta decidir si es posible dividir el conjunto de enteros en n conjuntos de tamaño 3, disjuntos, cuya suma sea igual, y que cubran todo S. Este problema está probado NP completo, y es posible resolverlo usando una caja negra que resuelva el problema del empaquetamiento.

¿Cómo reducimos este problema al de empaquetamiento? Sabemos que la cantidad de bins máxima es n, pero ¿cómo hacemos que los tamaños sean entre 0

y 1?. Acá es donde entra en juego T. Como la sumatoria de todos los elementos del conjunto es nT, entonces la suma de cada subconjunto deberá ser T. Sería equivalente a dividir los elementos en n envases de tamaño T. El problema de decisión de 3-Partition tendría solución si y solo sí es posible empaquetar los $3n$ elementos en n envases de tamaño T.

Sabiendo esto, si dividimos cada elemento por T entonces cada subconjunto tendría que sumar 1, y esto es posible saberlo usando el problema de decisión de empaquetamiento propuesto, preguntando: *¿Es posible almacenar los objetos en n envases de tamaño 1?*. Veamos un ejemplo.

Supongamos el siguiente conjunto de datos $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Hay 9 números, entonces $n = 3$. La suma de todos los números es 45, entonces $T = 15$. ¿Es posible dividir S en 3 subconjuntos que sumen 15?

Si dividimos cada valor por 15, obtenemos $S = \{1/15, 2/15, 3/15, 4/15, 5/15, 6/15, 7/15, 8/15, 9/15\}$. ¿Es posible empaquetar todos los elementos en 3 envases de tamaño 1? A simple vista podemos ver que con estos 3 conjuntos $\{1/15, 9/15, 5/15\}$, $\{2/15, 6/15, 7/15\}$, $\{3/15, 4/15, 8/15\}$, es posible almacenar los elementos en 3 envases. Por lo tanto, es posible dividir S en 3 subconjuntos de suma 15.

Realizar esta transformación tendría complejidad $O(n)$ con lo cual con un llamado al problema del empaquetamiento y un conjunto de pasos polinomiales, podemos resolver 3-Partition usando el problema del empaquetamiento propuesto. Entonces, **3-Partition es reducible polinomialmente al problema del empaquetamiento**.

Por lo tanto, si el problema del empaquetamiento se encuentra en NP, y como 3-Partition es NP completo y reducible al problema del empaquetamiento, por propiedad de transitividad, el problema del empaquetamiento es NP completo.

2 Ejercicio 2

2.1 Análisis del algoritmo y su complejidad

El algoritmo de Fuerza Bruta/Backtracking que implementamos, se basa en en la combinación de 2 partes:

1) Primero, buscamos todas las formas posibles (en adelante, combinaciones) de colocar n paquetes en diferente cantidad de envases de diferentes tamaños. Por ejemplo, para 2 paquetes hay dos posibles combinaciones, colocar ambos paquetes en un envase único, o colocar a cada paquete en un envase:

[[2 paquetes]] y [[1 paquete], [1 paquete]]

Para 3 paquetes hay tres posibles combinaciones:

[[1 paquete], [2 paquetes]] - [[1 paquete], [1 paquete], [1 paquete]] y [[3 paquetes]]

A continuación el código:

```
def get_combinations_rec(target, current_sum, start, output, result):
    if current_sum == target:
        output.append(copy.copy(result))

    for i in range(start, target):
        temp_sum = current_sum + i
        if temp_sum <= target:
            result.append(i)
            get_combinations_rec(target, temp_sum, i, output, result)
            result.pop()
        else:
            return

def get_combinations(packages):
    target = len(packages)
    min_containers = ceil(sum(packages))
    output = []
    result = []
    get_combinations_rec(target, 0, 1, output, result)
    output.append([target])
    output = list(filter(lambda comb: len(comb) >= min_containers, output))
    output.sort(key=len)
    return output
```

El algoritmo va guardando la suma parcial e iterando recursivamente hasta que llega a una combinación, y luego la agrega a la lista final de combinaciones. Esto tiene una complejidad de $O(2^n)$, siendo n la cantidad de paquetes.

Una vez obtenidas las combinaciones posibles, sabemos que la mínima cantidad de envases que va a tener la solución, es la suma de los tamaños de los paquetes redondeado para arriba, entonces, filtramos las combinaciones con menos cantidad de envases. Esto tiene una complejidad de $O(n)$

Luego, devolvemos las combinaciones de menor a mayor cantidad de envases, ya que una vez que encontremos la primer solución, sabremos que es una de las óptimas (ya que pueden haber mas de una solución con la mínima cantidad de envases posible). Esto tiene una complejidad de $O(nlogn)$

2) Segundo, buscamos todas las formas posibles (en adelante, permutaciones) de colocar n paquetes en n posiciones. Esto es el equivalente a buscar las distintas formas de distribuir n elementos de un arreglo. Usamos el arreglo que va desde 0 hasta n-1, haciendo referencia al indice de los paquetes en el arreglo de paquetes.

Por ejemplo, las diferentes permutaciones para un arreglo de indices de 2 paquetes es la siguiente:

[0, 1] - [1, 0]

Para 3 paquetes:

[0, 1, 2] - [0, 2, 1] - [1, 0, 2] - [1, 2, 0] - [2, 0, 1] - [2, 1, 0]

A continuación el código:

```
from itertools import permutations

def get_permutations(nums):
    for perm in permutations(nums):
        yield perm
```

Vimos que la librería de *itertools* tiene la función *permutations* que realiza esto, su complejidad es de $n!$. En adición para no tener que almacenar en memoria todas las posibles permutaciones, le agregamos la primitiva *yield* de python para que calcule y devuelva la siguiente permutación cuando sea necesario.

Usando estos 2 algoritmos, llegamos al algoritmo para hayar la solución exacta:

```
def packaging_backtracking(packages):
    combinations = get_combinations(packages)
    indexes = [i for i in range(len(packages))]
    for combination in combinations:
        permutations = get_permutations(indexes)
        for permutation in permutations:
            solution = []
            counter = 0
            valid_solution_set = True
            for container_size in combination:
                container = []
                for i in range(container_size):
                    container.append(packages[permutation[counter]])
```

```
            counter += 1
        if (sum(container) > 1):
            valid_solution_set = False
            break
        solution.append(container)
    if (valid_solution_set):
        return solution
```

Para cada combinación (que vienen de menor a mayor cantidad de envases) ubicamos a los paquetes en los envases de la combinación, con todas las posibles permutaciones del arreglo de paquetes. La cantidad de posibles combinaciones se calcula de la siguiente manera (colocar n paquetes en k envases, para cada k posible):

$$\sum_{k=T}^{n} \binom{n+k-1}{k-1} = \frac{(n+k-1)!}{(k-1)!(n-1)!}$$

Siendo T la suma de los tamaños de los paquetes redondeada para arriba:

$$T = \left\lceil \sum_{i=1}^{n} Ti \right\rceil$$

Si durante el agregado de un paquete a un envase, este último supera la cantidad de 1, se sigue con la siguiente permutación.

Una vez que se encuentra la primer solución válida se la devuelve. Sabemos que es la óptima ya que, como explicamos antes, las combinaciones vienen ordenadas por cantidad de envases de menor a mayor.

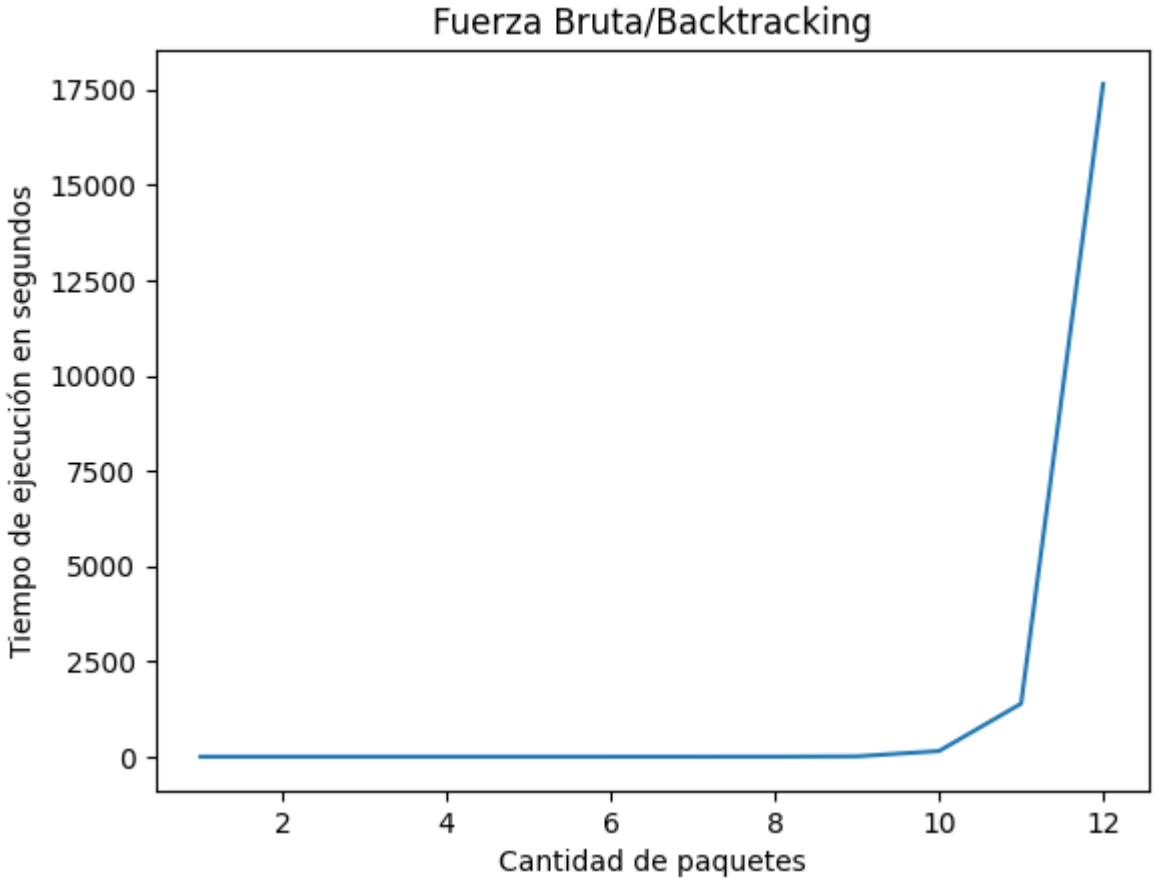
Para la complejidad del algoritmo total, el término de $O(2^n)$ queda opacado frente al término de $O(n!)$. Por lo tanto, la complejidad es la siguiente:

$$O(n! \sum_{k=T}^{n} \frac{(n+k-1)!}{(k-1)!(n-1)!})$$

.

2.2 Mediciones

Realizamos mediciones para n desde 1 hasta 12, ya que n mas grandes tardaba demasiado tiempo de lo que podía calcular nuestro hardware. Para cada n, realizamos 3 ejecuciones diferentes, promediamos el tiempo y lo graficamos. Este es el resultado:



Vemos que a medida que aumentamos, el tiempo que le toma al algoritmo crece abruptamente, como era esperado.

3 Ejercicio 3

Dada la aproximación propuesta por el enunciado, buscamos implementarla, y analizar que tan buena es comparándola con la solución exacta.

3.1 Análisis del algoritmo y su complejidad

El código de la implementación del algoritmo de aproximación es el siguiente:

```
def packaging_aproximation(packages):
    solution = []
    while (len(packages) > 0):
```

```
        container_size = 1.0
        container = []
        for package in packages:
            if (package < container_size or same_value(package, container_size)):
                container.append(package)
                container_size -= package
            else:
                break
        for package in container:
            packages.remove(package)
        solution.append(container)
    return solution
```

El algoritmo comienza con un envase vacío, y empieza a agregar paquetes en orden de llegada mientras entren en el envase. Una vez que el envase se llenó, elimina esos paquetes del arreglo original para no tenerlos en cuenta en siguientes iteraciones.

Como pasa una vez por cada paquete, la complejidad es $O(n)$, por lo que, comparado con la solución exacta, es mucho mejor opción. Ahora, es lo suficientemente buena como para elegirla sobre la solución exacta?.

Para eso, vamos a comparar las soluciones de ambos algoritmos para el mismo set de paquetes, y ver que tan mejor es la exacta de la aproximada, y también ver que tan peor puede llegar a ser la aproximada (buscaremos una cota de la relación entre las soluciones como fue sugerido en el enunciado).

De la solución aproximada, asumiendo que tiene una cantidad de envases par, tomamos envases consecutivos de a pares. Si o si, la suma de los tamaños de los paquetes de los pares de envases es mayor estricto a 1 (no puede ser igual a 1, ya que si no se hubieran puesto en el primer envase) y menor o igual a 2:

$$1 < T_i + T_{i+1} \leq 2 \ \forall \ i \ impar$$

Siendo *A(I)* la cantidad de envases de la solución aproximada, si sumamos la inecuación anterior para los tamaños de todos los paquetes:

$$\frac{A(I)}{2} < \sum_{i=1}^{n} T_i \leq A(I)$$

Siendo *z(I)* la cantidad de envases de la solución exacta, también sabemos:

$$\sum_{i=1}^{n} T_i \leq z(I)$$

Ya que no pueden haber menos envases que la suma de los tamaños de los paquetes. Por lo tanto, uniendo ambos resultados, nos queda que:

$$\frac{A(I)}{z(I)} \leq 2$$

Entonces, como peor escenario, la solución aproximada nos dará el doble de envases que la solución óptima.

Ahora, para una cantidad de envases impar, podemos pensar algo parecido. Tomamos de a pares de envases, y dejamos el último envase sin pareja (ya que hay una cantidad impar de envases). Para las parejas de envases tenemos el mismo resultado que antes:

$$1 < T_i + T_{i+1} \ \forall \ i \ impar$$

Y para el envase que quedó solo (n), lo único que sabemos es lo siguiente:

$$0 < T_n$$

Si hacemos la suma de todas las inecuaciones, nos queda lo siguiente:

$$\frac{A(I) - 1}{2} < \sum_{i=1}^{n} T_i$$

Como también sabemos que:

$$\sum_{i=1}^{n} T_i \leq z(I)$$

Entonces:

$$\frac{A(I) - 1}{2} < z(I)$$
$$A(I) < 2z(I) + 1$$

Ahora bien, como *z(I)* y *A(I)* son números enteros y la inecuación es estricta, podemos restar 1 del lado derecho y hacer a la inecuación menor o igual, quedando lo siguiente:

$$A(I) \leq 2z(I)$$

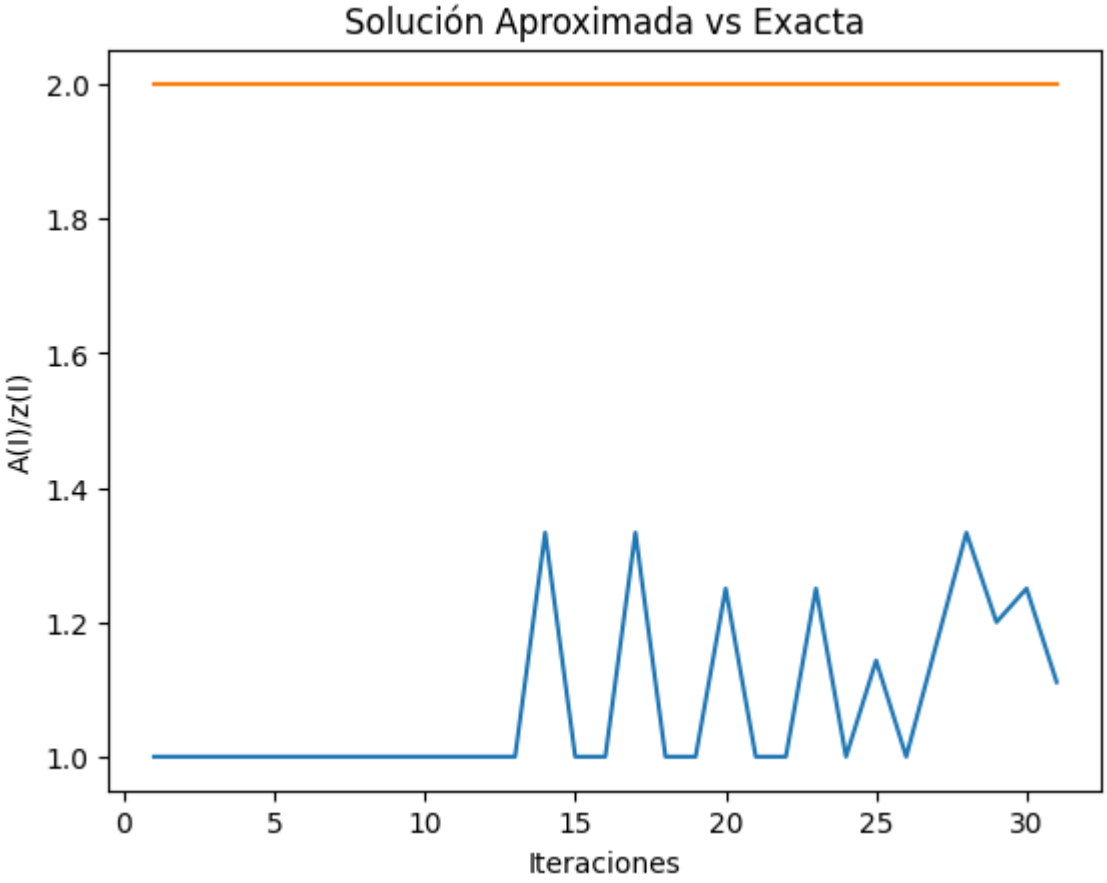
Por lo tanto:

$$\frac{A(I)}{z(I)} \leq 2$$

Como conclusión final, no importa cual sea la cantidad de envases de la solución aproximada, siempre se va a cumplir que la relación entre ésta y la solución óptima como mucho es 2.

3.2 Mediciones

Para corroborar esta cota, realizamos mediciones evaluando diferentes sets de paquetes para cada algoritmo (el mismo set en cada iteración), con n entre 1 y 11, para ver que la relación de las soluciones siempre diera menor o igual a 2. Este fue el resultado:



Como podemos observar, la relación siempre fue inferior a la cota calculada previamente, para las mas de 30 iteraciones que probamos.

También corrimos la aproximación para los archivos con datos de paquetes que nos proporcionaron en el Drive, estos fueron los resultados:

- env3.txt: 13 paquetes
  - Solución óptima: 3 envases
  - Solución aproximada: 4 envases
- env10.txt: 29 paquetes
  - Solución óptima: 10 envases
  - Solución aproximada: 12 envases
- env10b.txt: 28 paquetes

  - Solución óptima: 10 envases
  - Solución aproximada: 13 envases

- env10c.txt: 32 paquetes
  - Solución óptima: 10 envases
  - Solución aproximada: 13 envases

- env20.txt: 57 paquetes
  - Solución óptima: 20 envases
  - Solución aproximada: 26 envases

- env40.txt: 111 paquetes
  - Solución óptima: 40 envases
  - Solución aproximada: 53 envases

- env60.txt: 167 paquetes
  - Solución óptima: 60 envases
  - Solución aproximada: 81 envases

Se puede comprobar a ojo que la solución aproximada siempre es menor al doble de la óptima, haciendo cumplir la cota de la relación entre ambas explicada anteriormente.

4 Ejercicio 4

Proponemos la siguiente aproximación:

Tomamos un paquete, y lo metemos en el envase que entre mas justo. Si no entra en ningún envase, creamos uno nuevo con ese paquete dentro.

4.1 Análisis del algoritmo y su complejidad

Para realizar esto, utilizamos un heap de minimos para poder obtener los envases de menor a mayor tamaño restante por llenar. El código es el siguiente:

```
def packaging_aproximation_students(input_packages):
    packages = copy(input_packages)
    min_heap = []
    aux = []
    for package in packages:
        package_appended = False
        while (min_heap and not package_appended):
            (remaining_size, container) = heapq.heappop(min_heap)
```

```
            if (package < remaining_size or same_value(package, remaining_size)):
                container.append(package)
                remaining_size -= package
                package_appended = True
                heapq.heappush(min_heap, (remaining_size, container))
                break
            else:
                aux.append((remaining_size, container))
        for (remaining_size, container) in aux:
            heapq.heappush(min_heap, (remaining_size, container))
        aux = []
        if (not package_appended):
            new_container = [package]
            container_size = 1.0 - package
            heapq.heappush(min_heap, (container_size, new_container))
    return min_heap
```

El algoritmo comienza generando un envase con el primer paquete, y luego va desencolando del heap de minimos los envases creados hasta el momento, si el paquete actual no entra en el envase que se está observando, se continúa desencolando envases. Para no perder los envases descartados en la iteración actual que se fueron desencolando, se utiliza un arreglo auxiliar para luego de terminar de iterar sobre el heap volver a encolarlos.

En cuanto a la complejidad, para el paquete número i, se desencolan y encolan en el heap de mínimos una cantidad de veces que a lo sumo es i(en el caso por ejemplo, de que todos los paquetes tengan tamaño 1, siempre se va a tener que crear un envase nuevo y se va a recorrer todo el heap porque no hay espacio en ningún envase). Sabemos que tanto para encolar como para desencolar de un heap la complejidad es $O(logn)$. Por lo que la complejidad final queda:

$$O(2\sum_{i=1}^{n} i \ log \ n)$$

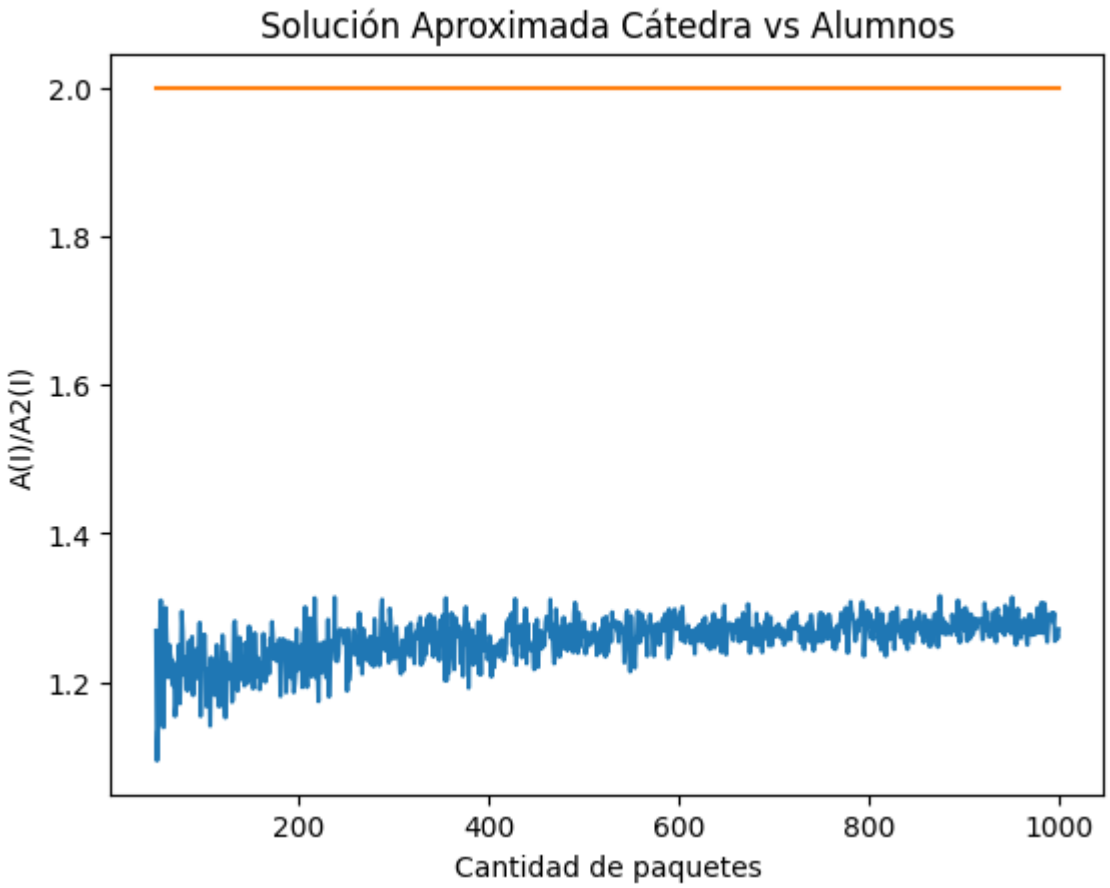
O, reduciendo la expresión:

$$O(n \ log \ n)$$

4.2 Mediciones

Medimos la relación entre la solución aproximada del ejercicio 3, y la solución de nuestro algoritmo, para cantidad de paquetes entre 50 y 1000 (de 1 en 1),

estos fueron los resultados:



En todas las iteraciones, la relación dió mayor a 1, por lo que aunque nuestro algoritmo tenga una peor complejidad, encuentra una mejor solución, en un tiempo razonable.

Ahora, corrimos esta nueva aproximación que los sets de datos del Drive para compararlos con la solución exacta y aproximada del punto anterior. Estos fueron los resultados:

- env3.txt: 13 paquetes
  - Solución óptima: 3 envases
  - Solución aproximada: 4 envases
  - Solución aproximada alumnos: 4 envases
- env10.txt: 29 paquetes

  - Solución óptima: 10 envases
  - Solución aproximada: 12 envases
  - Solución aproximada alumnos: 11 envases

- env10b.txt: 28 paquetes
  - Solución óptima: 10 envases
  - Solución aproximada: 13 envases
  - Solución aproximada alumnos: 11 envases

- env10c.txt: 32 paquetes
  - Solución óptima: 10 envases
  - Solución aproximada: 13 envases
  - Solución aproximada alumnos: 11 envases

- env20.txt: 57 paquetes
  - Solución óptima: 20 envases
  - Solución aproximada: 26 envases
  - Solución aproximada alumnos: 22 envases

- env40.txt: 111 paquetes
  - Solución óptima: 40 envases
  - Solución aproximada: 53 envases
  - Solución aproximada alumnos: 42 envases

- env60.txt: 167 paquetes
  - Solución óptima: 60 envases
  - Solución aproximada: 81 envases
  - Solución aproximada alumnos: 64 envases

Para todos los casos, se vió que ésta aproximación dió mejores soluciones que la propuesta en el punto 3, por lo que es una buena alternativa a la solución óptima.