



Trabajo Práctico N°1

Técnicas de Diseño

[75.29/95.06] Teoría de Algoritmos I
1^{er} Cuatrimestre 2023

Emanuel, Tomas	108026
Javes, Sofia	107677

Índice

Primera Parte	2
Teorema Maestro del Algoritmo Propuesto	2
Descripción del algoritmo con heaps	3
Implementación de ambos algoritmos	5
Análisis de complejidad	10
Conclusiones adicionales	12
Segunda Parte	13
Algoritmo greedy	13
Algoritmo de Programación Dinámica	15
Detalles de cada algoritmo	17
Pruebas	18

Primera Parte: Problema de K-merge por División y Conquista

Teorema Maestro del Algoritmo Propuesto

Analizando el algoritmo propuesto pudimos llegar a las siguientes conclusiones: Con la expresión del Teorema Maestro:

$$T(n) = AT(n/B) + f(n)$$

Donde:

- A - La cantidad de llamados recursivos en el algoritmo
- B - En cuanto se divide el problema inicial
- $f(n)$ - En referencia a la parte no recursiva del algoritmo. Es la complejidad de la parte no recursiva

Tomando en cuenta esto, la lista de arreglos se divide en 2 en cada llamado recursivo, por lo tanto $B = 2$, tenemos 2 llamados recursivos, osea que $A = 2$:

```
1  izq = k_merge(izq)
2  der = k_merge(der)
```

En cuanto a la complejidad de lo no recursivo, podemos decir que es $O(n)$ en el peor de los casos. Al dividir la lista en izq y der estamos a lo sumo usando la mitad de elementos, tomando por fuera las constantes de $/2$. Sabiendo que $K \cdot h = n$ nos queda $O(n)$ también con el algoritmo de merge:

```
1  izq = lista_arreglos[:medio] #O(medio) -> a lo sumo O((K*h)/2) -> O(K*h)
2  der = lista_arreglos[medio:] #O(fin - medio+1) -> a lo sumo es O((K*h)/2) -> O(K
3  *h)
```

Entonces, tomando en cuenta el Teorema maestro:

- K - Cantidad de arreglos ordenados
- H - Cantidad de elementos de cada arreglo
- n - $K \cdot H$, la cantidad de elementos totales
- La expresión: $T(n) = AT(n/B) + f(n)$
- Con: $A = 2$, $B = 2$, $f(n) = O(n)$

Para determinar la complejidad:

$$n^{\log_b a} \rightarrow n^{\log_2 2} \rightarrow n^1 \rightarrow n$$

Por lo tanto:

$$n = f(n) = n$$

Como $f(n)$ y $n^{\log_b a}$ son asintóticamente iguales:

$$T(n) = O(n^{\log_b a} * \log n) = O(n^1 * \log n) = O(n * \log n)$$

La complejidad computacional del algoritmo propuesto por división y conquista es:

$$T(n) = O(n * \log n)$$

Describir el algoritmo que utiliza heaps, y determinar su complejidad.

El algoritmo propuesto utiliza un heap para encolar y desencolar los elementos que se encuentran en todos los arreglos. En primer lugar, encolamos el primer elemento de cada arreglo. Una vez hecho esto (como todos los arreglos están ordenados, es el más pequeño), iniciamos una iteración hasta que el heap que utilizamos se encuentre vacío. El procedimiento es simple, desencolamos una vez el número del heap, y vemos cual es el siguiente en el arreglo que provino ese número, y lo encolamos al heap. Esto es análogo (desencolar del heap y ver cual es el siguiente del número que provino ese número) hasta que no tenemos más números en un arreglo, por lo que no encolamos más nada de este arreglo y seguimos desencolando del heap. Una vez que la posición de todos los arreglos se encuentre en el final, desencolamos elementos restantes del heap y los añadimos al arreglo resultado. Dicho arreglo resultará completamente ordenado.

Descripción gráfica:

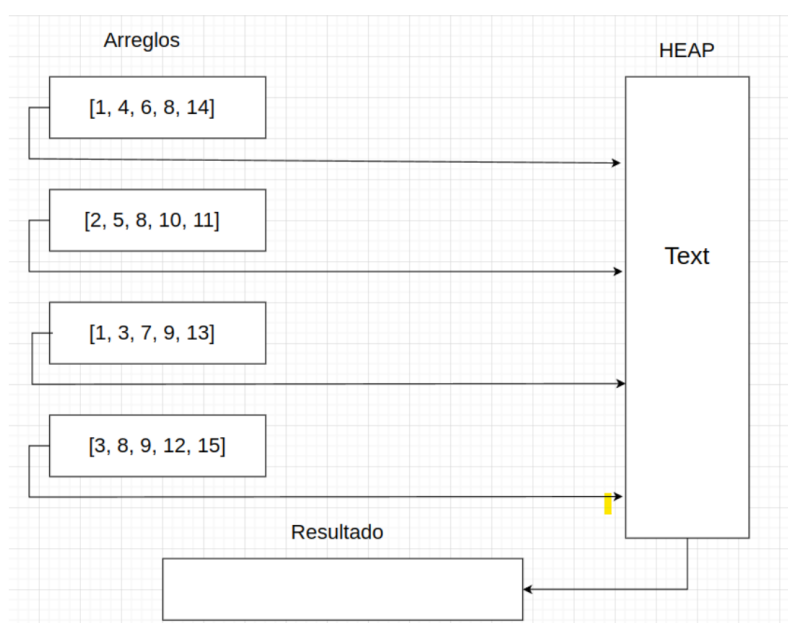
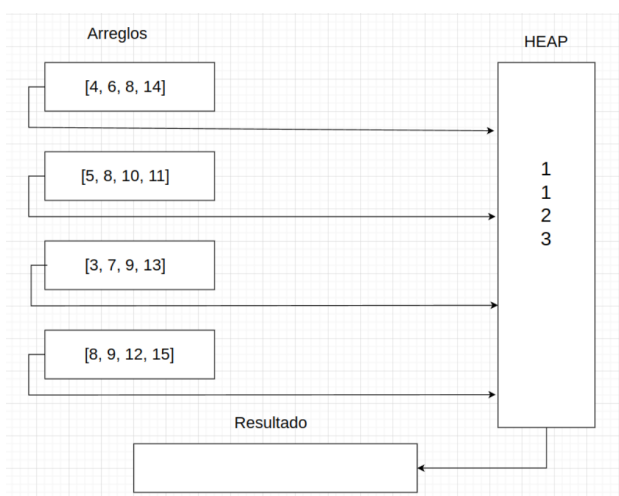
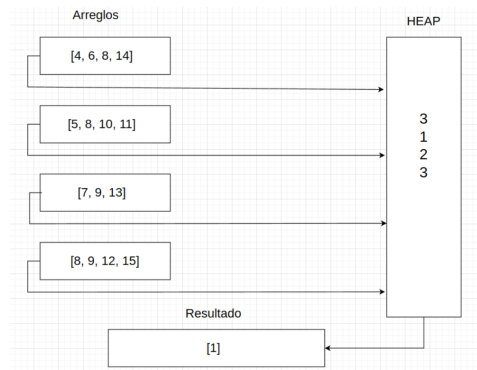


Figura 1: Situación inicial.

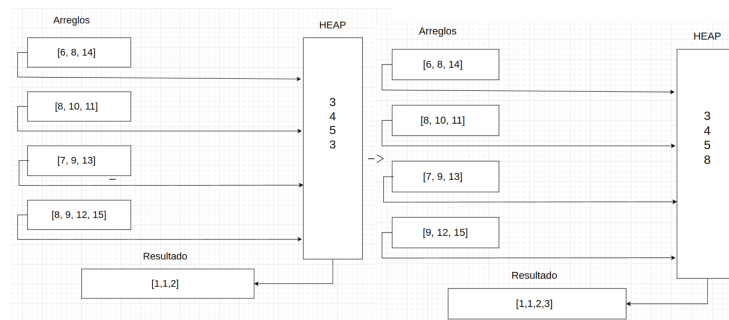
Luego encolamos los primeros elementos de cada arreglo:



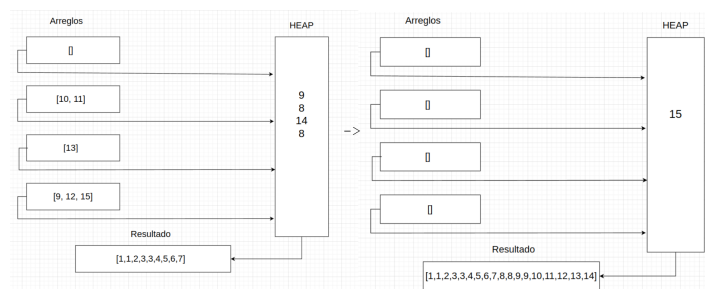
Desencolamos del heap el mínimo y del arreglo que provino ese mínimo en colamos el siguiente elemento y agregamos el mínimo desencolado al resultado:



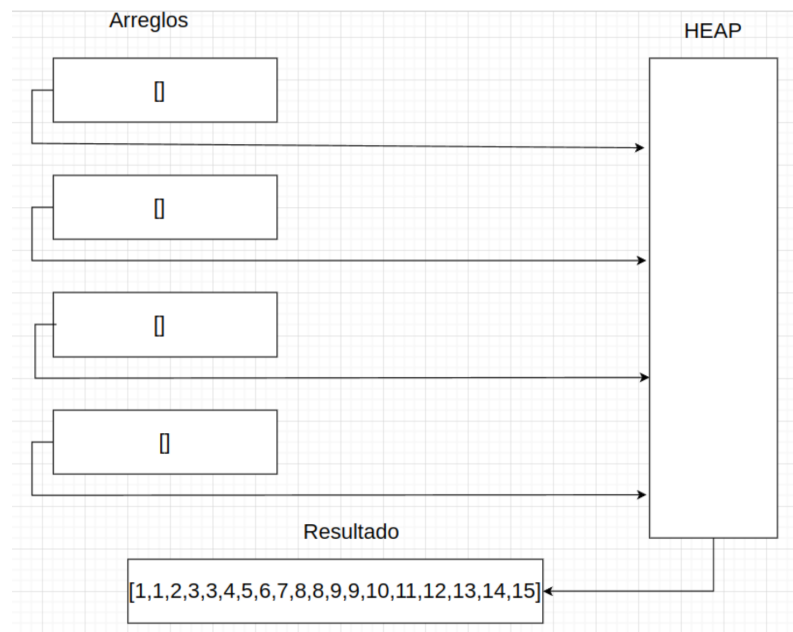
Seguimos con el mismo procedimiento, desencolar del heap y encolar al próximo del arreglo que provino, hasta quedarnos sin ningún numero mas de algún arreglo.



Si seguimos con el algoritmo: (Cabe resaltar que el desencolamiento cuando son dos elementos iguales no tiene ninguna preferencia, no debería tener una política de desencolamiento).



y por último desencolamos el último elemento del heap y nos queda el resultado:



Complejidad: Al principio encolamos los primeros elementos de todos los arreglos, como son k arreglo, y el encolamiento de los heaps son logaritmos, nos queda un $\log(k)$, siendo k los arreglos totales. Luego, el desencolar y encolar solamente es $\log k$, ya que como mucho tenemos k elementos en el heap (ya para lo último serían menos elementos en el heap, pero se toma lo peor), pero como lo tenemos que hacer hasta que cada uno de los arreglos se encuentre vacío, o sea un total de $k \cdot h$, porque cada arreglo tiene h elementos, quedaría en total:

$$O(k * h * \log k)$$

Tomando:

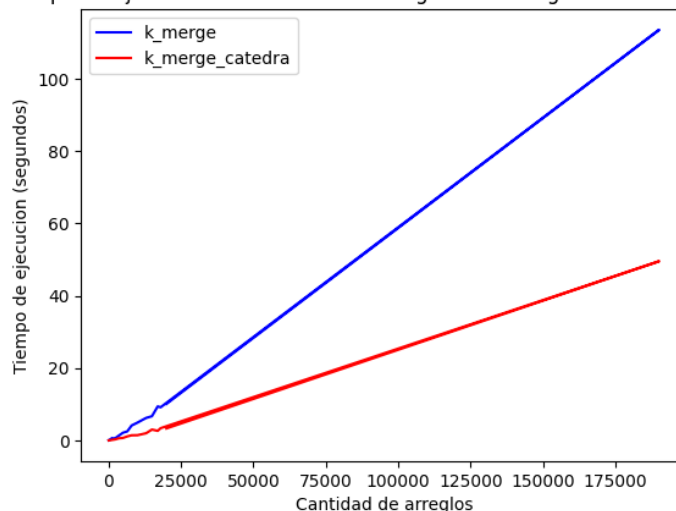
$$n = k * h \rightarrow O(n * \log k)$$

Implementar ambos algoritmos, y hacer mediciones (y gráficos) que permitan entender si las complejidades obtenidas para cada uno se condicen con la realidad.

Para poder corroborar y estar seguros de que las complejidades calculadas en un principio realizamos pruebas en donde vamos variando las 2 variables del problema k y h .

Para la primera prueba, mantenemos fijo el número de elementos por arreglos (100) y subimos el número de arreglos en total.

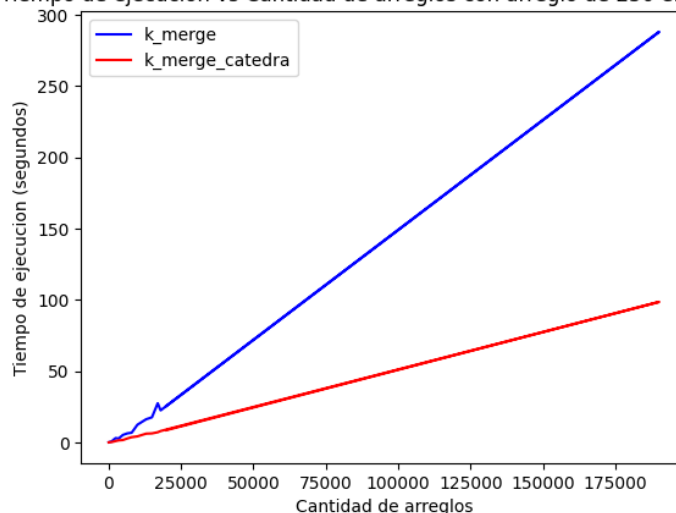
Tiempo de ejecución vs Cantidad de arreglos con arreglo de 100 elementos



En esta primera observación se puede ver como ambos algoritmos se comportan de manera similar a medida que aumentamos la cantidad de arreglos. Podemos ver como el gráfico de *kmerge* crece más rápido en comparación con el algoritmo de la cátedra. Lo que a grandes rasgos se podría creer que estamos trabajando con algoritmos de complejidad logarítmica, nos animaríamos a decir que el *kmegrecatedra* parece crecer de una forma más lineal.

Para la siguiente prueba realizamos lo mismo que la anterior pero aumentamos el valor fijo de los elementos por arreglo de 100 a 200.

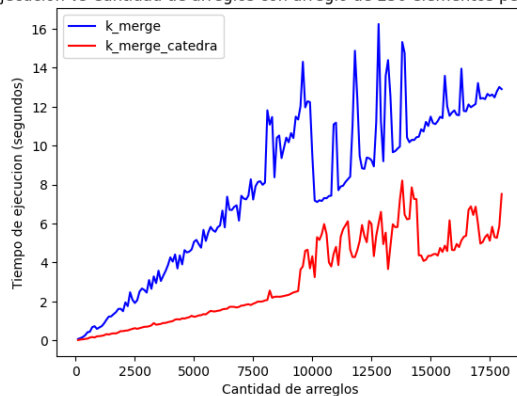
Tiempo de ejecución vs Cantidad de arreglos con arreglo de 250 elementos



Se puede ver que no hay mucha diferencia con el gráfico de la prueba anterior pero se puede notar un leve distanciamiento de las complejidades ya que pareciera como si el algoritmo de *kmerge* tardara un poco más que en la prueba anterior y el otro algoritmo parece tener menos pendiente. De igual forma podemos observar que la primera medición comienza en 50 segundos mientras que la de la prueba anterior comienza en 20. Entonces, al aumentar a 250 los elementos de cada arreglo pero mantener ese número fijo mientras aumentamos la cantidad de arreglos, no hay cambios notables con respecto a la prueba anterior.

Para la tercera prueba decidimos dejar fija la cantidad de arreglos pero aumentar la cantidad de elementos por arreglo. En esta prueba se puede ver como crece el tiempo de ejecución de forma más irregular.

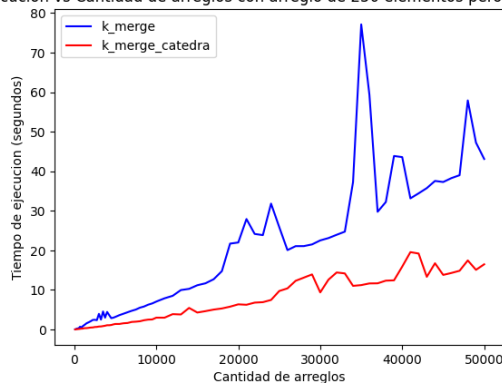
Tiempo de ejecución vs Cantidad de arreglos con arreglo de 250 elementos pero pasos mas chicos



Se observa que a partir de los 10.000 arreglos aproximadamente ambos algoritmos comienzan a tener crecimientos mas irregulares. Pero a grandes rasgos, se puede ver como siguen un patron de crecimiento dentro de todo lineal para el *kmegrecatedra* y logaritmico para el *kmerge* normal.

Como bien especifica el titulo del grafico, para esta prueba decidimos hacer algo similar que en la prueba anterior pero en este caso la cantidad de arreglos aumenta a pasos mas grandes, de a 10.000 elementos.

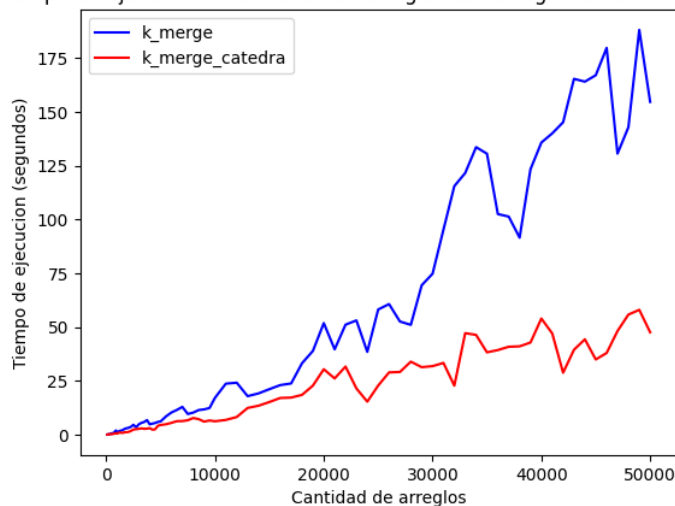
Tiempo de ejecución vs Cantidad de arreglos con arreglo de 250 elementos pero hasta 50 mil arreglos



Obtenemos resultado similares que la prueba anterior, pero en este caso al hacer pasos mas grandes, los saltos no son tantos. Igualmente podemos ver el mismo crecimiento hasta el gran salto que se da cuando la cantidad de arreglos es aproximadamente 32.000.

En esta prueba aumentamos también la cantidad de arreglos con la misma diferencia de pasos que la prueba anterior pero con la diferencia de que cada arreglo contiene 500 elementos, es decir, el doble con lo que estábamos testeando en la prueba anterior.

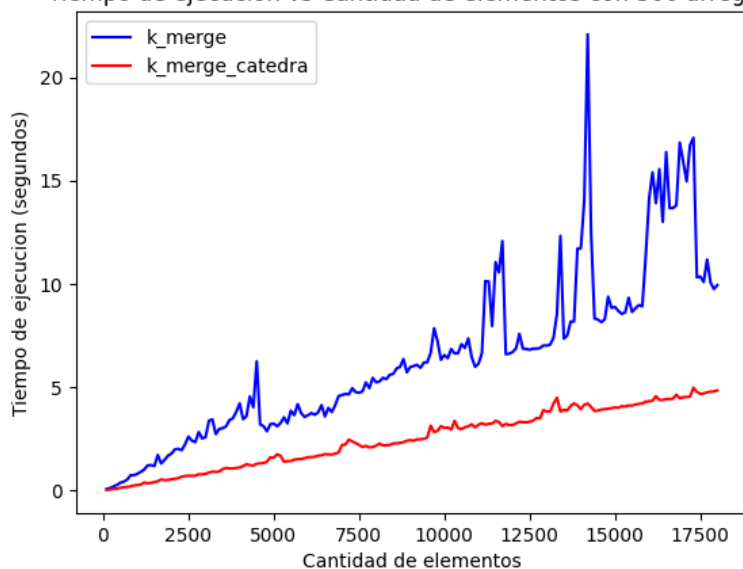
Tiempo de ejecución vs Cantidad de arreglos con arreglos de 500 elementos



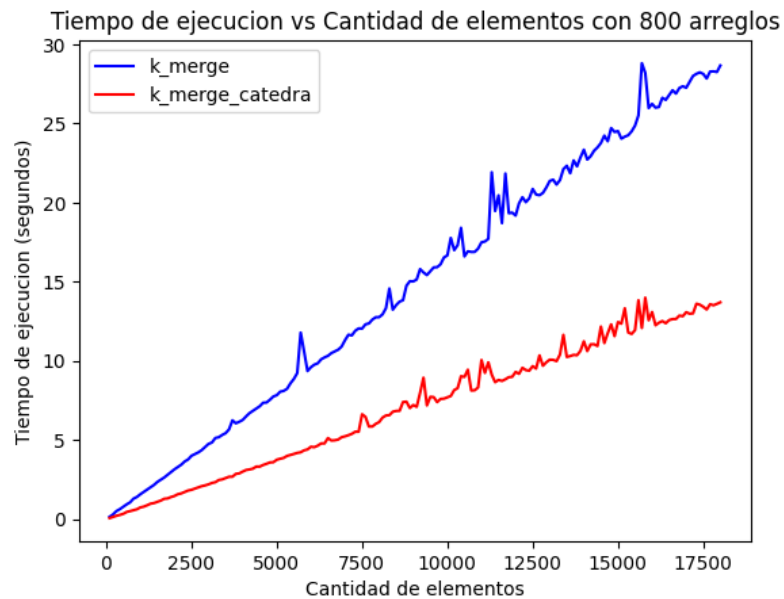
A medida que aumenta la cantidad de arreglos se puede ver que la diferencia entre las complejidades aumenta levemente también. De igual manera, el comportamiento del crecimiento continúa siendo similar a lo que estuvimos observando en los casos anteriores.

Para la sexta prueba hicimos variable la cantidad de elementos pero manteniendo fija la cantidad de arreglos, en 300.

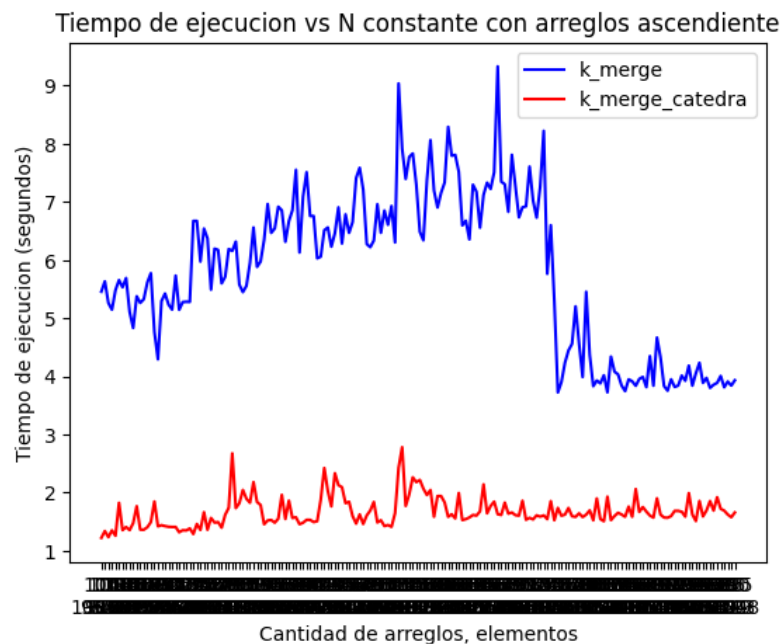
Tiempo de ejecución vs Cantidad de elementos con 300 arreglos



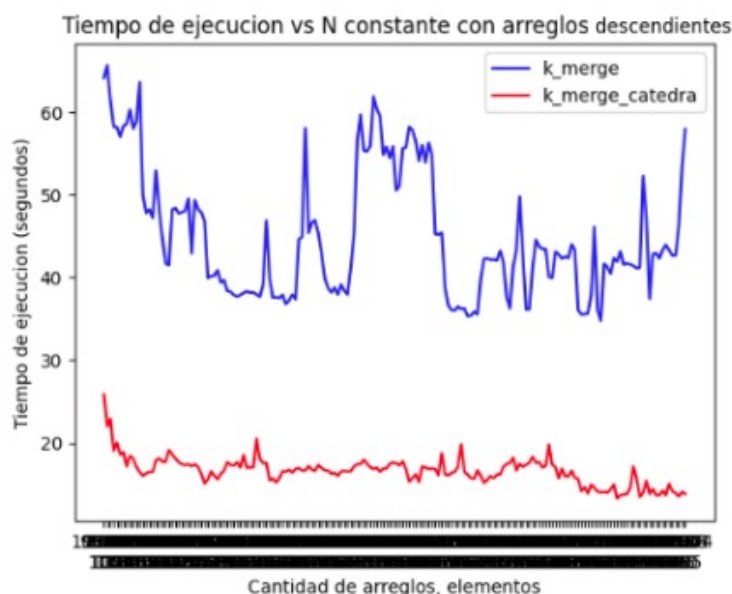
En este caso se puede observar que mientras aumentamos la cantidad de elementos el algoritmo de *k_merge* demora más que el algoritmo propuesto por la cátedra. El *k_merge* de la cátedra parece no tener tantas irregularidades como el otro algoritmo a la hora de ordenar. Cabe destacar de igual manera que los compartimientos no varían demasiado, a grandes rasgos parecen comportarse casi de manera lineal.



Para el último set de pruebas, nos pareció importante observar el comportamiento de los algoritmos intentando variar ambas variable (k y h) pero mientras una disminuye la otra aumenta, intentando así mantener la n constante.



En este caso se observa que mientras el $k_merge_catedra$ se mantiene a grandes rasgos constante, el k_merge parece que aumenta de forma lineal. Decidimos tomar como válido la subida hasta cuando hay una caída abrupta, nuestra hipótesis se basa en que de tener procesos corriendo a la par en la computadora, estos al haber terminado podrían haber afectado en como el algoritmo se ejecuta en un tiempo menor, ya que sube su prioridad en la cola de planificación



A grandes rasgos podemos observar que ambos algoritmos se comportan de igual forma (ignorando los grandes saltos).

Concluyendo con este set de pruebas podemos ver las diferencias entre las complejidades de los algoritmos dados en el problema. En este análisis podemos ver que la complejidad inicialmente propuesta con el teorema maestro para el `k_merge_catedra` puede que no sea el correcto ya que un algoritmo efectivamente logarítmico se comporta de la forma que se muestra el `k_merge`. Al haber una diferencia de crecimiento tanto variando h como k , podemos intuir que la complejidad propuesta inicialmente no es la correcta ya que, de ser logarítmica según n , debería comportarse de una forma similar al otro algoritmo.

Para verificar estas pruebas, decidimos agregar mas pruebas de lo mismo, solamente cambiando como generábamos los arreglos, esta vez utilizando la semilla de `time`, multiplicando por la variable del `for` y después agarrando el resto de la división con el rango que se nos pasa, siendo este primeramente 500. Por ultimo, decidimos cambiar el rango en donde se generaban los sets, siendo este un numero variable, para 250 elementos decidimos utilizar un rango de 1000 para que no estén tan repetidos, en un rango de 1000 utilizar otro numero mayor y así. Los resultados de estas se encuentran en las notebooks al final, notamos que cambiando la generación de arreglos y el rango de los elementos, los gráficos no tenían tantas inconsistencias como tenían los anteriores. Dado que se puede ver mas claramente en estos gráficos, nos vemos mas inclinados a saber que la complejidad empleada es la correcta.

En caso que la complejidad obtenida en el punto 1 no se condice con la realidad, indicar por qué (qué condición falla). Indicar la complejidad correcta

Como bien dijimos, gracias a los gráficos hechos en las pruebas, podemos constatar que la complejidad obtenida utilizando el Teorema Maestro no se condice con la realidad. En cambio, podemos observar, que el algoritmo a simple vista aparenta ser de complejidad $O(n)$.

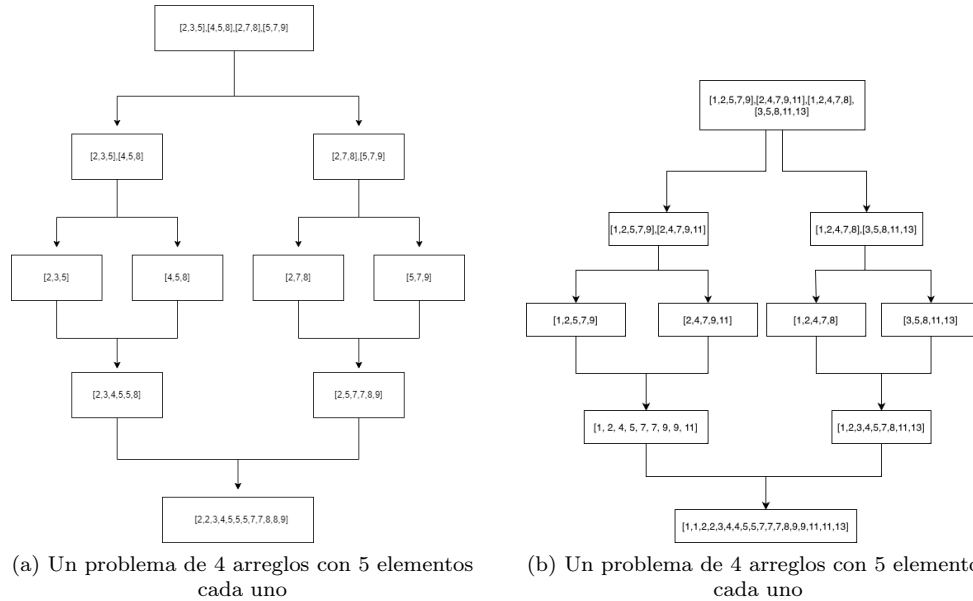
Empezando por el Teorema Maestro, y revisando las condiciones necesarias para poder utilizarlo, 1 de las 3 condiciones para efectivamente usarlo no se cumple en este caso. Siendo las 3 condiciones:

- A debe ser natural
- B es real y > 1 y es constante (siempre lo mismo)

- El caso base es constante (siempre lo mismo)

Se pueden afirmar las primeras 2 condiciones dado que A efectivamente dado que solo hay 2 llamados recursivos en el algoritmo, y B, es 2 ya que el problema siempre se divide en 2 partes, no en una variable.

En cuanto a la tercera condición, podemos hacer una comparación con el algoritmo de mergesort básico. En este algoritmo se puede aplicar el Teorema Maestro dado que el caso base siempre se aplica a 1 elemento. Por el otro lado, el algoritmo implementado en este Trabajo Practico, utiliza un caso base que no es constante, varia con la cantidad de los elementos de los arreglos.



De por sí, el Teorema Maestro toma en cuenta el h , es decir, la cantidad de elementos por arreglo para el calculo de complejidad, ya que en efecto, toma el n que contiene el h . Pero, en este caso, podemos ver, que la profundidad del arreglo siempre estara dada por la cantidad de arreglos, es decir, K . Es por esto que en este algoritmo se toma por demas la variable H en el calculo de la complejidad. Lo que resulta en una complejidad posiblemente mayor a la que en realidad es.

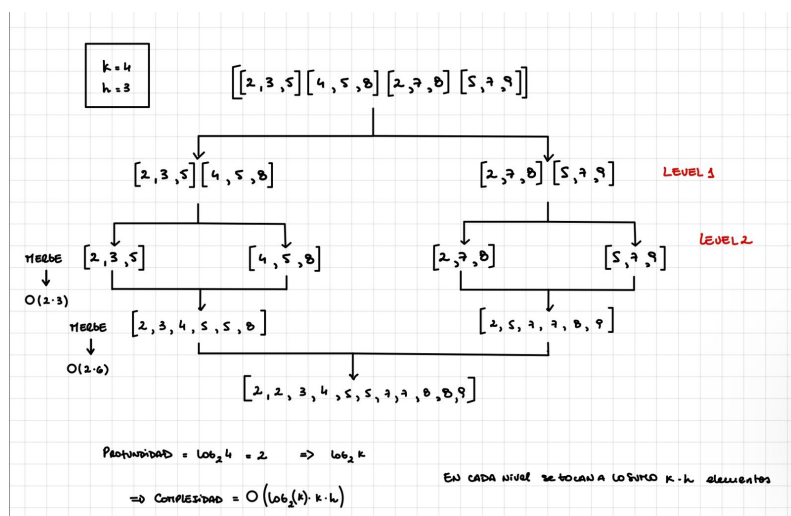
Al no poder aplicar el Teorema Maestro nos vimos en la necesidad de calcular la complejidad del algoritmo propuesto de forma tradicional. Primero definimos a partir de la notacion Big Oh, diciendo que va a llegar un punto n_0 donde se puede acotar las constantes. Esto quiere decir que trabajaremos con las cotas superiores.

$$T(n) = O(f(n))$$

$$T(n) \leq C f(n) \quad \forall n \geq n_0$$

Tomando los graficos de ayuda, podemos ver que la profundidad del arbol se basa en un $\log(k)$, en el ejemplo podemos ver, teniendo una cantidad 4 de arreglos, vamos a tener $\log_2 4 = 2$ niveles previo al merge. Luego, como hacemos un merge con todos los arreglos, teniendo $k \cdot h$ elementos, por lo que nos queda $O(k \cdot h \cdot \log(k))$.

Hicimos un Grafico para poder observar como se calcularia la complejidad:



Indicar cualquier conclusión adicional que les parezca relevante en base a lo analizado.

Teniendo en cuenta los gráficos hechos para respaldar nuestras respuestas a los algoritmos propuestos, se puede efectivamente verificar esta complejidad.

En primer lugar, podemos concluir con este tp que antes de utilizar el teorema maestro, debemos de asegurarnos que se cumplan cada una de las condiciones por mas obvias que parezcan. Una conclusión que nos deja pensando es la de porque en algunos casos puede parecer que nuestro algoritmo demora mas en la ejecución. Nuestra hipótesis para responder a este enigma es que el heap que se utiliza directo de la librería se encuentra efectivamente optimizado, dándonos así un mejor tiempo, aunque no se pierde el comportamiento general del algoritmo. Además de esto, hay algunos casos, donde se observa un corte en la continuidad de nuestro algoritmo, mostrando como se baja la linea. Esto se podría explicar por como se planifican los procesos en nuestra computadora, puede ser que un proceso grande se haya liberado justo en ese tiempo y es por eso que se obtuvo un tiempo menor que el de antes (Un ejemplo de esto el gráfico de la prueba de dejar N constante). En algunos gráficos, podemos notar como se refuta la idea de que el algoritmos propuesto esta acotado por un $\log(n)$. Un ejemplo puede ser en los gráficos de N constante. Tomando el N constante, estos gráficos deberían comportarse de manera igual (tomando en cuenta lo obtenido por Teorema Maestro), pero en la practica se puede observar un crecimiento/ decrecimiento logarítmicos en los respectivos gráficos, demostrando que la complejidad es en efecto la hallada.

Segunda Parte: ¡Problema de contrabando!

Describir un algoritmo Greedy que resuelva el caso propuesto.

Nuestra propuesta de algoritmo greedy es la siguiente. Por su característica de greedy, una consecuencia de este es una solución al menor tiempo posible, y en esto nos basamos para formular el algoritmo.

En primer lugar, nos centramos en un caso base como sería tener un soborno que requiera un solo tipo de producto, aplicándolo a nuestra problemática decidimos enfocarnos en el producto de cigarrillos. Como bien sabemos, el aduanero sólo nos pedirá aquello que tenemos (no nos pedirá cigarrillos de más ni productos que no poseemos). Un ejemplo sería, tener paquetes de cigarrillos de $[3,5,8,10,13,15]$ unidades y el aduanero nos pide 9 cigarrillos para poder pasar. En este caso, el resultado correcto sería pagar el soborno con un paquete de 10 cigarrillos sin importar que sea greedy o no porque esta es su solución óptima. Nuestra primera implementación, hacia una búsqueda lineal hasta encontrar un paquete que entrara en la cantidad de cigarrillos pedidos y lo devolvía.

Esto resolvería este primer acercamiento al problema, pero esto no es del todo así ya que nos preguntamos ¿Qué pasaría si el aduanero nos pidiera un soborno de 16 cigarrillos? ¿Cómo debería actuar nuestro algoritmo?

Entendemos que en el siguiente caso nuestro pago del soborno debería ser una caja de 15 y una de 3. El algoritmo al ser greedy sabemos que obtendrá una solución pero por su condición esta no siempre será la más óptima como lo sería si devolvieramos un paquete de 13 y un paquete de 3. La cantidad de paquetes será la misma pero el resultado será óptimo (no estaríamos devolviendo cigarrillos de más). El algoritmo greedy se centra principalmente en dar una solución que contenga lo pedido pero no siempre será lo óptimo.

Nuestra solución a este problema, es utilizar el máximo del arreglo ordenado, si vemos que ese máximo se sobrepasa del pedido por el aduanero, deberíamos hacer una combinación con ese monto, es por ello que nos guardamos una variable de incautado total para llevar cuenta de cuánto tenemos en total. En la próxima iteración, tendríamos, en el ejemplo de arriba, 15 guardado como incautado total y el arreglo quedaría para ver $[13,10,8,5,3]$. De nuevo chequeamos que el total incautado (15) más el elemento de mayor valor actual (13) no sobrepase el pedido total (16). Como el total incautado más el mayor valor actual es más que lo pedido, nuestro segundo elemento se encuentra en este arreglo. Por lo que nuevamente, hacemos una búsqueda binaria sobre el arreglo que nos quedó ($[13,10,8,5,3]$) utilizando el inicio=1 para no tener que copiar el arreglo) para encontrar el elemento que mejor nos quede con 15, que sería 3.

Como dijimos, se puede ver la deficiencia de este algoritmo, la mejor solución al problema sería devolver el paquete de 13 cigarrillos y el paquete de 3, pero se devuelve 15 y 3. Esto pasa para todos los casos en donde el soborno pedido sea mayor al máximo paquete de unidades que tengamos, no tendremos una solución exacta. Pero como es un algoritmo greedy, nos conformamos con esa solución. Luego mostraremos ejemplos donde la solución sea óptima y ejemplos donde no.

Ahora, en el caso de mas paquetes, se sigue la misma lógica. Si tuviéramos que entregar 36 cigarrillos, se agarra el máximo del arreglo ordenado $[15,13,10,8,5,3]$ (15), y se lo guarda como *incautado_{total}* y movemos el índice de inicio, verificamos si el nuevo máximo del arreglo, *osea* arreglo[inicio] mas el *incautado_{total}* es mayor que la cantidad de sobornos pedidas. En el caso de que fuera mayor, $15 + 13$, estaríamos en el caso de mas arriba, donde tendríamos que hacer la búsqueda binaria sobre los elementos restantes (un ejemplo sería con 25, $15 + 13 > 25$) y así poder encontrar que paquete deberíamos entregar en conjunto con el de 15 para llegar. Pero este caso no toma en cuenta eso, $15 + 13$ es menor que 36, por lo que deberíamos agregar el 13 al *incautado_{total}*, mover el índice inicio y verificar de vuelta. Por lo que en *incautado_{total}* nos queda $15 + 13$, donde, agarrando el nuevo máximo del arreglo, arreglo[inicio] (10), nos quedaría una mayor cantidad, ya que $15 + 13 + 10 > 36$, por lo que ahora si, hacemos una búsqueda binaria por los elementos restantes tomando en cuenta el índice inicio ($[10,8,5,3]$) para poder encontrar la solución para el problema, dando por resultado el 8, siempre utilizando un índice y no copiando el arreglo, así no complicamos la complejidad.

Casos en donde el resultado de nuestro algoritmo es óptimo podrían ser las siguientes, teniendo un arreglo de [3,5,8,10,13,15] nos pidieran algo que no sobrepase el 15, por ejemplo 9,11,5 o cualquier otro número ≤ 15 . Por el otro lado si nos pidieran algún número > 15 , donde combinando 15 + (otros máximos, 13, 10) + la solución que nos acerque al número no sea exacta teniendo otra combinación no exacta. Unos ejemplos de donde sería óptimo podría ser que nos pidieran :

Soborno Pedido (cigarrillos)	Incautado
32	15,3,5
18	15,3
40	15,13,10,3

Y ejemplos donde la solución no sea la óptima podrían ser:

Soborno Pedido (cigarrillos)	Incautado	Incautado Optimo
16	15,3	8,5,3 13,3
21	15,8	10, 8, 3
24	15,10	13, 8, 3

A continuacion detallamos un seguimiento de la resolucion greedy:

```

1
2 Tenemos las siguientes unidades disponibles -> [15,3,8,5,10,13]
3 Soborno pedido -> 16
4 Lo que entregamos -> []
5
6 Primero ordenamos el arreglo -> [15,13,10,8,5,3] inicio = 0
7 Observamos el maximo en la unidades disponibles: arreglo[inicio] = 15
8 15 > 16 ? No, tomamos el 15 y lo retiramos de la unidades disponibles subiendo
9 el indice donde se empieza
10 Lo que entregamos -> [15] inicio = 1
11
12 Repetimos nuevamente:
13 Unidades disponibles -> [13,10,8,5,3]
14 El maximo ? -> 13
15 15 + 13 > 16 ? Si. No tomamos el 13 y buscamos el elemento mas optimo haciendo
16 una busqueda binaria sobre el arreglo de unidades disponibles sabiendo que
17 deberiamos buscar el elemento mas cercano al 1 (16-15 = 1)
18
19 Teniendo 13,10,8,5,3 ya que tenemos un inicio en 1
20 -> elemento del medio = 8
21 15 + 8 > 16 ? Si, buscamos en la segunda mitad
22
23 Teniendo 8,5,3
24 -> elemento del medio = 5
25 15 + 5 > 16 ? Si, buscamos en la segunda
26
27 Teniendo 5,3
28 -> elemento del medio = 3
29 15+3 > 16, Si, pero ya no hay otra unidad mas chica por lo que nos quedamos con
30 la unidad 3 y la agregamos al soborno que devolveremos
31
32 Cumple con el soborno pedido ? 15 + 3 > 16 ? Si.
33 Devolvemos [15,3]
34

```

Un ejemplo donde devolvemos mas de 2 elementos

```

1
2 Tenemos las siguientes unidades disponibles -> [15,3,8,5,10,13]
3 Soborno pedido -> 34
4 Lo que entregamos -> []
5
6 Primero ordenamos el arreglo -> [15,13,10,8,5,3] inicio = 0

```

```

7 Observamos el maximo en la unidades disponibles: arreglo[inicio] = 15
8 15 > 34 ? No, tomamos el 15 y lo retiramos de la unidades disponibles subiendo
  el indice donde se empieza
9 Lo que entregamos -> [15] indice = 1
10
11 Repetimos nuevamente:
12 Unidades disponibles -> [13,10,8,5,3]
13 El maximo ? -> arreglo[inicio] = 13
14 15 + 13 > 34 ? No, tomamos el 13 y lo retiramos de la unidades disponibles
  subiendo el indice donde se empieza
15 Lo que entregamos -> [15,13] indice = 2
16
17 Repetimos nuevamente:
18 Unidades disponibles -> [10,8,5,3]
19 El maximo ? -> arreglo[inicio] = 10
20 15 + 13 + 10 > 34 ? Si. No tomamos el 10 y buscamos el elemento mas optimo
  haciendo una busqueda binaria sobre el arreglo de unidades disponibles sabiendo
  que deberiamos buscar el elemento mas cercano al 6 (34-(15+13) = 6)
21 indice = 2
22 Teniendo 10,8,5,3
23   -> elemento del medio = 5
24   15 + 13 + 5 > 34 ? no, buscamos en la primer mitad
25
26 Teniendo 10,8
27   -> elemento del medio = 8
28
29   y como no tenemos mas elementos en la parte derecha, devolvemos el 8 con el
  $incautadoTotal$
30
31 Cumple con el soborno pedido ? 15 + 13 + 8 > 34 ? Si.
32 Devolvemos [15,13,8]

```

Describir e implementar un algoritmo (que sea óptimo) que resuelva el problema utilizando programación dinámica.

En una primera instancia para poder resolver este problema intentamos hacerlo a través de una modificación del problema de knapsack (problema de la mochila). Partimos de ello intentando amoldarlo a nuestras necesidades y restricciones. Interpretamos el peso de la mochila como el soborno que se nos pedía pero en este caso, en vez de tratar de maximizar esto con los paquetes que teníamos intentaríamos setear esta variable como un peso mínimo al que debería llegar la mochila. Luego de intentar varias modificaciones al no obtener los resultados esperados se nos ocurrió interpretar el problema de otra manera. Donde, en vez de resolver el problema de intentar llenar la mochila con el peso mínimo resolveríamos de alguna forma su complemento". Esto significaba, tener un peso máximo (que sería la sumatoria de todos los valores de los paquetes disponibles) y tener un peso mínimo, el cual efectivamente seguiría siendo el soborno pedido. En este caso lo que hicimos fue hacer la diferencia entre este peso máximo con el peso mínimo. Con esto obtendríamos un nuevo peso que sería el peso máximo definitivo de una "nueva mochila". Una forma mas bien practica de verlo es, sin tener ninguna restriccion por parte del aduanero, nosotros quisieramos maximizar todos los productos que queremos llevar en la mochila, siendo este maximizante la suma de todos. Pero en el caso de que se nos ponga un limitante, quisieramos una vez mas maximizar lo que llevemos en la mochila con ese limitante impuesto. Al tener esto, podríamos intentar maximizar este peso máximo sabiendo que no nos pasaríamos de este, y por ende, lo que no metieramos en esta mochila (Osea, los que no entrasen en esa maximizacion), serían aquellos paquetes que debían ser utilizados para llegar al peso mínimo (soborno) de antes. Para verificar esto, pusimos como beneficio el mismo peso de los paquetes y realizamos una serie de pruebas y podemos decir, que, en efecto, funciona de la forma óptima que esperábamos.

Como bien sabemos, para poder realizar este problema con programacion dinamica hay que poder ser capaces de plantear los sub-problemas. En este caso, plantear estos como medida de la cantidad de elementos no es suficiente para saber de qué manera la presencia de un elemento

condiciona a otros. Es por esto que fue necesario agregar un arreglo extra con cada beneficio que traeria agregar cada paquete si decidieramos incluirlo. Para este arreglo, como no teniamos algun tipo de beneficio obligatorio asignamos el mismo valor del peso. Es decir, un paquete de 5 cigarrillos tendria un beneficio de 5 si se agrega, y así sucesivamente, como en un problema habitual de la mochila.

Justificar la complejidad de ambos algoritmos propuestos. Indicar casos (características y ejemplos) de deficiencias en el algoritmo greedy propuesto, para los cuales este no obtenga una solución óptima.

Tomando en cuenta que n = cantidad de cosas que el usuario tomo, h = cantidad de paquetes de una cosa, es decir que si el usuario compro 6 paquetes de cigarrillos, teniendo [15,12,14,7,2,4] una posible compra, tenemos $n = 1$ y $h = 6$ dado que hay 6 paquetes y hay una sola cosa, los cigarrillos. Mientras que en el caso de soborno tenemos algo parecido, aunque el aduanero nos pedira solamente un paquete, es decir que $h = 1$ siempre, lo que cambia es que cosas nos pedira, puede ser medialunas, naranjas del sahara, cupones para evadir el ente impositario, etc. Teniendo en cuenta la precondition de tener disponible las cosas que el aduanero nos pide, podriamos tener en el peor de los casos t = "cantidad de cosas que se nos pide al pasar", $t=n$, si el aduanero es muy estricto, aunque tomaremos dos variables, donde t siempre es $\leq n$. Para comenzar, decidimos que la forma que se recibian los datos serian dentro de un arreglo de estructuras del tipo:

```

1
2 class paquete:
3 def __init__(self, nombre, unidades):
4     self.nombre = nombre
5     self.unidades = unidades

```

donde el algoritmo greedy al igual que el de dinámica recibieron por ejemplo:

```

1
2 unidades_cigarillos = [3,5,8,10,13,15]
3 productos = []
4 unidades_coca = [12,11,7,15,2,6]
5 for unidad in unidades_coca:
6     productos.append(paquete("coca",unidad))
7 for unidad in unidades_cigarillos:
8     productos.append(paquete("cigarillos", unidad))
9
10 soborno_cigarillos = paquete("cigarillos", 29)
11 soborno_cocas = paquete("coca",7)
12 incautado = algoritmo(productos, [soborno_cigarillos,soborno_cocas]

```

y devolverian un diccionario con los paquetes de cada objeto que se dejarian en la aduana

Algoritmo greedy

En el comienzo del algoritmo greedy: decidimos guardar los productos que necesitaramos en un diccionario para poder acceder a aquellos más fácilmente, teniendo un $O(n)$, ya que se recorre todos los productos comprados por el usuario y los guarda. Teniendo en cuenta que guardar en un diccionario una lista haciendo una copia de la misma, hacer esto costaria $O(n*h)$ tomando el peor caso, donde uno sea mas grande que los otros o que todos tengan la misma longitud. Luego de eso llamamos al algoritmo un total de t veces, siendo t la cantidad de sobornos de cosas que el aduanero nos pide.

Dentro de la operacion, tenemos primero un sort, que si tomamos un algoritmo comparativo de ordenamiento tendríamos $O(h*\log(h))$ ya que tomamos el peor de los casos donde tengamos muchos productos de algo que se compro. Esto lo hacemos para poder hacer la siguiente comparacion, un while hasta que el primero que se esta viendo sea mayor que el soborno que se pide. Esto, en el caso [3,5,8,10,13,15], el aduanero nos podria pedir 51 elementos en un mal dia, teniendo que buscar por cada uno de los primeros elementos e ir sumandolos, determinando que deberiamos empezar con el 3. Esto podria tomar, en el peor de los casos $O(h)$ ya que vamos aumentando 1 en el indice y no hacemos .pop que llevaria la complejidad a $O(h^2)$ (tomamos que el append y el acceder a un diccionario es $O(1)$). Por el otro lado hacemos una busqueda binaria sobre el arreglo que nos queda, que en el peor de los casos seria cuando tenemos que buscar sobre toda la cantidad de productos, teniendo $O(\log h)$. Mientras que lo otro que se hace es preguntar si esta algo en un diccionario, y ponerle una cantidad a ese valor, siendo todo $O(1)$ segun las complejidades de python.

Llevamos en la funcion de operacion un total de $O(h \cdot \log(h) + h + \log(h))$, pero esta operacion se llama t veces, dado que serian t sobornos que se estan pidiendo, teniendo en el algoritmo greedy una complejidad de $O(n + t \cdot (h \cdot \log(h) + h + \log(h)))$ que podemos achicar en $O(n + t \cdot h \cdot \log(h))$.

Aclaracion, se devuelve el incautado en formato de diccionario. Si quisieramos devolverlo en formato de lista donde siguiendo el orden de los sobornos pedidos tendriamos que agregar una complejidad de $O(t)$, pero no pensamos que fuera mas entendible devolverlo de esta manera.

Algoritmo de programación dinámica

Al igual que el algoritmo greedy, empezamos con hacer un diccionario de los productos para poder tener los paquetes con sus respectivas unidades, que cuesta $O(n)$. Luego al empezar la operación dinámica, hacemos la suma de todos los elementos de los productos, que es $O(n)$. Cómo pensamos el problema como un problema de la mochila, la complejidad de serie: $O(n \cdot 2^m)$, donde W se calcula a partir de la suma total de todos los elementos restado al soborno que se nos pide, y m son los bits que se usan para escribir ese W . Luego de esto, vamos para atrás en la matriz y luego usando los índices que se devolvieron de la ejecución de la mochila usando W agarramos los productos que se deben incautar. Las dos llamadas son $O(n)$ dado que se revisan todos los elementos en el for, por lo que la llamada a la operación dinámica se resuelve en $O(n + n \cdot 2^m + n + n)$ y esto se llama por cada soborno que se toma, por lo que resumiendo seria $O(t \cdot n \cdot 2^m)$.

Pruebas

Para comenzar, en los dos algoritmos utilizamos un set para probar su correcto funcionamiento, dado que haciamos cambios en el algoritmo, queriamos saber si estos seguian funcionando de la manera en que fueron escritos, estas pruebas fueron escritas con los valores que deberian devolver, y estas fueron:

```

1
2 test_soborno_con_exactos_13_cigarrillos
3 test_soborno_con_exactos_3_cigarrillos
4 test_soborno_con_menos_cigarrillos_1
5 test_soborno_con_menos_cigarrillos_2
6 test_soborno_con_menos_cigarrillos_3
7 test_soborno_con_mas_cigarrillos_pero_suma_de_dos_exacta_1
8 test_soborno_con_mas_cigarrillos_pero_suma_de_dos_exacta_2
9 test_soborno_con_mas_cigarrillos_pero_suma_de_dos_exacta_3
10 test_soborno_con_mas_cigarrillos_pero_suma_de_dos_exacta_4
11 test_soborno_con_mas_cigarrillos_pero_suma_de_dos_no_exacta_1
12 test_soborno_con_mas_cigarrillos_pero_suma_de_dos_no_exacta_2
13 test_soborno_con_mas_cigarrillos_pero_suma_de_dos_no_exacta_3
14 test_soborno_con_mas_cigarrillos_pero_suma_de_dos_no_exacta_4
15 test_soborno_con_mas_cigarrillos_pero_suma_de_mas_de_dos_exacta_1
16 test_soborno_con_mas_cigarrillos_pero_suma_de_mas_de_dos_exacta_2
17 test_soborno_con_mas_cigarrillos_pero_suma_de_mas_de_dos_exacta_3
18 test_soborno_con_mas_cigarrillos_pero_suma_de_mas_de_dos_exacta_4
19 test_soborno_con_mas_cigarrillos_pero_suma_de_mas_de_dos_exacta_5
20 test_soborno_con_mas_cigarrillos_pero_suma_de_mas_de_dos_no_exacta_1
21 test_soborno_con_mas_cigarrillos_pero_suma_de_mas_de_dos_no_exacta_2
22 test_soborno_con_mas_cigarrillos_pero_suma_de_mas_de_dos_no_exacta_3
23 test_soborno_con_mas_cigarrillos_pero_suma_de_mas_de_dos_no_exacta_4
24 test_soborno_con_mas_cigarrillos_pero_suma_de_mas_de_dos_no_exacta_5
25 test_soborno_con_dos_o_mas_productos_1
26 test_soborno_con_dos_o_mas_productos_2

```

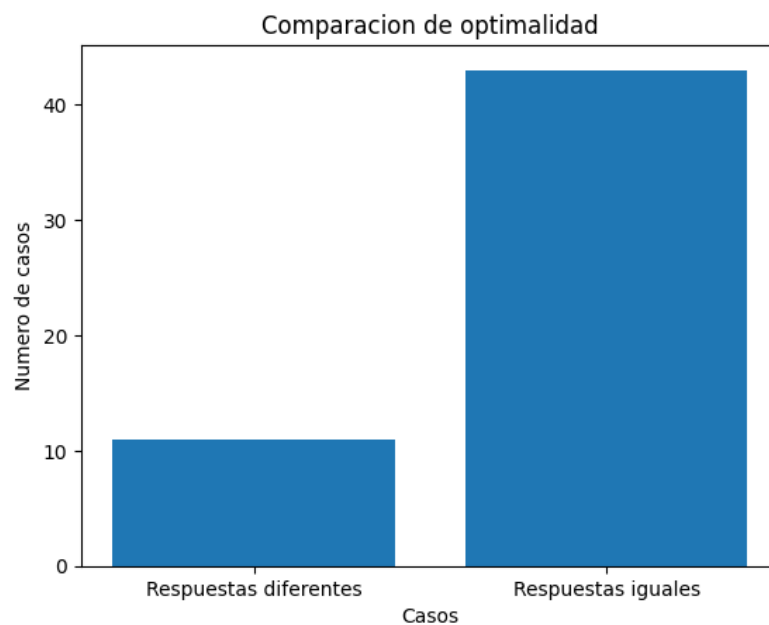
Dando un tick si el resultado era el correcto y una cruz si el resultado no era el esperado. Despues para el algoritmo de programacion dinamica, decidimos probar cada uno de los resultados de un set de datos era exacto, utilizando [3,5,8,10,13,15] para las unidades de los cigarrillos y haciendo una fuerza bruta para cada uno de los posibles resultados entre la suma de sus miembros, y luego comparando con el resultado de la suma de los miembros de la respuesta de programacion dinamica, para ver que efectivamente estuviessemos devolviendo un resultado correcto. Estos son algunas pruebas:

```

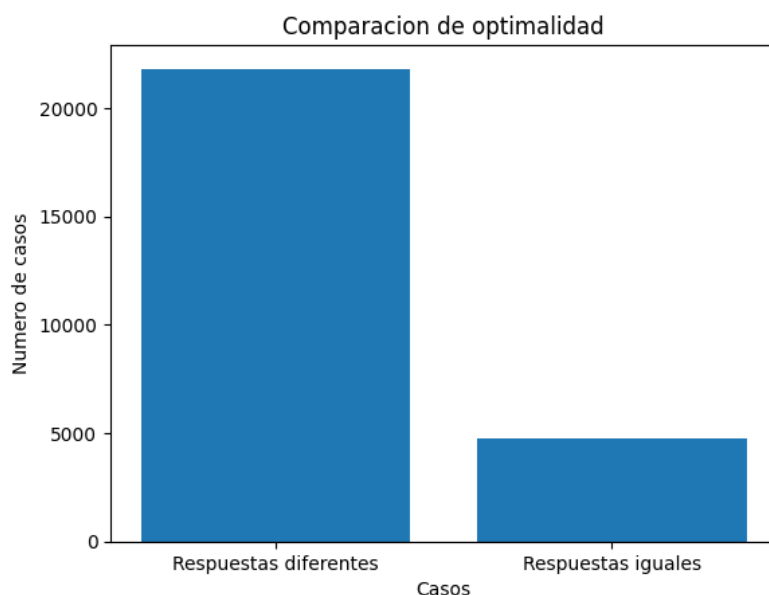
1 chequeo si es optimo con 16
2 chequeo si es optimo con 10
3 chequeo si es optimo con 13
4 chequeo si es optimo con 15
5 chequeo si es optimo con 18
6 chequeo si es optimo con 18
7 chequeo si es optimo con 21
8 chequeo si es optimo con 23
9 chequeo si es optimo con 26
10 chequeo si es optimo con 13
11 chequeo si es optimo con 16
12 chequeo si es optimo con 18
13 chequeo si es optimo con 21
14 chequeo si es optimo con 21
15 chequeo si es optimo con 24
16 chequeo si es optimo con 26
17 chequeo si es optimo con 29
18 chequeo si es optimo con 23
19 chequeo si es optimo con 26
20 chequeo si es optimo con 28

```

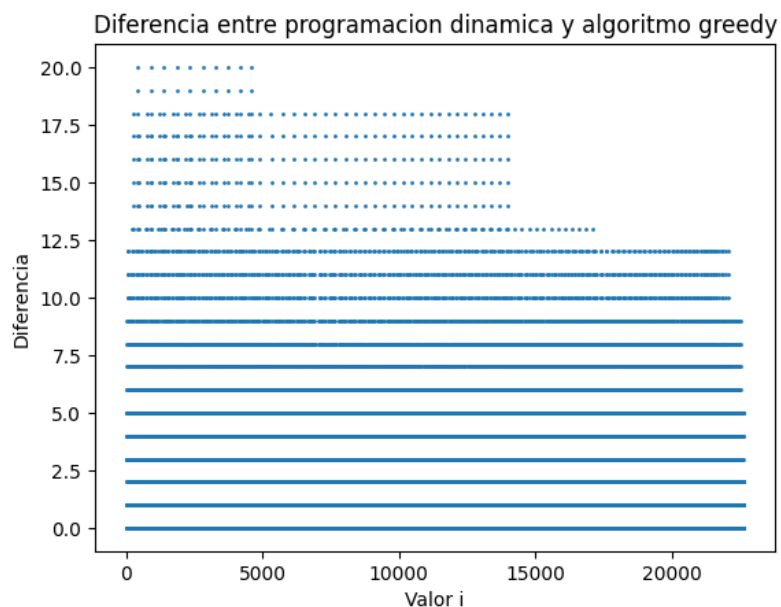
Por ultimo, utilizamos un programa que verificase la optimalidad de ambas soluciones. Es por esto, que utilizando el set anterior, fuimos por cada uno de los numeros hasta el 54, que es el resultado de la suma de todos los elementos, y verificamos si la respuesta de cada uno de los algoritmos era la misma:



Podemos ver que la diferencia de los casos aca no pareciese mucho, pero esto es porque no tenemos tantos elementos como para elegir, se podria aumentar la cantidad de elementos que nos traemos de contrabando para ampliar la diferencia y demostrar cuan optimo es el algoritmo de programacion dinamica. Para eso, utilizamos un set de datos de 100 elementos, donde esos elementos pueden tener un valor entre 1 y 500. Luego para cada uno de los valores de 1 hasta la suma de todos esos 100 elementos, decidimos probar y ver cuan diferentes eran los resultados de los algoritmos, y recibimos esto:



A modo de ejemplo, hicimos una prueba sobre el mismo set de datos anterior, pero en este caso queremos ver cuan lejos se encuentra la solución óptima, comparando el greedy con el de programación dinámica



Utilizamos otra prueba de volumen para el caso donde se nos pida dos tipos de sobornos, cigarrillos y coca, utilizando la misma tecnica previa a esto Por ultimo, si bien el algoritmo de programación dinámica siempre da una solución óptima, los tiempos de ejecución y de complejidad entre los algoritmos son muy diferentes, siendo el algoritmo greedy mucho mas rápido que el de programación dinámica. Ya con las pruebas de volumen nos podemos dar cuenta que se toma mucho tiempo para hallar la forma óptima, dando un tiempo de 4hs para correr la prueba del programa con los dos algoritmos. Es por ello, que antes de ejecutar o el algoritmo greedy o el de programación dinámica, debemos pensar si si o si necesitamos una solución óptima o una solución que puede entrar en el marco de óptimo en algunos casos, como es en el algoritmo greedy.