Universidad de Buenos Aires - FIUBA
75.29 Teoría de Algoritmos 1
TP1

Pernin Alejandro, *Padrón Nro. 92.216*
apernin@fi.uba.ar

Monsech Santiago, *Padrón Nro. 92.968*
smonsech@fi.uba.ar

17 de Mayo de 2023

Índice

1. División y Conquista

1.1. Análisis por Teorema Maestro

El algoritmo a analizar es el siguiente

```
Teniendo K arreglos ordenados de H elementos cada un algoritmo simil merge-sort

 - Caso base: cuando quede un único arreglo, simplemente devolver dicho arreglo.
 - En el caso general, dividir la cantidad de arreglos entre la primera mitad,
   y la segunda mitad, y luego invocar recursivamente para cada mitad de arreglos.
   Es decir, si tenemos cuatro arreglos, invocamos para los primeros 2,
   y luego para los segundos 2.
```

Si consideramos N como el tamaño total del problema, vemos que $N = K * H$ y acorde al teorema planteamos lo siguiente

- **a** (subproblemas): En cada recursión el problema general se divide en 2 (dos) subproblemas.
- **n/b** (tamaño subproblemas): Cada subproblema posee un tamaño de n/2, es decir la mitad del tamaño original
- $f(n)$: El costo de la división y combinación es $O(n)$

teniendo estos elementos expresamos

$$T(n) = 2 * T(n/2) + O(n)$$

siendo

$$O\left(n^{log_2(2)}\right) = f(n) = O(n)$$

entonces resulta que

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

1.2. Algoritmo de Heaps

El algoritmo de heaps se puede describir de la siguiente forma

Algorithm 1 K-merge con Heaps

```
1:  Crear array de resultado
2:  Crear Heap con el primer elemento de cada array                ▷ O(K)
3:  while Hayan elementos en los arrays do                         ▷ O(K*H)
4:      Remover la raiz del heap                                   ▷ O(log(K))
5:      Eliminarlo de su array original                            ▷ O(1)
6:      Agregarlo al resultado
7:      if El array al que pertenece el elemento no está vacio then
8:          Obtener el primer elemento y colocarlo en el heap      ▷ O(log(K))
9:      end if
10: end while
11: while Hayan elementos en el heap do                            ▷ O(K)
12:     Remover la raiz del heap                                   ▷ O(log(K))
13:     Agregarlo al resultado
14: end while
15: Devolver el resultado
```

A priori podemos aproximar una expresión de la forma

$$O(K) + O(K * H * log(K)) + O(K * log(K))$$

Siendo que $K > 1$ y $H > 1$ podemos acotar $O(K) < O(KH * log(K)$ y $O(K * log(K)) < (KH * log(K))$

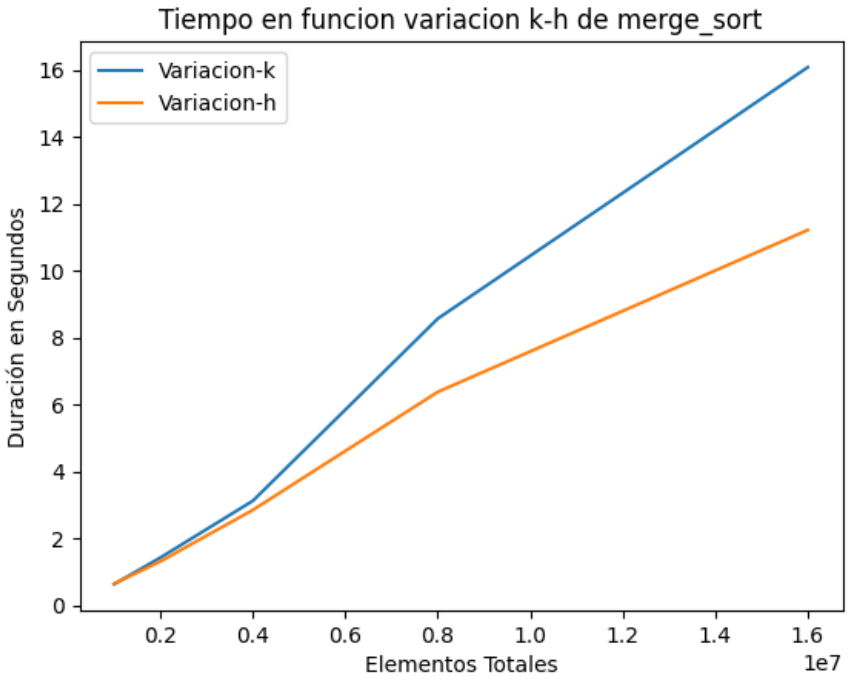
1.3. Implementación y Medición

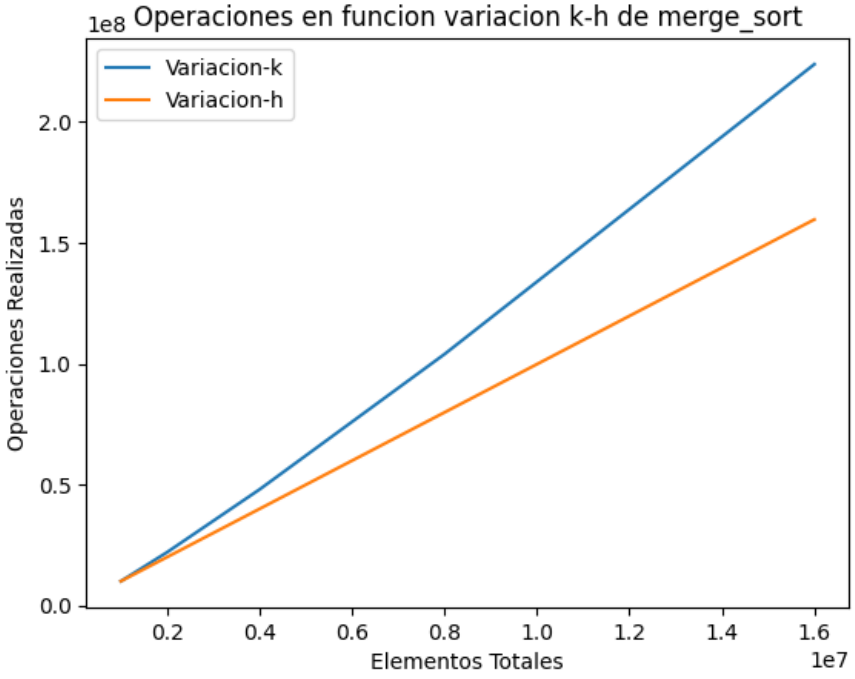
Para analizar la eficiencia de ambos algoritmos se realizaron múltiples pruebas variando los valores de **K** y **H**, midiendo tanto el tiempo que demora en obtener la solución, como contando la cantidad de operaciones realizadas.

El conteo de operaciones se realiza de forma aproximada, tomando considerandos acordes a la documentación disponible del lenguaje de implementación, en nuestro caso Python.

Las pruebas se realizaron mediante unos notebooks que pueden encontrarse en el repositorio

1.3.1. Algoritmo de Merge





Mediante el teorema del maestro se concluyó que la complejidad del algoritmo es de $O(n) = n * log(n)$ siendo $n = k * H$. En los gráficos observamos que para un mismo n alternando valores de h y k se obtienen resultados distintos, esto se debe a que el análisis del teorema del maestro se realiza sobre una sóla variable (la cantidad total de elementos) que no es completamente representativa del funcionamiento del algoritmo.

Si nos basamos en el desarrollo de la recurrencia de un merge-sort convencional, las llamadas recursivas *crean* un arbol binario de profundidad $log(n)$ y luego en cada una de estas capas se realiza trabajo por $O(n)$ (el merge en cuestión), por lo cuál la complejidad conocida del algoritmo es $N * log(n)$.

El algoritmo propuesto en contraposición, en vez de dividir los elementos totales, divide el problema en relación a la cantidad de listas, por lo cuál el arbol binario será de profundiad $log(k)$. La etapa de merge poseera la misma complejidad $O(n)$, por lo cual la complejidad total será de $k * h\ log(k)$

Esta relación es visible en los gráficos de operaciones, manteniendo k fijo y variando h se observa que el crecimiento es lineal. Mientras que manteniendo h fijo y variando k se observa un crecimiento mayor compatible con $k * log(k)$

1.3.1.1 Relación de Recurrencia

Si bien mencionamos como inferimos la complejidad real, expresaremos de forma formal el desarrollo de la recurrencia, comenzando con el caso base conocido de un merge sort tradicional

$$T(n) = 2 * T(n/2) + O(n)$$

donde poseíamos el caso base $T(1) = C_1$ sin embargo como mencionamos anteriormente nuestro algoritmo no divide en todo el espacio de elementos sino sólo en la cantidad de listas, por lo tanto el caso base será cuando haya una sola lista, es decir **H** elementos; por lo que expresamos nuestro caso base como $T(h) = C_1$

$$T(n/2) = 2\ T(n/4) + O(n/2)$$

reemplazando en la ecuación anterior

$$T(n) = 2\ [2\ T(n/4) + O(n/2)] + O(n) = 4\ T(n/4) + 2\ O(n)$$

Intuitivamente podemos predecir que luego de X recurrencias, la relación queda de la forma

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

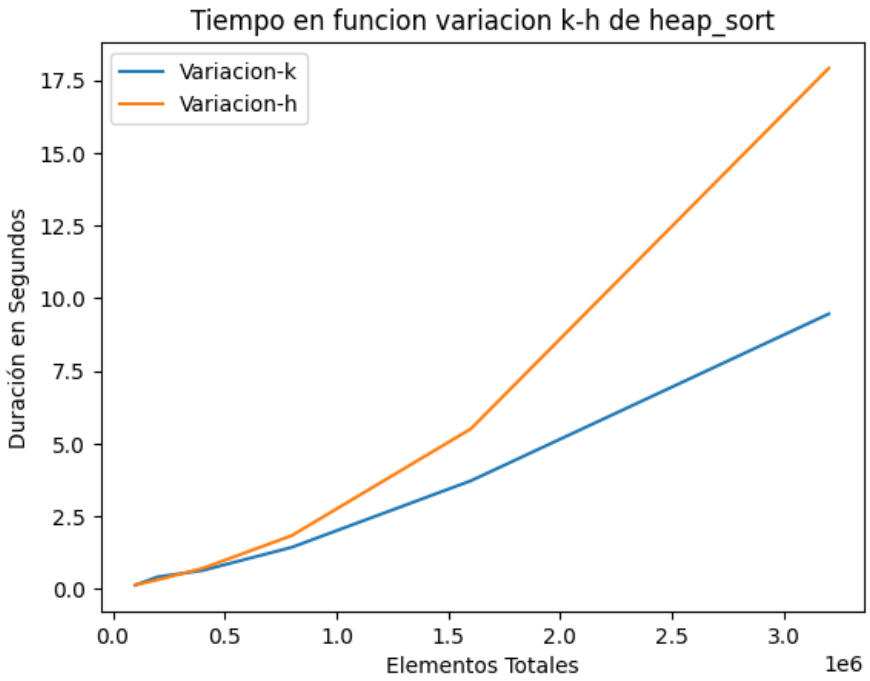
Recordando nuestro caso base $T(h) = C_1$ entonces $\frac{n}{2^X} = h \rightarrow n/h = 2^X \rightarrow X = log_2(n/h) = log_2(k)$. Volviendo a la expresión

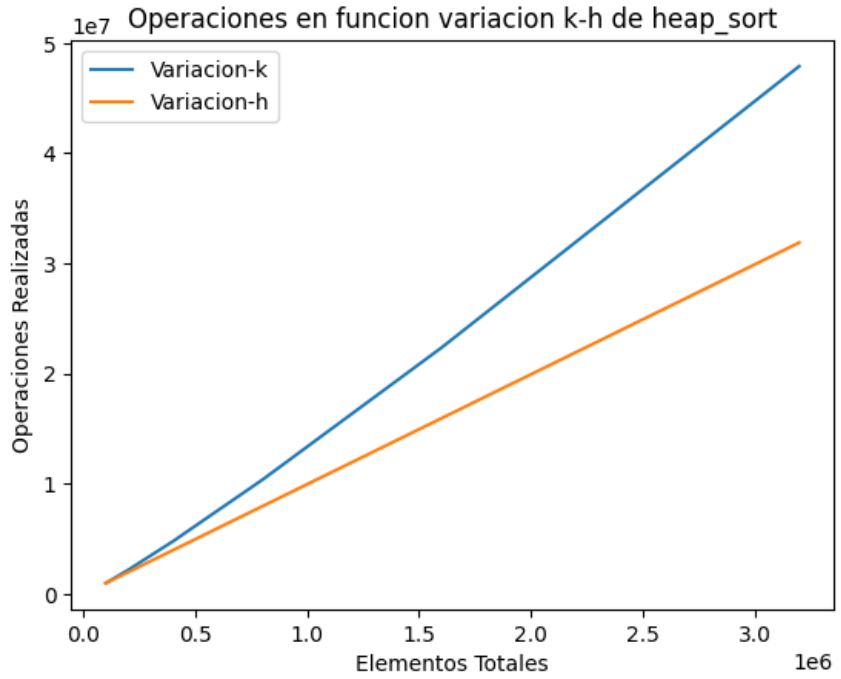
$$T(n) = log_2(k)\ O(n) + 2^{log_2(k)}\ T(h) = log_2(k)\ O(n) + k * C_1$$

donde resulta evidente que el término dominante es el logarímico por lo tanto finalmente obtenemos

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

1.3.2. Algoritmo de Heap





En este caso observamos que la complejidad en relación a las operaciones realizada es compatible con el análisis realizado previamente, sin embargo se observa una curiosidad en cuanto a los tiempos de ejecución del algoritmo.

Vemos como aún realizando (en teoría) menos operaciones en la **variación h**, el tiempo de ejecución es mayor en comparación a la **variación k**. Luego de realizar múltiples pruebas con distintos parámetros y revisar la implementación del algoritmo, llegamos a la conclusión que tal diferencia representa una disparidad entre lo analizado teóricamente con sus supuestos y la realidad de la ejecución.

Es decir, si bien el análisis de complejidad teórico es correcto, existen variables que afectan la ejecución y pueden modificar levemente los resultados. Por ejemplo dos operaciones de tiempo constante, pueden diferir entre sí su tiempo de ejecución y por ende afectar los resultados finales.

1.4. Conclusiones

Podemos concluir entonces que el Teorema Maestro posee sus limitaciones y que es necesario emplear otros recursos para encontrar expresiones que mejor definan el comportamiento de un algoritmo. También hallamos la importancia de relevar estos resultados mediante pruebas empíricas para ya sea corroborar los mismos como también darle un significado a cualquier discrepancia que uno pudiese encontrar

2. Problema del Contrabando

2.1. Solución de Algoritmo Greedy

Nuestra propuesta para resolver el problema utilizando un algoritmo Greedy consiste en plantear un simil al problema de la mochila, en donde el monto total que tenemos que dejar como soborno equivale al peso de la mochila y los productos que debemos entregar a los ítems que queremos poner en la mochila. El algoritmo lo podemos describir de la siguiente manera:

Algorithm 2 Cantidad mínima de ítems a entregar como sobrono

```
1: Agrupar los paquetes por cada tipo de producto
2: Ordenar los paquetes por unidades descendente
3: for all soborno do
4:     sobornos = []
5:     sobornado = 0
6:     for all paquetes do
7:         if sobornado <= soborno.qty then
8:             sobornos + paquete
9:             sobornado + paquete.qty
10:        end if
11:        if soborno.qty <= sobornado then
12:            break
13:        end if
14:    end for
15: end for
```

Dado que el problema tiene como supuesto que los paquetes no se pueden fraccionar para entregar la cantidad de soborno exacta, entonces nuestra propuesta greedy propone entregar paquetes con mayor cantidad de unidades y continúa hasta alcanzar o superar el monto a sobornar.

Esta primera solución corresponde a una Greedy, porque en cada paso busca contestar la decisión binaria de si incluye o no el paquete al soborno, para eso verifica si se alcanzo pagar el soborno con los demás paquetes anteriores, de no ser así considera que la mejor decisión es incluir el paquete a la solución para así garantizar llegar a una solución aunque esta no sea la óptima.

Propusimos empezar por los paquetes mas grandes, porque en general ir por lo mas pequeños primero, tiende a tener mayor pérdida de unidades para alcanzar el monto a sobornar.

Agregamos también una propuesta diferente de un algoritmo greedy, que encontramos que resuelve bien ciertos casos pero no performa mejor que el anterior en otros:

Algorithm 3 Cantidad mínima de ítems a entregar como sobrono alternativa

```
1: Agrupar los paquetes por cada tipo de producto
2: Ordenar los paquetes por unidades descendente
3: for all soborno do
4:     sobornos = []
5:     sobornado = soborno.qty
6:     total_unidades = SUM(paquete.qty)                    ▷ Calculo el total de unidades
7:     for all paquetes do
8:         total_unidades = total_unidades - paquete.qty
9:         if total_unidades = 0 then                       ▷ Es el último paquete
10:            if sobornado > 0 then
11:                sobornos + paquete
12:                sobornado -= paquete.qty
13:            else
14:                return                                   ▷ Se alcanzo el monto sin el último paquete
15:            end if
16:        else
17:            div_resto = sobornado / total_unidades
18:            if div_resto > 1 then
19:                sobornos + paquete
20:                sobornado -= paquete.qty
21:            else
22:                continue
23:            end if
24:        end if
25:    end for
26: end for
```

En esta alternativa proponemos en cada iteración calcular la relación entre lo que queda por sobornar y el total de unidades de los paquetes restantes, excluyendo al del paso actual. Con esta relación busca contestar la decisión binaria de si incluir o no el paquete a la solución, para esto analizar el resultado de la división, si el número es mayor que 1, quiere decir que lo pendiente a pagar como soborno supera el total del resto de los paquetes, por lo que no se puede excluir el paquete actual de la solución, caso contrario, si es menor que 1, la cantidad total supera a lo que falta para alcanzar el monto, entonces se puede prescindir del paquete actual y continuar repitiendo el análisis con los demas paquetes.

Como se puede apreciar, este algoritmo, busca evitar utilizar los paquetes de mayor unidades si es posible.

2.2. Solución de Algoritmo Programación Dinámica

Nuestra proupesta de solución utilizando programación dinámica utiliza la siguiente ecuación de recurrencia:

$$\text{OPT(n, bribe.qty)} = \text{MIN} \begin{cases} \text{OPT(n-1, bribe.qty)} \\ \text{OPT(n-1, bribe.qty - pj.qty)} + \text{pj.qty} \end{cases}$$

donde:

- n: es el índice del producto a en cada paso.
- *bribe.qty*: es la cantidad a sobornar
- $p_j.qty$ la cantidad de unidades que tiene el paquete j.

En cada paso se evalúa la decisión de si incluir o no el paquete en el soborno, en la rama de arriba el paquete del paso n no se incluye en el soborno el cual se podrá conservar, en la rama de abajo se refleja el caso de tener que perder el paquete para entregarlo como soborno, se ejecuta el óptimo con la cantidad a sobornar reducidas en las cantidad de unidades del producto j y se agrega el producto j a la lista de productos a entregar.

Para la implementación de la solución se utilizo una matriz que contiene:

- $\sum_{j=0}^{n} p_j$ columnas, porque se puede pagar soborno en exceso.
- n filas (una por paquete)

La tabla se popula con el siguiente algoritmo:

1. la primer fila y primer columna se cargan con el valor total a sobornar
2. iterando por cada fila, representando a cada producto (ordenados de manera descendente), para posición i,j, donde i representa el índice del producto y j representa el total de unidades aportadas, en cada celda se calcula $MIN(T(i-1,j), T(i-1, j-p_i.qty) - p_i.qty)$

Para obtener la solución final, el algoritmo resolverá empezando desde la fila n+1, seleccionando la primer columna que tenga un cero (indicando que se puede cubrir el pago del soborno sin excedente) o el máximo número negativo (indicando que por la propiedad de indivisibilidad se perderán unidades pero las mínimas posibles).
Desde ahí se ira obteniendo de manera recursiva la solución final paso a paso, descartando el producto i si T[i,j] = T[i-1, j], caso contrario incluirlo en la solución final y moverse en la tabla a la posición T[i-1][j-$p_i.qty$].

Para ejemplificar supongamos que tenemos un tipo de producto del cual se disponen 3 paquetes, con 2,2 y 1 unidades cada uno, y se debe pagar un soborno de 4 unidades del tipo de producto, la tabla tendria la siguiente forma

	0	1	2	3	4	5
p_0	4	4	4	4	4	4
p_1	4	4	2	2	2	2
p_2	4	4	2	2	0	0
p_3	4	3	2	1	0	-1

En el ejemplo anterior, partimos desde la fila p_3 seleccionando la columna 4, que resulta tener el valor de 0, como este valor es igual al de la fila anterior, quiere decir que no nos conviene incluir el paquete de 1 unidad en el soborno, calculamos la solución para p_2, como el valor es diferente al de la misma columna en p_1, entonces quiere decir que debemos incluirlo porque se formo restando 2 (cantidad de unidades del paquete p_2 al valor de la celda 2 en la posición (1,2).

Ahora para analizar el p_1 vamos a la posición (1,2) y vemos que es diferente al de la misma columna pero para el p_0 por lo que debemos incluirla en la solución, pues se formo restando 2 (cantidad de unidades del paquete p_1) con el valor en la celda (0,0). Finalmente nuestra solución constituirá de los paquetes p_1 y p_2 de dos unidades cada uno, para pagar el soborno de 4.

El algoritmo proupesto es:

Algorithm 4 Cantidad mínima de ítems a entregar como sobrono programación dinámica

```
1: Agrupar los paquetes por cada tipo de producto
2: Ordenar los paquetes por unidades descendente
3: for all soborno do
4:     sobornos = []
5:     Calcular la tabla de pagos
6:     n=cant productos + 1
7:     busco el best_match como el índice que tiene el MAX entre de todas las celdas <= 0
8:     sobornos = recurr-pay-bribe(tabla, n, best_match, paquetes)
9: end for
```

Algorithm 5 recurr-pay-bribe

```
1: if n=0 then                                                          ▷ caso base
2:     return []
3: end if
4: if tabla[n − 1][best_match] == table[n][best_match] then             ▷ Se descarta el producto
5:     return recur-pay-bribe(tabla, n-1, best_match, paq)
6: else                                                                 ▷ Se usará el producto como soborno
7:     paq = paquetes[n-1]
8:     return [paq] + recur-pay-bribe(tabla, n-1, best_match, paquetes)
9: end if
```

2.3. Análisis de las soluciones

Presentamos a continuación nuestras conclusiones en el estudio de soluciones propuestas.

2.3.1. Complejidad

Analizamos la complejidad temporal de las 3 soluciones, utilizando los variables como:

1. **k**: cantidad de paquetes
2. **x**: cantidad de tipos de producto
3. **w**: cantidad de unidades por producto

Algorithm 6 Complejidad algoritmo Greedy

```
1: Agrupar los paquetes por cada tipo de producto                    ▷ O(k)
2: Ordenar los paquetes por unidades descendente                     ▷ O(k log(k))
3: for all soborno do
4:     sobornos = []
5:     sobornado = 0
6:     for all paquetes do                                           ▷ O(x*k)
7:         if sobornado <= soborno.qty then
8:             sobornos + paquete
9:             sobornado + paquete.qty
10:        end if
11:        if soborno.qty <= sobornado then
12:            break
13:        end if
14:    end for
15: end for
```

Algorithm 7 Complejidad Greedy Alternativo

```
1: Agrupar los paquetes por cada tipo de producto                    ▷ O(k)
2: Ordenar los paquetes por unidades descendente                     ▷ O(k log(k))
3: for all soborno do
4:     sobornos = []
5:     sobornado = soborno.qty
6:     total_unidades = SUM(paquete.qty)                             ▷ O(x*k)
7:     for all paquetes do                                           ▷ (O(x*k)
8:         total_unidades = total_unidades - paquete.qty
9:         if total_unidades = 0 then
10:            if sobornado > 0 then
11:                sobornos + paquete
12:                sobornado -= paquete.qty
13:            else
14:                return
15:            end if
16:        else
17:            div_resto = sobornado / total_unidades
18:            if div_resto > 1 then
19:                sobornos + paquete
20:                sobornado -= paquete.qty
21:            else
22:                continue
23:            end if
24:        end if
25:    end for
26: end for
```

Para ambos algoritmos greedy la complejidad es de O(k*x)

Algorithm 8 Complejidad Programación Dinámica

```
1: Agrupar los paquetes por cada tipo de producto                                   ▷ O(k)
2: Ordenar los paquetes por unidades descendente                                    ▷ O(k log(k))
3: for all soborno do                                                               ▷ O(x)
4:     sobornos = []
5:     Calcular la tabla de pagos                                                   ▷ O(k*w)
6:     n=cant productos + 1
7:     busco el best_match como el índice que tiene el MAX entre de todas las celdas <= 0   ▷ O(w)
8:     sobornos = recurr-pay-bribe(tabla, n, best_match, paquetes)                  ▷ O(k)
9: end for
```

La complejidad del algoritmo por programación dinámica resulta entonces en el peor caso es de O(k*w*x) que es principalmente debido a la tabla que usa la solución.

2.3.2. Análisis de la Optimalidad

En nuestras pruebas encontramos que el algoritmo de programación dinámica alcanza siempre el óptimo global, sin embargo esto lo alcanza con el costo adicional de espacio (necesita una matriz calculada previamente) y el costo de tiempo, la generación de la matriz.

Por el contrario las soluciones Greedy pueden alcanzar el óptimo global pero hay escenarios en los cuales alcanzan la solución pero no es la mejor.

La primer solución del algoritmo Greedy toma la decisión en cada paso, evaluando si el monto total del soborno se alcanzo, independientemente de si incluir el paquete va a generar un excedente de unidades pérdidas, es por esto que esta solución puede no llegar al óptimo global cuando se llega casi al monto deseado y con el siguiente paquete se excede del monto a cubrir y como no verifica si hay otra ópcion con menos pérdida no llega a ser solución óptima.
Por ejemplo si se tiene paquetes con 10,8,5,2 unidades cada uno y un soborno de 12, entonces la solución con este algoritmo generará un soborno de 18, existiendo soluciones mejores (que tienen menos excedente) o inclusive el óptimo con el paquete de 10 y 2.

La solución alternativa, evita estos casos, dado que el algoritmo busca evitar utilizar los paquetes de unidades mas grandes, ya que en cada paso considera si se puede alcanzar o superar el monto del soborno sin incluir a la solución del paso actual, que siendo analizadas en orden descendente, de excluirse se estaría aprovechando ese paquete que tiene mas unidades que el resto. Sin embargo la debilidad de este algoritmo esta cuando el óptimo se puede alcanzar con exactitud con los paquetes descartados, generando una solución que no es la óptima.
Por ejemplo si se tuvieran paquetes con 10,8,5 y 2 unidades cada uno y un soborno de 12, la solución solo tendría los paquetes de 8 y 5 como solución, con un total de 13 unidades.

Proponemos como casos de prueba:

1. Caso 1: Paquetes: **10,8,5,2** — Soborno: **12**
2. Caso 2: Paquetes: **10,10,2,2,1,1** — Soborno: **4**
3. Caso 3: Paquetes: **10,9,4,1** — Soborno: **11**
4. Caso 4: Paquetes: **10,8,8,6,6,1** — Soborno: **18**
5. Caso 5: Paquetes: **17,10,4,4** — Soborno: **26**

En la siguiente tabla mostramos los resultados de los 3 algoritmos con los casos mencionados anteriormente:

	Greedy	Greedy Alt.	Prog. Dinámica	Óptimo
Caso 1	18 (10,8)	13 (8,5)	12 (10,2)	12
Caso 2	10 (10)	4 (2,1,1)	4 (2,2)	4
Caso 3	19 (10,9)	13 (9,4)	11 (1,10)	11
Caso 4	18 (10,8)	20 (8,6,6)	18 (10,8)	18
Caso 5	27(17,10)	27 (17,10)	27 (17,10)	27

Los casos de prueba se encuentran subidos en el repositorio. Estos pueden ejecutarse con el script que también se encuentra en el repositorio con el comando: *python optimality_test.py*

En caso de que se desee ejecutar los algoritmos con set randoms de tipos de productos y paquetes, se puede correr el script *main.py* que se encuentra dentro del repositorio, con el comando *python main.py*,pudiendose probar diferentes generaciones de set de datos, cambiando las constantes MAX_QTY (cantidad máxima de unidades por producto),MAX_PACKAGES (cantidad de paquetes por tipo de producto) y MAX_TYPES (tipos de producto sobre el que correran las soluciones). Estas constantes se pueden modificar al principio del archivo *aux.py* del repositorio. La ejecución del main genera set de datos nuevos en cada corrida dependiendo de las constantes, mostrando como esta conformado el dataset (paquetes con sus cantidades y los sobornos), el total de unidades usado por cada algoritmo, que paquetes uso cada uno y el tiempo de ejecución.

2.3.3. Pruebas de Stress

A continuación se mostrarán los resultados de las ejecuciones de las diferentes soluciones, en nuestras pruebas variando lo que consideramos 2 parámetros que pueden impactar en la performance:

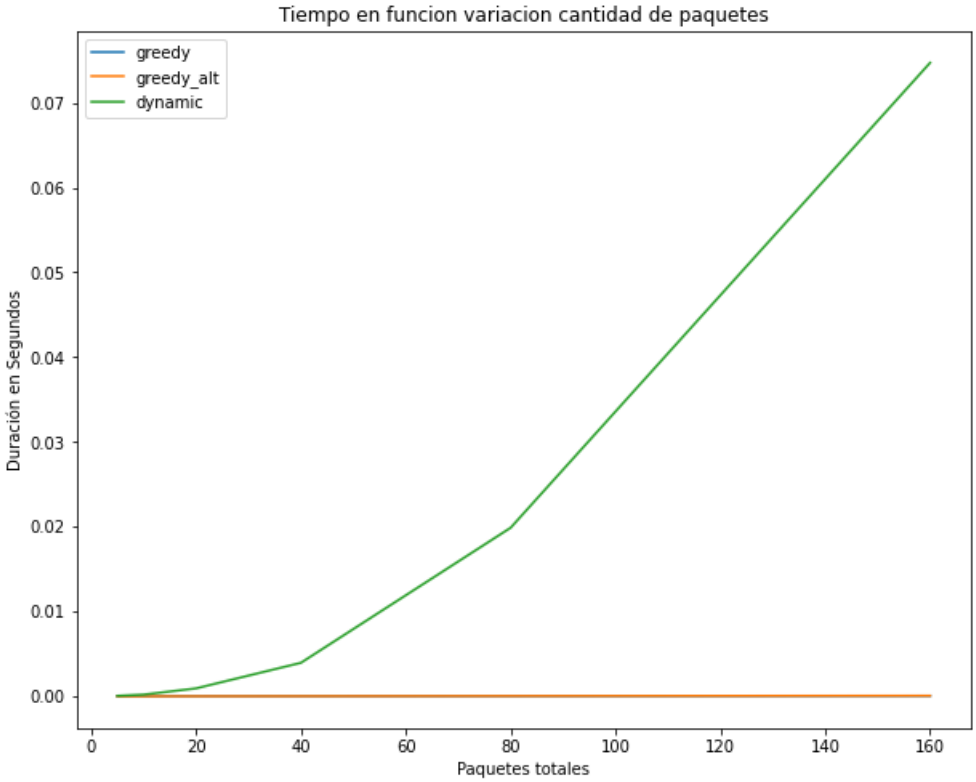
1. **MAX_QTY**: cantidad máxima de unidades en total que pueden tener cada paquete.
2. **MAX_PACKAGES**: cantidad máxima de paquetes que pueden tener un determinado tipo de producto

Para la realización de las pruebas, se desarrollo una implementación que para cada corrida, se arma un conjunto de paquetes con cierta cantidad de unidades por paquete, los valores de la cantidad de unidades se calcula de manera random entre 1 y MAX_QTY. Teniendo los paquetes, se calcula para todos los paquetes del mismo producto un soborno, cuya cantidad (como debe poder cubrirse con las existencias) se calcula como un número random entre 1 y la sumatoria de todas las unidades de los paquetes del tipo de producto.

El código de las pruebas puede accederse desde el noetbook del repositorio. Se realizaron 12 corridas con los siguientes valores iniciales.

1. MAX_QTY: 10
2. MAX_PACKAGES: 5

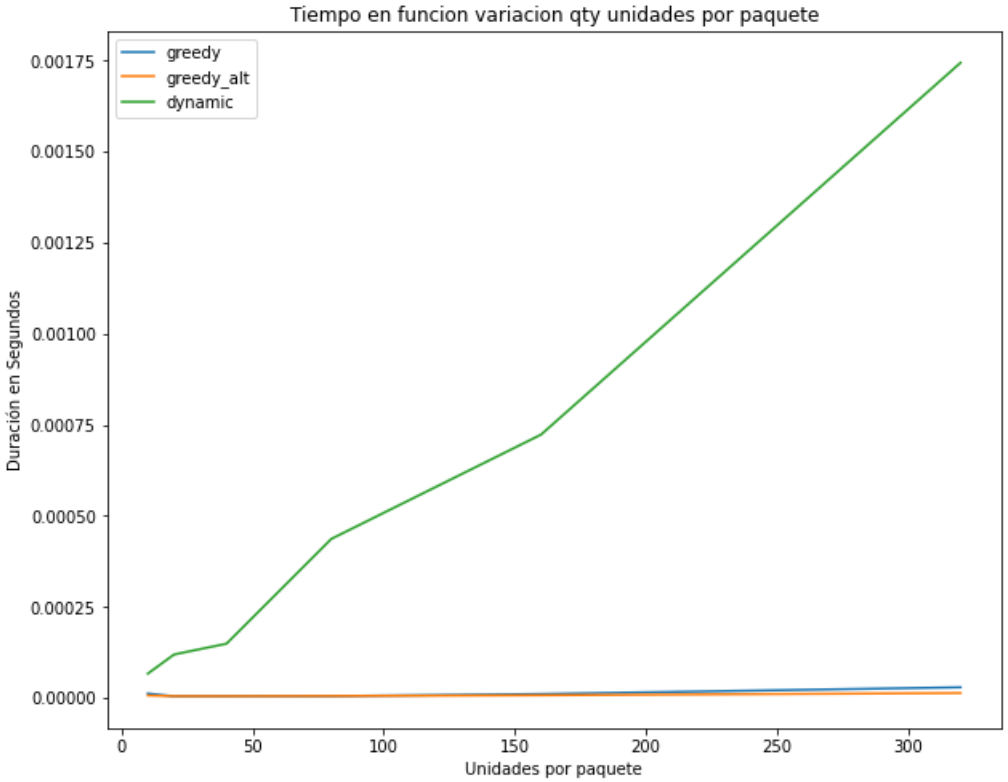
Las primeras 6 corridas se calcularon dejando la cantidad de unidades máxima por paquete fija y variando la cantidad de paquetes que se tendrían en existencia, la cantidad de tiposd distintos de productos se dejo fija en 1.



Con los resultados obtenidos por cada corrida y cada algoritmo

Qty packages	Qty per unit	Bribe	Greedy	Greedy Alt	Dynamic
5	10	11	17	11	11
5	20	42	42	42	42
5	40	100	100	100	100
5	80	168	170	168	168
5	160	265	268	265	265
5	320	157	160	157	157

Las proximas 6 corridas se dejaron fija la cantidad de paquetes de un tipo de producto y se variaron la cantidad máxima de unidades por paquete, la cantidad de tipos de producto se dejo fija en 1.



Qty packages	Qty per unit	Bribe	Greedy	Greedy Alt	Dynamic
5	10	4	10	4	4
10	10	9	18	12	12
20	10	56	64	57	57
40	10	61	64	76	64
80	10	324	331	331	331
160	10	491	607	497	497

Como se puede ver, el algoritmo de programación dinámica crece muy rápido a medida que aumentan los valores de los parámetros del problema. Esto se debe porque además de que recorre la tabla entera para encontrar el óptimo, tiene un tiempo de armado de la tabla que como esta resulta tener cantidad de paquetes igual a la cantidad de filas y el total de las columnas corresponde a la sumatoria total de todos las unidades de los paquetes, la tabla puede crecer muy rápidamente.

2.4. Correcciones

2.4.1. Parte 1

- Se corrigió un error de tipeo en la definición final de la complejidad del algoritmo propuesto
- Se agregó una demostración formal del desarrollo de la recurrencia, complementaria a la escrita preexistente.

2.4.2. Parte 2

Hacemos una corrección al algoritmo de programación dinámica, dado que aplicamos una operación de ordenamiento por cantidad de unidades por paquete, sin embargo la solución propuesta (la de utilizar

una tabla que calcula en función del valor calculado por el paso anterior y el valor actual) hace que sea irrelevante el ordenamiento, dado que el mismo proceso va a escoger los mejores productos.

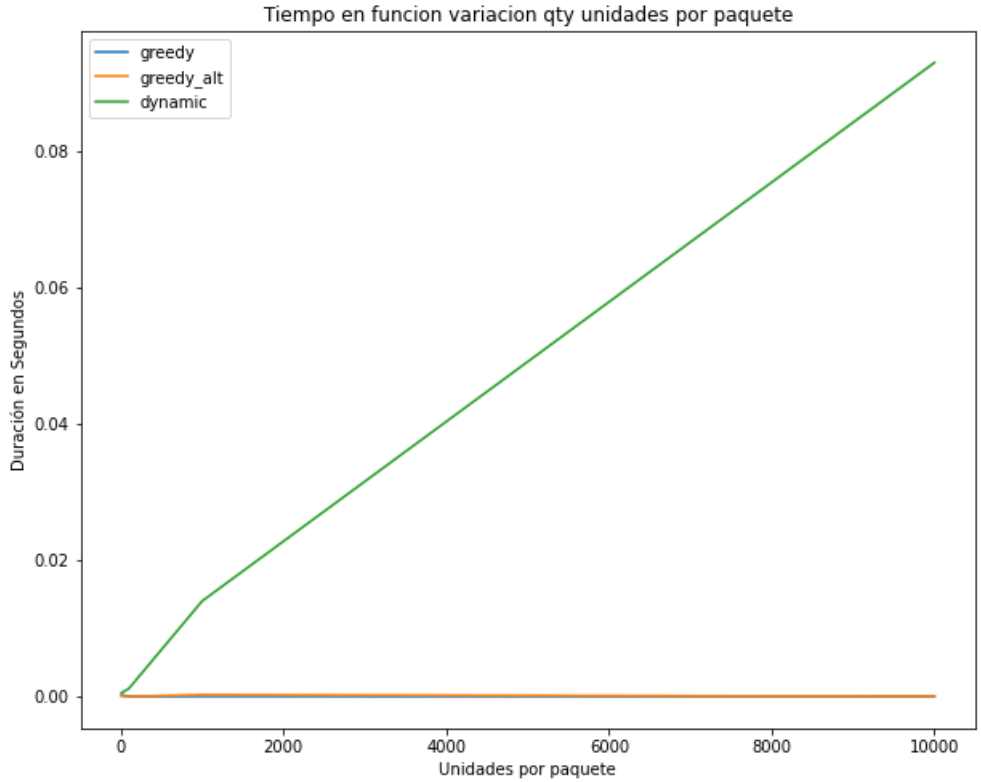
El cambio sobre el algoritmo no impacta en el calculo de su complejidad temporal , pues no es esta la operación que mas tiempo requiere para completar. La función *recur-pay-bribe* se mantiene como se propuso anteriormente, por lo que la complejidad temporal continúa siendo O(k*w*x).

Algorithm 9 Complejidad Programación Dinámica

```
1: Agrupar los paquetes por cada tipo de producto                                  ▷ O(k)
2: for all soborno do                                                              ▷ O(x)
3:     sobornos = []
4:     Calcular la tabla de pagos                                                  ▷ O(k*w)
5:     n=cant productos + 1
6:     busco el best_match como el índice que tiene el MAX entre de todas las celdas <= 0   ▷ O(w)
7:     sobornos = recurr-pay-bribe(tabla, n, best_match, paquetes)                 ▷ O(k)
8: end for
```

Respecto a las pruebas de cargas, repetimos el análisis pero esta vez llevando las variables a los límites que nos permitía correrlas. Variando en x10 (en vez de x2) en cada iteración.

Para las corridas variando la cantidad de unidades por paquetes:

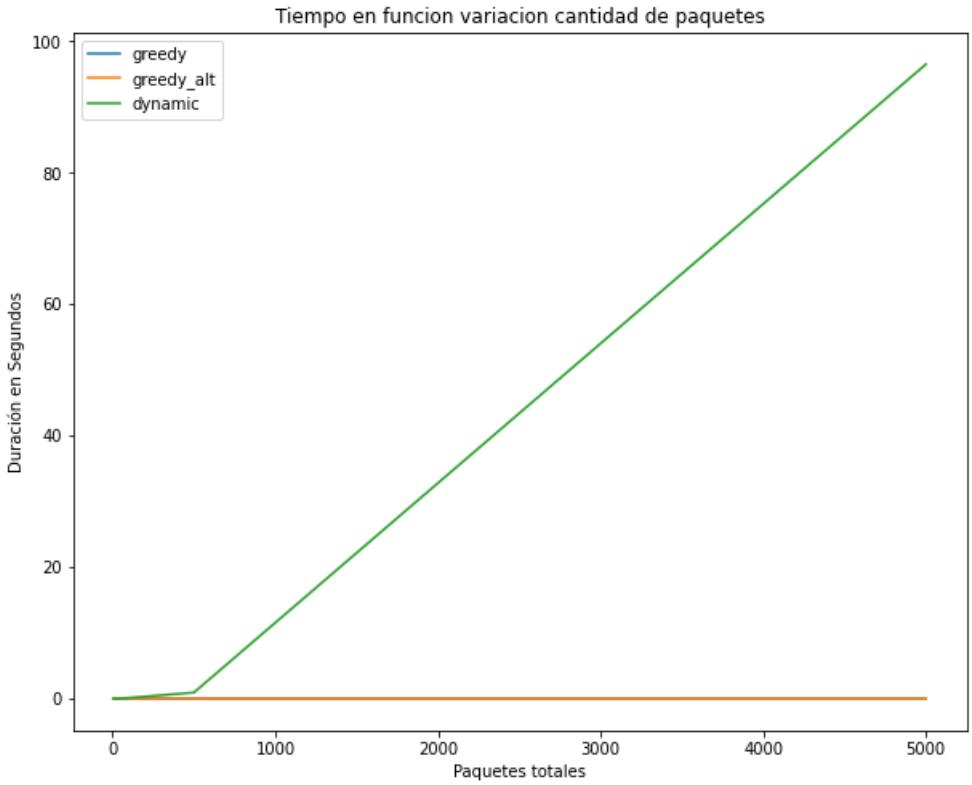


Podemos ver en la tabla siguiente, que en casi todas las ejecucciones los algoritmos greedy alcanzaron el óptimo, exceptuando la 3er corrida para el de greedy que presento un pago en exceso del 0.13 %, indicando una pérdida muy pequeña.

Qty packages	Qty per unit	Bribe	Greedy	Greedy Alt	Dynamic
5	10	28	28	28	28
5	100	238	238	238	238
5	1000	2294	2297	2294	2294
5	10000	10327	10327	10327	10327

Qty packages	Qty per unit	Bribe	Exceso Greedy	Exceso Greedy Alt
5	10	28	0 %	0 %
5	100	238	0 %	0 %
5	1000	2294	0.13 %	0 %
5	10000	10327	0 %	0 %

Para las ejecuciones variando la cantidad de paquetes:



Qty packages	Qty per unit	Bribe	Greedy	Greedy Alt	Dynamic
5	10	19	22	19	19
50	10	110	144	131	114
500	10	1831	2413	2050	1885
5000	10	10415	18337	11592	11592

Qty packages	Qty per unit	Bribe	Exceso Greedy	Exceso Greedy Alt
5	10	19	15.7 %	0 %
50	10	110	26.31 %	14.91 %
500	10	1831	28.01 %	0.08 %
5000	10	10415	58.18 %	0 %

En este caso vemos que el error que se presenta a variar la cantidad de paquetes, introduce una diferencia en el cálculo del óptimo, aunque el algoritmo de greedy alternativo pareciera sera una buena aproximación para calcular el óptimo sin tener que pasar por el alto tiempo de ejecución y consumo de recursos.

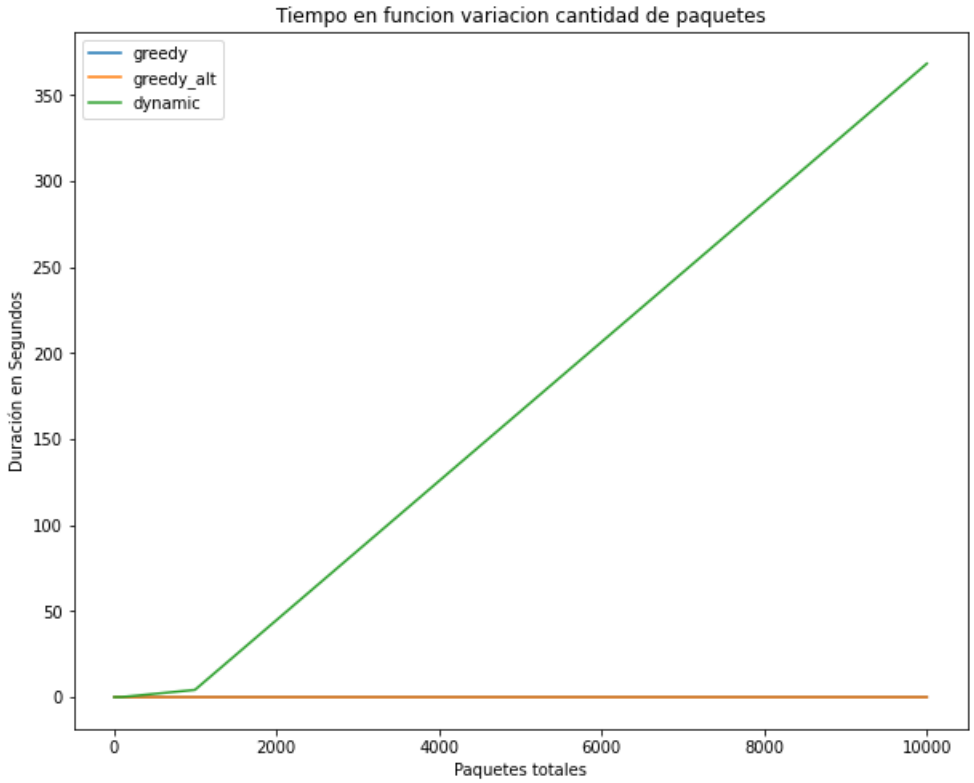
Para el caso del aumento de paquetes, no se pudo llevar a mas de 160 paquetes sin aumentar la cantidad de máxima de llamados recursivos que tiene python de manera default, aún así la cantidad de recursos que consumía (dado que consume memoria para los llamados recursivos) no pudimos superar las 50000 llamados recursivos sin llegar al tope de consumo de 32Gbs de memoria RAM.

Para poder incrementar el valor de qty de paquetes implementamos una versión iterativa para evitar el consumo de recursos de los llamados recursivos, esta implementación se encuentra en el repositorio, con el nombre *dynamic_programming_iterative*, también se incluyo su llamado en el *main.py*.

Algorithm 10 Complejidad Programación Dinámica Iterativo

```
1: Agrupar los paquetes por cada tipo de producto                                  ▷ O(k)
2: for all soborno do                                                              ▷ O(x)
3:     sobornos = []
4:     Calcular la tabla de pagos                                                  ▷ O(k*w)
5:     busco el best_match como el índice que tiene el MAX entre de todas las celdas <= 0   ▷ O(w)
6:     for all paquetes do                                                         ▷ O(k)
7:         if ultimo paquete then
8:             break
9:         end if
10:        if tabla[i][j] == tabla[i-1][j] then
11:            continue
12:        else
13:            sobornos + producto
14:            best_match -= producto.qty
15:        end if
16:    end for
17: end for
```

Esta alternativa nos permitió duplicar la cantidad de paquetes que habíamos podido lograr en la corrida con la versión recursiva.



Vemos que duplicar la cantidad de paquetes (la cantidad de filas de la matriz) hace que el tiempo sea mas del triple.

Qty packages	Qty per unit	Bribe	Greedy	Greedy Alt	Dynamic
10	10	51	52	52	52
100	10	8	97	14	14
1000	10	4268	4589	4293	4278
10000	10	26024	29921	26486	26043

Qty packages	Qty per unit	Bribe	Exceso Greedy	Exceso Greedy Alt
10	10	51	0 %	0 %
100	10	8	692.8 %	0 %
1000	10	4268	7.26 %	0.35 %
10000	10	26024	14.8 %	1.7 %

Podemos volver a ver que para la variación de paquetes, el algoritmo greedy alternativo, seria una buena opción si necesitamos obtener el óptimo pero no disponemos de los recursos o el tiempo.