Trabajo Práctico 1: Técnicas de Diseño

Teoría de Algorítmos I

Cátedra Buchwald

Abril, 2023

Integrantes

Nombre	**Padrón**
Bedoya, Gabriel	107602
Berenguel, Rafael	108225

Índice

1. Problema de K-merge por División y Conquista **3**

1.1. Complejidad del algoritmo propuesto . . . 3

1.2. Algoritmo que utiliza heaps . . . 4

1.3. Comparación de los algoritmos . . . 5

1.3.1. Resultados k-merge . . . 5

1.3.2. Resultados algoritmo con heap . . . 6

1.4. Comparación gráficos y complejidad . . . 7

1.5. Conclusiones finales . . . 9

2. Problema de contrabando **10**

2.1. Implementación del algoritmo greedy . . . 11

2.2. Implementación del algoritmo con programación dinámica . . . 12

2.3. Complejidad de los algoritmos y deficiencias . . . 13

2.3.1. Complejidad del algoritmo Greedy . . . 13

2.3.2. Complejidad del algoritmo que utiliza programación dinámica . . . 13

2.3.3. Casos de deficiencia en el algoritmo greedy . . . 14

2.4. Comparación de los algoritmos . . . 15

2.4.1. Pruebas con datos escritos a mano . . . 15

2.4.2. Prueba de volumen . . . 16

1. Problema de K-merge por División y Conquista

El problema de K-merge es el siguiente: se tienen K arreglos ordenados, y se quiere quiere obtener un único arreglo, también ordenado, con todos los elementos de los arreglos originales (inclusive si hay repetidos). Por simplicidad para los diferentes análisis se puede suponer que todos los arreglos tienen exactamente h elementos (por ende, la cantidad total de elementos es $n = K * h$).

Para resolver este problema, es posible que hayan visto en Algoritmos y Programación II un algoritmo que resuelve este problema utilizando un Heap. Nos referiremos a este como el algoritmo que utiliza Heaps.

La idea en este caso será plantear otra solución y analizarla. Se propone el siguiente algoritmo por división y conquista, con semejanzas a mergesort.

1. Caso base: cuando quede un único arreglo, simplemente devolver dicho arreglo.
2. 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. Al terminar los llamados recursivos, tenemos dos arreglos ordenados. Estos deberán ser intercalados ordenadamente, tal cual se realiza en mergesort.

   Consigna:

   1. Determinar, utilizando el Teorema Maestro, cuál sería la complejidad del algoritmo propuesto.
   2. Describir el algoritmo que utiliza heaps, y determinar su complejidad.
   3. Implementar ambos algoritmos, y hacer mediciones (y gráficos) que permitan entender si las complejidades obtenidas para cada uno se condicen con la realidad.
   4. En caso que la complejidad obtenida en el punto 1 no se condiga con la realidad, indicar por qué (qué condición falla). En dicho caso, se requiere llegar a la complejidad correcta (no solamente enunciarla, sino demostrar cuál es).
   5. Indicar cualquier conclusión adicional que les parezca relevante en base a lo analizado.

1.1. Complejidad del algoritmo propuesto

Vamos a calcular la complejidad de nuestro algoritmo utilizando el teorema maestro, el cual es de la siguiente forma:

$$T(n) = aT\left(\frac{n}{b}\right) + n^c \quad donde\ a \geq 1, b > 1$$

siendo:

- n es el tamaño del problema a resolver.
- a es el número de subproblemas en la recursión: es la cantidad de llamadas recursivas que realiza el algoritmo.
- b es en cuánto se divide el problema en cada llamada recursiva.
- $\frac{n}{b}$ el tamaño de cada subproblema. (Aquí se asume que todos los subproblemas tienen el mismo tamaño)
- $f(n) = n^c$ es el costo del trabajo hecho fuera de las llamadas recursivas, que incluye el costo de la división del problema y el costo de unir las soluciones de los subproblemas. Es decir, es el costo en tiempo de lo que cuesta dividir y combinar.

La solución será

- Si $log_b(a) < c \Rightarrow T(n) = O(n^c)$
- Si $log_b(a) = c \Rightarrow T(n) = O(n^c log(n^c))$
- Si $log_b(a) < c \Rightarrow T(n) = O(n^{log_b(a)})$

Como el array final que queremos conseguir va a ser de tamaño $k*h$ entonces n va a ser de ese tamaño.

Como realizamos dos llamadas recursivas, **a** será igual a 2

Como el problema se divide en dos partes cada llamada, **b** es 2

En el caso de nuestro *K-merge*, el $f(n)$ será el caso más costoso de hacer lo que hacemos fuera de la llamada recursiva, el cual sería el último merge, pues mezclamos los dos arrays finales, que serán los más grandes. Dado que la complejidad de merge es la suma de las longitudes de los dos arrays ordenados y que en nuestro caso este terminará siendo de longitud $k * h$, $f(n)$ tiene complejidad $O(k * h)$

Con esto nuestra ecuación queda

$$T(n) = 2T(\frac{k * h}{2}) + O(k * h)$$

Como $log_b(a) = log_2(2) = 1$ entonces estamos en el caso en que la complejidad es

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

1.2. Algoritmo que utiliza heaps

El algoritmo que utiliza heaps consiste en tener un heap al que nos referiremos como **heap de mínimos** en el cual insertaremos el menor elemento de cada array.

Luego al hacerle pop nos dará el menor de todos esos elementos, el cual también será el menor elemento de todos los arrays. Después de hacer esto tendremos que reemplazar el elemento que sacamos por el segundo menor del array de donde lo sacamos y repetimos el proceso hasta que hayamos introducido y sacado todos los elementos del heap.

Este algoritmo no es recursivo, por lo que para analizarlo no nos hará falta el teorema maestro.

Crear el heap de tamaño **k** (cantidad de arrays) nos tomará, dado que agregar elementos a un heap tiene complejidad *0(log(n))*, *O(k*log(k))*.

Luego haremos un ciclo *while* mientras el heap tenga elementos, y dado que nosotros querremos que todos lo elementos eventualmente pasen por el heap, esto tomará $k * h$ ciclos.

En cada uno de estos ciclos de while se va a extraer la raíz del heap y hacer otra inserción si es necesario, ambas acciones teniendo complejidad *O(log(n))* cada una, pero como en el heap habrá como máximo k elementos, entonces la complejidad será de *O(log(k))*. Con lo cual el ciclo terminaría con una complejidad *O(log(k))* y el while con una complejidad de *O(k*h*log(k))*

Con lo cual la complejidad de nuestro algoritmo es

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

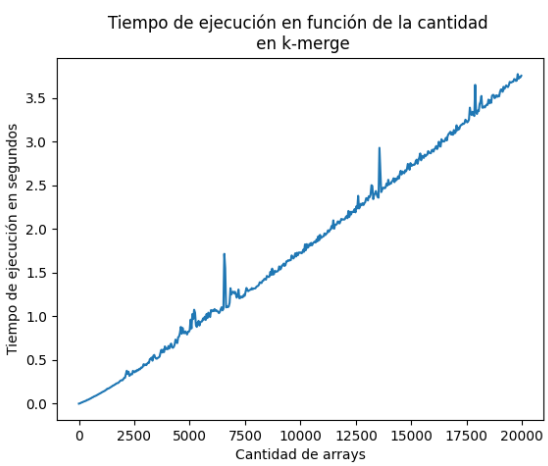
y nos quedamos con el más grande, que sería $O(k * h * log(k))$

1.3. Comparación de los algoritmos

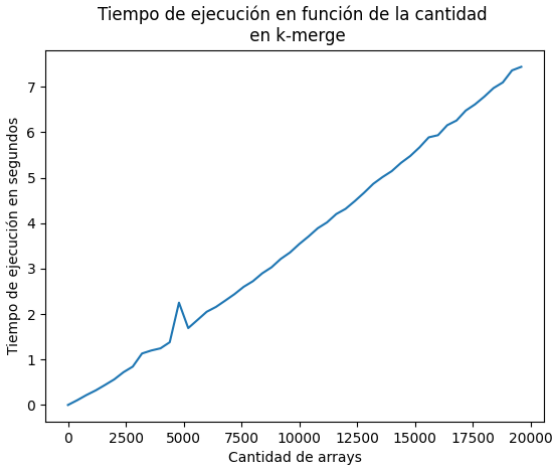
Tanto el algoritmo de *k-merge* que se nos pidió elaborar como el algoritmo de basado en heaps fueron ejecutados con cantidades distintas de arrays con longitudes distintas.

1.3.1. Resultados k-merge

En una primera instancia probamos el algoritmo k-merge repetidas veces, variando la cantidad de arrays que queríamos ordenar, pero dejando fija la longitud de estos, siendo estos lo resultados:



(a) Arrays de longitud 100

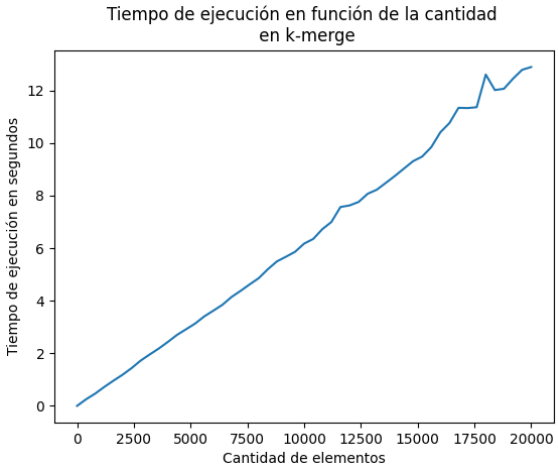


(b) Arrays de longitud 200

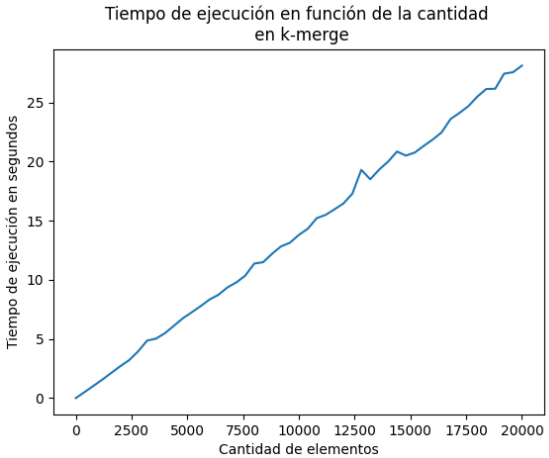
Figura 1: Tiempo de ejecución del algoritmo K-merge con diferente cantidad de arrays

Como podemos apreciar, a medida que aumenta la variable k, el tiempo que nos toma ejecutar nuestro algoritmo aumenta en proporción lineal.

Si probamos, por otro lado, aumentar la variable h obtenemos los siguientes resultados.



(a) 500 arrays



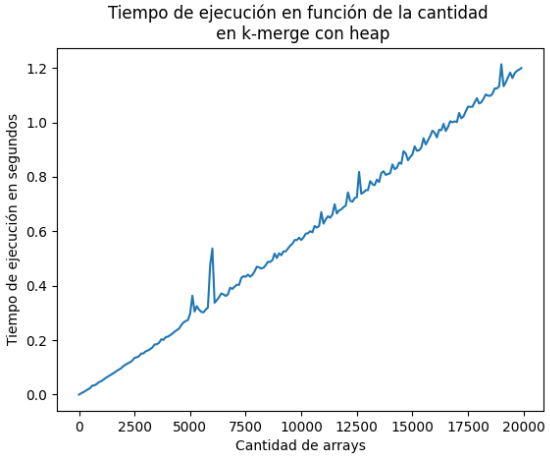
(b) 1000 arrays

Figura 2: Tiempo de ejecución del algoritmo K-merge con diferente cantidad elementos en los arrays

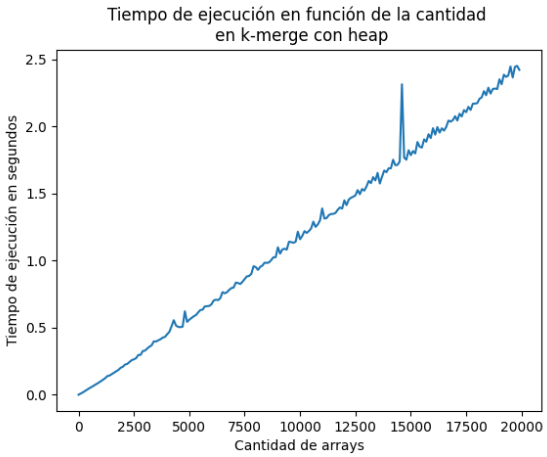
Como podemos apreciar en los gráficos, tanto al aumentar la variable k como al aumentar la variable h, el tiempo que el programa demora en ejecutarse es de carácter lineal.

1.3.2. Resultados algoritmo con heap

Al igual que con el algoritmo *k-merge*, probamos el algoritmo con heaps repetidas veces, variando la cantidad de arrays que queríamos ordenar, pero dejando fija la longitud de estos, siendo estos lo resultados:



(a) Arrays de longitud 100

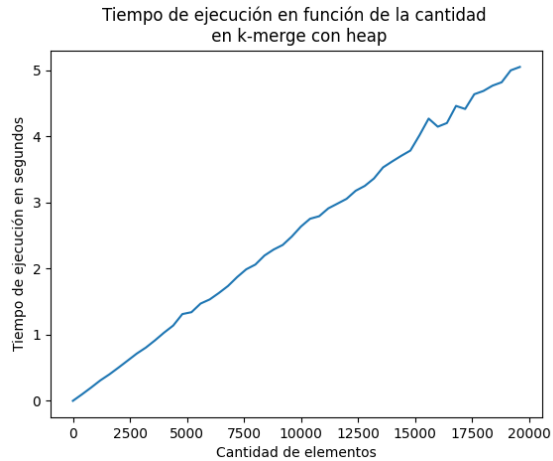


(b) Arrays de longitud 200

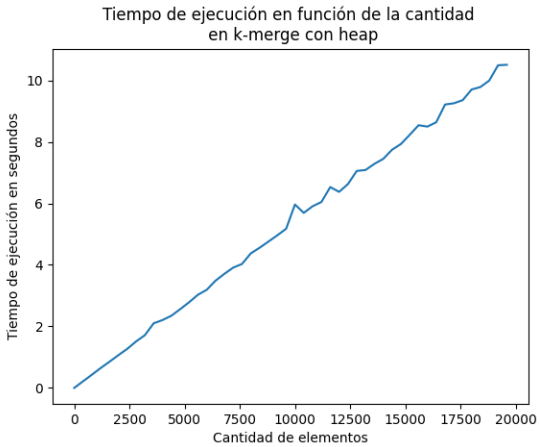
Figura 3: Tiempo de ejecución del algoritmo con heap con diferente cantidad de arrays

Como podemos apreciar, a medida que aumenta la variable k, el tiempo que nos toma ejecutar nuestro algoritmo aumenta en proporción lineal.

Si probamos, por otro lado, aumentar la variable h obtenemos los siguientes resultados.



(a) 500 arrays



(b) 1000 arrays

Figura 4: Tiempo de ejecución del algoritmo con heap con diferente cantidad elementos en los arrays

Como podemos apreciar en los gráficos, tanto al aumentar la variable k como al aumentar la variable h, el tiempo que el programa demora en ejecutarse es de carácter lineal.

1.4. Comparación gráficos y complejidad

Podemos ver que, a pesar de que la complejidad obtenida con el teorema maestro nos dió *O(k*h*log(k*h)*, en los gráficos este no pareciera ser el caso.

Si dejamos fijo h y aumentamos k, entonces esperamos que la complejidad sea lineal-logarítmica, y en este caso podemos apreciar cierto nivel de curvatura, con lo que no descartamos que pueda cumplirse esta complejidad, sin embargo aparenta ser más lineal.

En el caso de dejar fijo k y aumentar h es más evidente ver que esta complejidad parece lineal.

Revisamos las condiciones del teorema maestro para constatar estar cumpliéndolas, siendo las más importantes:

1. **a** tiene que ser natural.
2. **b** tiene que ser real mayor a 1 y constante.
3. El caso base es constante.

En una primera instancia consideramos que se cumplían todas, pues **a** y **b** cumplen sus condiciones y en el caso base siempre regresamos un array. Sin embargo, al replantearnos lo que implicaba que el caso base fuera constante, nos percatamos de que, a pesar de estar devolviendo siempre un solo array, la longitud de este varía dependiendo de **h**, lo cual hace que no sea constante.

Dado esta observación, vamos a proceder a calcular la complejidad de manera manual, pues el teorema maestro no nos será de utilidad en este caso.

Vamos a calcular la notación **big O**, la cual nos da la cota superior de complejidad, tomando en cada caso el peor caso.

La complejidad de nuestro algoritmo, el cual recibe los parámetros k y h, la vamos a denotar **T(k,h)**. Primero calculamos la complejidad del algoritmo sin calcular las de las llamadas recursivas, el cual sería:

$$T(k,h) = 2T\left(\frac{k}{2}, h\right) + O(kh)$$

Analicemos esta ecuación de complejidad. Nuestro algoritmo, cuando no está en el caso base, se llama a si mismo con la mitad de los arrays, con lo cual la complejidad de esa llamada es $T\left(\frac{k}{2}, h\right)$. Como hacemos dos llamadas recursivas entonces la multiplicamos por dos. Luego le sumamos la complejidad de hacer merge con los dos arrays que tenemos. La complejidad de hacer merge a los dos arrays va a ser *O(hk)* debido a que el merge de los arrays tiene una complejidad que es la suma de sus longitudes, que como nos tiene que dar el array final, que mide $k * h$, entonces esta será la complejidad.

Para facilitarnos analizar los cassos subsecuentes, vamos a hacer un cambio de variables, en donde $k = 2^x$, con lo que $x = log(k)$. Con esto nuestra ecuación nos queda:

$$T(2^x, h) = 2T\left(\frac{2^x}{2}, h\right) + O(2^x h)$$

$$T(2^x, h) = 2T(2^{x-1}, h) + O(2^x h)$$

Ahora vamos a calcular las ecuaciones de las llamadas recursivas:

$$T(2^{x-1}, h) = 2T(2^{x-2}, h) + O(2^{x-1} h)$$

$$T(2^{x-2}, h) = 2T(2^{x-3}, h) + O(2^{x-2} h)$$

Si reemplazamos estas ecuaciones en la original podemos hacernos una idea del tipo de ecuación que finalmente tendremos que escribir

$$T(2^x, h) = 2(2(2T(2^{x-3}, h) + O(2^{x-2}h)) + O(2^{x-1}h)) + O(2^x h)$$

Y de esta manera vamos a seguir hasta no poder dividir a la cantidad de arrays k en dos partes enteras. Como $k = 2^x$, entonces x es la cantidad de veces que podemos dividir a k por dos, con lo cual la ecuación que nos dará toda la complejidad es:

$$T(2^x, h) = 2\sum_{i=0}^{x-1}(2^i O(2^{x-1-i}h)) + O(2^x h)$$

Podemos multiplicar 2^i por el interior de la complejidad

$$T(2^x, h) = 2\sum_{i=0}^{x-1}(O(2^{x-1}h)) + O(2^x h)$$

También podemos meter el dos multiplicando a la sumatoria y hacer lo mismo

$$T(2^x, h) = \sum_{i=0}^{x-1} 2(O(2^{x-1}h)) + O(2^x h)$$

$$T(2^x, h) = \sum_{i=0}^{x-1}(O(2^x h)) + O(2^x h)$$

Como el interior de la sumatoria y el último término son el mismo, lo podemos agregar a esta

$$T(2^x, h) = \sum_{i=0}^{x}(O(2^x h)))$$

Como dentro de la sumatoria ya no tenemos al parámetro i, podemos directamente multiplicar por x

$$T(2^x, h) = xO(2^x h)$$

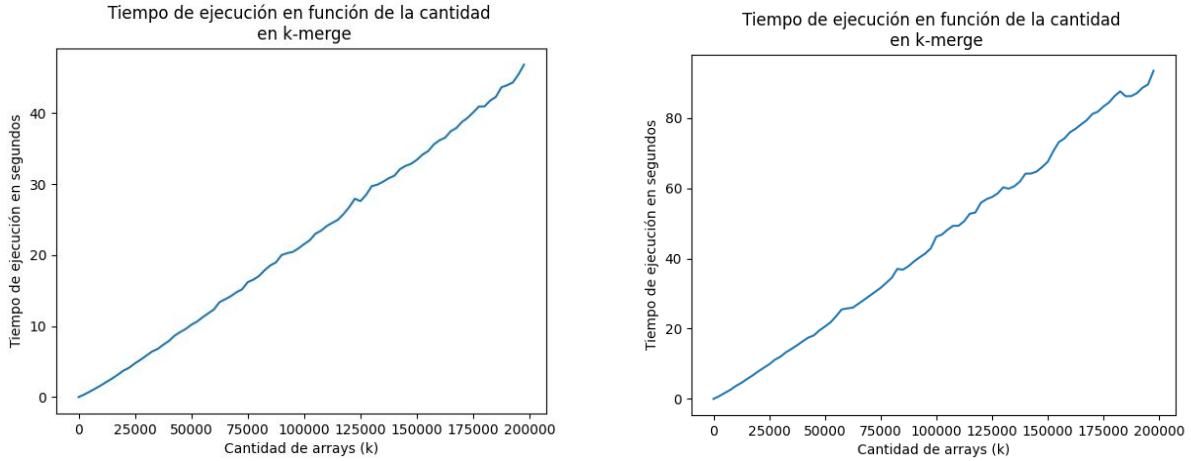
$$T(2^x, h) = O(x2^x h)$$

Recordando que $2^x = k$ y que $x = log(k)$ la complejidad nos queda

$$T(k, h) = O(log(k)kh)$$

Según esta complejidad, si nosotros dejamos fijo el parámetro k, entonces el tiempo de ejecución de nuestro algoritmo al variar el parámetro h va a aumentar de manera lineal, lo cual corresponde con lo visto en los gráficos en la figura 2.

Si, por otro lado, dejamos sin variar el parámetro h y variamos el k, entonces este tendría complejidad lineal-logarítmica, lo cual es un poco difícil de ver claramente con los gráficos, sin embargo se puede notar cierta curvatura, en especial si aumentamos el número de arrays mucho más.



(a) 100 elementos

(b) 200 elementos

Figura 5: Tiempo de ejecución del algoritmo K-merge con cantidades grandes de arrays

1.5. Conclusiones finales

Después de hacer este ejercicio, pudimos concluir una cuantas cosas.

La primera es que siempre se deben verificar las condiciones iniciales para la aplicación del teorema maestro antes de su uso, las cuales en el problema de k-merge se cumplieron la mayoría exceptuando el caso base constante y debido a esto, se tuvo que calcular de forma manual.

Por otro lado, se puede concluir que a pesar de que ambos algoritmos tengan una complejidad O(log(k)kh), no quiere decir que el tiempo de ejecución de ambos sea igual, esto se puede ver en los gráficos de la distribución de tiempos de ejecución de ambos, en el caso de 1000 arrays k-merge recursivo tardó más de 25 segundos, por el contrario el tiempo estimado k-merge con heaps fue de 10 segundos.

Gran parte de este problema radica en la implementación recursiva de k-merge, la cual al aumentar la cantidad de arrays provoca problemas de memoria debido a la gran cantidad de variables y sobrecargas de llamadas recursivas, logrando que el tiempo de ejecución aumente considerablemente. Este problema no se observa en k-merge en la versión de heaps, al ser completamente iterativo.

2. Problema de contrabando

Queremos pasar mercadería de contrabando de Genovia a Krakozhia. La mercadería viene en paquetes que no podemos abrir. Cada paquete i trae X_i unidades de un determinado tipo de producto j. Podríamos llegar a tener varios paquetes del mismo tipo de producto j, incluso con diferente cantidad de unidades. También podemos tener diferentes paquetes de diferentes productos. Es decir, cada paquete (in-abrible) es de una cantidad específica de un tipo específico, y en total para un tipo específico j tenemos la suma de X_i unidades, para todos los i que sean de ese tipo.

Para nuestro ejemplo, supongamos que tenemos un paquete que trae 8 cajetillas de cigarrillos sabor arándano. Otro paquete trae 5 cajetillas de lo mismos cigarrillos. Otro paquete puede traer 5 botellitas de 100ml de vodka radioactivo, etc. . .

Al pasar por la aduana, el corrupto funcionario puede indicarnos que "por acá no pasan sin dejarme al menos 6 cajetillas de cigarrillos de arándano". Ante la imposibilidad de abrir y/o separar los paquetes, es claro que en dicho caso nos conviene dejar el paquete de 8 (no podemos abrirlo para sacar 6 de allí. . . sino la movida sería muy evidente). Si el oficial hubiera dicho que hay que dejar al menos 10 cajetillas, habría sido necesario dejar ambos paquetes para un total de 13 unidades de dicho producto. Si este hubiera dicho que le dejemos una cajetilla de cigarrillos y una botellita de vodka, tendríamos que dejar el paquete de 5 botellitas de vodka y el paquete de 5 cajetillas de cigarrillos.

Consigna:

1. Describir e implementar un algoritmo greedy que, dado un input con los productos que se tienen, y lo pedido como soborno, nos permita salir airosos de la situación, con la mayor cantidad de productos posibles. Justificar por qué el algoritmo es, efectivamente, greedy. Considerar que siempre se nos pedirá una cantidad de productos en existencias (en nuestro ejemplo anterior, no nos habrían pedido que dejemos 7 botellas de vodka radioactivo, ni tampoco mandarinas del Sahara).
2. Con las mismas consideraciones que en el punto anterior, describir e implementar un algoritmo (que sea óptimo) que resuelva el problema utilizando programación dinámica.
3. Indicar y 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.
4. Implementar un programa que utilice ambos algoritmos, realizar mediciones y presentar resultados comparativos de ambas soluciones, en lo que refiere a su optimalidad de la solución (no de su complejidad). Incluir en la entrega del tp los sets de datos utilizados para estas simulaciones (que deben estar explicados en el informe). Estos deben incluir al menos una prueba de volumen, indicando cómo es que fueron generadas.

2.1. Implementación del algoritmo greedy

Los algoritmos greedy se caracterizan por intentar hallar la mejor solución posible fijándose únicamente en el óptimo local mientras recorren los elementos para decidir qué hacer.

En nuestro problema representamos los paquetes y la cantidad que contienen con diccionarios que tienen como llave el nombre del producto y como valor la cantidad en el paquete.

Nuestra función recibe una lista de diccionarios llamada *packets* en donde tendremos los paquetes disponibles y otra lista llamada *requested*, que tendrá diccionarios con el producto y la cantidad que se nos pide.

Para el problema que se nos planteó, decidimos hacer lo siguiente:

1. Primero ordenamos los paquetes por cantidad de productos de manera ascendente.
2. Creamos una lista vacía llamada *extracted* en donde guardaremos los paquetes que serán retenidos.
3. Iteramos todos los paquetes que tenemos disponibles.
4. Por cada paquete primero intentamos encontrar la cantidad que nos piden del producto en el paquete buscándola en la lista *requested*.
5. Nos fijamos en la cantidad que nos piden y, si esta es mayor a cero, entonces agregamos el paquete a la lista *extracted* y le restamos a la cantidad pedida del producto la cantidad de productos que había en el paquete.
6. Seguimos iterando y como vamos disminuyendo la cantidad que nos piden de cada producto eventualmente esta llegará a cero para todos los elementos cuando los paquetes en *extracted* sean suficientes para darle al guardia.

La razón por la que ordenamos ascendentemente y no de manera descendente es porque si tenemos que elegir un lado para tener menos pérdidas es preferible empezar por lo que tengan menos elementos.

Este algoritmo es efectivamente greedy, pues se avanza sobre cada elemento solo una vez y se decide si lo tomaremos o no, fijándose en la situación actual de cuantos productos falta cubrir y tomando una decisión en base a ello para avanzar al siguiente elemento y hacer lo mismo.

A pesar de que el algoritmo resuelve nuestro problema, la solución no es óptima, pues, al avanzar una vez sobre el vector y tener que decidir si agregarlo o no con solo la información disponible en ese momento, se pierde muchas combinaciones que podrían habernos ahorrado productos.

2.2. Implementación del algoritmo con programación dinámica

A la hora de intentar resolver este problema por programación dinámica primero intentamos relacionarlo con algún problema ya conocido de los vistos en clase. Tras evaluarlos todos, el problema que más se le parecía era el de la mochila, por lo que lo usamos como base.

Al principio lo planteamos muy parecido al de la mochila, pero con los pesos y el valor siendo el mismo elemento, buscando cumplir con el mínimo proporcionado minimizando el peso de la mochila para cada tipo de producto que nos pedían, utilizando condiciones distintas para poner elementos en la matriz.

Sin embargo, aunque esto funcionó en algunos casos, dependiendo del orden de las cantidades contenidas en las cajas nos daba resultados no óptimos, con lo que tuvimos que desistir de esta idea.

Finalmente nos decantamos por, en vez de intentar seleccionar los elementos que darle al guardia como soborno para minimizar las pérdidas, intentar seleccionar las cajas que **no** le íbamos a dar.

A nuestros elementos disponibles los podemos pensar como dos grupos, cajas que le doy al guardia y cajas que me quedo. La suma de los elementos de las cajas que le doy al guardia tiene que ser como mínimo la cantidad que nos pide, por lo cual el otro grupo, de las cajas que nos quedamos, puede ser como máximo **la cantidad total de elementos menos la cantidad requerida de elementos**.

Hagamos un ejemplo. Si tengo 4 cajas, con 3, 5, 10 y 3 sodas mágicas y el guardia me pide 7, entonces la cantidad total de elementos que tengo es 21, por lo que lo máximo que me podría quedar si todo saliera bien es 14, con lo que podríamos verlo como un problema de la mochila pero en el que intento maximizar la cantidad de cajas que puedo meter en una mochila con capacidad 14 y eso me dará la solución óptima. Pues si, como en este problema, termino eligiendo las cajas 10 y 3, me termino quedando con solo 13 sodas, pero eso es porque es la máxima cantidad que me podía quedar, minimizando así las pérdidas.

Ahora vamos a describir el algoritmo utilizado:

1. Primero separamos nuestras cajas por el tipo de elemento que contienen.
2. Por cada tipo de elemento vamos a hacer el problema de la mochila tomando como la capacidad de la mochila la diferencia entre la cantidad total de elementos que tenemos disponibles y la cantidad de elementos que el guardia requiere.
3. Para resolver este problema de la mochila creamos una tabla en donde las filas son las diferentes cajas, cada una teniendo como peso/valor la cantidad de elementos que contiene, y las columnas son los números del 0 a la capacidad de la "mochila".
4. Vamos llenando la tabla de izquierda a derecha y de arriba a abajo, fijándonos la máxima cantidad de elementos que podemos poner con las diferentes capacidades y reutilizando las soluciones anteriores.
5. Con nuestra tabla completa la podemos recorrer para encontrar qué elementos fueron seleccionados para que nos los quedemos.
6. Sabiendo qué elementos nos vamos a quedar devolvemos los elementos que le vamos a dar al guardia.

Este algoritmo resuelve nuestro problema de manera óptima, pues el algoritmo con programación dinámica para resolver el problema de la mochila es óptimo, con lo cual sabemos que nos estamos quedando con la mayor cantidad posible de elementos.

2.3. Complejidad de los algoritmos y deficiencias

2.3.1. Complejidad del algoritmo Greedy

El algoritmo greedy que implementamos consiste en llamar a una función llamada **extract packets greedy**, la cual recibe un diccionario llamado *packets* que tiene como llave el nombre del producto y como valor una lista con las cantidades en las cajas que tenemos y un diccionario llamado *requests* que es como *packets* pero en los valores está la cantidad requerida de cada producto. La cantidad de cajas que tenemos será n y la cantidad de pedidos que tenemos será m.

Lo primero que hacemos en la función es entrar en un ciclo for que tendrá m iteraciones, en el que iteraremos sobre los diferentes productos que me están pidiendo.

Luego ordenamos los elementos que son de ese producto con el método *sort* de las listas de python. Como estamos calculando la complejidad con *notación big O* nos tenemos que poner en el peor caso para este sort, el cual es que todas las cajas que tengo sean de este producto, haciendo que la complejidad del sort sea $O(n.log(n))$, que es la complejidad del método sort de las listas de python.

Posteriormente llamamos a una función llamada **extract amounts**, que recibe las cajas del producto siendo iterado y la cantidad pedidas de este. Dentro de esta función se hace un ciclo while que tendrá como máximo tantas iteraciones como cajas de ese producto y, dado que las acciones realizadas adentro del while tienen complejidad $O(1)$, la complejidad de la función **extract amounts** es $O(n)$.

Con esto calculado, podemos concluir que la complejidad de la función **extract packets** es de $0(m.n.log(n))$, debido a que el sort de los elementos es más costoso que la función **extract amounts**.

2.3.2. Complejidad del algoritmo que utiliza programación dinámica

El algoritmo con programación dinámica que implementamos consiste en llamar a una función llamada **extract packets dynamic**, la cual recibe un diccionario llamado *packets* que tiene como llave el nombre del producto y como valor una lista con las cantidades en las cajas que tenemos y un diccionario llamado *requests* que es como *packets* pero en los valores está la cantidad requerida de cada producto. La cantidad de cajas que tenemos será n y la cantidad de pedidos que tenemos será m.

Lo primero que hacemos en la función es entrar en un ciclo for que tendrá m iteraciones, en el que iteraremos sobre los diferentes productos que me están pidiendo.

Dentro de ese ciclo for vamos a llamar a una función llamada **min packets**, la cual recibe las cajas del producto siendo iterado y la cantidad pedida de este.

Dentro de esta función se lleva a cabo el problema de la mochila. Para calcular la complejidad de esta función vamos a, otra vez ponernos en el peor escenario para el algoritmos, que, dado que nos dicen que cualquier cantidad que nos pidan la tendremos disponible, sería cuando el guardia nos pide todos los elementos que tenemos, pues eso hará que tengamos la matriz más grande posible.

Denotaremos a la cantidad de elementos que tenemos al sumar todas las cajas como p, lo que hará que nuestra matriz sea de dimensiones $n.p$.

Luego de crear la matriz, la vamos a recorrer, lo cual tiene complejidad $O(n.p)$.

Luego vamos a hacer un ciclo while para recuperar los índices de las cajas que vamos a conservar, lo cual tiene complejidad $O(n)$.

Luego pasamos los índices a elementos lo cual también tiene una complejidad $O(n)$ por el caso en el cual todos los índices estén en el vector.

Teniendo estas tres secciones en cuenta, la complejidad mayor de estas es $O(n.p)$, con lo cual esa será la complejidad de nuestra función **min packets**.

Entonces podemos ver que la complejidad de nuestra función **extract packets dynamic** es $O(m.n.p)$.

2.3.3. Casos de deficiencia en el algoritmo greedy

Nuestro algoritmo greedy, a pesar de darnos una solución que cumple con lo pedido, no es óptimo, sino que tiene casos en los cuales va a dar más de lo que debería.

Un caso en el que no es óptimo es cuando va recorriendo el vector y recolectando las cajas y, por ejemplo, le falta uno para cumplir con lo pedido, pero el siguiente elemento es muy grande y lo terminás añadiendo a las cajas que das, cuando si simplemente hubieras dado ese puede que hubieras cumplido con lo que te pedían.

Veamos un ejemplo

$$cajas = [1, 1, 3, 4, 10]$$
$$piden = 10$$

En este caso, vamos a ver el 1, tomarlo y nos falta llenar 9. Así avanzaremos por el 1, 3 y 4 y nos faltaría llenar 1, pero el siguiente elemento es el 10, entonces lo añado, y termino dando 19 elementos cuando podría directamente haber dado el 10 y quedarme con los otros 9,

También se puede dar que logremos dar la cantidad que nos pidieron pero que más adelante en el vector tengamos una caja más adecuada.

$$cajas = [1, 2, 3, 4, 12, 20]$$
$$piden = 20$$

En este caso vamos a darle al guardia las primeras 5 cajas, que contendran 22 elementos, cuando más adelante teníamo una caja con 20 exacto.

2.4. Comparación de los algoritmos

2.4.1. Pruebas con datos escritos a mano

Vamos a comparar nuestro algoritmos con distitos set de datos para comparar los resultados que nos dan y constatar que en los casos en los que el algoritmo greedy no nos da el óptimo, el algoritmo por programación dinámica sí lo hará.

Por simplicidad, dado que lo que queremos es comparar la optimalidad de las soluciones, vamos a usar datos con solo un tipo de producto.

Si tuviéramos los datos

$$packets = pringles : [7, 8, 1]$$
$$target = pringles : 5$$

Es claro ver que la solución óptima es 7, pues es una caja que cumple y con la que solo perderemos dos.

Al introducir estos datos en nuestro algoritmo greedy este nos devuelve **[7,1]**, debido a que empieza por el uno, ve que no le alcanza y va por el siguiente, con el que sí completa lo pedido, pero se llevó los dos.

Por otra parte, el algoritmo con programación dinámica nos devuelve **[7]**, pues este es capaz de calcular la solución óptima.

Si, por otra parte nuestro datos fueran

$$packets = pringles : [7, 8, 2, 3]$$
$$target = pringles : 5$$

Entonces los dos algoritmos nos darían la solución óptima, pues al avanzar de menor a mayor, al sumar el 2 y el 3, ya se alcanzaría el 5.

Si tuviéramos los datos:

$$packets = pringles : [2, 3, 1, 2]$$
$$target = pringles : 5$$

Entonces otra vez ambos nos darían la solución óptima, sin embargo el algoritmo greedy me daría **[1,2,2]** mientras que el de programación dinámica me da **[2,3]** (aunque esto puede variar dependiendo del orden de los elementos, lo que no varía es que el dinámico da siempre el óptimo).

Si tuviéramos los datos:

$$packets = pringles : [4, 2, 1]$$
$$target = pringles : 5$$

Entonces, mientras que mi algoritmo con programación dinámica me devolverá **[4,1]**, mi algoritmo greedy me devolverá todas las cajas, **[4,2,1]**, siendo otra vez, no óptimo.

Con estos ejemplos es fácil de ver que, tanto en los casos en donde el algoritmo greedy falla como en los que funciona, el algoritmo con programación dinámica es capaz de obtener la solución óptima.

2.4.2. Prueba de volumen

Ahora, vamos a hacer una prueba de volumen, para ver cuantos productos más entrega el algoritmo greedy que el algoritmo con programación dinámica.

Para generar los datos que vamos a utilizar creamos una función que recibe la lista de productos que queremos que estén presentes, la cantidad mínima que se puede requerir de un producto, la cantidad máxima que se puede requerir de un producto, la cantidad mínima que se puede poner en una caja, la cantidad máxima que se puede poner en una caja y una semilla para garantizar reproducibilidad de los resultados.

Para los nombres de los productos vamos a poner números en formato string, con el objetivo de poder tener muchos nombres rápido.

Si vamos variando la cantidad de productos que requerimos y creando sets de datos al azar, vemos que en todos los casos la diferencia entre lo que entrega el algoritmo con programación dinámica y lo que entrega el algoritmo greedy es positiva, con lo que es siempre más eficiente.

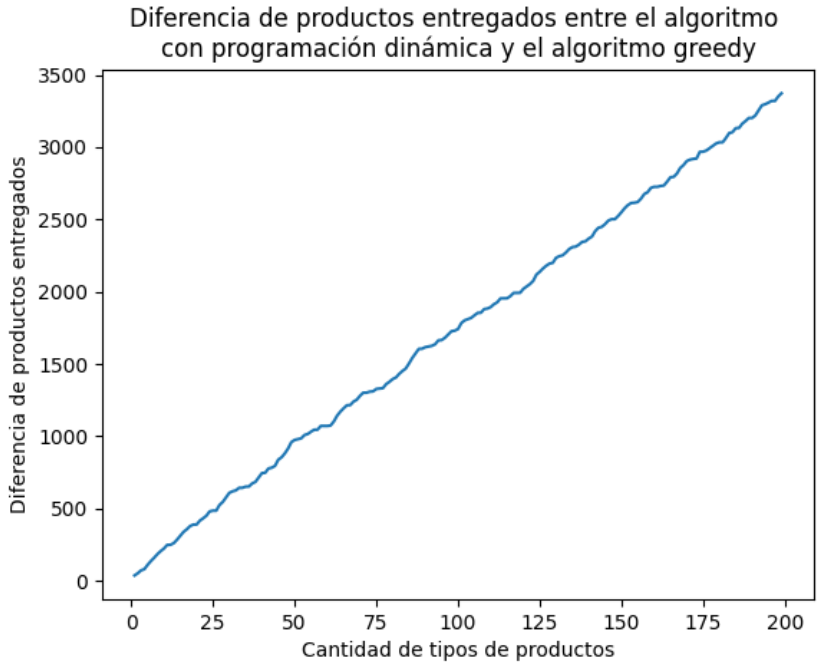


Figura 6: Diferencia de elementos entregados entre el algoritmos greedy y el algoritmo con programación dinámica

Sin embargo, si queremos ver más claramente que para cualquier mismo problema nuestro algoritmo con programación dinámica siempre nos va a dar una solución que cede una cantidad igual o menor a la del algoritmo greedy, podemos hacer un gráfico en el cual no variamos el set de datos, sino que tomamos en cuenta un producto y vamos aumentando, entonces vemos que cuando se agrega la diferencia del producto nuevo a la diferencia del producto anterior la función es creciente, pues nunca la diferencia que agreguemos va a ser negativa.

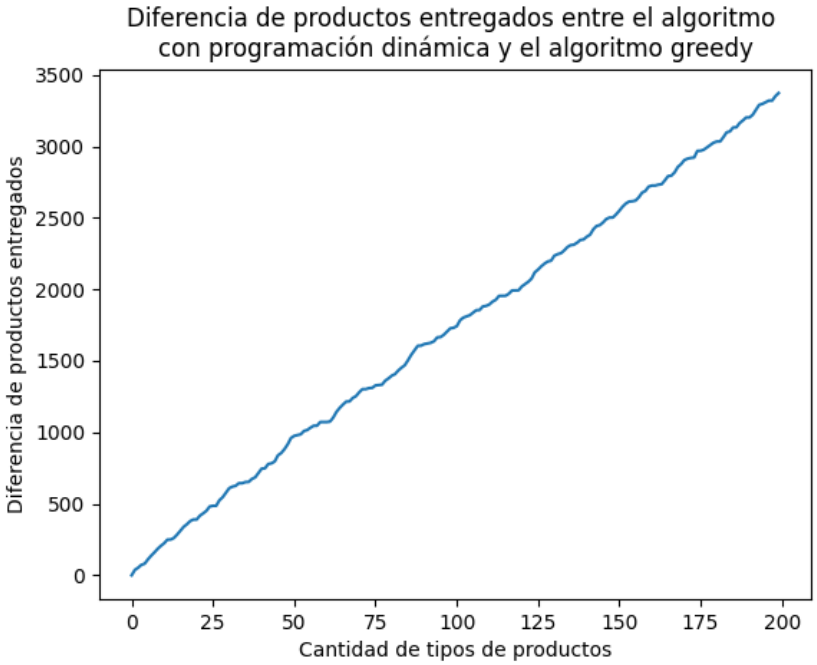


Figura 7: Diferencia de elementos entregados entre el algoritmos greedy y el algoritmo con programación dinámica en un mismo ser de datos

Con todo esto, se puede ver claramente que el algoritmo con programación dinámica alcanza el óptimo en las situaciones en las que el algoritmo greedy no lo hace.