# Practica 3 – Optimización apilamiento de contenedores

Marcos de Diego Martin, Marcos Almaraz Arranz

## INDICE

Análisis del problema

Resolución del problema

Analisis de Eficiencia

Conclusiones

## ANALISIS DEL PROBLEMA

En esta práctica abordamos el problema de apilar contenedores en un puerto comercial con el objetivo de maximizar el número de contenedores en una pila. Las restricciones del problema son las siguientes:

* Solo se puede colocar un contenedor directamente encima de otro.
* Un contenedor no puede soportar un peso mayor al de su capacidad máxima.
* Los contenedores deben respetar el orden ascendente de sus números de serie.

El problema implica manejar contenedores con dos propiedades: peso y resistencia. La solución óptima debe considerar:

* La resistencia de cada contenedor como límite para el peso total de los contenedores apilados sobre él.
* La necesidad de garantizar que los números de serie estén en orden ascendente.

Para solucionar este problema se utilizó un enfoque basado en programación dinámica, donde:

* Cada contenedor se considera como una posible base de pila.
* Se evalúan todas las combinaciones válidas respetando las restricciones.
* Se maximiza el número de contenedores en la pila más alta.

## IMPLEMENTACION

El código está implementado en Java con las siguientes clases principales:

* Contenedor: Representa un contenedor con atributos como id, peso y resistencia.
* Pila: Estructura que almacena una pila de contenedores, su peso total y una referencia a la pila anterior.
* ApiladorPRO: Clase principal que implementa el algoritmo y maneja la entrada y salida.

El núcleo del programa es el método encontrarMejorPila, que utiliza un arreglo de pilas para calcular la solución óptima. El proceso comienza con la lectura de los datos desde un archivo, los cuales se almacenan en una lista de objetos Contenedor, asumiendo que el formato de entrada es correcto.

El algoritmo implementa un enfoque basado en backtracking para buscar la pila óptima. En cada paso, evalúa si el contenedor en análisis puede soportar el peso de la pila formada hasta ese momento. Si es así, el contenedor se añade a la pila y pasa a ocupar su cima. Si no, el algoritmo retrocede para explorar otras posibilidades, verificando si es posible formar una pila más ligera con ese contenedor en una posición diferente.

Este proceso se repite de manera exhaustiva hasta que se evalúan todas las combinaciones posibles. Al finalizar, el programa determina la pila que cumple con las restricciones y maximiza el número de contenedores.

Finalmente, el programa imprime:

* El tiempo de ejecución.
* El tamaño máximo de la pila encontrada.
* Los identificadores de los contenedores que forman la pila óptima, ordenados.

## ANALISIS DE EFICIENCIA

**1. Complejidad Temporal**

La complejidad temporal se refiere a la cantidad de operaciones que realiza el algoritmo en función del tamaño de la entrada (en este caso, el número de contenedores n).

1. Bucle exterior: for (int i = n - 1; i >= 0; i--) { ... }

Este bucle recorre todos los contenedores en orden descendente. Como se ejecuta una vez por cada contenedor, tiene una complejidad de O(n).

1. Bucle interior: for (int j = mejorAltura; j >= 0; j--) { ... }

Este bucle recorre las alturas posibles de la pila. El número de iteraciones depende de la variable mejorAltura, que puede incrementarse durante la ejecución, pero en el peor caso puede llegar hasta n (cuando la pila crece al máximo). Por lo tanto, el número de iteraciones en el peor caso es O(n).

1. Operaciones dentro del bucle interior: Las operaciones dentro del bucle interior son principalmente comparaciones y asignaciones, las cuales son constantes O(1).
2. Reconstrucción de la pila: El bucle while (pilaActual != null && pilaActual.primero != null) recorre las pilas construidas de atrás hacia adelante. En el peor caso, recorrerá todos los contenedores, lo que da una complejidad de O(n).

Complejidad Temporal Total:

* El bucle exterior tiene complejidad O(n).
* El bucle interior tiene complejidad O(n) y como está anidado dentro del bucle exterior, la complejidad total de ambos bucles anidados es O(n^2).
* La reconstrucción de la pila tiene complejidad O(n), pero esta es menor que O(n^2), por lo que no afecta la complejidad total.

Resultado: La complejidad temporal total es O(n^2).

**2. Complejidad Espacial**

La complejidad espacial mide cuánta memoria necesita el algoritmo en función del tamaño de la entrada. Esto incluye las estructuras de datos que se usan durante la ejecución del algoritmo.

1. Arreglo de pilas (pilas): El arreglo pilas tiene un tamaño de n+1. Esto se debe a que el algoritmo podría apilar todos los contenedores en una pila de altura n. El tamaño de esta estructura es O(n).
2. Lista mejorPila: La lista mejorPila se llena con los contenedores que forman la mejor pila. En el peor caso, la lista tendrá n contenedores (si todos los contenedores se apilan). Por lo tanto, el tamaño de esta lista es también O(n).

Complejidad Espacial Total:

El espacio adicional requerido es principalmente para las pilas y la lista, ambas con tamaño O(n).

Resultado: La complejidad espacial total es O(n).

## CONCLUSIONES

Este programa se ha terminado por resolver mediante programación dinámica, desconocemos si se pudiera resolver de otra forma, por ejemplo, mediante el algoritmo de la mochila.   
De todas formas, el algoritmo ha sido probado y funciona adecuadamente. A pesar de dar soluciones diferentes a las devueltas en los ejemplos, entendemos que esto es porque existen varias soluciones validas y no porque nuestro programa falle o este mal hecho.  
Para conseguir resolver el problema hemos aprovechado todas las herramientas que se pueden imaginar, aun con ellas, ha sido una tarea ardua y que ha llevado mucho mas rato del que se podría esperar, muchos ajustes, correcciones y reinicios hasta llegar a la solución esperada.