Análisis Proyecto 3

Integrantes:
Ignacio Chaparro - 202220577
Juan Diego Osorio – 202220148

Universidad de Los Andes,
Diseño y Análisis de Algoritmos

Explicación de la Solución

Para solucionar este problema, partimos de que el objetivo del problema es ordenar un arreglo de números utilizando el número de flips mínimo. Un flip lo interpretamos como tomar los elementos desde la posición k hasta n y los invertimos, entonces, por ejemplo, si tenemos un arreglo [1,3,2,4,5] y hacemos un flip en k=2, entonces estamos agarrando el subarreglo [2,4,5] y lo invertimos, quedando el subarreglo [5,4,2], y el arreglo [1,3,5,4,2]. En primer lugar, cabe recalcar que el arreglo que entra por parámetro representa una pila de pancakes, donde sus posiciones arrancan desde 0 hasta n-1, siendo n el número de elementos (pancakes), de este modo, la posición 0 es la parte de debajo de la pila de pancakes por lo que es donde se deben encontrar los pancakes más grandes. Entonces, grosso modo, el objetivo es ordenar el arreglo en orden descendente, considerando que el número es proporcional al tamaño del pancake. Dicho esto, el enfoque que le dimos a la solución fue utilizar un grafo de permutaciones y por medio de búsqueda en anchura, es decir BFS, encontramos la secuencia de flips mínima.

Primero, definimos un objetivo, el cual siempre va a ser el arreglo entra por parámetro ordenado descendentemente, y además se define el punto de inicio el cual será simplemente el arreglo tal cual entra. Esto lo hacemos con el objetivo de que si el arreglo ya entra ordenado simplemente retorne directamente un arreglo vació, lo cual quiere decir que no se realizó ningún flip, y así nos ahorramos el correr todo el algoritmo.

Segundo, una vez revisado ese caso inicial, pasamos al algoritmo en sí, debido a que realizaremos BFS para hallar la secuencia de flips, entonces inicializamos la cola, la cual contiene el arreglo de inicio y una lista vacía donde registraremos los flips realizados. Del mismo modo, inicializamos un set ‘visited’ en el cual vamos a almacenar los arreglos que ya hayamos visitado y de esta forma evitamos los ciclos.

Tercero, en esta parte hacemos la exploración de las permutaciones, para ello hacemos un recorrido total por medio de un while siempre y cuando la cola no esté vacía. En cada iteración extraemos el arreglo actual y también la secuencia de flips de la cola, luego hacemos un llamado a la función que genera las permutaciones y le pasamos el arreglo actual, esta función lo que hace es que genera todas las permutaciones posibles del arreglo, aplicando un flip desde cada índice ‘k’ hasta el final del arreglo y cada permutación se guarda junto con el índice del flip aplicado. Una vez se finaliza este llamado entonces se itera sobre cada permutación del arreglo que este llamado retornó, y revisamos si la nueva permutación es igual a la meta, de ser así entonces devolvemos la secuencia de flips. Si la nueva permutación no ha sido visitada, la agregamos al set ‘visited’, y la agregamos a la cola junto con su secuencia de flips.

Adicionalmente, implementamos un algoritmo con un enfoque greedy del algoritmo Pancake Sorting. Con este algoritmo lo que hacemos principalmente es que, en cada iteración, identificamos el índice del elemento más grande del arreglo que no se encuentra en su posición correcta (teniendo en cuenta que debe ordenarse en orden descendente) y realizamos dos flips, el primer flip es para mover este elemento a lo más alto de la pila y el segundo es para moverlo a su posición final que le corresponde, este proceso se puede evidenciar en la Ilustración 1 Ejemplo de los dos 'flips'. Repetimos este proceso para todos los elementos, asegurando que en cada paso colocamos el siguiente elemento más grande en la posición correcta, acumulando la secuencia de flips necesarios en una lista que retornamos al final. Un ejemplo ilustrativo de este proceso completo se encuentra en Ilustración 2 Ejemplo del proceso completo de ordenamiento mediante 'flips '. Con un arreglo [1,4,3,2,5].

Ilustración 1 Ejemplo de los dos 'flips'.

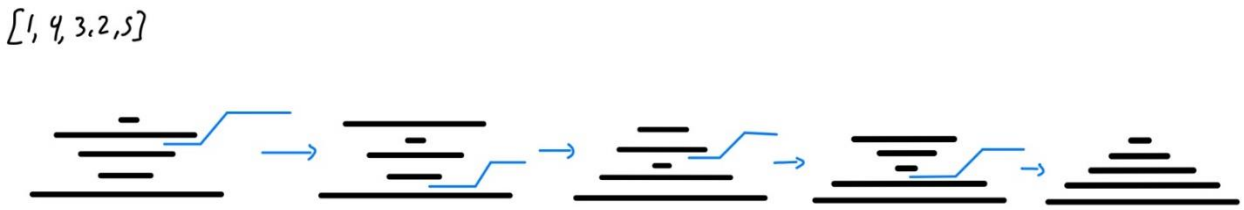

Ilustración 2 Ejemplo del proceso completo de ordenamiento mediante 'flips '.

Para solucionar este problema decidimos entonces implementar ambos algoritmos, por lo tanto, utilizamos el algoritmo del grafo de permutaciones para pilas de pancakes menores o iguales a 10 y para mayores a este número utilizamos el greedy, esto debido a que la implementación del grafo es muy compleja en tiempo y se demoraba mucho para tamaños de pila mayores a 10. Incluso llegamos a implementar una optimización potencial con una cola de prioridad, donde elaboramos una función de fitness, la cual nos indica que tan ordenado se encuentra la permutación, y así evaluar que tan buena era una permutación y la ingresábamos al heap junto con la permutación y la secuencia de flips, no obstante, esta solución nos dio algo más alejado que con el algoritmo greedy por lo que no valía la pena implementarla, no obstante, creemos que sí la función de fitness la mejoráramos de tal manera que nos indique que tan cerca está la permutación de la correcta, pues no siempre que tan ordenada está nos indica que se alcanzará la menor cantidad de flips y esto fue lo que evidenciamos al comparado con el greedy, así lograríamos optimizar más el algoritmo.

Análisis de Complejidad Temporal y Espacial

Complejidad Temporal

Por un lado, tenemos nuestra solución, realizando un recorrido BFS a lo largo del grafo de estados, estados representados por todas las posibles permutaciones del arreglo de pancakes inicial y por todos los posibles flips que cada una de estas permutaciones podaría tener. En este caso, la cantidad de vértices seria N*N!, ya que todas las posibles permutaciones serian N! Y cada una de estas tendría N posibles flips. Añadido a esto, se debe tener en cuenta la complejidad de la función generate_permutations(), la cual tiene una complejidad $O(N)$, esta función es llamada cada vez que se recorre un nuevo estado del grafo, por ende, esto sumaria una complejidad $O(N^2)$. Finalmente, la complejidad temporal de esta solución de búsqueda, teniendo en cuenta la generación de todas las permutaciones y su respectiva exploración seria $O(N^2 \cdot N!)$.

Por otro lado, tenemos la solución greedy, la cual se aplica para para los tamaños que la primera solución no es capaz de manejar. Esta solución, al tener que buscar el índice del mayor elemento del arreglo N veces y luego realizar los respectivos flips en los índices necesarios, primero para llevar el pancake más grande a la parte superior y luego invierte toda la pila de pancakes sin mover los que ya estén ordenados. Cada flip, por la función que utilizamos con python, tendrían complejidad O(NlogN). Estas tres operaciones, encontrar le máximo, voltear el pancake más grande y voltear toda la pila se ejecutan N veces. Por lo tanto, su complejidad temporal seria$O\big(N(N + 2N \log N)\big)$ lo que se convertiría en $O(N^2 + 2N^2 \log N)$.

Complejidad Espacial

Nuestra primera solución, la búsqueda con BFS, al tener que almacenar todos los posibles estados en la cola que utilizamos para el recorrido, va a tener que almacenar N*N! estados. Entiéndanse los estados como todas las posibles permutaciones con todos sus posibles flips. Por lo tanto, la complejidad espacial de esta solución de búsqueda seria $O(N \cdot N!)$.

Por otra parte, la solución greedy que implementamos, a medida que recorre el arreglo, almacena las posiciones en que realiza los flips en una estructura aparte. En el peor de los casos va a tener que realizar 2N flips (2 por cada pancake en la pila). Añadido a esto, cada flip crea una estructura auxiliar, la cual retorna como el nuevo arreglo, generando N estructuras auxiliares en el peor caso. Por ende, la complejidad espacial de este segundo algoritmo seria $O(2N + N)$.

Respuestas a los escenarios de comprensión de problemas algorítmicos.

Para este primer escenario lo que cambiaría en nuestra implementación es que inicialmente, antes de iniciar con el proceso de ordenar los pancakes, se buscaría cuál es el elemento que se encuentra más alejado de su posición, el nuevo reto que significaría sería para hallar esto, para lo cual podemos implementar un algoritmo ávaro donde por cada elemento del arreglo miramos la diferencia entre la posición donde se encuentra el elemento y donde debería estar, al que tenga esta mayor diferencia haremos el switch, poniendo esta ficha en su posición correcta y la que estaba en dicha posición la ponemos en donde se encontraba la de mayor diferencia inicialmente. Para lograr este switch tendríamos que agarrar ambos elementos y accediendo a los índices de donde se encuentran y actualizando sus valores intercambiando los elementos. Una vez hecho esto, ya habríamos resuelto el elemento que más nos habría costado poner en la posición correcta, por lo que habríamos reducido significativamente la cantidad de flips. Esto para nuestro algoritmo avaro. Por otro lado, para nuestro grafo de estados, esta nueva modificación aumentaría drásticamente la cantidad de estados, ya que en las permutaciones y en la creación de estos, se deben agregar los estados para cada posible intercambio de cada índice con cualquier otro índice. Esto aumentaría la n^n estados nuevos que computar. Esto no representaría un gran reto ya que solo se debería modificar la creación de los estados con las permutaciones, sin embargo, a la hora de computar tal cantidad de estados si afectará en gran medida la complejidad del algoritmo.

Para el segundo escenario, en el cual se busca maximizar la cantidad de flips, se buscaría buscar por el grafo de estados, el estado que mayor cantidad de flips tenga en él. En la creación de los estados y las permutaciones se buscaría aumentar los “espacios” entre pancakes, espacios entre pancakes que los flips se encargan de corregir. La búsqueda por el grafo se realizaría buscando la mayor cantidad de espacios posibles entre pancakes, de tal manera que se deba ejecutar la mayor cantidad de flips para poder ordenar los pancakes. Esto representaría un reto, ya que se debería modificar la búsqueda en el grafo.