Práctica 1 - Permutaciones Gráciles

1st Imar Nayeli Jimenez Arango
1007424872
imar.jimenez@udea.edu.co

2nd Oscar Andrés Gutierrez Rivadeneira
1193515015
oscar.gutierrezr@udea.edu.co

I. Marco teórico

A. Permutación

Una permutación es un arreglo ordenado de todos los elementos de un conjunto, donde importa el orden en que aparecen estos elementos. Por ejemplo, si se tiene el conjunto $\{1,2,3\}$, algunas permutaciones posibles son (1,2,3), (1,3,2), (2,1,3), etc. En general, un conjunto de n elementos tiene exactamente $n!$ permutaciones.

B. Permutación grácil

Las permutaciones gráciles son un tipo especial de permutaciones con ciertas propiedades numéricas particulares. Se dice que una permutación de números enteros positivos es grácil si, al restar cada par de elementos consecutivos, los resultados son números enteros positivos diferentes entre sí, formando exactamente el conjunto de números del 1 hasta $n-1$.

C. Importancia de una permutación grácil

Las permutaciones gráciles están estrechamente relacionadas con la teoría de grafos, especialmente con el etiquetado grácil (Graceful Labeling), un problema clásico en matemáticas discretas. Este etiquetado es fundamental en el estudio de grafos y tiene aplicaciones en comunicación, distribución de frecuencias en telecomunicaciones y algoritmos eficientes de asignación.

Calcular el número de permutaciones gráciles es un problema difícil debido a su crecimiento factorial, la ausencia de algoritmos eficientes conocidos y las estrictas restricciones combinatorias involucradas. Esto lo convierte en un área activa de investigación dentro de la matemática discreta y la computación.

II. Procedimiento

El procedimiento implementado en este programa tiene como objetivo generar y contar las permutaciones gráciles de un conjunto de números del 1 al n. Para lograrlo, se emplea un enfoque basado en recursión.

En la Figura 1, se observa que el algoritmo inicia solicitando un valor n al usuario y validando que esté dentro del rango permitido. Posteriormente, se invoca la función recursiva `generate_graceful(0)` (ver Figura 2), la cual construye las permutaciones número por número. En cada nivel de recursión, se intenta colocar un número no utilizado en la secuencia, asegurando que la diferencia con el número anterior sea única y válida. Para optimizar el proceso y reducir iteraciones innecesarias, se emplean arreglos auxiliares (`used` y `diff_used`) que marcan los números y diferencias ya utilizados, evitando combinaciones inválidas.

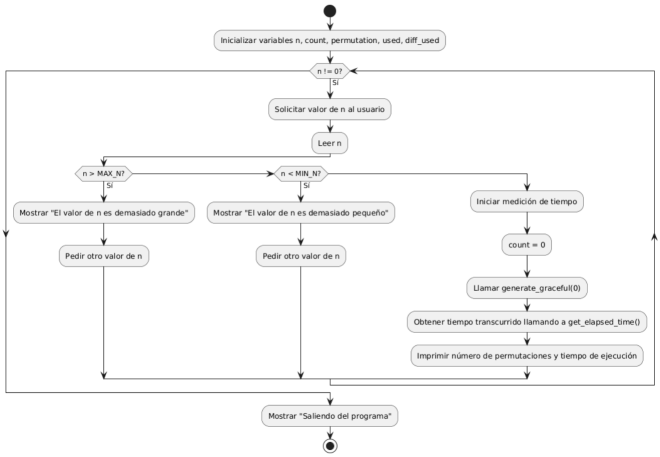

Fig. 1. Diagrama de flujo del programa.

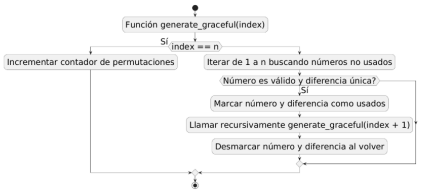

Fig. 2. Diagrama de flujo para la función que encuentra las permutaciones gráciles.

Cuando se alcanza una permutación completa, se incrementa un contador y la función retrocede en la recursión, deshaciendo los cambios para explorar otras combinaciones. Este enfoque permite recorrer únicamente los caminos viables, limitando el número de iteraciones y mejorando la eficiencia del algoritmo. Finalmente, se imprime el número total de permutaciones encontradas junto con el tiempo de ejecución, lo que permite evaluar el desempeño del método.

III. Resultados y análisis

Tras la implementación del algoritmo, se realizaron pruebas para evaluar su desempeño. En la Figura 3 se observa la relación entre el número de elementos y el tiempo requerido para generar todas las permutaciones gráciles. Se evidencia un crecimiento exponencial en la duración de la ejecución, lo que confirma la complejidad del problema.

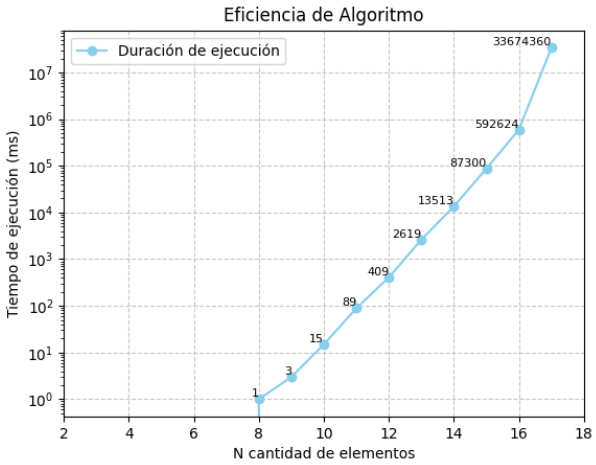

Fig. 3. Gráfico de eficiencia del programa.

Para valores pequeños de n, el tiempo de ejecución es prácticamente instantáneo. Sin embargo, conforme n aumenta, los tiempos crecen drásticamente, alcanzando varios segundos e incluso minutos para $n \geq 14$. Este comportamiento se debe a la naturaleza recursiva del algoritmo y la necesidad de evaluar múltiples combinaciones posibles.

Otro aspecto relevante observado es la variabilidad en los tiempos de ejecución dependiendo del equipo utilizado y la carga del sistema operativo. Procesadores con mayor capacidad de cómputo y menor carga de procesos concurrentes ejecutaron el algoritmo más rápido, mientras que sistemas con menor rendimiento experimentaron tiempos considerablemente mayores. Esto indica que la eficiencia del algoritmo está influenciada no solo por su complejidad, sino también por las condiciones del entorno de ejecución.

Estos resultados resaltan la importancia de explorar técnicas avanzadas para mejorar la eficiencia del algoritmo, como heurísticas de poda más agresivas o enfoques alternativos como programación dinámica y metaheurísticas.

IV. Conclusiones

El desarrollo de la práctica permitió analizar la implementación y el comportamiento de un algoritmo basado en recursión y backtracking para la generación de permutaciones gráciles. Se confirmó que, debido a la naturaleza del problema, el tiempo de ejecución crece exponencialmente con el tamaño de n, lo que lo hace computacionalmente ineficiente para valores grandes.

Se observó que la eficiencia del algoritmo no solo depende de su implementación, sino también del hardware y la carga de procesos en el sistema operativo. Esto resalta la importancia de considerar el entorno de ejecución al analizar el rendimiento de algoritmos con alta demanda computacional.

Finalmente, los resultados sugieren la necesidad de explorar estrategias más eficientes, como técnicas de poda más agresivas, heurísticas o métodos alternativos. Estas optimizaciones podrían reducir significativamente el tiempo de cómputo y permitir el análisis de instancias más grandes del problema.