UNIDAD TEMÁTICA 9 PRÁCTICOS DOMICILIARIOS INDIVIDUALES - 4

Ejercicio 1

El algoritmo de Shell Sort es una mejora del ordenamiento por inserción. La clave para su eficiencia radica en la elección adecuada de la secuencia de incrementos (gaps) que se utilizan para reducir el intervalo entre los elementos comparados.Existen distintos tipos de secuencias sumamente populares para este algoritmo, algunos ejemplos de esta son:

- La de **Hibbard, que utiliza 1, 3, 7, 15, 31, 2^k -1**, propuesta por **Hibbard en 1963**. Esta mejora el rendimiento sobre la secuencia original de Shell. Esta secuencia mejora el rendimiento sobre la secuencia original de Shell debido a que los incrementos son potencias de dos menos uno, lo que permite una mejor distribución de los elementos durante las fases iniciales del ordenamiento. Al usar esta secuencia, se logra una mejor eficiencia al reducir el número de comparaciones y movimientos necesarios para ordenar los elementos.

- **Robert Sedgewick** propuso varias secuencias en 1986, siendo una de las más conocidas. En la que los incrementos **son 1, 5, 19, 41, 109.** De acuerdo con esta estrategia, se pretende proporcionar incrementos que sean suficientemente grandes en las primeras fases del ordenamiento para incrementar la eficiencia, y posteriormente reducir gradualmente los incrementos para ajustar el orden.

- Propuesta por **H. Tokuda en 1992**, en esta secuencia los incrementos **son 1, 4, 9, 20, 46, 103.** Los incrementos en esta secuencia están diseñados para minimizar el número de comparaciones y movimientos, proporcionando una mejora significativa en el rendimiento del ShellSort, especialmente en listas grandes.

Ejercicio 2

El algoritmo de QuickSort es muy eficiente y su rendimiento depende en gran medida de la elección del pivote. Algunas de las estrategias más recomendadas para elegir el pivote son el pivote medio (Median-of-Three), que elige el pivote como la mediana de tres elementos (el primero, el medio y el último). Esta estrategia reduce la probabilidad de peores casos. Otra estrategia es el pivote aleatorio, que elige un pivote aleatorio cada vez que se ejecuta el algoritmo, ayudando a evitar peores casos en entradas específicas. La mediana de

medianas, que divide la lista en grupos de cinco, encuentra la mediana de cada grupo y luego utiliza la mediana de esas medianas como el pivote, es eficiente pero más compleja de implementar.

En cuanto a la implementación en librerías modernas, Java utiliza una versión dual-pivot de QuickSort en su clase Arrays. En Python, la función sort() de las listas usa Timsort, una combinación de MergeSort y QuickSort. C# utiliza introspective sort, que combina QuickSort, MergeSort y HeapSort.

Ejercicio 3

```
Algoritmo SonDisjuntosOrdenados(A, B)
    Entrada: Conjunto A de tamaño m, Conjunto B de tamaño n
    Salida: Verdadero si los conjuntos son disjuntos, Falso en caso contrario

    // Ordenar ambos conjuntos
    Ordenar(A)
    Ordenar(B)

    //Inicializar punteros para ambos conjuntos
    i ← 0
    j ← 0

    // Recorrer ambos conjuntos comparando elementos
    Mientras i < m y j < n hacer
        Si A[i] < B[j] entonces
            i ← i + 1
        Sino si A[i] > B[j] entonces
            j ← j + 1
        Sino // A[i] == B[j]
            Retornar Falso // Encontramos un elemento común
    Fin Mientras

    // Si no encontramos elementos comunes, los conjuntos son disjuntos
```

Retornar Verdadero

Fin Algoritmo

Tiempo de ejecución: El tiempo de ejecución del algoritmo depende de ambos conjuntos y el tiempo para recorrer y comparar los elementos. Ordenar el conjunto (A) tiene una complejidad de (O(m log m)) y ordenar el conjunto (B) tiene una complejidad de (O(n log n)), donde (m) y (n) son los tamaños de los conjuntos (A) y (B), respectivamente. Después de ordenar, recorrer ambos conjuntos y comparar los elementos tiene una complejidad de (O(m + n)). Por lo tanto, la complejidad total del algoritmo es (O(m log m + n log n)).