UT9 - Práctico Domiciliario 4

UT9 - Práctico Domiciliario 4

Ejercicio 1 - ShellSort: Secuencias de Incrementos Modernas

1. **Secuencia de Ciura (2001)**

- Incrementos: `1, 4, 10, 23, 57, 132, 301, 701...`
- Muy eficiente en la práctica, especialmente en arreglos pequeños (<100.000 elementos).
- Usada por implementaciones estándar de ShellSort.
- Fuente: *Ciura, M. (2001). Best Increments for the Average Case of Shellsort.*

2. **Tokuda mejorado (Lee, 2021)**

- Incrementos aproximados: `1, 4, 9, 20, 45, 102, 230, 516...`
- Calculados en base a fórmulas matemáticas para mejorar el rendimiento.
- Fuente: *Lee, M.-S. (2021). An Improved Empirical Study of Shellsort.*

3. **Gonnet & Baeza-Yates (1991)**

- Secuencia definida recursivamente: hk=⌊5 hk−1−111⌋h*k* = *left**lfloor* *frac{5*,*h*{k-1} - 1}{11} \right\rfloor con h0=Nh_0 = N
- Fuente: *Gonnet, G. H., & Baeza-Yates, R. A. (1991). Handbook of Algorithms and Data Structures.*

Estas secuencias se pueden probar reemplazando la lista de incrementos en la implementación de Shellsort.

Ejercicio 2 - QuickSort: Funciones de Pivote

1. **Pivote Aleatorio**

- Selección aleatoria para evitar el peor caso.
- Usado en Python, JavaScript y otros lenguajes.
- Fuentes: GeeksForGeeks, Wikipedia.

2. **Mediana de Tres**

- Pivote es la mediana entre el primer, medio y último elemento.
- Muy eficiente, usado en C++ STL y otras implementaciones.
- Fuente: Bentley & McIlroy (1993).

3. **Mediana de Medianas (BFPRT)**

- Garantiza selección de pivote en tiempo lineal.
- Costoso en práctica, poco usado en implementaciones comunes.

4. **QuickSort Dual-Pivote (Java desde v7)**

- Usa dos pivotes para particionar en tres partes.
- Mejora el rendimiento promedio.
- Fuente: Yaroslavskiy, Bentley y Bloch.

Librerías modernas como Python usan Timsort (mezcla de mergesort e inserción).

Ejercicio 3 - Disjunción de Conjuntos

- **Algoritmo con HashSet:**
  - Insertar elementos del conjunto más pequeño en un set (hash table).
  - Recorrer el otro conjunto verificando intersección.
  - Complejidad: $O(m + n)$, ideal cuando $m \ll n$.
- **Algoritmo con Ordenamiento + Dos Punteros:**
  - Ordenar ambos conjuntos.
  - Recorrer simultáneamente con dos punteros para detectar intersección.
  - Complejidad: $O(m \log m + n \log n)$.

Ambos métodos son eficientes para determinar si dos conjuntos son disjuntos.

Ejercicio 4 - Introsort y Mejora de QuickSort

Para evitar el peor caso ($O(n^2)$) en QuickSort se usa Introsort:

- Se limita la profundidad de recursión a $2 * \log_2(n)$.
- Si se supera ese límite, se cambia a HeapSort ($O(n \log n)$ garantizado).
- Para subconjuntos pequeños (≤16 elementos) se usa InsertionSort, más eficiente en ese rango.
- Esta estrategia es usada en implementaciones estándar en C++ y Java.

Pseudocódigo simplificado:

```
function introsort(arr, lo, hi, depthLimit):
    if hi - lo < 16:
        insertionSort(arr, lo, hi)
    else if depthLimit == 0:
        heapSort(arr, lo, hi)
    else:
        p = partition(arr, lo, hi)
        introsort(arr, lo, p - 1, depthLimit - 1)
        introsort(arr, p + 1, hi, depthLimit - 1)
```
