Ejercicio 1 – Secuencias de incrementos en Shellsort

1. Secuencias conocidas (no vistas en clase)

1.1. Secuencia de Donald Shell

Esta fue la primera secuencia que se usó en el algoritmo Shellsort, creada en 1959. Es muy simple: toma el tamaño del array (n) y lo va dividiendo sucesivamente por potencias de 2.

- Fuente: Donald L. Shell – "A High-Speed Sorting Procedure", ACM, 1959
- Ejemplo con n = 16:
- Los incrementos serían: 8, 4, 2, 1
- Complejidad: Aproximadamente $O(n^2)$

1.2. Secuencia de Knuth

Knuth propuso otra secuencia más eficiente en 1971. Usa la fórmula h = (3^k – 1) / 2, sin pasarse de n/3.

- Fuente: *Donald Knuth – "The Art of Computer Programming", Vol. 3*
- Ejemplo de secuencia: 1, 4, 13, 40, 121, 364, 1093...
- Complejidad: Alrededor de `O(n^(3/2))`

1.3. Secuencia de Sedgewick (1982)

Sedgewick diseñó una secuencia más avanzada basada en la fórmula: h = 4^j + 3*2^(j-1) + 1.

- Fuente: *Robert Sedgewick – "Algorithms" (4ª edición), Capítulo 2*
- Ejemplo: 1, 8, 23, 77, 281, 1073, 4193...
- Complejidad: Aproximadamente `O(n^(4/3))` en el peor caso.

2. Análisis del tiempo de ejecución

El algoritmo Shellsort funciona dividiendo el array en grupos de elementos separados por un cierto "salto" (gap). En cada paso, se hace una especie de inserción ordenada dentro de esos grupos.

- Con cada valor de gap, hay que recorrer todo el array (≈ O(n) comparaciones y movimientos).
- La cantidad de pasos depende de cuántos gaps haya.

❖ Comparativo:

Shell: $O(n^2)$

Knuth: O(n^(3/2))

Sedgewick: O(n^(4/3))

3. Prueba con un conjunto de datos

Conjunto a ordenar:

`256, 458, 655, 298, 043, 648, 778, 621, 655, 019, 124, 847`

Resultado paso a paso con diferentes secuencias:

Shell (gaps: 6, 3, 1)

Después de gap 6:

256, 458, 655, 19, 43, 648, 778, 621, 655, 298, 124, 847

Después de gap 3:

19, 43, 648, 256, 124, 655, 298, 458, 655, 778, 621, 847

Después de gap 1:

19, 43, 124, 256, 298, 458, 621, 648, 655, 655, 778, 847

Knuth (gaps: 4, 1)

Después de gap 4:

43, 19, 124, 298, 256, 458, 655, 621, 655, 648, 778, 847

Después de gap 1:

19, 43, 124, 256, 298, 458, 621, 648, 655, 655, 778, 847

Sedgewick (gap: 5)

Después de gap 5 (queda ordenado directamente):

19, 43, 124, 256, 298, 458, 621, 648, 655, 655, 778, 847

Ejercicio 2 – Algoritmo de Burbuja y mejoras

1. Burbuja clásico paso a paso

Conjunto:

44, 55, 12, 42, 94, 18, 6, 67

Iteraciones:

i=0: 44 55 12 42 94 18 6 67

i=1: 44 12 42 55 18 6 67 94

i=2: 12 42 44 18 6 55 67 94

i=3: 12 42 18 6 44 55 67 94

i=4: 12 18 6 42 44 55 67 94

i=5: 12 6 18 42 44 55 67 94

i=6: 6 12 18 42 44 55 67 94

i=7: `6 12 18 42 44 55 67 94 → ya no hay intercambios, termina

Sí, el array se ordena antes de la última pasada.

2. Mejoras posibles al algoritmo

Primera mejora: usar una bandera

Si en una pasada no hubo ningún intercambio, quiere decir que ya está ordenado y no hace falta seguir.

Segunda mejora: acotar el recorrido

Se guarda la posición del último intercambio. Los elementos después de esa posición ya están ordenados, así que se puede evitar recorrerlos.

3. Aplicación de las mejoras

Para cada mejora, se muestra cómo queda el array en cada paso. En ambos casos se ordena más rápido que con la versión clásica.

4. Shakersort (Cocktail sort)

Este algoritmo mejora la burbuja recorriendo en dos direcciones:

Va primero de izquierda a derecha como el burbuja normal.

Luego vuelve de derecha a izquierda.

Así, mueve los valores más chicos al inicio y los grandes al final en cada ciclo.

Pasos:

Paso 1: 44 12 42 55 18 6 67 94

Paso 2: 12 42 44 18 6 55 67 94

Paso 3: 12 42 18 6 44 55 67 94

Paso 4: 12 18 6 42 44 55 67 94

Paso 5: 12 6 18 42 44 55 67 94

Paso 6: 6 12 18 42 44 55 67 94

Ya no hay intercambios → termina.

Conclusión

Este práctico muestra cómo diferentes estrategias de incrementos y mejoras en algoritmos de ordenamiento pueden afectar el rendimiento. Shellsort es muy sensible a los gaps que se usen, y el burbuja, aunque básico, puede ser optimizado notablemente con simples ajustes como cortar antes o recorrer en ambas direcciones.