Algoritmos y Estructuras de Datos
UT9-PD2

Santiago Blanco

25-06-2025

Ejercicio #1

Secuencias de Shellsort importantes

1. Secuencia original de Donald Shell

Donald Shell introdujo Shellsort en 1959, y esta fue la primera secuencia de incrementos utilizada. Es muy simple y se basa en divisiones sucesivas de N por potencias de 2.

$\rightarrow$ Donald L. Shell. "A High-Speed Sorting Procedure", Communications of the ACM, 1959.

Formula general: $h_k = \left\lfloor \frac{n}{2^k} \right\rfloor$

- Ejemplo para $N = 16$: 8, 4, 2, 1
- **Complejidad:** $O(N^2)$

2. Secuencia Knuth

Propuesta en 1971 por Donald Knuth en "The Art of Computer Programming, Vol. 3"

Formula general: $\frac{3^k - 1}{2}$, no mayor que $\left\lceil \frac{n}{3} \right\rceil$

- Ejemplo: 1, 4, 13, 40, 121, 364, 1093…
- **Complejidad:** $O\left(n^{\frac{3}{2}}\right)$

3. Secuencia Sedgewick (1982)

Robert Sedgewick introdujo una secuencia basada en $4^j + 3 \cdot 2^{j-1} + 1$, con orden $O\left(n^{\frac{4}{3}}\right)$.

$\rightarrow$ *Algorithms 4th edition*, Robert Sedgewick. - Capítulo 2: Sorting

- Ejemplo: 1, 8, 23, 77, 281, 1073, 4193, 16577, …
- **Implementación**: calcular hasta que el gap ≤ N, luego iterar decreciendo.
- **Complejidad**: $O\left(n^{\frac{4}{3}}\right)$ peor caso.

Pseudocódigo de Shellsort

```
ShellSort(A[1..N], gaps[1..t])
COM
  PARA CADA h en gaps (de mayor a menor) HACER
    PARA i DESDE h + 1 HASTA N HACER
      temp ← A[i]
      j ← i
      MIENTRAS j > h y A[j - h] > temp HACER
        A[j] ← A[j - h]
        j ← j - h
      FIN MIENTRAS
      A[j] ← temp
    FIN PARA
  FIN PARA
FIN
```

5. Análisis complejo de tiempo

- **Cada paso con gap** requiere recorrer el array y hacer inserciones con distancia, que implica $O(n)$ comparaciones y $O(n)$ movimientos.
- El número de pasos: $d = |\text{gaps}|$.
  - **Shell**: tiempo total $O(n^2)$.
  - **Knuth:** Total $O\left(n^{\frac{3}{2}}\right)$
  - **Sedgewick**: $d = \Theta(\log n)$ -> $O\left(n^{\frac{4}{3}}\right)$

La secuencia en cuestión

Con Shell

Gaps: 6, 3, 1

- Después del gap 6:
  - 256, 458, 655, 19, 43, 648, 778, 621, 655, 298, 124, 847
- Después del gap 3:
  - 19, 43, 648, 256, 124, 655, 298, 458, 655, 778, 621, 847
- Después del gap 1:
  - 19, 43, 124, 256, 298, 458, 621, 648, 655, 655, 778, 847

Con Knuth

Gaps: 4, 1

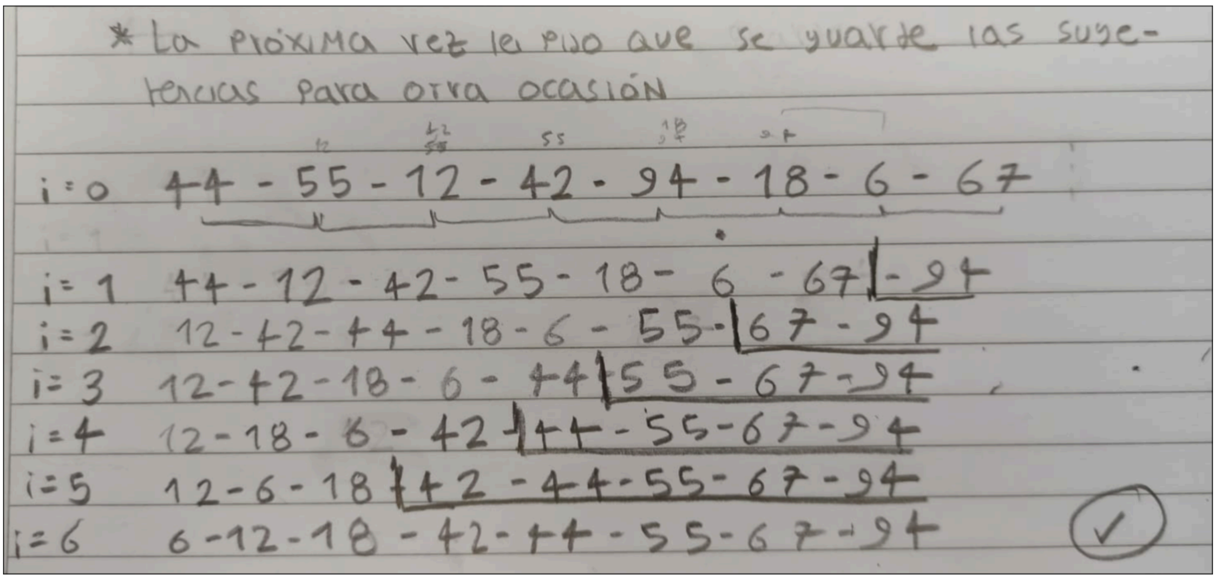
- Después del gap 4:
  - 43, 19, 124, 298, 256, 458, 655, 621, 655, 648, 778, 847
- Después del gap 1:
  - 19, 43, 124, 256, 298, 458, 621, 648, 655, 655, 778, 847

Con Sedgewick

Gap: 5

- Después del gap 5:
  - 19, 43, 124, 256, 298, 458, 621, 648, 655, 655, 778, 847

Ejercicio #2


* La próxima vez le pido que se guarde las sugerencias para otra ocasión

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

2.

Vamos al bobelsort con una primera mejora:
Agregamos una bandeirantes (huboIntercambio). Si en una iteración completa no se intercambia ningún elemento, entonces el array ya está ordenado.

```
bubbleSort(Arr):
COM
    n <- Arr.length()
    REPETIR
        huboIntercambio <- FALSO
        para i <- 0 hasta n-2 hacer
            si A[i] > A[i+1] entonces
                intercambiar A[i] con A[i+1]
                huboIntercambio <- VERDADERO
        n <- n - 1
    HASTA que NO huboIntercambio
FIN
```

```
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 (no hay intercambios, terminemus)
```

—

Vamos a por otra mejora:

Si en una iteración, el último swap ocurrió en una posición determinada, los elementos después de esa posición ya están ordenados. Podemos acortar el rango.

```
bubbleSortDos(arr):
COM
    n <- arr.length()
    REPETIR
        ultimaPos <- 0
        para i <- 0 hasta n-2 hacer
            si A[i] > A[i+1] entonces
                intercambiar A[i] con A[i+1]
                ultimaPos <- i+1
        n <- ultimaPos
    HASTA que n <= 1
FIN
```

```
i = 0 (n=8):     44   55   12   42   94   18    6   67  -> últimaPos = 7
i = 1 (n=7):     44   12   42   55   18    6   67   94  -> últimaPos = 6
i = 2 (n=6):     12   42   44   18    6   55   67   94  -> últimaPos = 5
i = 3 (n=5):     12   42   18    6   44   55   67   94  -> últimaPos = 4
i = 4 (n=4):     12   18    6   42   44   55   67   94  -> últimaPos = 3
```

```
i = 5 (n=3):     12    6   18   42   44   55   67   94  -> últimaPos = 2
i = 6 (n=2):      6   12   18   42   44   55   67   94  -> últimaPos = 1
i = 7 (n=1):                          (terminate)
```

Shakersort

Shakersort recorre el array en ambas direcciones en cada iteración:

- Primero de izquierda a derecha (bubblesort normal).
- Luego de derecha a izquierda (como burbuja inversa).

Ergo, los pares más grandes van al final y las más pequeñas al principio en cada ciclo.

```
Inicio:            44   55   12   42   94   18    6   67

-> izq a der:
Paso 1:            44   12   42   55   18    6   67   94
-> izq a der:
Paso 2:            12   42   44   18    6   55   67   94
-> Izquierda a derecha:
Paso 3:            12   42   18    6   44   55   67   94
-> izq a der:
Paso 4:            12   18    6   42   44   55   67   94
-> Izquierda a derecha:
Paso 5:            12   6   18   42   44   55   67   94
-> izq a der:
Paso 6:             6   12   18   42   44   55   67   94
-> Siguiente iteración: no swaps -> termina
```