Algoritmos de Ordenamiento y Archivos de Texto



Fundamentos de Programación / Algoritmos y Programación I

Semana 6 · Clase Teórica · Prof. Servetto

- Ordenamiento: selección, inserción, burbuja, merge sort, quick sort
- Archivos de texto: open, read, write, with
- Temas holísticos: compiladores e intérpretes, procesos del SO

Agenda de la clase

1	**El problema del ordenamiento**	Definición, clasificación, conexión con búsqueda
2	**Algoritmos cuadráticos $O(n^2)$**	Selección, inserción, burbuja — correctitud y complejidad
3	**Merge sort: divide y vencerás**	Fusión, O(n log n), estabilidad, trade-off memoria
4	**Quick sort: particionar y vencer**	Buen pivote vs. caso degenerado $O(n^2)$
5	**Archivos de texto en Python**	open, read, write, with, lectura línea a línea
6	**Compiladores e intérpretes [Holístico]**	Fases de compilación, bytecode, CPython, dis
7	**Procesos del SO [Holístico]**	Ciclo de vida, E/S y ordenamiento externo

1. El Problema del Ordenamiento

Dada una secuencia de n elementos, producir una permutación en orden no decreciente.

Contrato de la función de ordenamiento

Entrada: secuencia datos de n elementos comparables

Salida: datos reordenada: datos[0] ≤ datos[1] ≤ … ≤ datos[n−1]

Poscondición: la salida es permutación de la entrada

Precondición: elementos admiten comparación con <

Clasificación de algoritmos

Comparativos: determinan orden comparando pares

In-place: O(1) memoria extra (vs. O(n) auxiliar)

Estables: preservan orden relativo de iguales

Conexión con Semana 5: ordenar es el costo de poder buscar con binaria. Si ordenar cuesta O(n log n) y buscar O(log n), conviene ordenar una vez y buscar muchas veces.

El ordenamiento es, junto con la búsqueda, uno de los problemas más estudiados en la historia de la computación.

2a. Ordenamiento por Selección

En cada pasada, seleccionar el mínimo de la zona desordenada y ubicarlo en su posición definitiva.

Pseudocódigo

```
para i ← 0 hasta n − 2:
    pos_min ← i
    para j ← i+1 hasta n − 1:
        si datos[j] < datos[pos_min]:
            pos_min ← j
    si pos_min ≠ i:
        intercambiar datos[i] ↔
datos[pos_min]
```

Análisis de eficiencia

- Comparaciones: siempre n(n−1)/2
- No se adapta al estado inicial
- Intercambios: máximo n−1 (O(n))
- In-place: O(1) memoria extra
- NO es estable

Complejidad: $O(n^2)$ siempre

Traza: [64, 25, 12, 22, 11]

i	Sublista	pos_min	Resultado
0	[64,25,12,22,11]	4 (11)	[11, 25, 12, 22, 64]
1	[25,12,22,64]	2 (12)	[11, 12, 25, 22, 64]
2	[25,22,64]	3 (22)	[11, 12, 22, 25, 64]
3	[25,64]	3 (25)	**[11, 12, 22, 25, 64]** ✓

2b. Ordenamiento por Inserción

Tomar cada elemento e insertarlo en su posición correcta dentro de la zona ya ordenada.

```
para i ← 1 hasta n − 1:        # determinado
    clave ← datos[i]
    j ← i − 1
    mientras j ≥ 0 y datos[j] > clave:
        datos[j+1] ← datos[j]   # desplazar
        j ← j − 1
    datos[j+1] ← clave          # insertar
```

Peor caso (inverso): $O(n^2)$

Mejor caso (ordenada): $O(n)$

Caso promedio: $O(n^2)$

Estable: Sí

In-place: $O(1)$

Bucle externo: for/range() (determinado: exactamente n−1 iteraciones). Bucle interno: while (condición compuesta indeterminada: j ≥ 0 AND datos[j] > clave). La evaluación en cortocircuito evita acceso fuera de rango.

Ventaja clave: se adapta al grado de desorden. Excelente para listas pequeñas o casi ordenadas. Timsort (sorted() de Python) usa inserción internamente para sublistas cortas.

2c. Ordenamiento Burbuja

Comparar pares adyacentes; el mayor «burbujea» al final. Optimización: parada temprana.

```
i ← 0; hubo_intercambio ← True
mientras i < n-1 y hubo_intercambio:
    hubo_intercambio ← False
    para j ← 0 hasta n-2-i:
        si datos[j] > datos[j+1]:
            intercambiar datos[j] ↔ datos[j+1]
            hubo_intercambio ← True
    i ← i + 1
```

Condición compuesta del while

i < n−1:
pasadas restantes
hubo_intercambio:
la pasada anterior hizo al menos un swap
Sin break:
la condición compuesta maneja ambos criterios.

Peor caso: O(n²) | Mejor caso (con optim.): O(n) | Estable: Sí | In-place: O(1)

Comparación de algoritmos cuadráticos

Aspecto	Selección	Inserción	Burbuja
Mejor caso	O(n²)	**O(n) ✓**	**O(n) ✓**
Swaps	**O(n) ✓**	O(n²) desplaz.	O(n²)
Estable	No	**Sí**	**Sí**
Cuándo usar	Swaps costosos	Listas pequeñas/casi ord.	Pedagógico

3. Merge Sort: Divide y Vencerás

Dividir por la mitad, ordenar recursivamente, fusionar. O(n log n) siempre.

1 Dividir

Partir la lista por la mitad

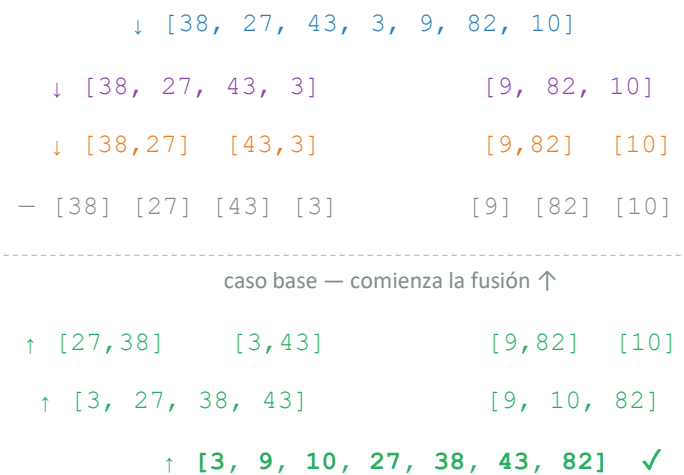
2 Vencer

Ordenar recursivamente cada mitad (caso base: 0 o 1 elementos)

3 Combinar

Fusionar las dos mitades ordenadas en una sola secuencia ordenada

Ejemplo: [38, 27, 43, 3, 9, 82, 10]



Merge Sort: fusión y análisis de complejidad

La operación de fusión (merge) es la clave del algoritmo.

```
def fusionar(izq, der):
    resultado = []
    i, j = 0, 0
    while i < len(izq) and j < len(der):
        if izq[i] <= der[j]:   # <= para
estabilidad
            resultado.append(izq[i]); i += 1
        else:
            resultado.append(der[j]); j += 1
    # copiar los restantes...
    return resultado
```

¿Por qué O(n log n)?

Nivel 0: 1 fusión de n elem → O(n)

Nivel 1: 2 fusiones de n/2 → O(n)

Nivel 2: 4 fusiones de n/4 → O(n)

Cada nivel: O(n) total

¿Cuántos niveles? $\log_2(n)$

Total = O(n) × O(log n) = O(n log n)

Propiedades: estable (usar ≤) · O(n) memoria auxiliar · O(n log n) en todos los casos

Trade-off: se gana eficiencia temporal (O(n log n) en lugar de $O(n^2)$) pero se paga con espacio adicional O(n). Los algoritmos cuadráticos son in-place (O(1) memoria extra). n >> 1 es equivalente a n // 2 pero más eficiente a nivel de ALU.

4. Quick Sort: Particionar y Vencer

Elegir pivote, particionar (menores|pivote|mayores), recurrir. O(n log n) promedio.

Merge sort: dividir es trivial, combinar es costoso (fusionar)
Quick sort: dividir es costoso (particionar), combinar es trivial (concatenar)

Caso degenerado: ya ordenada + pivote último

```
[3,9,10,27,38,43,82]   piv=82   n=7
[3,9,10,27,38,43]      piv=43   n=6
[3,9,10,27,38]         piv=38   n=5
[3,9,10,27]            piv=27   n=4
[3,9,10]               piv=10   n=3
[3,9]                  piv=9    n=2
[3]                    base     n=1
```

n−1 niveles × O(n) comp. = $O(n^2)$

Buen pivote (último elemento): [38,27,43,3,9,82,10]

```
                [3,9] | 10 | [38,27,43,82]
        [3] | 9 | []          [38,27,43] | 82 | []
                        [38,27] | 43 | []
                [] | 27 | [38]
```

→ [3, 9, 10, 27, 38, 43, 82] ✓

Mitigación del peor caso

- Pivote aleatorio
- Mediana de tres (primer, medio, último)
- Cambiar a inserción para sublistas pequeñas
- **Versión in-place: Semana 11**

Tabla Resumen de Complejidades

Algoritmo	Mejor	Promedio	Peor	Memoria	Estable
Selección	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	No
Inserción	**O(n)**	$O(n^2)$	$O(n^2)$	O(1)	**Sí**
Burbuja	**O(n)**	$O(n^2)$	$O(n^2)$	O(1)	**Sí**
Merge Sort	**O(n log n)**	**O(n log n)**	**O(n log n)**	O(n)	**Sí**
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)	No
Python sorted()	**O(n)**	O(n log n)	O(n log n)	O(n)	**Sí**

Timsort (sorted() y .sort()): algoritmo híbrido de Tim Peters (2002). Combina inserción (sublistas cortas) + merge sort (fusión global). Aprovecha «corridas» parcialmente ordenadas en datos reales. Mejor caso O(n) cuando ya está ordenado.

Aspecto	Merge Sort	Quick Sort
Dónde está el trabajo	En la fusión (después)	En la partición (antes)
Localidad de caché	Menor (listas dispersas)	Mayor (datos contiguos)
En la práctica	Predecible; ideal para disco	Más rápido en RAM; base de qsort()

5. Archivos de Texto en Python

Persistencia de datos: información que sobrevive al programa.

1. ABRIR
`open(ruta, modo)`

→

2. OPERAR
`read / write / ...`

→

3. CERRAR
`close() / with`

Modos de apertura

Modo	Si no existe	Si existe
`'r'`	Error	Lee desde inicio
`'w'`	Lo crea	¡Borra contenido!
`'a'`	Lo crea	Escribe al final

Context manager: with

```
# Recomendado: cierre automático
with open('datos.txt', 'r') as f:
    contenido = f.read()
# f.close() se ejecuta siempre,
# incluso si ocurre un error
```

Tres formas de lectura

`f.read()`
Todo a un string
Carga todo en RAM

`f.readlines()`
Lista de líneas
Carga todo en RAM

`for linea in f:`
Línea a línea
Memoria constante ✓

6. Compiladores e Intérpretes [Holístico]

Fases de compilación: del código fuente al ejecutable o bytecode.



Ejemplo: x = a + 1

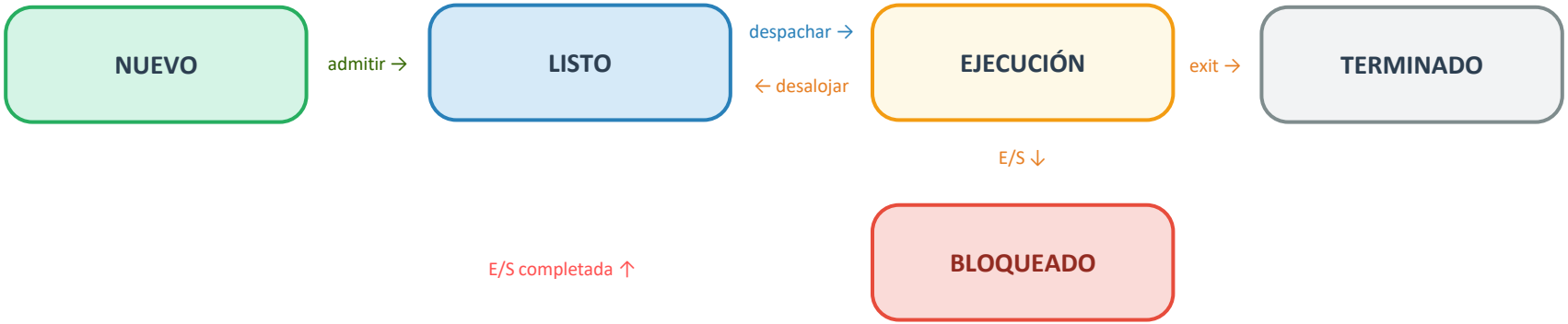
Léxico	Sintáctico	Semántico	Generación

`[ID:'x'] [OP:'=']`
`[ID:'a'] [OP:'+']`
`[NUM:1]`
	`Asign`
	`/ \`
	`x Suma`
	`/ \`
	`a 1`
		¿'a' declarada? ✓
		¿Tipos sumables? ✓
		¿'x' asignable? ✓
			`LOAD_FAST a`
			`LOAD_CONST 1`
			`BINARY_ADD`
			`STORE_FAST x`

Python = caso híbrido: CPython compila a bytecode (.pyc) y luego la PVM lo interpreta. Verificable con el módulo dis. Cada vez que ejecutamos «python programa.py», CPython realiza las 4 fases internamente.

7. Procesos del SO y su Ciclo de Vida

Cuando ejecutamos nuestro programa, el SO crea un proceso para gestionarlo. [Holístico]



¿Qué pasa al leer un archivo?

Cada read/write → EJECUCIÓN → BLOQUEADO. El SO despacha otro proceso. Al completarse la E/S → LISTO → EJECUCIÓN. Los programas con mucho acceso a disco son lentos por los bloqueos, no por la CPU.

Ordenamiento externo

Si los datos no caben en RAM → merge sort en disco: (1) crear corridas ordenadas en memoria, (2) fusionar por pasadas. Merge sort es ideal por su acceso secuencial (minimiza seeks de disco).

Visión de conjunto

⬅️ **Semanas 1–5**

Notación O grande, modularización, patrones de secuencias, jerarquía de memoria → base para los algoritmos de ordenamiento.

📌 **Semana 6 (hoy)**

Selección, inserción, burbuja ($O(n^2)$), merge sort y quick sort ($O(n \log n)$). Archivos de texto. Compiladores/intérpretes. Procesos del SO.

➡️ **Semana 7**

Punteros, referencias y modelo de memoria: id(), mutabilidad, aliasing. Paso por referencia en funciones.

🔮 **Semanas posteriores**

Recursividad (S8), estructuras de datos (S9), quick sort in-place (S11). Proyecto final: elegir y justificar algoritmo.

La elección del algoritmo de ordenamiento es un ejemplo paradigmático de decisión de diseño algorítmico: no hay respuesta universal, sino análisis de compromisos.