Algoritmos de Búsqueda y Complejidad Algorítmica

Fundamentos de Programación / Algoritmos y Programación I
Semana 5 · Clase Teórica · Prof. Servetto

- Búsquedas lineal, binaria y por interpolación lineal
- Notación O grande

Agenda de la clase

1	**El problema de la búsqueda**	Definición formal, contrato de la función
2	**Búsqueda lineal**	Algoritmo, traza, casos de prueba, análisis O(n)
3	**Optimizaciones de búsqueda lineal**	Move-to-Front, Transposición, inserción al comienzo
4	**Complejidad algorítmica: O grande**	Definición, reglas, complejidades comunes
5	**Búsqueda binaria**	Iterativa, recursiva (anticipo), correctitud
6	**Búsqueda por interpolación**	Estimación de posición, análisis de conveniencia
7	**Temas holísticos**	Jerarquía de memoria, sistema de archivos

1. El Problema de la Búsqueda

Dada una secuencia de *n* elementos y un valor **clave**, determinar si la clave se encuentra en la secuencia y reportar su posición.

Contrato de la función de búsqueda

Entrada:

secuencia datos y valor clave

Salida:

índice i tal que datos[i] == clave, o −1

Precondición:

datos es indexable; clave es comparable

Conexión con Semanas 1–4

Semana 3:

Patrón de búsqueda con condición de corte

Semana 4:

Funciones con contrato y docstring

La búsqueda es una de las operaciones más fundamentales en computación.

2. Búsqueda lineal: idea y pseudocódigo

Recorrer la secuencia elemento por elemento, comparando cada uno con la clave.

Pseudocódigo

```
i ← 0
mientras i < n y datos[i] ≠ clave:
    i ← i + 1
si i < n:
    retornar i
sino:
    retornar −1
```

Análisis de eficiencia

Mejor caso:
clave en posición 0 → 1 comparación

Peor caso:
clave ausente o en última posición → n comp.

Caso promedio:
(n + 1) / 2 comparaciones

Complejidad: O(n)

Condición compuesta con cortocircuito: si i < n es False, Python no evalúa datos[i] → evita acceso fuera de rango.

3. Optimizaciones de la Búsqueda Lineal

Si un elemento fue buscado recientemente, es probable que se busque de nuevo pronto (localidad temporal).

Inserción al comienzo

Elementos nuevos se insertan en posición 0 (no al final).

Los más recientes se encuentran primero.

Costo: O(n) en arreglo, O(1) en lista enlazada.

Move-to-Front

Al encontrar la clave, moverla a la posición 0.

Adaptación inmediata: la siguiente búsqueda es O(1).

Riesgo: una búsqueda aislada desplaza a las frecuentes.

Transposición

Al encontrar la clave, intercambiarla con el elemento anterior.

Ascenso gradual: las claves frecuentes suben de a uno.

Más estable que MTF ante búsquedas esporádicas.

Las tres heurísticas explotan el mismo principio que la caché del procesador: localidad de referencia.

Comparación de heurísticas

Aspecto	Sin optimización	Move-to-Front	Transposición
Tras encontrar	No modifica la lista	Mueve a posición 0	Swap con el anterior
Adaptación	Ninguna	Inmediata	Gradual
Estabilidad	Orden fijo	Baja	Alta
Costo reorg.	0	O(n) en arreglo	O(1) siempre
Mejor para	—	Distribución cambiante	Distribución estable

Ejemplo: datos = [A, B, C, D, E], búsquedas: D, D, D, D

Move-to-Front: 4 + 1 + 1 + 1 = 7 comparaciones
Transposición: 4 + 3 + 2 + 1 = 10 comparaciones
Sin optimización: 4 + 4 + 4 + 4 = 16 comparaciones

4. Notación O grande: la idea central

Describe cómo crece el tiempo de ejecución cuando crece el tamaño de la entrada n.

Definición

$T(n) \in O(f(n))$ si existen $c > 0$ y $n_0 \geq 1$ tales que $T(n) \leq c \cdot f(n)$ para todo $n \geq n_0$

Reglas prácticas

1. **Constantes no importan:**
   $O(3n) = O(n)$
2. **Términos menores no importan:**
   $O(n^2 + 3n) = O(n^2)$

Notación	Nombre	Ejemplo	n=1.000	n=1.000.000
O(1)	Constante	Acceder a datos[i]	1	1
O(log n)	Logarítmica	Búsqueda binaria	~10	~20
O(n)	Lineal	Búsqueda lineal	1.000	1.000.000
O(n log n)	Lineal-log.	Merge sort	~10.000	~20.000.000
O(n^2)	Cuadrática	Burbuja	1.000.000	10^{12}

5. Búsqueda binaria: dividir para conquistar

⚠ **Precondición: la secuencia DEBE estar ordenada. Si no se cumple, el resultado es incorrecto.**

① Examinar el elemento central de la región

② Si coincide con la clave → terminar

③ Si clave < central → buscar en mitad izquierda

④ Si clave > central → buscar en mitad derecha

⑤ Repetir hasta encontrar o región vacía

Eficiencia

En cada paso se descarta la mitad.

`n → n/2 → n/4 → ... → 1`

Pasos: $\lceil \log_2(n) \rceil$

Complejidad: O(log n)

n = 1.000.000

Lineal: hasta 1.000.000 comp.

Binaria: hasta 20 comp.

50.000× más rápida

Correctitud: invariante del bucle

Invariante: si la clave está en datos, entonces está en datos[izq..der]

Inicialización

izq = 0, der = n−1
Abarca toda la secuencia.
Invariante: trivialmente verdadero.

Mantenimiento

Si datos[medio] < clave:
izq ← medio + 1
Descarta solo elementos menores → invariante se preserva.

Terminación

izq > der → región vacía.
Si la clave existiera, estaría en ∅: contradicción.
Retornar −1 es correcto.

Errores clásicos en la búsqueda binaria

1. Desbordamiento: (izq+der)//2 puede fallar en C/Java. Usar izq + (der−izq)//2
2. izq < der en vez de izq <= der: omite región de 1 elemento
3. izq = medio en vez de izq = medio + 1: bucle infinito

6. Búsqueda por interpolación: estimar la posición

En lugar de ir siempre al medio, estimar la posición de la clave por interpolación lineal.

```
pos = izq + ⌊(clave − datos[izq]) × (der − izq) / (datos[der] − datos[izq])⌋
```

Distribución	Interpolación	Binaria	Recomendación
Uniforme	**O(log log n)**	O(log n)	Interpolación ✓
Aprox. uniforme (DNIs, timestamps)	**~O(log log n)**	O(log n)	Interpolación ✓
Sesgada / exponencial	**O(n)**	O(log n)	Binaria ✓
Desconocida	Impredecible	**O(log n)**	Binaria (segura)

n = 1.000.000: $\log_2(n) \approx 20$ vs. $\log_2(\log_2(n)) \approx 4$ → 5× menos comparaciones en distribución uniforme

Comparación: Lineal vs. Binaria vs. Interpolación

Aspecto	Lineal	Binaria	Interpolación
Precondición	Ninguna	Ordenada	Ordenada + numérica
Peor caso	O(n)	O(log n)	O(n) si sesgada
Caso promedio	O(n)	O(log n)	O(log log n) si uniforme
n = 10^6	1.000.000	~20	~4 (unif.) o 10^6 (sesg.)
Cuándo usar	Datos desordenados o pequeños	Siempre segura si ordenado	Distribución conocida y uniforme

¿Cuándo conviene ordenar y luego buscar?

Con m búsquedas: lineal sin ordenar = m × O(n) vs. ordenar + binaria = O(n log n) + m × O(log n)

Ejemplo: n = 10.000, m = 100 → lineal ≈ 1.000.000 ops vs. binaria ≈ 134.400 ops (7× más rápida)

Anticipo: búsqueda binaria recursiva

La estructura dividir-y-examinar se presta naturalmente a una formulación recursiva.

```
def busqueda_binaria_rec(datos, clave, izq, der):
    if izq > der:
        return -1              # caso base
    medio = (izq + der) // 2
    if datos[medio] == clave:
        return medio           # caso base
    elif clave < datos[medio]:
        return busqueda_binaria_rec(
            datos, clave, izq, medio-1)
    else:
        return busqueda_binaria_rec(
            datos, clave, medio+1, der)
```

Conexión con la pila de ejecución

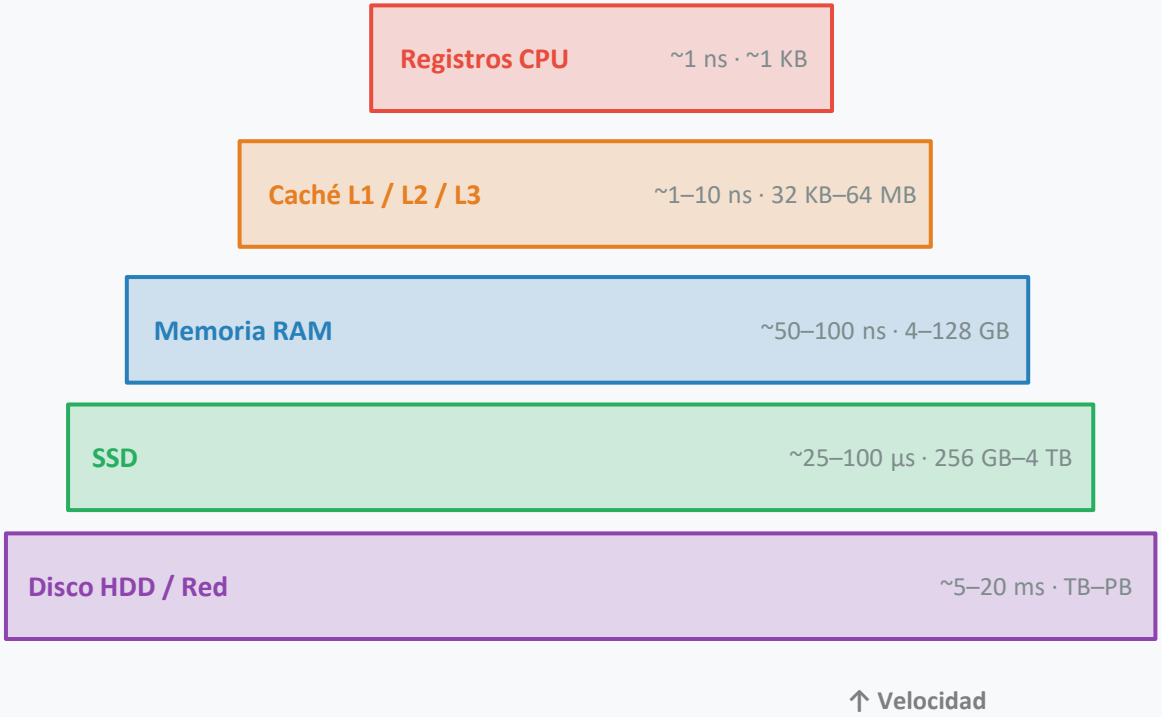
Cada llamada recursiva apila un nuevo frame con izq, der, medio.

Profundidad máxima: $\lceil \log_2(n) \rceil$
Para n = 1.000.000: solo 20 frames.

La recursividad como técnica general se estudia en Semana 8.

Misma complejidad O(log n) que la versión iterativa. La iterativa usa O(1) memoria extra; la recursiva O(log n).

7. Jerarquía de memoria [Holístico I]



Conexión con los algoritmos de búsqueda

Búsqueda lineal: excelente localidad espacial (posiciones contiguas) → aprovecha la caché.

Búsqueda binaria: salta posiciones distantes → más cache misses, pero solo O(log n) accesos.

Para datos en disco: cada acceso ~5 ms.
Binaria con 20 accesos = 100 ms.
Lineal con 10^6 accesos = 83 minutos.

7. Sistema de archivos del SO [Holístico II]

Rutas y directorios

Linux: /home/alumno/tp1.py
Windows: C:\Users\alumno\tp1.py

. = directorio actual
.. = directorio padre
Estructura jerárquica (árbol)

Permisos Unix: -rwxr-xr--

Tipo | Propietario | Grupo | Otros
- | rwx (7) | r-x (5) | r-- (4)

r=4 (lectura), w=2 (escritura), x=1 (ejecución)
Octal 754 → binario 111 101 100
Conexión directa con Semana 2

Operaciones: open → read → write → close

El SO gestiona acceso vía syscalls.
Descriptor de archivo (fd): entero
que identifica la conexión abierta.
fd 0=stdin, 1=stdout, 2=stderr

Caché de buffer (page cache)

RAM como intermediaria del disco.
Lectura: busca primero en caché (µs);
si no, accede al disco (ms).
Escritura: write-back + flush.
El mismo principio de localidad.

Visión de conjunto

⬅️ **Semanas 1–4**	Patrones de búsqueda con condición de corte → base de búsqueda lineal. Funciones y pila → modularización y anticipo recursivo.
📌 **Semana 5 (hoy)**	Búsqueda lineal + optimizaciones, binaria (iterativa y recursiva), interpolación, O grande, invariante de bucle. Jerarquía de memoria y sistema de archivos.
➡️ **Semana 6**	Ordenamiento (selección, inserción, burbuja, merge sort). Complemento natural: ordenar para poder buscar con binaria. $O(n^2)$ vs $O(n \log n)$.
🔮 **Semanas posteriores**	Recursividad (Semana 8), tablas hash con O(1) (Semana 9), proyecto final: elegir y justificar la estrategia de búsqueda.

La elección del algoritmo de búsqueda es un ejemplo paradigmático de decisión de diseño algorítmico.