

EXAMEN HOY · LUNES 14 DE SEPTIEMBRE · A LA HORA DE CLASE**GUÍA MEGA — PRIMER PARCIAL**

Estructuras de Datos · CN122 · LIS-2032 · UDLAP

Dr. José Luis Zechinelli Martini · Curso en lenguaje C

Examen: hoy 13:00–14:15, aula CN 122 (confírmalo) · Vale 15 % de la materia**Alcance: sesiones 1 a 11 (10-ago al 14-sep)****De dónde sale esta guía**

Está armada con las **guías del profesor de las 11 sesiones** que van del semestre, tus notas de clase y tus **Actividades 1 (polinomios) y 2 (arreglos de registros)**. Cada respuesta numérica (direcciones, `sizeof`, trazas de pila, cola y recursión) se comprobó compilando y ejecutando el código en C con gcc de 64 bits. **Supuesto:** el parcial cubre hasta hoy (colas lineales). La cola circular (sesión 12) viene como adelanto por si el profe la pregunta; si no entra, sáltala. Como el examen suele traer **5–8 preguntas objetivas de teoría**, la sección 3 desarrolla todas las definiciones y la sección 4 trae 77 reactivos para practicarlas.

Plan para las horas que quedan (orden estricto)

10:35–10:50 **§2 Chuleta maestra** completa.
 10:50–11:15 **§3 Teoría a fondo:** definiciones exactas para las 5–8 preguntas objetivas.
 11:15–11:35 **§4 Banco de teoría objetiva** sin ver la clave; califícate y relea lo que falles.
 11:35–12:00 **§5–§10:** solo los recuadros verdes (direcciones, `sizeof`, Horner, binaria).
 12:00–12:20 **§11 Pilas:** infijo→postfijo y evaluación postfija a mano.
 12:20–12:35 **§12 Recursión y §13 Colas:** “¿qué imprime?”, Hanói, trazas.
 12:35–12:50 **§14 Simulacro** partes B–E; clave en §15.
 12:50 **§16 Checklist** de 2 minutos y camino a CN 122.

Índice

1	Mapa del parcial: qué entra y qué preguntan	2
2	Chuleta maestra (léela entera)	2
3	Teoría a fondo (para las 5–8 preguntas objetivas)	4
4	Banco de teoría objetiva (77 reactivos)	9
5	Memoria en C (sesión 2)	14
6	Arreglos (sesión 3)	16
7	Matrices y arreglos multidimensionales (sesión 4)	17
8	Polinomios con arreglos (sesión 5 y Actividad 1)	18
9	Registros: struct , relleno, union , enum (sesión 6)	20
10	Arreglos de registros, ordenamiento y búsqueda (sesiones 7 y 8)	21
11	Pilas (sesión 9)	23
12	Recursión y su relación con la pila (sesión 10)	26
13	Colas (sesión 11; circular = adelanto de la 12)	28
14	Simulacro de examen (sin ver la clave)	30
15	Clave del simulacro (resuelta y verificada)	32
16	Checklist de 2 minutos antes de entrar	34

1. Mapa del parcial: qué entra y qué preguntan

S	Fecha	Tema	Pregunta típica	Prioridad
1	10-ago	Almacenar, organizar, recuperar; arreglo vs. lista; escala	¿Por qué no existe “la mejor” estructura?	★
2	12-ago	Memoria en C: stack/heap, apuntadores, <code>malloc/free</code>	Encontrar fuga, colgante, doble <code>free</code>	★★★★
3	17-ago	Arreglos: $a[i] \equiv *(a+i)$, $O(1)$, decaimiento	Calcular direcciones; trampa de <code>sizeof</code>	★★★★
4	19-ago	Matrices: orden por renglones, caché, dinámicas, dispersas	<code>&m[i][j]</code> con fórmula; renglones vs. columnas	★★★★
5	24-ago	Polinomios densos y dispersos; suma, producto, Horner	Horner a mano; suma dispersa	★★
6	26-ago	<code>struct</code> , <code>.</code> y <code>-></code> , relleno, <code>union</code> , <code>enum</code>	<code>sizeof</code> con relleno y reordenar campos	★★★★
7	31-ago	Arreglos de registros: alta, baja, búsqueda; AoS/SoA	Baja con corrimiento vs. con el último	★★
8	02-sep	Ordenar por clave, <code>strcmp</code> , índices, búsqueda binaria	Traza de binaria; ¿conviene ordenar?	★★★★
9	07-sep	Pilas LIFO con arreglo; paréntesis; infijo→postfijo	Conversión y evaluación postfija	★★★★
10	09-sep	Recursión y pila de llamadas; Hanói; recursión de cola	¿Qué imprime?; $T(n) = 2T(n-1) + 1$	★★★★
11	14-sep	Colas FIFO lineales: frente/final, falso lleno	Traza de cola; por qué falla la lineal	★★
A1	—	Actividad 1: polinomios en x, y como matriz	$m[i][j] = \text{coef. de } x^i y^j$	★
A2	—	Actividad 2: <code>&emp[i].address</code>	$\text{BASE} + i(N + M) + N$	★★★★

La idea que el profe repite en todas las sesiones

Elegir estructura es elegir qué operación será rápida y cuál lenta. No hay estructura superior: hay estructuras adecuadas a un patrón de uso. En preguntas abiertas, justifica siempre con *costo de la operación que domina* (acceso, inserción, búsqueda) y con *memoria* (tiempo vs. espacio). Razonar sobre el **costo total** (p. ej. ordenar una vez y buscar muchas) es exactamente lo que se espera.

2. Chuleta maestra (léela entera)

Complejidad de lo visto hasta hoy

Estructura	Acceso por posición	Buscar	Insertar	Eliminar
Arreglo desordenado	$O(1)$	$O(n)$ secuencial	$O(1)$ al final; $O(n)$ en medio	$O(n)$ con corrimiento; $O(1)$ con el último
Arreglo ordenado	$O(1)$	$O(\log n)$ binaria	$O(n)$ (correr)	$O(n)$ (correr)
Pila con arreglo	solo el tope	—	push $O(1)$	pop $O(1)$
Cola lineal	solo el frente	—	enqueue $O(1)$	dequeue $O(1)$, pero falso lleno
Cola circular	solo el frente	—	$O(1)$	$O(1)$, reutiliza todo
Lista ligada (adelanto)	$O(n)$	$O(n)$	$O(1)$ si tienes el nodo	$O(1)$ si tienes el anterior

n	$\log_2 n$	n	$n \log_2 n$	n^2
10^3	≈ 10	10^3	$\approx 10^4$	10^6
10^6	≈ 20	10^6	$\approx 2 \times 10^7$	10^{12} (imposible)

Fórmulas que tienes que escribir sin pensar

- Arreglo: $\&a[i] = \text{BASE} + i \cdot \text{sizeof}(T)$. $a[i] \equiv *(a+i) \equiv i[a]$.
- Matriz T $m[F][C]$ (C , por renglones): $\&m[i][j] = \text{BASE} + (i \cdot C + j) \cdot \text{sizeof}(T)$.
- Por columnas (Fortran/MATLAB): $\text{BASE} + (j \cdot F + i) \cdot \text{sizeof}(T)$.
- 3D T $t[D1][D2][D3]$: $\text{BASE} + ((i \cdot D2 + j) \cdot D3 + k) \cdot \text{sizeof}(T)$.
- Arreglo de registros: $\&v[i].\text{campo} = \text{BASE} + i \cdot \text{sizeof}(\text{struct}) + \text{offsetof}(\text{struct}, \text{campo})$.
- Actividad 2 (`char name[N]; char address[M];`): $\&emp[i].\text{address} = \text{BASE} + i(N + M) + N$.
- Relleno: cada campo empieza en múltiplo de su alineación; el **sizeof** total es múltiplo de la *mayor* alineación.
- Horner: $b_n = a_n$, $b_k = a_k + x \cdot b_{k+1}$, $p(x) = b_0$. $O(n)$ contra $O(n^2)$.
- Binaria: $\text{medio} = \text{ini} + (\text{fin} - \text{ini}) / 2$; a lo más $\lfloor \log_2 n \rfloor + 1$ vueltas.
- Hanói: $T(n) = 2T(n-1) + 1$, $T(1) = 1 \Rightarrow T(n) = 2^n - 1$.
- Fibonacci ingenuo: $C(n) = C(n-1) + C(n-2) + 1$ llamadas $= 2F(n+1) - 1$.
- Cola circular: $\text{final} = (\text{final} + 1) \% \text{CAP}$, $\text{frente} = (\text{frente} + 1) \% \text{CAP}$.

Tamaños en 64 bits (gcc, Windows y Linux)

`char` 1 · `short` 2 · `int` 4 · `float` 4 · `double` 8 · cualquier apuntador 8. La alineación de cada tipo básico es igual a su tamaño; la de un arreglo es la de su elemento (`char name[40]` alinea a 1). Nunca supongas: en el código usa **sizeof**.

3. Teoría a fondo (para las 5–8 preguntas objetivas)

Cómo usar esta sección

Las preguntas objetivas (opción múltiple, verdadero/falso, relacionar columnas) premian la **definición exacta** y la **palabra clave** correcta. Cada concepto trae qué es, por qué importa y, cuando aplica, el distractor con el que suelen confundirte. Léela una vez con atención y luego resuelve el banco de la sección 4 sin mirar.

3.1. Conceptos generales (sesión 1)

Estructura de datos. Forma de organizar datos en memoria *junto con las operaciones* que se permiten sobre ellos, para usarlos eficientemente. Toda estructura resuelve tres operaciones fundamentales: **almacenar** (insertar), **organizar** (mantener un orden o relación) y **recuperar** (encontrar un dato dado un criterio). Ninguna las resuelve todas igual de bien.

Tipo de dato abstracto (TDA). Especificación de un conjunto de valores y sus operaciones que dice *qué* se hace, no *cómo*. La pila es un TDA; “pila con arreglo” es una implementación. Una misma interfaz admite varias implementaciones.

Estática vs. dinámica. Estática: su capacidad se fija al crearla y no cambia (arreglos; pilas y colas implementadas con arreglo). Dinámica: crece y decrece en ejecución pidiendo y liberando memoria (listas ligadas, árboles, grafos). El temario del curso clasifica: *estáticas*: arreglos, pilas y colas; *dinámicas*: listas ligadas, grafos y árboles. **Ojo**: “estática” aquí no es la palabra reservada **static** de C.

Lineal vs. no lineal. Lineal: cada elemento tiene a lo más un antecesor y un sucesor (arreglo, pila, cola, lista). No lineal: un elemento puede tener varios sucesores (árbol, grafo).

Homogénea vs. heterogénea. Homogénea: todos los elementos son del mismo tipo (arreglo). Heterogénea: campos de tipos distintos (registro).

Eficiencia. Se mide en **tiempo** (número de operaciones en función del tamaño n) y en **espacio** (memoria adicional). Casi siempre se intercambia uno por el otro.

Notación O grande. $f(n) = O(g(n))$ si existen constantes $c > 0$ y n_0 tales que $f(n) \leq c g(n)$ para todo $n \geq n_0$. Es una **cota superior** del crecimiento: se ignoran constantes y términos de menor orden ($3n^2 + 5n + 7 = O(n^2)$). Ω es cota inferior y Θ cota ajustada (superior e inferior a la vez).

Mejor, promedio y peor caso. Búsqueda secuencial: mejor caso 1 comparación (está al inicio), peor caso n (está al final o no está), promedio $\approx n/2$, es decir $O(n)$. Si la pregunta no aclara, se refiere al **peor caso**.

Jerarquía de órdenes. $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$. Con n pequeño casi todo sirve; con $n = 10^8$ la diferencia entre $O(n)$ y $O(\log n)$ separa lo imposible de lo trivial.

3.2. Memoria y apuntadores (sesión 2)

Variable. Tiene nombre, tipo, dirección y valor. El **alcance** (dónde se puede usar el nombre) no es lo mismo que el **tiempo de vida** (cuánto tiempo existe su memoria).

Segmentos de un programa. **Código** (instrucciones); **datos** estáticos y globales (viven todo el programa); **montículo** o heap (memoria pedida con **malloc**, vive hasta **free**); **pila** o stack (locales y marcos de llamada, se libera sola al salir de la función).

Automática, estática y dinámica. Automática: la local normal, en la pila, muere al salir de la función. **static local**: vive en el segmento de datos; su alcance es local pero **conserva su valor entre llamadas**. Dinámica: en el heap, sin nombre, se usa solo por apuntador.

Marco de pila (registro de activación). Bloque que se apila en cada llamada con los parámetros, las variables locales y la dirección de retorno; se desapila al regresar.

Apuntador. Variable cuyo valor es una **dirección de memoria**. Su tipo indica cómo interpretar lo apuntado y cuánto avanza la aritmética. $\&x$ da la dirección de x ; $*p$ (indirección o desreferencia) da el contenido apuntado; en la declaración `int *p` el asterisco solo indica el tipo.

NULL. Indica que el apuntador no apunta a nada válido. Desreferenciarlo es comportamiento indefinido (normalmente el programa truena).

void *. Apuntador genérico: guarda cualquier dirección, pero no se desreferencia sin convertirlo. `malloc`, `calloc` y `realloc` regresan `void *`.

Apuntador a apuntador. `int **pp` guarda la dirección de un `int *`. Se usa en matrices dinámicas y para que una función modifique un apuntador del llamador.

Paso de parámetros en C. Siempre por valor: la función recibe copias. El “paso por referencia” se simula pasando la dirección: `swap(&a, &b)`.

sizeof. Operador (no función) que da el tamaño en bytes de un tipo o expresión; se resuelve en compilación (salvo arreglos de longitud variable). Su resultado es `size_t`.

malloc, calloc, realloc, free. `malloc(n)`: n bytes **sin inicializar** (basura). `calloc(k, tam)`: $k \cdot \text{tam}$ bytes **en cero**. `realloc(p, n)`: cambia el tamaño conservando el contenido; **puede mover el bloque** a otra dirección y, si falla, regresa `NULL` dejando intacto el bloque original. `free(p)` libera; `free(NULL)` no hace nada. Las tres primeras pueden fallar: verifica `NULL`.

Errores de memoria. Fuga: perder la última referencia a un bloque sin liberarlo. **Apuntador colgante:** usar memoria ya liberada, o la dirección de una local que ya murió. **Doble liberación:** `free` dos veces sobre la misma dirección. **Desbordamiento de búfer:** escribir fuera de los límites. **Apuntador salvaje:** usar un apuntador sin inicializar. El compilador no los detecta y ninguno falla de inmediato; `valgrind` los encuentra al ejecutar.

Fragmentación. Hay memoria libre suficiente en total, pero no en un solo bloque contiguo: un `malloc` grande puede fallar.

3.3. Arreglos y matrices (sesiones 3 y 4)

Arreglo. Colección **finita, homogénea y ordenada** de elementos en **memoria contigua**, accesibles por índice. En C los índices van de 0 a $n - 1$ y el arreglo **no guarda su tamaño**.

Dirección base. La del primer elemento. En expresiones, el nombre del arreglo equivale a `&a[0]`.

Acceso aleatorio (directo). Llegar a cualquier posición cuesta lo mismo, $O(1)$, porque la dirección se *calcula*: $\text{BASE} + i \cdot \text{tamaño}$. Lo contrario es el acceso *secuencial*: hay que recorrer (lista ligada).

Límites. C **no verifica** índices: `a[n]` compila y corrompe memoria en silencio (comportamiento indefinido).

Decaimiento. Al usar un arreglo en una expresión o pasarlo a una función se convierte en apuntador a su primer elemento y se pierde el tamaño. Excepciones: `sizeof(a)` y `&a` en el ámbito donde se declaró.

Aritmética de apuntadores. `p + k` avanza $k \cdot \text{sizeof}(*p)$ bytes. Restar dos apuntadores al mismo arreglo da el **número de elementos**, no de bytes. $a[i] \equiv *(a + i) \equiv i[a]$.

Cadena. Arreglo de `char` terminado en `'\0'`. `strlen` no cuenta el terminador: copiar una cadena exige `strlen(s) + 1` bytes.

Costos del arreglo. Acceso $O(1)$; insertar o borrar en medio $O(n)$ por el corrimiento; tamaño fijo; necesita un bloque contiguo libre.

Arreglo multidimensional. En C es un arreglo de arreglos guardado **por renglones** (row-major): la última dimensión es la que varía más rápido. Fortran y MATLAB guardan por columnas (column-major).

Localidad de referencia. *Espacial*: se accede a datos cercanos en memoria. *Temporal*: se reusan datos recientes. La caché trae bloques (líneas de caché, típicamente de 64 bytes), así que recorrer memoria contigua produce más aciertos. Por eso el orden de los ciclos cambia el tiempo real **sin cambiar la complejidad**.

Matriz dinámica. Con `int **`: renglones independientes, no contiguos, que pueden medir distinto (matriz irregular); requiere $F + 1$ `malloc` y $F + 1$ `free`, renglones primero. Con un bloque `int *`: contigua, un `malloc` y un `free`, índice manual `i*C + j`.

Matriz dispersa. La mayoría de sus elementos son cero; se guardan solo los no nulos como tercias (renglón, columna, valor). Ahorra espacio y el acceso deja de ser $O(1)$.

3.4. Polinomios (sesión 5)

Representación densa. Arreglo indexado por exponente, de tamaño grado $+1$. Acceso $O(1)$ al coeficiente de cualquier grado; desperdicia memoria cuando hay huecos.

Representación dispersa. Solo pares (coeficiente, exponente) no nulos, ordenados por exponente. Memoria proporcional al número de términos; encontrar un exponente exige recorrer.

Costos. Suma densa $O(\max(n, m))$; suma dispersa $O(n+m)$ por mezcla; producto $O(n \cdot m)$; evaluación directa $O(n^2)$; Horner $O(n)$.

Lección. La representación se elige por la **densidad esperada**. Y reorganizar un cálculo (Horner) mejora la complejidad sin cambiar el resultado.

3.5. Registros (sesión 6)

Registro (struct). Estructura **heterogénea**: agrupa campos (miembros) de tipos distintos que describen a una misma entidad.

typedef. Crea un **alias** de un tipo existente; no crea un tipo nuevo.

Acceso a campos. Punto con la variable (`a.campo`); flecha con apuntador (`p->campo`), que equivale a `(*p).campo`. El punto tiene mayor precedencia que el asterisco, por eso `*p.campo` es un error.

Alineación y relleno. El procesador exige que ciertos tipos empiecen en direcciones múltiplo de su alineación, así que el compilador inserta **bytes de relleno** (padding) entre campos y al final. `offsetof(tipo, campo)` da el desplazamiento. El compilador **respeto el orden de declaración** (no reordena): acomodar de mayor a menor lo haces tú.

Asignación y comparación. `r1 = r2` copia todos los campos (si hay apuntadores copia las direcciones, no lo apuntado). Los registros **no se comparan con ==**: se compara campo por campo.

Paso a funciones. Por valor se copia el registro completo. Por apuntador viajan 8 bytes; con `const` se documenta que no se modifica.

union. Todos los miembros comparten la misma dirección; su tamaño es el del **miembro mayor** (ajustado a la alineación) y solo es válido el último miembro escrito.

enum. Constantes enteras con nombre; si no se asignan valores empiezan en 0 y aumentan de uno en uno. Junto con una `union` forma un **registro variante**: el `enum` indica qué miembro es válido.

Registro autorreferenciado. Contiene un apuntador a su propio tipo (`struct Nodo *sig`); por eso la estructura necesita nombre de etiqueta. Es la base de las listas ligadas.

3.6. Arreglos de registros, búsqueda y ordenamiento (sesiones 7 y 8)

Tabla. Arreglo de registros: renglones homogéneos con columnas heterogéneas, como una tabla de base de datos. El **contador** n (elementos lógicos) es tan importante como la capacidad física.

Clave. Campo que identifica un registro o por el que se busca (matrícula). Clave primaria: única para cada registro.

Operaciones. Alta (inserción), baja (eliminación), consulta (búsqueda), modificación y listado.

Baja física vs. lógica. Física: se quita el dato moviendo elementos, con corrimiento ($O(n)$, conserva el orden) o sustituyendo con el último ($O(1)$, no conserva el orden). Lógica: se marca **activo** = 0 sin mover datos; las búsquedas deben ignorar los inactivos y hay que **compactar** cada tanto.

AoS vs. SoA. Arreglo de estructuras (registros completos consecutivos) vs. estructura de arreglos (un arreglo por campo). SoA aprovecha mejor la caché cuando se recorre repetidamente un solo campo.

Búsqueda secuencial (lineal). Revisa uno por uno; **no requiere orden**; $O(n)$ en el peor caso; sirve en cualquier estructura que se pueda recorrer.

Búsqueda binaria. Compara con el elemento medio y descarta la mitad. Requiere **datos ordenados por la clave** y **acceso aleatorio** (por eso no se aplica directamente a una lista ligada). A lo más $\lfloor \log_2 n \rfloor + 1$ comparaciones: unas 20 para un millón de datos. El medio se calcula $\text{ini} + (\text{fin} - \text{ini}) / 2$ para evitar desbordamiento.

Ordenar por clave. La comparación usa un campo y el intercambio mueve el registro completo. Las cadenas se comparan con `strcmp` (orden lexicográfico); `==` compara direcciones.

Ordenamiento indirecto. Se ordenan índices o apuntadores dejando quietos los datos: evita copiar registros grandes y permite **varios órdenes simultáneos** sin duplicar información (la idea de un índice de base de datos).

Estable e in situ. **Estable:** conserva el orden relativo de los elementos con clave igual (inserción sí lo es; selección con intercambio no). **In situ:** usa memoria adicional $O(1)$.

Costo total. Ordenar cuesta $O(n \log n)$ una vez y se amortiza si habrá muchas búsquedas. Para una sola búsqueda la secuencial es más barata.

3.7. Pilas (sesión 9)

Pila. TDA lineal **LIFO** (último en entrar, primero en salir): inserciones y eliminaciones solo en un extremo, el **tope**.

Operaciones. `push` (apilar), `pop` (desapilar), `peek` o `top` (consultar sin sacar), `isEmpty`, `isFull`. Todas $O(1)$.

Implementación con arreglo. `tope` empieza en -1 (vacía); llena cuando `tope == MAX - 1`. **Overflow** (desbordamiento): `push` en pila llena. **Underflow** (subdesbordamiento): `pop` en pila vacía. En C nadie los detecta por ti.

Aplicaciones. Pila de llamadas del programa, deshacer, botón “atrás” del navegador, paréntesis balanceados, conversión y evaluación de expresiones, recorrido en profundidad (DFS), backtracking. Una pila **invierte** el orden de llegada.

Notaciones. **Infija** ($A + B$): operador en medio; necesita paréntesis y precedencia. **Prefija** o **polaca** ($+ A B$): operador antes. **Postfija** o **polaca inversa** ($A B +$): operador después; **no necesita paréntesis ni reglas de precedencia** y se evalúa de izquierda a derecha con una pila en $O(n)$.

Qué guarda la pila. Al convertir infijo $\rightarrow$ postfijo guarda **operadores y paréntesis de apertura**. Al evaluar postfijo guarda **operandos y resultados parciales**. El orden relativo de los operandos **no cambia** entre infija y postfija: solo se mueven los operadores.

3.8. Recursión (sesión 10)

Recursión. Técnica en la que una función se llama a sí misma sobre una instancia más pequeña del problema.

Caso base y paso recursivo. Caso base: se resuelve sin llamada recursiva y detiene la recursión. Paso recursivo: reduce el problema acercándolo al caso base. Si falta cualquiera de los dos, la recursión no termina.

Directa vs. indirecta. Directa: f llama a f . Indirecta o mutua: f llama a g y g llama a f .

Lineal vs. múltiple. Lineal: una llamada por invocación (factorial). Múltiple o en árbol: varias llamadas (Fibonacci, Hanói); suele crecer exponencialmente.

Recursión de cola. La llamada recursiva es la **última operación** y no queda trabajo pendiente; el compilador puede convertirla en ciclo (el estándar de C no lo garantiza). `return n * factorial(n - 1);` **no** es de cola: la multiplicación queda pendiente.

Memoria. Cada llamada activa ocupa un marco en la pila del sistema; la memoria es proporcional a la **profundidad** máxima. Factorial recursivo usa $O(n)$ de pila; el iterativo, $O(1)$.

Desbordamiento de pila. Ocurre sin caso base o con profundidad excesiva, aun con un algoritmo correcto: la pila del sistema tiene tamaño limitado.

Equivalencia. Todo algoritmo recursivo puede escribirse de forma iterativa (con una pila explícita si hace falta). La recursión es la pila del sistema haciendo ese trabajo.

Relación de recurrencia. Ecuación que expresa el costo $T(n)$ en términos de valores menores; se resuelve desenrollando o por sustitución. Hanói: $T(n) = 2T(n - 1) + 1 = 2^n - 1$.

Cuándo pasar a iterativo. Profundidad grande (riesgo de desbordamiento), recálculo exponencial (Fibonacci ingenuo repite subproblemas) o cuando importa más el desempeño que la claridad.

3.9. Colas (sesión 11)

Cola. TDA lineal **FIFO** (primero en entrar, primero en salir). Opera en **dos extremos**: se inserta por el **final** y se extrae por el **frente**. Conserva el orden de llegada.

Operaciones. `enqueue` (encolar), `dequeue` (desencolar), `front` (consultar el primero), `isEmpty`, `isFull`.

Cola lineal con arreglo. `frente` y `final` solo avanzan. **Falso lleno:** `final` llega al último índice y la cola se reporta llena aunque haya espacio libre al inicio. Remedio ingenuo: correr los elementos en cada `dequeue`, $O(n)$.

Cola circular (búfer circular). Los índices avanzan con aritmética modular, $(i + 1) \% CAP$: todas las operaciones son $O(1)$ y se reutiliza todo el espacio. Como con solo `frente == final` vacía y llena se ven iguales, se lleva un **contador** o se **sacrifica una celda**.

Variantes (cultura general). Bicola o *deque*: inserta y extrae por ambos extremos. Cola de prioridad: sale el elemento más urgente, no el más antiguo (con montículo, $O(\log n)$).

Aplicaciones. Planificación de procesos del sistema operativo, colas de impresión y de peticiones a un servidor, búfer entre productor y consumidor (UART), recorrido en anchura (BFS).

Distractores clásicos en opción múltiple

- Pila = LIFO, cola = FIFO. Overflow es en `push`; underflow es en `pop`.
- Binaria exige *dos* cosas: datos ordenados y acceso aleatorio. Secuencial no exige orden.
- `sizeof(struct) ≥` suma de campos (relleno); `sizeof(union) =` miembro mayor alineado.

- `malloc` deja basura; `calloc` pone ceros; `realloc` puede mover el bloque.
- `p->c` $\equiv$ `(*p).c`; `*p.c` no es lo mismo. Cadenas con `strcmp`, no `==`.
- C guarda matrices por renglones; Fortran y MATLAB por columnas.
- Recorrer por columnas no cambia la complejidad, cambia el tiempo (caché).
- Horner es $O(n)$, no $O(\log n)$. Hanói es $2^n - 1$, no 2^n .
- “Recursión de cola” no tiene nada que ver con la estructura cola.
- Postfija (`A B +`) $\neq$ prefija (`+ A B`). En postfija el orden de los operandos se conserva.
- Estructura “estática” (tamaño fijo) $\neq$ variable `static` de C (vida de todo el programa).
- C pasa todo por valor; “por referencia” es pasar una dirección.

4. Banco de teoría objetiva (77 reactivos)

Instrucciones

Una sola respuesta por reactivo; unos 20 segundos cada uno. No mires la clave (sección 15) hasta terminar. Cada error: vuelve a la definición en la sección 3.

Opción múltiple

Conceptos generales

T1. Una estructura de datos se define mejor como:

- | | |
|---|--|
| (a) Un lenguaje para almacenar información en disco | (b) Una forma de organizar datos en memoria junto con las operaciones permitidas sobre ellos |
| (c) Un algoritmo de ordenamiento | (d) Un tipo de dato primitivo de C |

T2. Un tipo de dato abstracto (TDA) especifica:

- | | |
|--|--|
| (a) La implementación con arreglos | (b) El tamaño en bytes de cada elemento |
| (c) La dirección de memoria de los datos | (d) Los valores y las operaciones, sin fijar cómo se implementan |

T3. ¿Cuál afirmación sobre la elección de estructura es correcta?

- | | |
|--|---|
| (a) Elegir estructura es elegir qué operación será rápida y cuál lenta | (b) El arreglo supera a la lista en todas las operaciones |
| (c) La lista supera al arreglo en todas las operaciones | (d) Con muchos datos cualquier estructura sirve igual |

T4. Órdenes de menor a mayor crecimiento:

- | | |
|---|---|
| (a) $O(n)$, $O(\log n)$, $O(n \log n)$, $O(n^2)$ | (b) $O(1)$, $O(n^2)$, $O(n)$, $O(2^n)$ |
| (c) $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$ | (d) $O(n \log n)$, $O(n)$, $O(\log n)$, $O(1)$ |

T5. $f(n) = 3n^2 + 5n + 7$ es:

- | | |
|--------------|-----------------|
| (a) $O(1)$ | (b) $O(n)$ |
| (c) $O(n^2)$ | (d) $O(\log n)$ |

T6. Según el temario del curso, son estructuras estáticas:

- | | |
|-----------------------------|------------------------------|
| (a) Arreglos, pilas y colas | (b) Listas, árboles y grafos |
| (c) Árboles y grafos | (d) Listas ligadas y colas |

T7. Una estructura es no lineal cuando:

- (a) Sus elementos son de tipos distintos
- (b) Su tamaño cambia en ejecución
- (c) Se guarda en el heap
- (d) Un elemento puede tener más de un sucesor

Memoria y apuntadores

T8. Las variables locales (automáticas) de una función se almacenan en:

- (a) El montículo (heap)
- (b) El segmento de código
- (c) La pila (stack)
- (d) El segmento de datos estáticos

T9. ¿Qué función reserva memoria e inicializa todos sus bytes en cero?

- (a) `calloc`
- (b) `malloc`
- (c) `realloc`
- (d) `free`

T10. Usar un apuntador después de liberar su memoria es:

- (a) Fuga de memoria
- (b) Doble liberación
- (c) Desbordamiento de pila
- (d) Apuntador colgante

T11. Perder la única referencia a un bloque reservado sin liberarlo produce:

- (a) Apuntador colgante
- (b) Fuga de memoria
- (c) Underflow
- (d) Error de compilación

T12. Si `int *p = &x;`, la expresión `*p` representa:

- (a) El valor almacenado en `x`
- (b) La dirección de `x`
- (c) La dirección de `p`
- (d) El tamaño de `x`

T13. En C, los argumentos de una función se pasan:

- (a) Siempre por referencia
- (b) Por referencia si son enteros
- (c) Siempre por valor; la referencia se simula pasando direcciones
- (d) Por valor solo si son apuntadores

T14. Si `realloc` no puede agrandar el bloque en su lugar:

- (a) Siempre regresa `NULL`
- (b) Borra el contenido anterior
- (c) Libera todo el heap
- (d) Puede mover el bloque a otra dirección copiando el contenido

Arreglos y matrices

T15. El acceso a `a[i]` es $O(1)$ porque:

- (a) El arreglo está ordenado
- (b) La dirección se calcula como `BASE + i * sizeof(tipo)`, sin recorrer
- (c) C verifica los límites
- (d) Los elementos viven en el heap

T16. ¿Qué hace C si accedes a `a[i]` con `i` fuera de rango?

- (a) Lanza una excepción
- (b) Ajusta `i` al último índice
- (c) Nada: es comportamiento indefinido
- (d) Siempre detiene el programa

T17. Dentro de `void f(int a[])`, el parámetro `a` es:

- (a) Un apuntador al primer elemento
- (b) Una copia del arreglo
- (c) Un arreglo que conoce su tamaño
- (d) Una referencia constante

T18. Para `double *p`, la expresión `p + 1` avanza:

- (a) 1 byte
- (b) 4 bytes
- (c) Depende del valor apuntado
- (d) `sizeof(double)` bytes

T19. Insertar un elemento en la posición 0 de un arreglo con n elementos cuesta:

- (a) $O(1)$
- (b) $O(n)$
- (c) $O(\log n)$
- (d) $O(n^2)$

T20. C almacena las matrices:

- (a) Por renglones
- (b) Por columnas
- (c) Como listas ligadas
- (d) En el orden que decida el sistema operativo

T21. Recorrer una matriz por renglones es más rápido que por columnas porque:

- (a) Tiene menor complejidad asintótica
- (b) Hace menos operaciones
- (c) Aprovecha la localidad de referencia y la caché
- (d) El compilador elimina un ciclo

T22. En un parámetro que recibe una matriz estática se puede omitir:

- (a) Ninguna dimensión
- (b) La segunda dimensión
- (c) Ambas dimensiones
- (d) La primera dimensión

T23. Representar una matriz dispersa con tercias:

- (a) Ahorra espacio y hace el acceso más lento
- (b) Ahorra espacio y acelera el acceso
- (c) Siempre gasta más espacio
- (d) Solo sirve para matrices cuadradas

Polinomios

T24. La regla de Horner evalúa un polinomio de grado n en:

- (a) $O(1)$
- (b) $O(n)$
- (c) $O(\log n)$
- (d) $O(n^2)$

T25. Para representar $x^{1000} + 1$ conviene:

- (a) La representación densa
- (b) Una pila
- (c) Una matriz de 1001×1001
- (d) La representación dispersa

T26. El producto de dos polinomios de n y m términos cuesta:

- (a) $O(n + m)$
- (b) $O(\max(n, m))$
- (c) $O(n \cdot m)$
- (d) $O(\log(n \cdot m))$

Registros

T27. `sizeof` de un `struct` puede ser mayor que la suma de sus campos debido a:

- (a) El relleno por alineación
- (b) El `typedef`
- (c) El operador flecha
- (d) Los `enum`

T28. `p->campo` equivale a:

- (a) `*p.campo`
- (b) `(*p).campo`
- (c) `&p.campo`
- (d) `p.campo`

T29. El tamaño de una `union` es:

- (a) La suma de sus miembros
- (b) Siempre 8 bytes
- (c) El de su primer miembro
- (d) El de su miembro más grande, ajustado a la alineación

T30. En `enum Color {ROJO, VERDE, AZUL};` el valor de `AZUL` es:

- (a) 0
- (b) 1
- (c) 2
- (d) 3

T31. Pasar un registro grande como `const Alumno *a` sirve para:

- (a) Evitar la copia y documentar que no se modificará
- (b) Copiar todo el registro
- (c) Obligar a modificarlo
- (d) Reducir el relleno

Arreglos de registros, búsqueda y ordenamiento

T32. Dar de baja reemplazando con el último elemento:

- | | |
|--|--|
| (a) Conserva el orden y cuesta $O(n)$ | (b) Conserva el orden y cuesta $O(1)$ |
| (c) No conserva el orden y cuesta $O(n)$ | (d) No conserva el orden y cuesta $O(1)$ |

T33. El borrado lógico consiste en:

- | | |
|---|--|
| (a) Liberar la memoria del registro con <code>free</code> | (b) Marcar el registro como inactivo sin mover datos |
| (c) Recorrer los elementos hacia la izquierda | (d) Ordenar el arreglo |

T34. La búsqueda binaria requiere:

- | | |
|---|------------------------|
| (a) Que no haya elementos repetidos | (b) Datos desordenados |
| (c) Datos ordenados por la clave y acceso aleatorio | (d) Una lista ligada |

T35. Para comparar dos cadenas en C se usa:

- | | |
|-------------------------|-------------------------|
| (a) <code>strcmp</code> | (b) <code>==</code> |
| (c) <code>sizeof</code> | (d) <code>strlen</code> |

T36. El ordenamiento indirecto:

- | | |
|-----------------------------------|---|
| (a) Mueve los registros completos | (b) Solo funciona con enteros |
| (c) Siempre es $O(n)$ | (d) Ordena índices o apuntadores y deja quietos los datos |

T37. Máximo de comparaciones de la búsqueda binaria con 1 000 000 de elementos, aproximadamente:

- | | |
|-----------|-------------|
| (a) 10 | (b) 20 |
| (c) 1 000 | (d) 500 000 |

Pilas

T38. Una pila sigue la política:

- | | |
|-------------------|---------------|
| (a) LIFO | (b) FIFO |
| (c) Por prioridad | (d) Aleatoria |

T39. Hacer `pop` en una pila vacía produce:

- | | |
|---------------|---------------------|
| (a) Overflow | (b) Falso lleno |
| (c) Underflow | (d) Fuga de memoria |

T40. La notación postfija (polaca inversa) tiene la ventaja de:

- | | |
|------------------------------|--|
| (a) Necesitar más paréntesis | (b) Evaluarse de derecha a izquierda con una cola |
| (c) Servir solo para sumas | (d) No requerir paréntesis ni reglas de precedencia para evaluarse |

T41. Al convertir de infijo a postfijo, la pila almacena:

- | | |
|---------------------------|---|
| (a) Los operandos | (b) Los operadores y los paréntesis de apertura |
| (c) El resultado numérico | (d) Nada |

T42. ¿Cuál de estas aplicaciones **no** usa una pila?

- | | |
|--------------------------------------|---------------------------|
| (a) Cola de impresión | (b) Deshacer en un editor |
| (c) Verificar paréntesis balanceados | (d) Llamadas a funciones |

T43. Al pasar de infijo a postfijo, el orden relativo de los operandos:

- | | |
|-----------------|-------------------------------|
| (a) Se invierte | (b) Se ordena alfabéticamente |
| (c) Se conserva | (d) Depende de la precedencia |

Recursión

T44. Una función recursiva correcta necesita:

- (a) Solo el paso recursivo
- (b) Una variable global
- (c) Un ciclo `while`
- (d) Un caso base y un paso que se acerque a él

T45. Cada llamada recursiva activa ocupa:

- (a) Un bloque en el heap
- (b) Un marco (registro de activación) en la pila
- (c) Espacio en el segmento de código
- (d) Nada de memoria

T46. Movimientos para resolver las Torres de Hanói con n discos:

- (a) $2^n - 1$
- (b) n^2
- (c) $2n$
- (d) $n!$

T47. La recursión de cola se caracteriza porque:

- (a) Usa una cola FIFO
- (b) Tiene dos llamadas recursivas
- (c) La llamada recursiva es la última operación
- (d) No tiene caso base

T48. `return n * factorial(n - 1);` es un ejemplo de:

- (a) Recursión de cola
- (b) Recursión indirecta
- (c) Iteración
- (d) Recursión que no es de cola: queda pendiente la multiplicación

T49. Fibonacci recursivo ingenuo es ineficiente porque:

- (a) Recalcula muchas veces los mismos subproblemas
- (b) Usa variables globales
- (c) No tiene caso base
- (d) Es recursión de cola

T50. Si `f` llama a `g` y `g` llama a `f`, la recursión es:

- (a) Directa
- (b) Indirecta (mutua)
- (c) De cola
- (d) Lineal

Colas

T51. Una cola sigue la política:

- (a) LIFO
- (b) Por tamaño
- (c) FIFO
- (d) Por prioridad

T52. En una cola se inserta por _____ y se extrae por _____:

- (a) el final; el frente
- (b) el frente; el final
- (c) el tope; el tope
- (d) el medio; el final

T53. El “falso lleno” de la cola lineal con arreglo ocurre cuando:

- (a) El arreglo está totalmente ocupado
- (b) `frente` vale -1
- (c) Se usa aritmética modular
- (d) `final` llega al último índice aunque haya espacio libre al inicio

T54. La cola circular resuelve el falso lleno usando:

- (a) Corrimiento de elementos en cada `dequeue`
- (b) Aritmética modular en los índices
- (c) Un arreglo más grande
- (d) Una pila auxiliar

T55. Sin contador ni celda sacrificada, la condición `frente == final` en una cola circular:

- (a) Indica siempre cola llena
- (b) Indica siempre cola vacía
- (c) No distingue por sí sola vacía de llena
- (d) Nunca ocurre

T56. Para atender peticiones en orden de llegada usarías:

- (a) Una pila (b) Un arreglo ordenado
(c) Un registro (d) Una cola

T57. ¿Qué estructura invierte el orden en que llegan los datos?

- (a) Pila (b) Cola
(c) Cola circular (d) Arreglo con búsqueda binaria

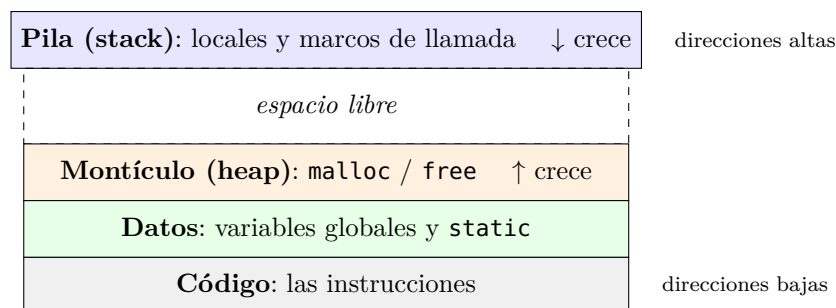
Verdadero o falso

- V1.** `malloc` inicializa en cero la memoria que reserva.
V2. Una variable `static` declarada dentro de una función conserva su valor entre llamadas.
V3. `free(NULL)` provoca un error.
V4. En una pila con arreglo, `tope == MAX - 1` significa pila llena.
V5. El compilador de C reordena los campos de un `struct` para reducir el relleno.
V6. Todo algoritmo recursivo puede reescribirse de forma iterativa.
V7. Dos registros se pueden comparar directamente con `==`.
V8. Una cola conserva el orden de llegada de los datos.
V9. La búsqueda secuencial requiere que el arreglo esté ordenado.
V10. El ordenamiento por selección con intercambio es estable.

Relaciona columnas

Concepto	Descripción
1 Fuga de memoria	A Marcar un registro como inactivo sin mover datos
2 Apuntador colgante	B Evaluación de un polinomio en $O(n)$
3 Relleno (padding)	C <code>pop</code> sobre una pila vacía
4 Orden por renglones	D Bytes que inserta el compilador para respetar la alineación
5 Regla de Horner	E Instancia que se resuelve sin llamada recursiva
6 Borrado lógico	F Usar memoria que ya fue liberada
7 Underflow	G Notación que no necesita paréntesis
8 Caso base	H Perder la referencia a memoria sin liberarla
9 Falso lleno	I <code>final</code> llega al fin del arreglo con espacio libre al inicio
10 Notación postfija	J Forma en que C guarda las matrices en memoria

5. Memoria en C (sesión 2)



- **Pila:** se libera sola al salir de la función. **Montículo:** solo si tú llamas `free`.

- `int *p`; guarda una dirección. `&x` da la dirección; `*p` da el contenido; `NULL` = no apunta a nada válido.
- `malloc(n)` reserva n bytes (o regresa `NULL`); `calloc(n, tam)` reserva e inicializa en cero; `realloc` cambia el tamaño; `free` libera. **Cada `malloc` lleva exactamente un `free`.**

Patrón correcto: reservar, llenar, redimensionar, liberar (el ejercicio de la sesión)

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int n = 5;
    int *a = malloc(n * sizeof *a);           /* pedir memoria */
    if (a == NULL) return 1;                  /* SIEMPRE verificar */
    for (int i = 0; i < n; i++) a[i] = i * i;

    int *tmp = realloc(a, 2 * n * sizeof *a); /* crecer al doble */
    if (tmp == NULL) { free(a); return 1; }    /* si falla, a sigue valido */
    a = tmp;
    for (int i = n; i < 2 * n; i++) a[i] = i * i;
    n *= 2;

    free(a);                                  /* un malloc -> un free */
    a = NULL;                                 /* evita apuntador colgante */
    return 0;
}
```

Por qué `tmp`: si escribes `a = realloc(a, ...)` y falla, `a` queda en `NULL` y pierdes el bloque original: fuga.

Los tres errores que “van a perseguirte todo el semestre”

1. **Fuga de memoria**: perder la última referencia a un bloque sin liberarlo.
2. **Apuntador colgante**: usar un apuntador después de `free`, o regresar la dirección de una variable local.
3. **Doble liberación**: `free` dos veces sobre la misma dirección (también vía un alias).

Ninguno truena de inmediato: el programa parece funcionar y falla después, en otro lado. Por eso se usa `valgrind`.

Ejercicio resuelto 3.1 — Encuentra los errores

```
int *crear(int n) {
    int v[n];                                /* (1) arreglo LOCAL: vive en la pila */
    for (int i = 0; i <= n; i++)             /* (2) i <= n escribe v[n]: fuera de rango */
        v[i] = i;
    return v;                                /* (1) regresa direccion que muere: colgante */
}

int main(void) {
    int *a = malloc(10 * sizeof(int));        /* (3) no se verifica NULL */
    a = malloc(20 * sizeof(int));             /* (4) FUGA: se perdio el primer bloque */
    free(a);
    a[0] = 1;                                /* (5) uso despues de free: colgante */
    free(a);                                  /* (6) doble liberacion */
}
```

Correcciones: en `crear` usar `malloc` y `i < n`; verificar `NULL`; `free` del primer bloque antes de

reasignar (o usar `realloc`); tras `free(a)` poner `a = NULL`.

Ejercicio resuelto 3.2 — ¿Qué imprime? (verificado)

```
int x = 5, y = 10;
int *p = &x, *q = &y;
*p = *q + 1;      /* x = 11 */
p = q;           /* ahora p apunta a y */
*p = 20;         /* y = 20 */
printf("%d %d %d %d\n", x, y, *p, *q);
```

Salida: 11 20 20 20. Clave: `p = q` copia la *dirección*; `*p = ...` escribe en lo apuntado.

Paso por referencia (por qué existen los apuntadores en funciones)

```
void swap_mal(int x, int y) { int t = x; x = y; y = t; } /* copias: no cambia nada */
void swap(int *x, int *y) { int t = *x; *x = *y; *y = t; } /* uso: swap(&a, &b); */
```

6. Arreglos (sesión 3)

`int a[6], BASE = 1000` →

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
1000	1004	1008	1012	1016	1020

- Un arreglo es un **bloque contiguo** y un nombre que apunta al primer elemento. C **no guarda el tamaño** ni verifica límites.
- `a + i` no suma *i* bytes: suma $i \cdot \text{sizeof}(\text{tipo})$. Por eso el acceso es $O(1)$: una multiplicación y una suma.
- Se paga: tamaño fijo, insertar/borrar en medio $O(n)$ por corrimiento, y necesita un bloque contiguo libre.
- **Decaimiento**: al pasar un arreglo a una función solo viaja la dirección. `void f(int a[], int n)` es lo mismo que `int *a`. **Siempre pasa la longitud**.

La trampa de `sizeof` (sale en exámenes)

```
void f(int a[]) { printf("%zu\n", sizeof(a) / sizeof(a[0])); } /* 8/4 = 2 (!) */
int main(void) {
    int a[10];
    printf("%zu\n", sizeof(a)); /* 40: aquí a SI es arreglo */
    f(a);                       /* imprime 2, no 10 */
}
```

Otro clásico: `for (i = 0; i <= n; i++)` escribe `a[n]`, fuera del arreglo, y C no lo impide.

Código de la práctica: máximo, invertir in situ, rotar, insertar

```
int maximo(const int a[], int n) {
    int m = a[0];
    for (int i = 1; i < n; i++)
        if (a[i] > m) m = a[i];
    return m;
}
```

```

void invertir(int a[], int ini, int fin) {    /* invierte a[ini..fin] in situ */
    while (ini < fin) {
        int t = a[ini]; a[ini] = a[fin]; a[fin] = t;
        ini++; fin--;
    }
}

void rotar_derecha(int a[], int n, int k) {    /* O(n) tiempo, O(1) extra */
    k %= n;
    invertir(a, 0, n - 1);    /* 1 2 3 4 5 -> 5 4 3 2 1 */
    invertir(a, 0, k - 1);    /* k = 2:      4 5 3 2 1 */
    invertir(a, k, n - 1);    /*              4 5 1 2 3 */
}

/* inserta x en pos; n = cantidad actual, cap = capacidad */
int insertar(int a[], int *n, int cap, int pos, int x) {
    if (*n == cap || pos < 0 || pos > *n) return 0;
    for (int i = *n; i > pos; i--)    /* corre a la derecha: n - pos movimientos */
        a[i] = a[i - 1];
    a[pos] = x;
    (*n)++;
    return 1;
}

```

Ejercicios resueltos 4.1 — Direcciones y aritmética de apuntadores

1. `int a[8]` con BASE `0x1000`: $&a[5] = 0x1000 + 5 \cdot 4 = 0x1014$.
2. `double d[10]`, BASE `0x2000`, `double *p = &d[2]`; Entonces $p + 3 = \&d[5] = 0x2000 + 5 \cdot 8 = 0x2028$. $(p+3) - p$ vale **3** (elementos), pero la distancia en bytes es **24**.
3. ¿Qué imprime?

```

int a[] = {10, 20, 30, 40, 50};
int *p = a + 1;    /* p apunta a a[1] */
printf("%d %d %d\n", *p + 1, *(p + 2), p[-1]);    /* 21 40 10 */
int *q = a;    q += 4;
printf("%td %d\n", q - a, *(a + 4));    /* 4 50 */

```

$*p + 1$ es $(*p) + 1 = 21$ (no $*(p+1)$). $p[-1]$ es legal: $*(p-1) = a[0]$.

7. Matrices y arreglos multidimensionales (sesión 4)

matriz lógica `m[2][3]`

<code>m[0][0]</code>	<code>m[0][1]</code>	<code>m[0][2]</code>
<code>m[1][0]</code>	<code>m[1][1]</code>	<code>m[1][2]</code>

memoria real

<code>m[0][0]</code>	<code>m[0][1]</code>	<code>m[0][2]</code>	<code>m[1][0]</code>	<code>m[1][1]</code>	<code>m[1][2]</code>
desplaz. 0	desplaz. 1	desplaz. 2	desplaz. 3	desplaz. 4	desplaz. 5

- La memoria es lineal: `C` guarda **por renglones** (row-major). En `m[F][C]`, `m[i][j]` está en el desplazamiento $i \cdot C + j$.
- Para calcularlo el compilador necesita `C`: en un parámetro puedes omitir la primera dimensión, **no**

la segunda: `void f(int m[][10], int F).`

- **Localidad:** recorrer por renglones lee posiciones contiguas y aprovecha la línea de caché; por columnas salta C elementos en cada paso. **Misma complejidad $O(F \cdot C)$, tiempo real muy distinto:** la disposición física de los datos, no el algoritmo, decide el desempeño.

```
for (i = 0; i < F; i++)           /* por renglones: contiguo, rapido */
    for (j = 0; j < C; j++)
        s += m[i][j];

for (j = 0; j < C; j++)           /* por columnas: salta C, lento */
    for (i = 0; i < F; i++)
        s += m[i][j];
```

Matrices dinámicas: las dos formas y cómo liberar

```
/* (a) arreglo de apuntadores: renglones sueltos (pueden medir distinto) */
int **m = malloc(F * sizeof(int *));
for (int i = 0; i < F; i++)
    m[i] = malloc(C * sizeof(int));
/* ... uso: m[i][j] ... */
for (int i = 0; i < F; i++)      /* PRIMERO cada renglon */
    free(m[i]);
free(m);                         /* AL FINAL el arreglo de apuntadores */

/* (b) bloque contiguo: un malloc, un free, mejor localidad */
int *b = malloc(F * C * sizeof(int));
/* uso: b[i * C + j] */
free(b);
```

(En examen agrega la verificación de NULL tras cada malloc.) Si liberas `m` antes que los renglones, pierdes sus direcciones: **fuga de toda la matriz**.

Ejercicios resueltos 5.1 — Direcciones en matrices (verificados)

1. `int m[4][6]`, `BASE = 2000`: $\&m[2][3] = 2000 + (2 \cdot 6 + 3) \cdot 4 = 2000 + 60 = \mathbf{2060}$.
2. La misma, pero por columnas: $2000 + (3 \cdot 4 + 2) \cdot 4 = 2000 + 56 = \mathbf{2056}$.
3. `int t[3][4][5]`, `BASE = 0`: $\&t[1][2][3] = ((1 \cdot 4 + 2) \cdot 5 + 3) \cdot 4 = 33 \cdot 4 = \mathbf{132}$.
4. **Inverso:** en `int m[5][7]` un elemento está 52 bytes después de `BASE`. $52/4 = 13$ elementos $\Rightarrow i = 13/7 = 1$, $j = 13 \bmod 7 = 6 \Rightarrow m[1][6]$.

Matrices dispersas: tercias (renglón, columna, valor)

$$M_{4 \times 5} = \begin{pmatrix} 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 7 \\ 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \end{pmatrix}$$

Tercias: (0, 2, 3), (1, 4, 7), (2, 0, 5), (3, 3, 2).

Densa: $20 \cdot 4 = 80$ bytes. Tercias: $4 \cdot 12 = 48$ bytes (cada tercia son 3 int).

Conviene si $3k < F \cdot C$, es decir $k < FC/3$. Se ahorra espacio y **se paga con acceso más lento** (hay que buscar la tercia): otra vez tiempo contra espacio.

8. Polinomios con arreglos (sesión 5 y Actividad 1)

- **Densa:** índice = exponente, contenido = coeficiente. $3x^4 + 2x + 7 \rightarrow \{7, 2, 0, 0, 3\}$. Acceso $O(1)$ a cualquier grado; desperdicia con huecos ($x^{1000} + 1$ usa 1001 celdas para 2 términos).

- **Dispersa**: solo pares (coeficiente, exponente), ordenados por exponente. Ahorra memoria; buscar un exponente exige recorrer.
- **Suma**: densa, elemento a elemento. Dispersa, **mezcla** de dos listas ordenadas comparando exponentes.
- **Producto**: cada término por cada término, exponentes se suman: $O(n \cdot m)$; grado del resultado = suma de grados (dimensiona antes).
- **Horner**: $p(x) = a_0 + x(a_1 + x(a_2 + \dots))$: $O(n)$ en lugar de $O(n^2)$ (calcular cada x^k cuesta k multiplicaciones: $0 + 1 + \dots + n = n(n+1)/2$).

Código: producto denso, Horner y suma dispersa

```
void mult_densa(const int p[], int gp, const int q[], int gq, int r[]) {
    for (int k = 0; k <= gp + gq; k++) r[k] = 0;    /* grado resultado = gp+gq */
    for (int i = 0; i <= gp; i++)
        for (int j = 0; j <= gq; j++)
            r[i + j] += p[i] * q[j];                /* O(n*m) */
}

int horner(const int a[], int g, int x) {           /* a[k] = coef de x^k; 0(g) */
    int b = a[g];
    for (int k = g - 1; k >= 0; k--)
        b = a[k] + x * b;
    return b;
}

typedef struct { int coef, exp; } Termino;          /* ordenados por exp descendente */

int suma_dispersa(const Termino p[], int np, const Termino q[], int nq, Termino r[]) {
    int i = 0, j = 0, k = 0;
    while (i < np && j < nq) {
        if (p[i].exp > q[j].exp) r[k++] = p[i++];
        else if (p[i].exp < q[j].exp) r[k++] = q[j++];
        else {
            int c = p[i].coef + q[j].coef;           /* mismo exponente */
            if (c != 0) {                             /* 0 no se guarda */
                r[k].coef = c; r[k].exp = p[i].exp; k++;
            }
            i++; j++;
        }
    }
    while (i < np) r[k++] = p[i++];
    while (j < nq) r[k++] = q[j++];
    return k;                                         /* terminos de r */
}
```

Ejercicios resueltos 6.1 — Horner, suma dispersa y producto

(a) **Horner**: $p(x) = 3x^4 + 2x + 7$ en $x = 2$. Coeficientes de mayor a menor: 3, 0, 0, 2, 7.

k	a_k	$b_k = a_k + 2 \cdot b_{k+1}$	
4	3	3	(se copia)
3	0	$0 + 2 \cdot 3 = 6$	
2	0	$0 + 2 \cdot 6 = 12$	
1	2	$2 + 2 \cdot 12 = 26$	
0	7	$7 + 2 \cdot 26 = 59$	comprobación: $48 + 4 + 7 = 59$

(b) **Suma dispersa:** $P = 5x^6 + 3x^2 + 1$, $Q = 4x^6 - 3x^2 + 2x$. Exp. 6: $5 + 4 = 9$. Exp. 2: $3 - 3 = 0$ (no se guarda). Exp. 1: solo en Q . Exp. 0: solo en P . Resultado: $9x^6 + 2x + 1$ (3 términos).

(c) **Producto:** $(2x + 1)(x^2 - 3)$: grado $1 + 2 = 3 \Rightarrow$ arreglo de 4 celdas. $= 2x^3 + x^2 - 6x - 3$.

Actividad 1 (tu entrega): polinomio en dos variables

`typedef int Polinomio[N][N];` con `matriz[i][j]` = coeficiente de $x^i y^j$. Suma = suma de matrices celda a celda. Producto: `r[i+k][j+l] += p[i][j] * q[k][l]`, **verificando** que $i+k$ y $j+l$ no pasen de **GRADO** (si no, escribes fuera de la matriz).

9. Registros: **struct**, **relleno**, **union**, **enum** (sesión 6)

```
typedef struct {
    int    matricula;
    char   nombre[50];
    float  promedio;
} Alumno;

Alumno a;          a.matricula = 170123;
Alumno *p = &a;    p->promedio = 9.1f;      /* igual a (*p).promedio */
                  /* ERROR: *p.promedio significa *(p.promedio) */

void imprimir(const Alumno *a) {           /* 8 bytes, sin copiar, no modifica */
    printf("%d %s %.1f\n", a->matricula, a->nombre, a->promedio);
}
```

- **Por valor** se copia todo el **struct** (costoso si es grande). **Por apuntador** viajan 8 bytes (lo habitual). `const Alumno *` documenta que no se modifica.
- **Relleno:** el compilador mete bytes para que cada campo caiga en múltiplo de su alineación, y completa el final para que `sizeof` sea múltiplo de la mayor alineación. Por eso `sizeof` suele ser mayor que la suma de los campos.
- **Reordenar de mayor a menor** reduce el relleno.

Ejercicios resueltos 7.1 — **sizeof** con relleno (todos verificados con gcc 64 bits)

Registro	Desplazamientos	sizeof
{char a; int b; char c;}	a@0, 3 relleno, b@4, c@8, 3 relleno final	12
{int b; char a; char c;}	b@0, a@4, c@5, 2 relleno final	8
{char c; double d; int i;}	c@0, 7, d@8, i@16, 4 final	24
{double d; int i; char c;}	d@0, i@8, c@12, 3 final	16
Alumno (int, char[50], float)	matricula@0, nombre@4.53, 2, promedio@56	60
{int i; char c[5];}	i@0, c@4.8, 3 final	12
union {int i; double d; char s[3];}	todos en @0; mide lo del mayor	8

Receta en examen: (1) recorre los campos en orden; (2) si el desplazamiento actual no es múltiplo de la alineación del campo, sube al siguiente múltiplo; (3) al final sube al múltiplo de la mayor alineación.

Actividad 2 (tu entrega) — dirección de `emp[i].address`

```
#define N 40
#define M 40
struct emp_type { char name[N]; char address[M]; } emp[100];
```

Los campos son `char` (alineación 1) $\Rightarrow$ no hay relleno, `sizeof` = $N + M = 80$. El elemento i empieza i registros después de `BASE` y `address` empieza N bytes dentro:

$$\&emp[i].address = \text{BASE} + i(N + M) + N = \text{BASE} + 80i + 40$$

$i = 7 \Rightarrow \text{BASE} + 600$; $i = 99 \Rightarrow \text{BASE} + 7960$.

Contraejemplo `struct emp_mixto {int id; char name[40]; char address[40]; double salario;};` `id@0, name@4, address@44, 4 relleno, salario@88` $\Rightarrow$ `sizeof` = 96, y `&v[i].address` = `BASE + 96i + 44`. La fórmula general siempre es `BASE + i · sizeof + offsetof`.

union + enum: registro variante

```
typedef enum { CIRCULO, RECTANGULO } TipoFigura;

typedef struct {
    TipoFigura tipo;                /* dice que campo de la union es valido */
    union {
        struct { double radio; } c;
        struct { double base, altura; } r;
    } u;                          /* comparten memoria: solo uno vale a la vez */
} Figura;

double area(const Figura *f) {
    return f->tipo == CIRCULO ? 3.14159 * f->u.c.radio * f->u.c.radio
                               : f->u.r.base * f->u.r.altura;
}
```

10. Arreglos de registros, ordenamiento y búsqueda (sesiones 7 y 8)

Un arreglo de registros es una **tabla**: renglones homogéneos, columnas heterogéneas. Alumno grupo[100]; int n = 0; — el contador `n` es tan importante como el arreglo.

Alta, búsqueda y las dos bajas

```
#define CAP 100

int alta(Alumno g[], int *n, Alumno nuevo) {
    if (*n == CAP) return 0;          /* lleno */
    g[(*n)++] = nuevo;                /* 0(1) */
    return 1;
}

int buscar(const Alumno g[], int n, int matricula) { /* secuencial 0(n) */
    for (int i = 0; i < n; i++)
        if (g[i].matricula == matricula) return i;
    return -1;
}

void baja_corrimiento(Alumno g[], int *n, int i) { /* conserva el orden, 0(n) */
    for (int k = i; k < *n - 1; k++)
```

```

        g[k] = g[k + 1];
        (*n)--;
    }

    void baja_con_ultimo(Alumno g[], int *n, int i) {        /* rompe el orden, O(1) */
        g[i] = g[*n - 1];
        (*n)--;
    }

```

Ejemplo: [A, B, C, D, E], baja de C. Con corrimiento: [A, B, D, E] (2 movimientos). Con el último: [A, B, E, D] (1 movimiento). **La elección depende de si el orden importa**, no de cuál es “mejor”.

- **Borrado lógico:** campo **activo**; dar de baja es poner 0. No mueve datos, pero las búsquedas deben saltar inactivos y hay que *compactar* cada tanto.
- **AoS vs. SoA:** arreglo de estructuras (registros completos seguidos) vs. estructura de arreglos (un arreglo por campo). Si siempre recorres *un solo campo*, SoA aprovecha mucho mejor la caché: para promediar 1000 alumnos SoA lee 4000 bytes contiguos de **promedio**; AoS recorre 60000 bytes.

Ordenar por un campo (el intercambio mueve el registro completo)

```

void ordenar_por_promedio(Alumno g[], int n) {        /* seleccion, O(n^2), desc. */
    for (int i = 0; i < n - 1; i++) {
        int mayor = i;
        for (int j = i + 1; j < n; j++)
            if (g[j].promedio > g[mayor].promedio) mayor = j;
        Alumno t = g[i]; g[i] = g[mayor]; g[mayor] = t;
    }
}

void ordenar_por_nombre(Alumno g[], int n) {        /* insercion, estable */
    for (int i = 1; i < n; i++) {
        Alumno x = g[i];
        int j = i - 1;
        while (j >= 0 && strcmp(g[j].nombre, x.nombre) > 0) { /* NUNCA == con cadenas */
            g[j + 1] = g[j];
            j--;
        }
        g[j + 1] = x;
    }
}

/* Ordenamiento INDIRECTO: se ordenan indices, los datos no se mueven */
void ordenar_indices(const Alumno g[], int idx[], int n) {
    for (int i = 0; i < n; i++) idx[i] = i;
    for (int i = 1; i < n; i++) {
        int x = idx[i], j = i - 1;
        while (j >= 0 && g[idx[j]].matricula > g[x].matricula) {
            idx[j + 1] = idx[j];
            j--;
        }
        idx[j + 1] = x;
    }
} /* listado: for (k = 0; k < n; k++) imprimir(&g[idx[k]]); */

```

`strcmp(a, b)` regresa `< 0`, `0` o `> 0`. `a == b` compara **direcciones**, no texto. Con índices (o apuntadores) puedes tener *varios órdenes a la vez* sin duplicar datos: es lo que hace un índice

de base de datos.

Búsqueda binaria

```
int binaria(const Alumno g[], int n, int matricula) { /* g ORDENADO por matricula */
    int ini = 0, fin = n - 1;
    while (ini <= fin) {
        int medio = ini + (fin - ini) / 2; /* no (ini+fin)/2: desborda */
        if (g[medio].matricula == matricula) return medio;
        if (g[medio].matricula < matricula) ini = medio + 1;
        else fin = medio - 1;
    }
    return -1;
}
```

Ejercicio resuelto 8.1 — Trazas de binaria en [3, 8, 12, 17, 25, 31, 40, 52]

Buscar 31

ini	fin	medio	a[medio]
0	7	3	17 < 31 → ini = 4
4	7	5	31 encontrado

2 vueltas.

Buscar 20

ini	fin	medio	a[medio]
0	7	3	17 < 20 → ini = 4
4	7	5	31 > 20 → fin = 4
4	4	4	25 > 20 → fin = 3

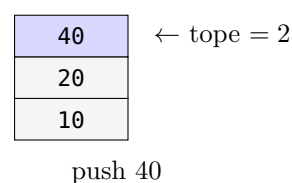
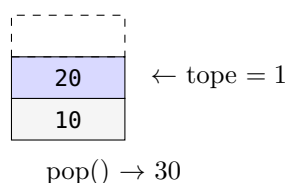
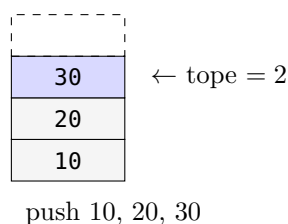
ini > fin: no está (3 vueltas).

Peor caso con $n = 8$: $\lfloor \log_2 8 \rfloor + 1 = 4$ vueltas (p.ej. buscar 60).

¿Cuándo conviene ordenar para buscar? (razonamiento de costo total)

Ordenar cuesta $O(n \log n)$. Con k búsquedas: secuencial $\approx k \cdot n$; ordenar y binaria $\approx n \log n + k \log n$. Con $n = 10^6$: ordenar $\approx 2 \times 10^7$. **5 búsquedas:** $5 \times 10^6 < 2 \times 10^7 \Rightarrow$ **no conviene**. **50 búsquedas:** $5 \times 10^7 > 2 \times 10^7 + 1000 \Rightarrow$ **sí conviene**. El punto de equilibrio es $k \approx \log_2 n \approx 20$ (peor caso). Una sola búsqueda: siempre secuencial.

11. Pilas (sesión 9)



LIFO: el último en entrar es el primero en salir; todo ocurre en **un solo extremo**. push, pop, peek, vacía, llena: todas $O(1)$, ninguna recorre. Usos: pila de llamadas, deshacer, paréntesis balanceados, expresiones, recorrido en profundidad.

Implementación con arreglo (tope = -1 es vacía)

```
#define MAX 100
typedef struct {
    int datos[MAX];
    int tope; /* indice del ultimo; -1 = vacia */
} Pila;
```

```

void inicializar(Pila *p)      { p->tope = -1; }
int vacia(const Pila *p)      { return p->tope == -1; }
int llena(const Pila *p)      { return p->tope == MAX - 1; }

int push(Pila *p, int x) {
    if (llena(p)) return 0;      /* overflow: C no lo detecta por ti */
    p->datos[++p->tope] = x;      /* primero sube, luego escribe */
    return 1;
}

int pop(Pila *p, int *x) {
    if (vacía(p)) return 0;      /* underflow */
    *x = p->datos[p->tope--];      /* primero lee, luego baja */
    return 1;
}

int peek(const Pila *p, int *x) {
    if (vacía(p)) return 0;
    *x = p->datos[p->tope];
    return 1;
}

```

Ejercicio resuelto 9.1 — Secuencias de push/pop

(a) push(1) push(2) push(3) pop() push(4) pop() pop() $\Rightarrow$ salen **3, 4, 2**; queda [1] con tope = 0.

(b) Si se meten 1, 2, 3, 4 *en ese orden* (con pop intercalados), ¿qué salidas son posibles?

Salida	¿Posible?	Por qué
2 1 4 3	Sí	push1 push2 pop pop push3 push4 pop pop
3 4 2 1	Sí	push1..3 pop(3); push4 pop(4); pop(2); pop(1)
2 4 3 1	Sí	push1 push2 pop(2); push3 push4 pop(4) pop(3) pop(1)
4 1 2 3	No	al salir 4, el tope es 3: el 1 está enterrado
1 4 2 3	No	tras sacar 4 el tope es 3, no 2

Regla rápida: cuando sale un número, los menores que siguen dentro deben salir en orden *decreciente*.

Paréntesis balanceados (“aparece en exámenes”)

```

int balanceado(const char *s) {
    char pila[MAX]; int tope = -1;
    for (; *s; s++) {
        char c = *s;
        if (c == '(' || c == '[' || c == '{') {
            if (tope == MAX - 1) return 0;
            pila[++tope] = c;      /* apertura: push */
        } else if (c == ')' || c == ']' || c == '}') {
            char abre = (c == ')') ? '(' : (c == ']') ? '[' : '{';
            if (tope == -1 || pila[tope] != abre) return 0; /* cierre sin pareja */
            tope--;               /* coincide: pop */
        }
    }
    return tope == -1;          /* sobraron aperturas? */
}

```

$\{[(())]\}$ → sí (la pila termina vacía). $[()]$ → no: al leer $)$ el tope es $[$. $(()$ → no: termina con $($ en la pila.

11.1. Infijo → postfijo

Algoritmo (con $*$ y $/$ por encima de $+$ y $-$, todos asociativos a la izquierda)

1. **Operando** → directo a la salida.
2. $($ → push.
3. $)$ → pop a la salida hasta encontrar $($; la $($ se descarta.
4. **Operador** op → mientras el tope sea operador con precedencia $\geq$ que op , pop a la salida; luego push op .
5. **Fin** → vaciar la pila a la salida.

Ejercicio resuelto 9.2 — $A + B * (C - D) / E$

Paso	Lee	Acción	Pila (tope a la der.)	Salida
1	A	operando		A
2	+	pila vacía: push	+	A
3	B	operando	+	A B
4	*	prec(+) < prec(*): push	+ *	A B
5	(	push	+ * (	A B
6	C	operando	+ * (	A B C
7	-	tope es (: push	+ * (-	A B C
8	D	operando	+ * (-	A B C D
9	)	pop hasta (	+ *	A B C D -
10	/	prec(*) $\geq$ prec(/): pop *; prec(+) < : push	+ /	A B C D - *
11	E	operando	+ /	A B C D - * E
12	fin	vaciar		A B C D - * E / +

Otro (verificado): $A * (B + C) - D / E \rightarrow A B C + * D E / -$.

Evaluación de postfijo

```

int evaluar_postfijo(const char *s) {           /* operandos de un dígito */
    int pila[MAX]; int tope = -1;
    for (; *s; s++) {
        char c = *s;
        if (c == '(') continue;
        if (isdigit((unsigned char)c)) pila[++tope] = c - '0';
        else {
            int b = pila[tope--];                /* OJO: sale primero el operando
DERECHO */
            int a = pila[tope--];
            switch (c) {
                case '+': pila[++tope] = a + b; break;
                case '-': pila[++tope] = a - b; break;
                case '*': pila[++tope] = a * b; break;
                case '/': pila[++tope] = a / b; break;
            }
        }
    }
}

```

```
return pila[tope];
}
```

Ejercicio resuelto 9.3 — Evaluar $5\ 3\ 2\ * +\ 8\ 4\ /\ -$

Lee	Acción	Pila
5	push	5
3	push	5 3
2	push	5 3 2
*	$b = 2, a = 3: 3 \cdot 2$	5 6
+	$b = 6, a = 5: 5 + 6$	11
8	push	11 8
4	push	11 8 4
/	$b = 4, a = 8: 8/4$	11 2
-	$b = 2, a = 11: 11 - 2$	9

Equivale a $5 + 3 \cdot 2 - 8/4 = 9$.

Error típico: hacer $b - a$ en vez de $a - b$. En $-$ y $/$ el orden importa: el *primer pop* es el operando *derecho*.

Otro: $6\ 2\ /\ 3\ 1\ -\ * = (6/2) \cdot (3 - 1) = 6$.

12. Recursión y su relación con la pila (sesión 10)

- Dos cosas **no opcionales**: **caso base** que detiene y **paso recursivo** que se acerca al caso base.
- Cada llamada crea un **marco** en la pila del programa (parámetros, locales, dirección de retorno). Sin caso base, o con recursión muy profunda: **desbordamiento de pila**, aunque el algoritmo sea correcto.
- **Recursión de cola**: la llamada es la última operación; el compilador *puede* convertirla en ciclo (con optimización suele pasar, pero el estándar no lo garantiza).
- Conviene pasar a iterativo si: la profundidad puede ser grande, hay recálculo exponencial (Fibonacci ingenuo) o importa más el desempeño que la claridad.

Código de la sesión

```
long factorial(int n) {
    if (n <= 1) return 1;           /* caso base */
    return n * factorial(n - 1);    /* paso: problema mas chico */
}

long factCola(int n, long acc) {    /* recursion de cola: llamar con acc = 1 */
    if (n <= 1) return acc;
    return factCola(n - 1, n * acc); /* la llamada es lo ultimo */
}

void hanoi(int n, char origen, char destino, char aux) {
    if (n == 1) {
        printf("%c -> %c\n", origen, destino);
        return;
    }
    hanoi(n - 1, origen, aux, destino); /* n-1 discos fuera del camino */
    printf("%c -> %c\n", origen, destino); /* el disco grande a su lugar */
    hanoi(n - 1, aux, destino, origen); /* n-1 discos encima */
}

long fib(int n) {                  /* 0(2^n): recalcula todo */
    if (n < 2) return n;
    return fib(n - 1) + fib(n - 2);
}
```

```

}

long fib_iter(int n) {                               /* O(n) tiempo, O(1) memoria */
    long a = 0, b = 1;
    for (int i = 0; i < n; i++) { long t = a + b; a = b; b = t; }
    return a;
}

```

Ejercicio resuelto 10.1 — Marcos de `factorial(4)`

Marco (de abajo hacia arriba)	n	Queda esperando	Regresa
<code>factorial(4)</code>	4	<code>4 * factorial(3)</code>	24
<code>factorial(3)</code>	3	<code>3 * factorial(2)</code>	6
<code>factorial(2)</code>	2	<code>2 * factorial(1)</code>	2
<code>factorial(1)</code>	1	caso base	1

Profundidad máxima: 4 marcos a la vez. Se *desapilan* en orden inverso: 1, 2, 6, 24.

Ejercicio resuelto 10.2 — ¿Qué imprime? (verificados)

```

void f(int n) { if (n == 0) return; f(n - 1); printf("%d ", n); f(n - 1); }
void g(int n) { if (n > 0) { printf("%d ", n); g(n - 2); printf("%d ", n); } }

```

Método: escribe la salida como fórmula y construye de abajo hacia arriba.

- $f: S(n) = S(n-1), n, S(n-1)$. $S(1) = 1$; $S(2) = 1\ 2\ 1$; $S(3) = 1\ 2\ 1\ 3\ 1\ 2\ 1$ ($2^3 - 1 = 7$ números).
- $g: S(n) = n, S(n-2), n$ y $S(\leq 0)$ vacío. $S(1) = 1\ 1$; $S(3) = 3\ 1\ 1\ 3$; $S(5) = 5\ 3\ 1\ 1\ 3\ 5$.

Ejercicio resuelto 10.3 — Torres de Hanói con $n = 3$ (de A a C usando B)

#	Movimiento	Disco
1	A → C	1
2	A → B	2
3	C → B	1
4	A → C	3
5	B → A	1
6	B → C	2
7	A → C	1

Recurrencia (desenrollar):

$$\begin{aligned}
 T(n) &= 2T(n-1) + 1 \\
 &= 4T(n-2) + 2 + 1 \\
 &= 2^k T(n-k) + (2^k - 1) \\
 &\stackrel{k=n-1}{=} 2^{n-1} \cdot 1 + 2^{n-1} - 1 = 2^n - 1.
 \end{aligned}$$

$n = 4 \rightarrow 15$, $n = 6 \rightarrow 63$, $n = 10 \rightarrow 1023$. $n = 64$:
 $\approx 1.8 \times 10^{19}$ s $\approx 5.8 \times 10^{11}$ años.

Primer movimiento: n impar $\rightarrow$ A→C; n par $\rightarrow$ A→B.

Ejercicio resuelto 10.4 — Por qué Fibonacci ingenuo es exponencial

Llamadas: $C(0) = C(1) = 1$, $C(n) = C(n-1) + C(n-2) + 1$.

n	0	1	2	3	4	5	6	30
fib(n)	0	1	1	2	3	5	8	832 040
llamadas	1	1	3	5	9	15	25	2 692 537

fib(30) hace casi 2.7 millones de llamadas; fib_iter(30) hace 30 vueltas. La recursión es elegante, no barata.

13. Colas (sesión 11; circular = adelanto de la 12)

FIFO: el primero en entrar es el primero en salir. Opera en **dos extremos**: se inserta por **final** y se extrae por **frente**. Usos: planificación de procesos, colas de impresión y de peticiones, búfer productor-consumidor, recorrido en anchura.

Cola lineal con arreglo y dos índices

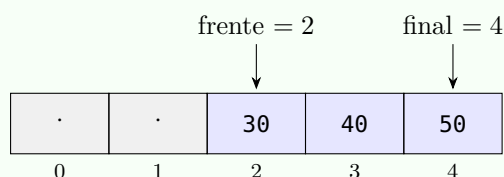
```
#define CAP 5
typedef struct {
    int datos[CAP];
    int frente, final;           /* inicio: frente = 0, final = -1 */
} ColaLineal;

int enqueue_lineal(ColaLineal *q, int x) {
    if (q->final == CAP - 1) return 0; /* "llena"... aunque haya huecos al inicio */
    q->datos[++q->final] = x;
    return 1;
}

int dequeue_lineal(ColaLineal *q, int *x) {
    if (q->frente > q->final) return 0; /* vacía */
    *x = q->datos[q->frente++];
    return 1;
}
```

Ejercicio resuelto 11.1 — El falso lleno (verificado)

CAP = 5: enqueue 10, 20, 30; dequeue (sale 10); dequeue (sale 20); enqueue 40, 50.



enqueue(60): final == CAP-1
 ⇒ se reporta **llena** con solo 3
 elementos y 2 celdas libres.

Soluciones: (1) recorrer todo en cada **dequeue** (frente siempre en 0) cuesta $O(n)$ por extracción;
 (2) **arreglo circular**: reutiliza el inicio con costo $O(1)$. Es la solución correcta.

Cola circular con contador (la más clara)

```
typedef struct {
    int datos[CAP];
    int frente, final, cont;     /* inicio: frente = 0, final = -1, cont = 0 */
} ColaCircular;
```

```

int enqueue(ColaCircular *q, int x) {
    if (q->cont == CAP) return 0;           /* llena de verdad */
    q->final = (q->final + 1) % CAP;         /* da la vuelta */
    q->datos[q->final] = x;
    q->cont++;
    return 1;
}

int dequeue(ColaCircular *q, int *x) {
    if (q->cont == 0) return 0;             /* vacía */
    *x = q->datos[q->frente];
    q->frente = (q->frente + 1) % CAP;
    q->cont--;
    return 1;
}

```

¿Por qué el contador? Con solo `frente == final` no distingues vacía de llena. **Alternativa:** sacrificar una celda. Ahí `final` apunta a la siguiente celda libre (inicio `frente = final = 0`): vacía si `frente == final`; llena si `(final + 1) % CAP == frente`; elementos = `(final - frente + CAP) % CAP`; caben `CAP - 1`.

Ejercicio resuelto 11.2 — Traza de cola circular, CAP = 5 (verificada)

Operación	datos[0..4]	frente	final	cont	Sale
inicio		0	-1	0	
enq 10, 20, 30, 40	10 20 30 40 .	0	3	4	
deq	. 20 30 40 .	1	3	3	10
deq	. . 30 40 .	2	3	2	20
enq 50	. . 30 40 50	2	4	3	
enq 60	60 . 30 40 50	2	0	4	(final da la vuelta)
enq 70	60 70 30 40 50	2	1	5	llena
enq 99	rechazada: cont == CAP	2	1	5	
deq	60 70 . 40 50	3	1	4	30
enq 80	60 70 80 40 50	3	2	5	

Orden lógico final (de `frente` a `final`): **40 50 60 70 80**. (· = celda libre; físicamente puede conservar el valor viejo, pero es basura.)

Pila vs. cola en una tabla

	Pila	Cola
Política	LIFO	FIFO
Extremos	uno (tope)	dos (frente y final)
Con arreglo	tope de -1 a MAX-1	lineal: falso lleno; circular: módulo
Errores	overflow / underflow	llena / vacía (y falso lleno)
Usos	llamadas, deshacer, paréntesis, postfijo, DFS	procesos, impresión, búferes, BFS

14. Simulacro de examen (sin ver la clave)

Instrucciones

20 minutos. Papel y pluma, sin compilador. Supón `int` de 4 bytes, `double` y apuntadores de 8. Cuando termines, califícate con la sección 15.

A. Verdadero o falso

- A1. En C, `a[i]` e `i[a]` son equivalentes.
- A2. Dentro de `void f(int a[]), sizeof(a)/sizeof(a[0])` da la longitud del arreglo.
- A3. `calloc` inicializa la memoria en cero.
- A4. Una fuga de memoria provoca un error inmediato.
- A5. La búsqueda binaria funciona sobre cualquier arreglo.
- A6. En una pila implementada con arreglo, `push` y `pop` son $O(1)$.
- A7. Recorrer una matriz por columnas en vez de por renglones cambia su complejidad.
- A8. En una cola circular sin contador, la condición `frente == final` basta para distinguir vacía de llena.

B. Cálculo

- B1. `int m[6][8]` con `BASE = 5000`. Dirección de `m[4][5]` por renglones y por columnas.
- B2. `int t[2][3][4]` con `BASE = 0`. Desplazamiento en bytes de `t[1][2][3]`.
- B3. `struct S {char a; double b; char c; int d;};` ¿`sizeof`? ¿Cuál es el mínimo reordenando?
- B4. Con Horner, evalúa $p(x) = 2x^3 - 4x^2 + 3x - 5$ en $x = 3$ mostrando la tabla.
- B5. Suma dispersa: $P = 6x^5 - 2x^3 + x$ y $Q = -6x^5 + 4x^3 + 7$. ¿Cuántos términos guarda el resultado?
- B6. Traza de binaria en `[2, 5, 9, 14, 21, 30, 33, 47, 50]` buscando 33 y buscando 4.

C. Pilas y colas

- C1. Convierte a postfijo: $(A + B) * (C - D) / E + F$.
- C2. Evalúa el postfijo: `9 3 / 4 2 - * 1 +`.
- C3. ¿Está balanceada `{([)][]}`? Si no, ¿qué hay en la pila cuando falla?
- C4. Se meten 1, 2, 3, 4, 5 en orden. ¿Es posible la salida `3 2 5 4 1`? ¿Y `4 5 3 1 2`?
- C5. Cola circular con contador, `CAP = 6`: `enq 1, 2, 3, 4, 5; deq; deq; enq 6, 7, 8; deq`. Da `frente`, `final`, `cont` y el orden lógico final.
- C6. Cola lineal, `CAP = 4`: `enq a, b, c, d; deq; deq; enq e`. ¿Qué pasa y por qué?

D. Recursión

- D1. ¿Qué imprime `r(3)`?

```
void r(int n) {
    if (n <= 0) return;
    r(n - 2);
    printf("%d ", n);
    r(n - 1);
}
```

- D2. Hanói: ¿cuántos movimientos con 6 discos? ¿Cuál es el primer movimiento con 4 discos de A a C usando B?
- D3. ¿Cuántas llamadas hace `fib(6)` en la versión ingenua?
- D4. Escribe `int potencia(int b, int e)` recursiva. ¿Cuántos marcos hay a la vez con `e = 5`?

E. Código y memoria

- E1. Encuentra todos los errores:

```
char *dup(const char *s) {
    char *r = malloc(strlen(s));
    strcpy(r, s);
    return r;
}
int main(void) {
    char *a = dup("hola");
    char *b = a;
    free(a);
    printf("%s\n", b);
    free(b);
}
```

- E2. ¿Qué imprime?

```
int x = 1, y = 2, *p = &x, *q = &y;
*q = *p + *q;
p = q;
*p = *p * 2;
printf("%d %d\n", x, y);
```

- E3. Escribe `int pop(Pila *p, int *x)` que regrese 0 en underflow.

F. Preguntas abiertas (3 líneas cada una)

- F1. ¿Cuándo das de baja con corrimiento y cuándo reemplazando con el último? ¿Qué es el borrado lógico y qué cuesta?
- F2. AoS vs. SoA: ¿qué son y cuándo conviene SoA?
- F3. Tienes 10^6 registros desordenados. ¿Conviene ordenarlos si harás 50 búsquedas? ¿Y si harás 5?
- F4. ¿Por qué una función recursiva correcta puede terminar en desbordamiento de pila?

15. Clave del simulacro (resuelta y verificada)

Banco de teoría objetiva

T1 b	T2 d	T3 a	T4 c	T5 c	T6 a	T7 d	T8 c	T9 a	T10 d
T11 b	T12 a	T13 c	T14 d	T15 b	T16 c	T17 a	T18 d	T19 b	T20 a
T21 c	T22 d	T23 a	T24 b	T25 d	T26 c	T27 a	T28 b	T29 d	T30 c
T31 a	T32 d	T33 b	T34 c	T35 a	T36 d	T37 b	T38 a	T39 c	T40 d
T41 b	T42 a	T43 c	T44 d	T45 b	T46 a	T47 c	T48 d	T49 a	T50 b
T51 c	T52 a	T53 d	T54 b	T55 c	T56 d	T57 a			
V1 F	V2 V	V3 F	V4 V	V5 F	V6 V	V7 F	V8 V	V9 F	V10 F
1-H	2-F	3-D	4-J	5-B	6-A	7-C	8-E	9-I	10-G

Por qué (los que más se fallan)

- **T6:** el temario clasifica pilas y colas como estáticas porque se implementan con arreglo; con listas ligadas serían dinámicas.
- **T13:** C no tiene paso por referencia; `swap(&a, &b)` pasa *copias de direcciones*.
- **T14:** `realloc` puede mover el bloque; por eso se asigna primero a un `tmp`.
- **T22:** el compilador necesita el número de columnas para calcular $i \cdot C + j$; el de renglones no.
- **T29:** la `union` mide lo de su miembro mayor; el `struct`, la suma más el relleno.
- **T37:** $\lfloor \log_2 10^6 \rfloor + 1 = 19 + 1 = 20$.
- **T43:** $A + B * C \rightarrow A \ B \ C * +$: las letras quedan en el mismo orden.
- **T48:** al regresar `factorial(n - 1)` todavía falta multiplicar por `n`.
- **T55:** vacía y llena se ven iguales; por eso se usa contador o se sacrifica una celda.
- **V3:** el estándar permite `free(NULL)` y no hace nada.
- **V10:** selección con `[2a, 2b, 1]`: el primer intercambio da `[1, 2b, 2a]` y los dos 2 quedan invertidos.

Simulacro práctico

A. Verdadero o falso

A1 V: ambos son $*(a+i)$. **A2** F: `a` es apuntador, da $8/4 = 2$. **A3** V. **A4** F: nada falla de inmediato. **A5** F: requiere arreglo ordenado por la clave. **A6** V. **A7** F: sigue siendo $O(F \cdot C)$; solo empeora el uso de caché. **A8** F: vacía y llena se ven igual; hace falta contador o sacrificar una celda.

B. Cálculo

- B1. Renglones: $5000 + (4 \cdot 8 + 5) \cdot 4 = 5000 + 148 = \mathbf{5148}$. Columnas: $5000 + (5 \cdot 6 + 4) \cdot 4 = 5000 + 136 = \mathbf{5136}$.
- B2. $((1 \cdot 3 + 2) \cdot 4 + 3) \cdot 4 = 23 \cdot 4 = \mathbf{92}$.
- B3. `a@0, 7, b@8, c@16, 3, d@20` $\Rightarrow$ **24**. Reordenado `{double b; int d; char a; char c;}`: 0, 8, 12, 13, 2 *final* $\Rightarrow$ **16**.
- B4. Coeficientes 2, -4, 3, -5: $b_3 = 2$; $b_2 = -4 + 3 \cdot 2 = 2$; $b_1 = 3 + 3 \cdot 2 = 9$; $b_0 = -5 + 3 \cdot 9 = \mathbf{22}$. (Comprobación: $54 - 36 + 9 - 5 = 22$.)
- B5. x^5 : $6 - 6 = 0$ **no se guarda**; x^3 : $-2 + 4 = 2$; x solo en P ; constante solo en Q . Resultado $2x^3 + x + 7$: **3 términos**.
- B6. Buscar 33: (ini 0, fin 8, medio 4: $21 < 33 \rightarrow$ ini 5); (5, 8, medio 6: **33**) $\Rightarrow$ encontrado en índice 6, 2 vueltas. Buscar 4: (0, 8, 4: $21 > 4 \rightarrow$ fin 3); (0, 3, 1: $5 > 4 \rightarrow$ fin 0); (0, 0, 0: $2 < 4 \rightarrow$ ini 1) $\Rightarrow$ no está, 3 vueltas.

C. Pilas y colas

- C1. **A B + C D - * E / F +**. Claves: en / se saca * (misma precedencia); en + se saca /.
- C2. $9/3 = 3$; $4 - 2 = 2$; $3 \cdot 2 = 6$; $6 + 1 = 7$.
- C3. **No**. Pila: { { ({ ({ ({ { [; al leer } el tope es [. En la pila queda { [.
- C4. **3 2 5 4 1: sí** (push 1, 2, 3; pop 3, pop 2; push 4, 5; pop 5, pop 4, pop 1). **4 5 3 1 2: no** (tras sacar 3 el tope es 2, no 1).
- C5. Tras enq 1..5: **final = 4, cont = 5**. Dos deq (salen 1 y 2): **frente = 2, cont = 3**. enq 6 $\rightarrow$ **final = 5**; enq 7 $\rightarrow$ **final = 0**; enq 8 $\rightarrow$ **final = 1, cont = 6** (llena). deq (sale 3). Final: **frente = 3, final = 1, cont = 5**, orden lógico **4 5 6 7 8**.
- C6. Tras 4 enq, **final = 3 = CAP-1**. Dos deq dejan **frente = 2**. **enqueue(e)** se **rechaza: falso lleno** con 2 elementos y 2 celdas libres al inicio. Se arregla con arreglo circular.

D. Recursión

- D1. $S(n) = S(n-2)$, n , $S(n-1)$. $S(1) = 1$; $S(2) = S(0), 2$, $S(1) = 2$ **1**; $S(3) = S(1), 3$, $S(2) = 1$ **3 2 1**.
- D2. $2^6 - 1 = 63$. Con 4 discos (par) el primer movimiento es **A $\rightarrow$ B**.
- D3. $C(6) = 25$ llamadas.
- D4.

```
int potencia(int b, int e) {
    if (e == 0) return 1;           /* caso base */
    return b * potencia(b, e - 1); /* e baja hasta 0 */
}
```

Con $e = 5$ se apilan $e = 5, 4, 3, 2, 1, 0$: **6 marcos**.

E. Código y memoria

- E1. (1) `malloc(strlen(s))` no deja lugar para `'\0'`: debe ser `strlen(s) + 1`; `strcpy` desborda 1 byte. (2) No verifica `NULL`. (3) `printf` de `b` tras `free(a)`: **apuntador colgante** (`b` y `a` son la misma dirección). (4) `free(b)`: **doble liberación**.
- E2. $*q = 1 + 2 \rightarrow y = 3$; $p = q$; $*p = 3 * 2 \rightarrow y = 6$. Imprime **1 6**.
- E3. Ver §11: `if (vacía(p)) return 0; *x = p->datos[p->tope-]; return 1;`

F. Preguntas abiertas (respuesta modelo)

- F1. Corrimiento si el orden importa ($O(n)$); con el último si no importa ($O(1)$). Borrado lógico: marcar **activo = 0** sin mover datos; cuesta que las búsquedas salten inactivos y hay que compactar periódicamente.
- F2. AoS: arreglo de registros completos. SoA: un arreglo por campo. SoA conviene cuando se recorre repetidamente un solo campo (p.ej. promediar), porque ese campo queda contiguo y aprovecha la caché.
- F3. Ordenar $\approx n \log_2 n \approx 2 \times 10^7$. 50 búsquedas secuenciales $\approx 5 \times 10^7$: **sí conviene** ordenar y usar binaria. 5 búsquedas $\approx 5 \times 10^6$: **no conviene**. Se decide por costo total, no por la operación aislada.
- F4. Cada llamada ocupa un marco en la pila del sistema, que es de tamaño limitado. Si la profundidad es muy grande (aunque haya caso base), los marcos no caben: desbordamiento. Solución: versión iterativa o recursión de cola.

16. Checklist de 2 minutos antes de entrar

Errores que cuestan puntos

- Verificar `NULL` tras `malloc`; `realloc` a un `tmp`; un `free` por `malloc`; `p = NULL` tras liberar.
- No usar `sizeof(a)` dentro de la función: pasar `n`. Ciclos con `i < n`, nunca `i <= n`.
- Matriz de apuntadores: liberar renglones primero, luego el arreglo.
- Suma dispersa: **no guardar coeficientes 0**. Horner empieza por el coeficiente de mayor grado.
- Relleno: múltiplo de la alineación del campo; tamaño final múltiplo de la mayor alineación.
- `p->campo = (*p).campo`; `*p.campo` es otra cosa. Cadenas con `strcmp`, nunca `==`.
- Binaria solo en arreglo ordenado; `medio = ini + (fin - ini) / 2`.
- `pop` verifica vacía, `push` verifica llena. En postfijo: primer `pop` = operando derecho.
- Recursión: caso base *y* paso que se acerca. Hanói $2^n - 1$.
- Cola lineal $\Rightarrow$ falso lleno. Circular $\Rightarrow \% \text{ CAP}$ y contador (o celda sacrificada).
- Teoría: pila LIFO (overflow en `push`, underflow en `pop`); cola FIFO; binaria = ordenado + acceso aleatorio; `union` mide lo del miembro mayor; `calloc` pone ceros; `static` local conserva su valor.
- En preguntas abiertas: **operación dominante + costo + memoria**. Esa es la respuesta que busca el profe.

¡Éxito en el parcial!