TÉCNICA DE DISEÑO

DIVIDE Y CONQUISTA

Claudia Pereira – Liliana Martinez

Divide y conquista

Técnica de diseño de algoritmos resuelve un problema recursivamente, aplicando tres pasos en cada nivel de la recursión:

- **Divide** el problema en un número de *subproblemas* que son instancias menores del mismo problema.
- **Conquista** los subproblemas resolviéndolos recursivamente. Si el tamaño del subproblema es suficientemente pequeño, lo resuelve de manera directa.
- **Combina** las soluciones de los subproblemas para obtener la solución al problema original.

Divide y Conquista: Problemas

Características de los problemas

- El problema debe admitir una formulación recursiva.
- Los subproblemas deben ser del mismo tipo que el problema original, pero con datos de tamaño estrictamente menor.
- El tamaño de los datos que manipulen los subproblemas ha de ser lo mas parecido posible

Divide y Conquista: Esquema algorítmico

```
Tipo_Solución  DyC (P) {

    if ( SIMPLE (P) ) // P pequeño -> su solución es directa
      return Solucion_Directa (P);

    else {  // P es grande

      DIVIDE P en k Subproblemas P1, P2,... Pk,  k>1;

      return ( COMBINA ( DyC(P1) , DyC(P2), ..., DyC(Pk) );
    }
}
```

Divide y Conquista: Análisis de Eficiencia

```
Tipo_Solución DyC (P) {
  if ( SIMPLE (P) )
     return Solucion_Directa (P);
  else{
     DIVIDE P en k Subproblemas P1, P2,... Pk,  k>1;
     return ( COMBINA ( DyC(P1) , DyC(P2), ..., DyC(Pk) );
      }
}
```

$$T(n) = \begin{cases} g(n) & \text{n pequeño} \\ T(n_1) + T(n_2) + \ldots + T(n_k) + \mathbf{f(n)} & \text{n suficientemente grande} \end{cases}$$

T(n) es el tiempo de ejecución de DyC con para una entrada de tamaño n,

g(n) es el tiempo para resolver directamente las entradas pequeñas

f(n) es el tiempo de dividir P en subproblemas y combinar las soluciones.

Divide y Conquista: Análisis de Eficiencia

- Nunca resuelve un problema más de una vez.
- **Divide** y **Combina** deben ser eficientes.
- El tamaño de los subproblemas debe ser lo mas parecido posible.
- Si el subproblema es suficientemente pequeño

  → evitar generar nuevas llamadas recursivas

Divide y Conquista: Método de ordenamiento Mergesort

```
MERGE-SORT (A, i, d)
{
  if  ( i < d )
      { m ← ⌊(i + d)/2⌋               Divide
      MERGE-SORT(A, i, m)
      MERGE-SORT(A, m + 1, d)         Conquista
      MERGE(A, i, m, d)               Combina
      }
}
```

$$T(n) = \begin{cases} c_0 & n <= 1 \\ 2\,T(n/2) + cn_1 + c_2 & n > 1 \end{cases}$$

Ventaja: el tiempo requerido por mergesort es proporcional a n log n.

Desventaja: requiere espacio adicional proporcional a n (para el arreglo auxiliar de la función merge).

QUICKSORT

QUICKSORT

Quicksort es un método que aplica la técnica divide y conquista para ordenar los elementos almacenados en un arreglo:

- **Divide**: Selecciona un elemento y particiona el arreglo en dos subarreglos (posiblemente vacíos) A[i…p-1] y A[p+1.. j] tal que:
  - El elemento seleccionado queda ordenado (en su posición final p)
  - los elementos en A[i…p-1] son menores o igual que A[p] y
  - los elementos en A[p+1..j] son mayores que A[p]

- **Conquista**: Ordena los subarreglos A[i…p-1] y A[p+1..j] llamando recursivamente al quicksort

A [i … p-1] ≤ A[p]	A[p]	A [p+1 … j] > A[p]
i	p	j

QUICKSORT

Quicksort es un método que aplica la técnica divide y conquista para ordenar los elementos almacenados en un arreglo:

- **Divide**: Selecciona un elemento y particiona el arreglo en dos subarreglos (posiblemente vacíos) A[i…p-1] y A[p+1.. j] tal que:
  - El elemento seleccionado, queda ordenado (en su posición final)
  - los elementos en A[i…p-1] son menores o igual que A[p] y
  - los elementos en A[p+1..j] son mayores que A[p]
- **Conquista**: Ordena los subarreglos A[i…p-1] y A[p+1..j] llamando recursivamente al quicksort
- **Combina**: No hay necesidad de combinar las soluciones (por la forma que divide, ordena los subarreglos, luego todo el arreglo queda ordenado)

Ordena el arreglo sobre si mismo => no requiere almacenamiento adicional

QUICKSORT: Esquema algorítmico

```
// ordena los elementos de A[i], A[i+1],..., A[j-1], A[j] ascendentemente

void QUICKSORT (Type A[], int i, int j) {
   if  (i < j)  { // Si hay más de un elemento divide el problema de
                  // ordenar a en dos subproblemas
      // p es la posición del pivote
      int p =  PARTICION ( A, i, j );                           Divide
      //resuelve los subproblemas
      QUICKSORT (A, i, p-1);
      QUICKSORT (A, p+1, j);                                    Conquista
     //No hay necesidad de combinar las soluciones.
     }
}
```

QUICKSORT : Partición

Primer paso: selecciona un pivote (por ejemplo, a[1])

	1	2	3	4	5	6	7	8	9
a:	65	70	50	80	85	60	55	75	45

Pivote

QUICKSORT : Partición

Primer paso: selecciona un pivote (por ejemplo, a[1])

	1	2	3	4	5	6	7	8	9
a:	65	70	50	80	85	60	55	75	45

Pivote

Segundo paso: reordena los otros elementos de modo tal que:

- el pivote queda ordenado
- los elementos menores al pivote quedan a su izquierda
- los elementos mayores al pivote quedan a su derecha

QUICKSORT : Partición

1	2	3	4	5	6	7	8	9
65	**70**	50	80	85	60	55	75	45
Pivote	i							j

Mientras i <= j

Mientras a[i] ≤ pivote -> avanza i

QUICKSORT : Partición

1	2	3	4	5	6	7	8	9
65	**70**	50	80	85	60	55	75	**45**

Pivote i j

Mientras i <= j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

QUICKSORT : Partición

1	2	3	4	5	6	7	8	9
65	**45**	50	80	85	60	55	75	**70**

i **j**

Pivote

Mientras i <= j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT : Partición

1	2	3	4	5	6	7	8	9
65	**45**	50	80	85	60	55	75	**70**

i

j

Pivote

Mientras i <= j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT : Partición

1	2	3	4	5	6	7	8	9
65	**45**	**50**	80	85	60	55	75	**70**

Pivote (1) — i (3) — j (8)

Mientras i <= j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT : Partición

1	2	3	4	5	6	7	8	9
65	**45**	**50**	**80**	85	60	55	75	**70**

Pivote

i

j

Mientras i <= j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT : Partición

1	2	3	4	5	6	7	8	9
65	**45**	**50**	**80**	85	60	55	**75**	**70**
			i				**j**	
Pivote								

Mientras i <= j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT : Partición

1	2	3	4	5	6	7	8	9
65	**45**	**50**	**80**	85	60	**55**	**75**	**70**

Pivote

i

j

Mientras i <= j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT : Partición

1	2	3	4	5	6	7	8	9
65	**45**	**50**	**55**	85	60	**80**	**75**	**70**

i

j

Pivote

Mientras i <= j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT : Partición

1	2	3	4	5	6	7	8	9
65	**45**	**50**	**55**	85	60	**80**	**75**	**70**

i j

Pivote

Mientras i <= j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT : Partición

1	2	3	4	5	6	7	8	9
65	**45**	**50**	**55**	**85**	60	**80**	**75**	**70**

Pivote

i j

Mientras i <= j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT : Partición

1	2	3	4	5	6	7	8	9
65	45	50	55	85	60	80	75	70

i j

Pivote

Mientras i <= j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT : Partición

1	2	3	4	5	6	7	8	9
65	**45**	**50**	**55**	**60**	**85**	**80**	**75**	**70**

Pivote i j

Mientras i <= j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT : Partición

1	2	3	4	5	6	7	8	9
65	**45**	**50**	**55**	**60**	**85**	**80**	**75**	**70**

Pivote

j i

Mientras i <= j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT : Partición

1	2	3	4	5	6	7	8	9
65	**45**	**50**	**55**	**60**	**85**	**80**	**75**	**70**

j i

Pivote

Mientras i <= j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT : Partición

1	2	3	4	5	6	7	8	9
65	**45**	**50**	**55**	**60**	**85**	**80**	**75**	**70**

j i

Pivote

i > j =>

Intercambia el pivote con a[j]

QUICKSORT : Partición

1	2	3	4	5	6	7	8	9
60	**45**	**50**	**55**	**65**	**85**	**80**	**75**	**70**

Pivote

i > j =>

Intercambia el pivote con a[j]

QUICKSORT : Partición

1	2	3	4	5	6	7	8	9
60	**45**	**50**	**55**	**65**	**85**	**80**	**75**	**70**

Pivote

elementos
menores al pivote

elementos
mayores al pivote

QUICKSORT

1	2	3	4	5	6	7	8	9
60	**45**	**50**	**55**	**65**	**85**	**80**	**75**	**70**

Pivote

elementos
menores al pivote

elementos
mayores al pivote

Una vez realizada la partición, cada subarreglo es ordenado llamando recursivamente a quicksort

QUICKSORT

1	2	3	4	5	6	7	8	9
45	50	55	60	65	70	75	80	85

Pivote

elementos
menores al pivote

elementos
mayores al pivote

Una vez ordenados los subarreglos, todo el arreglo está ordenado,

→ No hace falta combinar.

QUICKSORT

1	2	3	4	5	6	7	8	9
60	45	50	55	65	85	80	75	70

Pivote

elementos
menores al pivote

elementos
mayores al pivote

La partición…

¿siempre divide en subproblemas de igual tamaño?

QUICKSORT

Ordenemos los elementos menores al pivote…

1	2	3	4	5	6	7	8	9
60	45	50	55	65	85	80	75	70

i

j

Pivote

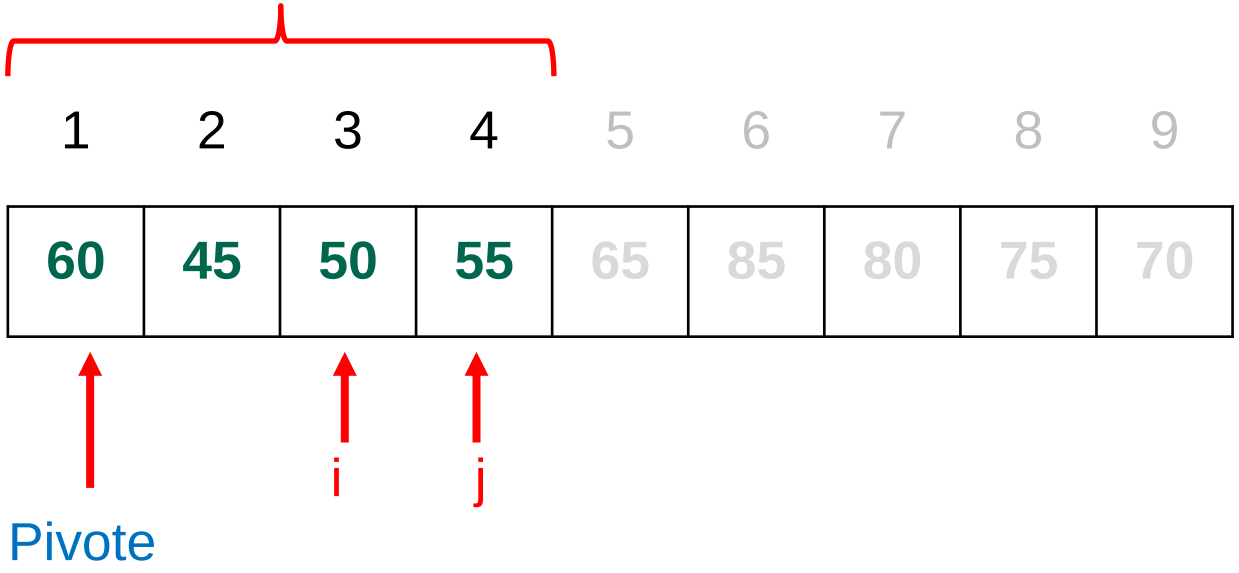
Mientras i <= j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT



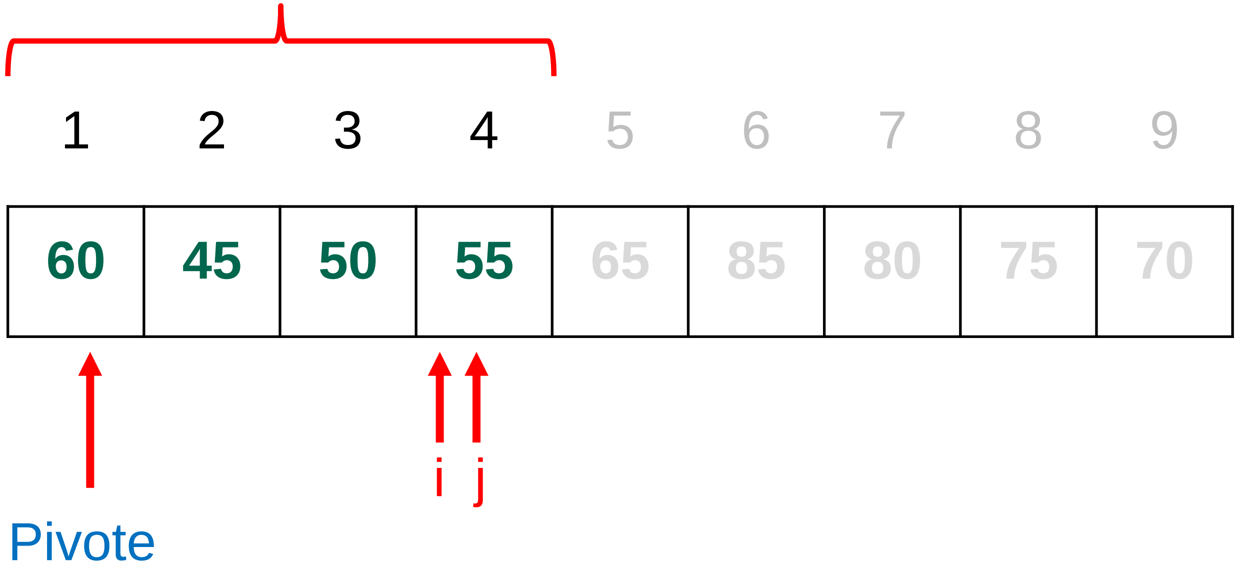
Mientras i <= j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT



Mientras i <= j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT

1	2	3	4	5	6	7	8	9
60	45	50	55	65	85	80	75	70

j i

Pivote

Mientras i <= j

Mientras a[i] ≤ pivote -> avanza i

Mientras a[j] > pivote -> retrocede j

intercambia a[i] con a[j] y avanza i y retrocede j

QUICKSORT

1	2	3	4	5	6	7	8	9
60	**45**	**50**	**55**	**65**	**85**	**80**	**75**	**70**

j i

Pivote

i > j =>

Intercambia el pivote con a[j]

QUICKSORT

1	2	3	4	5	6	7	8	9
55	45	50	60	65	85	80	75	70

Pivote

i > j =>

Intercambia el pivote con a[j]

QUICKSORT

1	2	3	4	5	6	7	8	9
55	45	50	60	65	85	80	75	70

Pivote

3 elementos
menores al pivote

0 elementos
mayores al pivote

⇨ Partición desbalanceada

QUICKSORT: Complejidad temporal

```
// ordena los elementos de A[i], A[i+1],..., A[j-1], A[j] ascendentemente

void QUICKSORT (Type A[], int i, int j) {
  if  (i < j)  { // Si hay más de un elemento divide el problema de
                  // ordenar a en dos subproblemas
     // p es la posición del pivote
     int p =  PARTICION ( A, i, j );
     //resuelve los subproblemas
     QUICKSORT (A, i, p-1);
     QUICKSORT (A, p+1, j);
    //No hay necesidad de combinar las soluciones.
    }
}
```

QUICKSORT: Complejidad temporal

El **peor caso** ocurre cuando la **partición** produce:

un problema con 0 elementos

un problema con n-1 elementos

p	a [i+1 … j] > p
i	j

un problema con n-1 elementos

un problema con 0 elementos

a [i … j-1] ≤ p	p
i	j

QUICKSORT: Complejidad temporal

El **peor caso**: la partición desbalanceada ocurre en cada llamada recursiva =>

$$T(n) \leq \begin{cases} c_0 & n \leq 1 \\ T(n-1) + c_1 n + c_2 & n > 1 \end{cases}$$

```
void QUICKSORT (Type A[], int i, int j)
{if  (i < j)
    {
      int p =  PARTICION ( A, i, j );
      QUICKSORT (A, i, p-1);
      QUICKSORT (A, p+1, j);
    }
}
```

El tiempo de particionar $\in O(n)$

El tiempo de llamar recursivamente sobre un arreglo de tamaño n-1 es T(n-1)

El tiempo de llamar recursivamente sobre un arreglo de tamaño 0 $\in O(1)$

QUICKSORT: Complejidad temporal

El **peor caso**: la partición desbalanceada ocurre en cada llamada recursiva =>

$$T(n) \leq \begin{cases} c_0 & n \leq 1 \\ T(n-1) + c_1 n + c_2 & n > 1 \end{cases}$$

```
void QUICKSORT (Type A[], int i, int j)
{if  (i < j)
     {
       int p =  PARTICION ( A, i, j );
       QUICKSORT (A, i, p-1);
       QUICKSORT (A, p+1, j);
     }
}
```

$$T(n) \in O(n^2)$$

El tiempo en el peor de los casos:

- No es mejor que el ordenamiento por inserción
- El peor tiempo ocurre cuando el arreglo ya está ordenado

QUICKSORT: Complejidad temporal

El **mejor caso** ocurre cuando la partición produce:

un problema con $\lfloor n/2 \rfloor$ elementos   un problema con $\lceil n/2 \rceil$-1 elementos

a [i … p-1] $\leq$ p	p	a [p+1 … j] > p

i   j

QUICKSORT: Complejidad temporal

El **mejor caso**: la partición balanceada ocurre en cada llamada recursiva =>

$$T(n) \leq \begin{cases} c_0 & n \leq 1 \\ 2\,T(n/2) + c_1 n + c_2 & n > 1 \end{cases}$$

```
void QUICKSORT (Type A[], int i, int j)
{if  (i < j)
     {
       int p =  PARTICION ( A, i, j );
       QUICKSORT (A, i, p-1);
       QUICKSORT (A, p+1, j);
     }
}
```

$$T(n) \in O(n \log n)$$

El tiempo de llamar recursivamente sobre un arreglo de tamaño a lo sumo n/2

El tiempo de particionar $\in$ O (n)

QUICKSORT: Complejidad temporal

El tiempo de ejecución depende de la partición:

*** partición balanceada** ⇨ el algoritmo corre asintóticamente tan rápido como el ordenamiento **mergesort**.

*** partición desbalanceada** ⇨ el algoritmo corre asintóticamente tan lento como el ordenamiento por **inserción**.

*** caso promedio** ⇨ es mucho más cercano al mejor caso que al peor caso. **El tiempo esperado es O (n log n)**

QUICKSORT: Complejidad temporal

Mejoras

- ✓ Si el peor caso se da cuando el arreglo está ordenado:

  en lugar de seleccionar el primer elemento como pivote →

  - elegir la mediana de algunos valores del arreglo (Ej: primero, medio y último)
  - seleccionar un pivote al azar

- ✓ Cuando los subproblemas son pequeños, entonces usar un algoritmo de ordenamiento iterativo simple como el de inserción

QUICKSORT: Ejercicios adicionales

1. Dado el siguiente arreglo, particionarlo tomando el primer elemento del arreglo como pivote.

M	**I**	**P**	**R**	**I**	**M**	**E**	**R**	**E**	**J**	**E**	**M**	**P**	**L**	**O**
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

2. Modificar el algoritmo Quicksort para seleccionar el k-ésimo elemento más pequeño.

3. ¿Cómo particionar el arreglo cuando existen muchos elementos repetidos?

Problema de la
Programación de Torneos de Tenis*

* Aho, Hopcroft, Ullman - Estructura de Datos y Algoritmos

Problema del Torneo de Tenis

Se debe organizar un torneo de tenis con **n jugadores** en donde:

- ✓ n es potencia de dos
- ✓ Cada jugador ha de jugar exactamente una vez contra cada uno de sus posibles n–1 competidores,
- ✓ Cada jugador debe tener un encuentro diario, durante n-1 días.

El programa del torneo es una tabla de
n filas por n-1 columnas

T[i,j] representa el jugador que debe jugar con i el j-ésimo día

Día / **Jugador**	**1**	**2**	**…**
1			
2			
3			
…			

Problema del Torneo de Tenis

La técnica D&C

- **Si n = 2, caso base, sólo hay dos jugadores**, basta enfrentar uno contra el otro.

Jugador \ Día	1
1	2
2	1

- **Si n > 2**, **Divide y Conquista:** la técnica construye un programa para la mitad de los jugadores, aplicando recursivamente el algoritmo, buscando un programa para la mitad de esos jugadores, y así sucesivamente.

Jugador \ Día	1	2	3	4	5	6	7
1							
2							
3							
4							
5							
6							
7							
8							

- **Combina**: A partir de la solución para la mitad de los jugadores, hemos llenado el cuadrante superior izquierdo de la tabla, es fácil llenar los otros tres cuadrantes.

Problema del Torneo de Tenis

Ejemplo n= 8

- **Si n > 2**: la técnica construye un programa para la mitad de los jugadores

Día / Jugador	1	2	3	4	5	6	7
1							
2							
3							
4							
5							
6							
7							
8							

Problema del Torneo de Tenis

n= **4**

Jugador \ Día	1	2	3	4	5	6	7
1							
2							
3							
4							
5							
6							
7							
8							

Problema del Torneo de Tenis

n= 4

- **Si n > 2**: la técnica construye un programa para

  la mitad de los jugadores

Día / Jugador	1	2	3	4	5	6	7
1							
2							
3							
4							
5							
6							
7							
8							

Problema del Torneo de Tenis

n= 2

Día / Jugador	1	2	3	4	5	6	7
1							
2							
3							
4							
5							
6							
7							
8							

Problema del Torneo de Tenis

n= 2

Caso Base: 2 jugadores → los enfrentamos

Jugador \ Día	1	2	3	4	5	6	7
1							
2							
3							
4							
5							
6							
7							
8							

Problema del Torneo de Tenis

n= 2

Caso Base: 2 jugadores → los enfrentamos

Día / Jugador	1	2	3	4	5	6	7
1	**2**						
2	**1**						
3							
4							
5							
6							
7							
8							

Problema del Torneo de Tenis

n= 4

Una vez resuelto el caso base, *retorna de la recursión* y procede a construir la solución:

Jugador \ Día	1	2	3	4	5	6	7
1	2						
2	1						
3							
4							
5							
6							
7							
8							

Problema del Torneo de Tenis

n= 4

Una vea resuelto el caso base, *retorna de la recursión* y procede a construir la solución:

1°) llena la mitad inferior izquierda:
enfrenta a los jugadores de numeración más alta (suma n/2) a la solución obtenida para la numeración más baja.

2°) llena el cuadrante superior derecho:
El día n/2 se enfrenta a los jugadores de menor numeración con los de mayor numeración y el resto de los días se permutan cíclicamente.

3°) llena el cuadrante inferior derecho:
Análogamente al cuadrante superior derecho, el día n/2 se enfrenta a los jugadores de mayor numeración con los de menor numeración y el resto de los días se permutan cíclicamente, pero en sentido contrario al cuadrante superior.

Jugador \ Día	1	2	3	4	5	6	7
1	2	3	4				
2	1	4	3				
3	4	1	2				
4	3	2	1				
5							
6							
7							
8							

Problema del Torneo de Tenis

n= 8

Una vez resuelto el primer cuadrante, *retorna de la recursión* y procede a construir la solución:

Jugador \ Día	1	2	3	4	5	6	7
1	2	3	4				
2	1	4	3				
3	4	1	2				
4	3	2	1				
5							
6							
7							
8							

Problema del Torneo de Tenis

n= 8

Llenó el primer cuadrante, retorna de la recursión
y procede a construir la solución:

1°) llena la mitad inferior izquierda:
enfrenta a los jugadores de numeración más alta
(suma n/2) a la solución obtenida para la numeración más baja.

2°) llena el cuadrante superior derecho:
El día n/2 se enfrenta a los jugadores de menor numeración
con los de mayor numeración y el resto de los días se permutan cíclicamente.

3°) llena el cuadrante inferior derecho:
Análogamente al cuadrante superior derecho, el día n/2 se enfrenta a los jugadores de mayor numeración con los de menor numeración y el resto de los días se permutan cíclicamente, pero en sentido contrario al cuadrante superior.

Jugador \ Día	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	1	4	3	6	7	8	5
3	4	1	2	7	8	5	6
4	3	2	1	8	5	6	7
5	6	7	8	1	4	3	2
6	5	8	7	2	1	4	3
7	8	5	6	3	2	1	4
8	7	6	5	4	3	2	1

Problema del Torneo de Tenis

Algoritmo:

```
Torneo ( Tabla, n)
{
    if ( n == 2 )   // caso base
        enfrentar a los dos jugadores
    else
        {                divide

        Torneo (tabla, n/2);   ←   conquista
        llenar cuadrante inferior izq;
        llenar cuadrante superior derecho;       combina
        llenar cuadrante inferior derecho;
        }
}
```

BIBLIOGRAFÍA

- Cormen, T.; Lieserson, C.; Rivest, R. **Introduction to Algorithms. 4th** Edition. The MIT Press. 2022.

- Horowitz, E.; Sahni, S.; Rajasekaran, S. **Computer Algorithms.** Computer Science Press.1998.

- Brassard, G.; Bratley, P. Prentice-Hall. **Fundamentos de Algoritmia**. 1997.

- Aho, Hopcroft, Ullman - Estructura de Datos y Algoritmos. Addison-Wesley, 1988