Introducción a la Computación (Matemática)

Primer Cuatrimestre de 2019

Recursión Algorítmica

Recursión algorítmica

Es uno de los conceptos centrales en Computación.

La solución a un problema depende de la solución a instancias de menor tamaño del mismo problema.

Recursión algorítmica: ejemplo (factorial)

Encabezado: $Fact : n \in \mathbb{Z} \to \mathbb{Z}$
Precondición: $\{n = n_0 \wedge n_0 \geq 0\}$
Poscondición: $\{RV = n_0!\}$

Recursión algorítmica: ejemplo (factorial)

Encabezado: $Fact : n \in \mathbb{Z} \to \mathbb{Z}$
Precondición: $\{n = n_0 \wedge n_0 \geq 0\}$
Poscondición: $\{RV = n_0!\}$

// Algoritmo iterativo
$RV \leftarrow 1$
while $(n > 0)$ $\{$
    $RV \leftarrow RV * n$
    $n \leftarrow n - 1$
$\}$

Recursión algorítmica: ejemplo (factorial)

Encabezado: $Fact : n \in \mathbb{Z} \to \mathbb{Z}$
Precondición: $\{n = n_0 \land n_0 \geq 0\}$
Poscondición: $\{RV = n_0!\}$

```
// Algoritmo iterativo
RV ← 1
while (n > 0) {
    RV ← RV * n
    n ← n − 1
}
```

```
// Algoritmo recursivo
if (n = 0) {
  RV ← 1
} else {
  RV ← Fact(n − 1) * n
}
```

Recursión algorítmica: ejemplo (producto)

Encabezado: $Prod : n \in \mathbb{Z} \times m \in \mathbb{Z} \to \mathbb{Z}$
Precondición: $\{n = n_0 \wedge n_0 \geq 0 \wedge m = m_0 \wedge m_0 \geq 0\}$
Poscondición: $\{RV = n_0 * m_0\}$

Recursión algorítmica: ejemplo (producto)

Encabezado: $Prod : n \in \mathbb{Z} \times m \in \mathbb{Z} \to \mathbb{Z}$
Precondición: $\{n = n_0 \wedge n_0 \geq 0 \wedge m = m_0 \wedge m_0 \geq 0\}$
Poscondición: $\{RV = n_0 * m_0\}$

// Algoritmo iterativo
$RV \leftarrow 0$
while $(m > 0)$ {
    $RV \leftarrow RV + n$
    $m \leftarrow m - 1$
}

Recursión algorítmica: ejemplo (producto)

Encabezado: $Prod : n \in \mathbb{Z} \times m \in \mathbb{Z} \to \mathbb{Z}$
Precondición: $\{n = n_0 \wedge n_0 \geq 0 \wedge m = m_0 \wedge m_0 \geq 0\}$
Poscondición: $\{RV = n_0 * m_0\}$

```
// Algoritmo iterativo
RV ← 0
while (m > 0) {
      RV ← RV + n
      m ← m − 1
}
```

```
// Algoritmo recursivo
if (m = 0) {
   RV ← 0
} else {
   RV ← Prod(n, m − 1) + n
}
```

Recursión algorítmica

1. Resolver el problema para los casos base.

2. Suponiendo que se tiene resuelto el problema para instancias de menor tamaño, modificar dichas soluciones para obtener una solución al problema original.

La recursión ofrece otra forma de ciclar o repetir código.

Recursión algorítmica

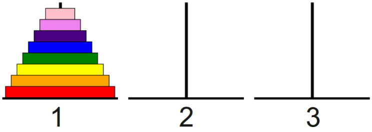
Herramienta poderosa para encontrar algoritmos para problemas no triviales, mediante técnicas como Divide and conquer o Backtracking.

Recursión algorítmica

Herramienta poderosa para encontrar algoritmos para problemas no triviales, mediante técnicas como Divide and conquer o Backtracking.

Por ej., ¿se acuerdan del problema de las Torre de Hanoi?

- Mover N discos de la estaca 1 a la 3.
- Mover de a un disco por vez.
- No se puede colocar un disco sobre otro de menor tamaño.



Hoy vamos a ver cómo resolverlo usando D&C.

Divide and conquer

- Táctica político-militar de dudoso origen, frecuentemente atribuida a Julio César.
- Consiste en dividir al enemigo, de modo que cada una de las partes sea más fácil de derrotar que el todo.

Divide and conquer

- Táctica político-militar de dudoso origen, frecuentemente atribuida a Julio César.
- Consiste en dividir al enemigo, de modo que cada una de las partes sea más fácil de derrotar que el todo.

En Computación, la técnica de D&C tiene tres etapas:

1. **Divide:** Dividir el problema en varios subproblemas de menor tamaño.

Divide and conquer

- Táctica político-militar de dudoso origen, frecuentemente atribuida a Julio César.
- Consiste en dividir al enemigo, de modo que cada una de las partes sea más fácil de derrotar que el todo.

En Computación, la técnica de D&C tiene tres etapas:

1. **Divide:** Dividir el problema en varios subproblemas de menor tamaño.
2. **Conquer:** Resolver cada subproblema recursivamente. Si un subproblema es lo suficientemente pequeño (un caso base), resolverlo en forma directa.

Divide and conquer

- Táctica político-militar de dudoso origen, frecuentemente atribuida a Julio César.
- Consiste en dividir al enemigo, de modo que cada una de las partes sea más fácil de derrotar que el todo.

En Computación, la técnica de D&C tiene tres etapas:

1. **Divide:** Dividir el problema en varios subproblemas de menor tamaño.
2. **Conquer:** Resolver cada subproblema recursivamente. Si un subproblema es lo suficientemente pequeño (un caso base), resolverlo en forma directa.
3. **Combine:** Combinar las soluciones de los subproblemas en una solución del problema original.

Ejemplo de D&C: Mergesort

Problema: Ordenar un arreglo A de enteros de tamaño n.

1. **Divide:**

Ejemplo de D&C: Mergesort

Problema: Ordenar un arreglo A de enteros de tamaño n.

1. **Divide:** Dividir A en 2 subarreglos de tamaño $\sim \frac{n}{2}$.
2. **Conquer:**

Ejemplo de D&C: Mergesort

Problema: Ordenar un arreglo A de enteros de tamaño n.

1. **Divide:** Dividir A en 2 subarreglos de tamaño $\sim \frac{n}{2}$.
2. **Conquer:** Ordenar cada subarreglo recursivamente. Si un subarreglo tiene tamaño 1 (caso base), no hacer nada.
3. **Combine:**

Ejemplo de D&C: Mergesort

Problema: Ordenar un arreglo A de enteros de tamaño n.

1. **Divide:** Dividir A en 2 subarreglos de tamaño $\sim \frac{n}{2}$.
2. **Conquer:** Ordenar cada subarreglo recursivamente.
   Si un subarreglo tiene tamaño 1 (caso base), no hacer nada.
3. **Combine:** Combinar los 2 subarreglos ordenados (función *merge*).

Ejemplo de D&C: Mergesort

Problema: Ordenar un arreglo A de enteros de tamaño n.

1. **Divide:** Dividir A en 2 subarreglos de tamaño $\sim \frac{n}{2}$.
2. **Conquer:** Ordenar cada subarreglo recursivamente.
   Si un subarreglo tiene tamaño 1 (caso base), no hacer nada.
3. **Combine:** Combinar los 2 subarreglos ordenados (función *merge*).

Si *merge* tiene orden lineal, entonces *mergesort* tiene $O(n \log n)$. (Demostración: Más adelante.)

Ejemplo de D&C: Búsqueda Binaria

4	7	23	41	44	59	97	134	165	187	210	212	249	280	314
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Buscamos el número **97**...

4	7	23	41	44	59	97	134
0	1	2	3	4	5	6	7

165	187	210	212	249	280	314
8	9	10	11	12	13	14

Ejemplo de D&C: Búsqueda Binaria

4	7	23	41	44	59	97	134	165	187	210	212	249	280	314
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Buscamos el número **97**...

4	7	23	41	44	59	97	134
0	1	2	3	4	5	6	7

165	187	210	212	249	280	314
8	9	10	11	12	13	14

4	7	23	41
0	1	2	3

44	59	97	134
4	5	6	7

44	59
4	5

97	134
6	7

97
6

134
7

97 ✓
6

Ejemplo de D&C: Búsqueda Binaria Recursiva

$$Buscar : x \in \mathbb{Z} \times A \in \mathbb{Z}[\,] \rightarrow está \in \mathbb{B} \times pos \in \mathbb{Z}$$

Ejemplo de D&C: Búsqueda Binaria Recursiva

$$\mathit{Buscar} : x \in \mathbb{Z} \times A \in \mathbb{Z}[\,] \rightarrow \mathit{está} \in \mathbb{B} \times \mathit{pos} \in \mathbb{Z}$$
$$(\mathit{está}, \mathit{pos}) \leftarrow \mathit{BuscarDesdeHasta}(x, A, 0, |A| - 1)$$

Ejemplo de D&C: Búsqueda Binaria Recursiva

$$\mathit{Buscar} : x \in \mathbb{Z} \times A \in \mathbb{Z}[\,] \to \mathit{está} \in \mathbb{B} \times \mathit{pos} \in \mathbb{Z}$$
$$(\mathit{está}, \mathit{pos}) \leftarrow \mathit{BuscarDesdeHasta}(x, A, 0, |A| - 1)$$

$$\mathit{BuscarDesdeHasta} : x \in \mathbb{Z} \times A \in \mathbb{Z}[\,] \times \mathit{izq} \in \mathbb{Z} \times \mathit{der} \in \mathbb{Z}$$
$$\to \mathit{está} \in \mathbb{B} \times \mathit{pos} \in \mathbb{Z}$$

Ejemplo de D&C: Búsqueda Binaria Recursiva

$$\mathit{Buscar} : x \in \mathbb{Z} \times A \in \mathbb{Z}[\,] \rightarrow \mathit{está} \in \mathbb{B} \times \mathit{pos} \in \mathbb{Z}$$

$$(\mathit{está}, \mathit{pos}) \leftarrow \mathit{BuscarDesdeHasta}(x, A, 0, |A| - 1)$$

$$\mathit{BuscarDesdeHasta} : x \in \mathbb{Z} \times A \in \mathbb{Z}[\,] \times \mathit{izq} \in \mathbb{Z} \times \mathit{der} \in \mathbb{Z} \rightarrow \mathit{está} \in \mathbb{B} \times \mathit{pos} \in \mathbb{Z}$$

$\mathit{med} \leftarrow (\mathit{izq} + \mathit{der})$ div 2
if $(A[\mathit{med}] < x)$ {
    $(\mathit{está}, \mathit{pos}) \leftarrow \mathit{BuscarDesdeHasta}(x, A, \mathit{med} + 1, \mathit{der})$
} else {
    $(\mathit{está}, \mathit{pos}) \leftarrow \mathit{BuscarDesdeHasta}(x, A, \mathit{izq}, \mathit{med})$
}

Ejemplo de D&C: Búsqueda Binaria Recursiva

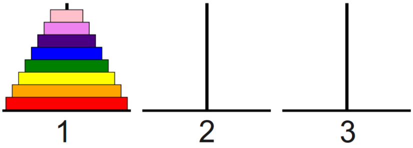
$$Buscar : x \in \mathbb{Z} \times A \in \mathbb{Z}[\,] \rightarrow está \in \mathbb{B} \times pos \in \mathbb{Z}$$

$$(está, pos) \leftarrow BuscarDesdeHasta(x, A, 0, |A| - 1)$$

$$BuscarDesdeHasta : x \in \mathbb{Z} \times A \in \mathbb{Z}[\,] \times izq \in \mathbb{Z} \times der \in \mathbb{Z}$$
$$\rightarrow está \in \mathbb{B} \times pos \in \mathbb{Z}$$

if $(izq = der)$ {
  $(está, pos) \leftarrow (x = A[izq],\ izq)$
} else {
  $med \leftarrow (izq + der)$ div 2
  if $(A[med] < x)$ {
    $(está, pos) \leftarrow BuscarDesdeHasta(x, A, med + 1, der)$
  } else {
    $(está, pos) \leftarrow BuscarDesdeHasta(x, A, izq, med)$
  }
}

Ejemplo de D&C: Torre de Hanoi



Objetivo: Mover N discos de la estaca 1 a la 3.

Restricciones:

- Mover de a un disco por vez.
- No se puede poner un disco sobre otro de menor tamaño.

Demo: http://www.cut-the-knot.org/recurrence/hanoi.shtml

Ejemplo de D&C: Torre de Hanoi

Hanoi(n, *desde*, *hacia*, *otra*):
    if ($n > 1$):
        Hanoi($n - 1$, *desde*, *otra*, *hacia*)
        Mover el disco superior de *desde* a *hacia*.
        Hanoi($n - 1$, *otra*, *hacia*, *desde*)
    else:
        Mover el disco superior de *desde* a *hacia*.

Por ejemplo, el llamado para resolver Hanoi de 8 discos es:
Hanoi(8, 1, 3, 2).

Recursión

$Sumatoria : n \in \mathbb{Z} \to \mathbb{Z}$
if $(n = 0)$ $\{$
  $RV \leftarrow 0$
$\}$ else $\{$
  $RV \leftarrow Sumatoria(n-1) + n$
$\}$

Recursión

La solución a un problema depende de la solución a instancias de menor tamaño del mismo problema.

1. Resuelvo el problema para los casos base.
2. Supongo que tengo resuelto el problema para instancias de menor tamaño; modifico dichas soluciones para obtener una solución al problema original.

Recursión

La solución a un problema depende de la solución a instancias de menor tamaño del mismo problema.

1. Resuelvo el problema para los casos base.
2. Supongo que tengo resuelto el problema para instancias de menor tamaño; modifico dichas soluciones para obtener una solución al problema original.

La recursión es útil para escribir código simple y claro, pero...

- ¡Cuidado con el consumo de memoria!

La recursión y el consumo de memoria

$Sumatoria : n \in \mathbb{Z} \to \mathbb{Z}$
if $(n = 0)$ $\{$
  $RV \leftarrow 0$
$\}$ else $\{$
  $RV \leftarrow Sumatoria(n-1) + n$
$\}$

La recursión y el consumo de memoria

$Sumatoria : n \in \mathbb{Z} \to \mathbb{Z}$
if $(n = 0)$ {
  $RV \leftarrow 0$
} else {
  $RV \leftarrow Sumatoria(n-1) + n$
}

Para cada llamado a una función, se crea un nuevo espacio de variables en la memoria.

Si se ejecutan muchos llamados recursivos, el programa puede morir por falta de memoria.

Ejemplo: `sumatoria.py`.

Cálculo de complejidad temporal

$Sumatoria : n \in \mathbb{Z} \to \mathbb{Z}$
if $(n = 0)$ {
  $RV \leftarrow 0$
} else {
  $RV \leftarrow Sumatoria(n-1) + n$
}

Cálculo de complejidad temporal

$Sumatoria : n \in \mathbb{Z} \to \mathbb{Z}$
if $(n = 0)$ { $\quad (2)$
$\quad RV \leftarrow 0 \quad (1)$
} else {
$\quad RV \leftarrow Sumatoria(n-1) + n \quad (5 + T(n-1))$
}

$T(0) = 3$
$T(n) = T(n-1) + 7$ para $n > 0$.

Cálculo de complejidad temporal

$Sumatoria : n \in \mathbb{Z} \to \mathbb{Z}$
if $(n = 0)$ { $\quad (2)$
$\quad RV \leftarrow 0 \quad (1)$
} else {
$\quad RV \leftarrow Sumatoria(n-1) + n \quad (5 + T(n-1))$
}

$T(0) = 3$
$T(n) = T(n-1) + 7$ para $n > 0$.

¿Podemos encontrar una definición no recursiva para $T(n)$?

Cálculo de complejidad temporal

$Sumatoria : n \in \mathbb{Z} \rightarrow \mathbb{Z}$
if $(n = 0)$ { $\quad (2)$
$\quad RV \leftarrow 0 \quad (1)$
} else {
$\quad RV \leftarrow Sumatoria(n-1) + n \quad (5 + T(n-1))$
}

$T(0) = 3$
$T(n) = T(n-1) + 7$ para $n > 0$.

¿Podemos encontrar una definición no recursiva para $T(n)$?
$T(n) = 7n + 3$ (Debe probarse que ambas def's son equivalentes.)

Cálculo de complejidad temporal

$Sumatoria : n \in \mathbb{Z} \rightarrow \mathbb{Z}$
if $(n = 0)$ { $\quad (2)$
$\quad RV \leftarrow 0 \quad (1)$
} else {
$\quad RV \leftarrow Sumatoria(n-1) + n \quad (5 + T(n-1))$
}

$T(0) = 3$
$T(n) = T(n-1) + 7$ para $n > 0$.

¿Podemos encontrar una definición no recursiva para $T(n)$?
$T(n) = 7n + 3$ (Debe probarse que ambas def's son equivalentes.)

Finalmente, $T(n) \in O(n)$.

Cálculo de complejidad temporal

$T(0) = 3$
$T(n) = T(n-1) + 7$ para $n > 0$.

¿Y si no encontramos una definición no recursiva para $T(n)$?

Cálculo de complejidad temporal

$T(0) = 3$
$T(n) = T(n-1) + 7$ para $n > 0$.

¿Y si no encontramos una definición no recursiva para $T(n)$?

Podemos probar por inducción que $T(n) \in O(n)$.
O sea, qvq $\exists c, n_0 \in \mathbb{N}_{>0}$ tq $T(n) \leq c \cdot n, \forall n \geq n_0$.

Cálculo de complejidad temporal

$T(0) = 3$
$T(n) = T(n-1) + 7$ para $n > 0$.

¿Y si no encontramos una definición no recursiva para $T(n)$?

Podemos probar por inducción que $T(n) \in O(n)$.
O sea, qvq $\exists c, n_0 \in \mathbb{N}_{>0}$ tq $T(n) \leq c \cdot n, \forall n \geq n_0$.

Elegimos $c = 10, n_0 = 1$.

- Caso base:
- Paso inductivo:

Cálculo de complejidad temporal

$T(0) = 3$
$T(n) = T(n-1) + 7$ para $n > 0$.

¿Y si no encontramos una definición no recursiva para $T(n)$?

Podemos probar por inducción que $T(n) \in O(n)$.
O sea, qvq $\exists c, n_0 \in \mathbb{N}_{>0}$ tq $T(n) \leq c \cdot n, \forall n \geq n_0$.

Elegimos $c = 10, n_0 = 1$.

- Caso base: $T(1) = T(0) + 7 = 10 = c \cdot n$ ✓
- Paso inductivo:
  Para $n \geq 1$, sup. $T(n) \leq c \cdot n$, qvq $T(n+1) \leq c(n+1)$.
  $$T(n+1) \stackrel{def}{=} T(n) + 7 \stackrel{HI}{\leq} 10n + 7 \leq 10n + 10 = c(n+1) \checkmark$$

Entonces, $T(n) \in O(n)$.

Recursión algorítmica: otro ejemplo (Fibonacci)

$Fib : n \in \mathbb{Z} \rightarrow \mathbb{Z}$
if $(n = 0)$ $\{$
  $RV \leftarrow 0$
$\}$ else if $(n = 1)$ $\{$
  $RV \leftarrow 1$
$\}$ else $\{$
  $RV \leftarrow Fib(n-1) + Fib(n-2)$
$\}$

Recursión algorítmica: otro ejemplo (Fibonacci)

$Fib : n \in \mathbb{Z} \to \mathbb{Z}$
if $(n = 0)$ { $\quad O(1)$
$\quad RV \leftarrow 0 \quad O(1)$
} else if $(n = 1)$ { $\quad O(1)$
$\quad RV \leftarrow 1 \quad O(1)$
} else {
$\quad RV \leftarrow Fib(n-1) + Fib(n-2) \quad T(n-1) + T(n-2) + O(1)$
}

$T(0) = O(1)$
$T(1) = O(1)$
$T(n) = T(n-1) + T(n-2) + O(1) \quad (n \geq 2)$

Recursión algorítmica: otro ejemplo (Fibonacci)

$Fib : n \in \mathbb{Z} \rightarrow \mathbb{Z}$
if $(n = 0)$ { $\quad O(1)$
$\quad RV \leftarrow 0 \quad O(1)$
} else if $(n = 1)$ { $\quad O(1)$
$\quad RV \leftarrow 1 \quad O(1)$
} else {
$\quad RV \leftarrow Fib(n-1) + Fib(n-2) \quad T(n-1) + T(n-2) + O(1)$
}

$$T(0) = O(1)$$
$$T(1) = O(1)$$
$$T(n) = T(n-1) + T(n-2) + O(1) \quad (n \geq 2)$$

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

Recursión algorítmica: otro ejemplo (Fibonacci)

$Fib : n \in \mathbb{Z} \to \mathbb{Z}$
if $(n = 0)$ { $\quad O(1)$
$\quad RV \leftarrow 0 \quad O(1)$
} else if $(n = 1)$ { $\quad O(1)$
$\quad RV \leftarrow 1 \quad O(1)$
} else {
$\quad RV \leftarrow Fib(n-1) + Fib(n-2) \quad T(n-1) + T(n-2) + O(1)$
}

$T(0) = O(1)$
$T(1) = O(1)$
$T(n) = T(n-1) + T(n-2) + O(1) \quad (n \geq 2)$

$T(n) \in O(2^n)$!!! Algoritmo iterativo: $O(n)$.

Recursión

La recursión es útil para escribir código simple y claro, pero...

- ¡Cuidado con el consumo de memoria!

Recursión

La recursión es útil para escribir código simple y claro, pero...

- ¡Cuidado con el consumo de memoria!
- ¡Cuidado con la complejidad temporal!

La recursión no es mala o buena *per se*.

Sólo hay que tener cuidado.

A veces nos lleva a algoritmos ineficientes (ej: Fibonacci) y otras veces a algoritmos muy eficientes (ej: Mergesort).

Complejidad de Mergesort

Problema: Ordenar un arreglo A de enteros de tamaño n.

1. **Divide:** Dividir A en 2 subarreglos de tamaño $\sim \frac{n}{2}$.
2. **Conquer:** Ordenar cada subarreglo recursivamente.
   Si un subarreglo tiene tamaño 1, no hacer nada.
3. **Combine:** Combinar los 2 subarreglos ordenados.

Complejidad de Mergesort

Problema: Ordenar un arreglo A de enteros de tamaño n.

1. **Divide:** Dividir A en 2 subarreglos de tamaño $\sim \frac{n}{2}$. $O(1)$
2. **Conquer:** Ordenar cada subarreglo recursivamente. $2T(\frac{n}{2})$
   Si un subarreglo tiene tamaño 1, no hacer nada. $O(1)$
3. **Combine:** Combinar los 2 subarreglos ordenados. $O(n)$

Complejidad de Mergesort

Problema: Ordenar un arreglo A de enteros de tamaño n.

1. **Divide:** Dividir A en 2 subarreglos de tamaño $\sim \frac{n}{2}$. $O(1)$
2. **Conquer:** Ordenar cada subarreglo recursivamente. $2T(\frac{n}{2})$
   Si un subarreglo tiene tamaño 1, no hacer nada. $O(1)$
3. **Combine:** Combinar los 2 subarreglos ordenados. $O(n)$

Por lo tanto: $T(1) = O(1)$
$T(n) = 2T(\frac{n}{2}) + O(n)$

Complejidad de Mergesort

Queremos ver que $T(n) \in O(n \log n)$.

Complejidad de Mergesort

Queremos ver que $T(n) \in O(n \log n)$.

$T(n) = \; 2T(\frac{n}{2}) + O(n)$

Complejidad de Mergesort

Queremos ver que $T(n) \in O(n \log n)$.

$$
\begin{aligned}
T(n) &= 2T(\tfrac{n}{2}) + O(n) \\
&\leq 2T(\tfrac{n}{2}) + c \cdot n \\
&\leq 2(2T(\tfrac{n}{4}) + c \cdot \tfrac{n}{2}) + c \cdot n &&= 4T(\tfrac{n}{4}) + 2 \cdot c \cdot n \\
&\leq 4(2T(\tfrac{n}{8}) + c \cdot \tfrac{n}{4}) + 2 \cdot c \cdot n &&= 8T(\tfrac{n}{8}) + 3 \cdot c \cdot n \\
&\ldots \\
&\leq 2^k \cdot T(\tfrac{n}{2^k}) + k \cdot c \cdot n \\
&\approx n \cdot T(1) + \log_2 n \cdot c \cdot n && \text{porque } k \approx \log_2 n \\
&= O(n) + O(n \log n) \\
&= O(n \log n)
\end{aligned}
$$

Por lo tanto, Mergesort tiene $O(n \log n)$.

Ejemplo de D&C: Par más cercano

Dados n puntos (x_i, y_i) en el plano, encontrar el par de puntos más cercanos entre sí (considerar la distancia euclideana).

Algoritmo exhaustivo: $O(n^2)$.
Algoritmo *divide and conquer*: $O(n \log n)$.

Repaso de la clase de hoy

- Recursión algorítmica.
- Producto, factorial.
- Divide & Conquer.
- Mergesort, Hanoi, Búsqueda binaria, par más cercano.
- Complejidad de algoritmos recursivos.
- Consumo de memoria de la recursión.

Próximos temas

- Tipos abstractos de datos: uso e implementación.
- Backtracking.