Práctico 1.3: Ejercicio 2

Algoritmos y Estructuras de Datos II - 2020
FaMAF - UNC

Práctico 1.3: Ejercicio 2 (b) y (c): Encontrar la cima

2. Dado un arreglo a : **array**$[1..n]$ **of nat** se define una *cima* de a como un valor k en el intervalo $1, \ldots, n$ tal que $a[1..k]$ está ordenado crecientemente y $a[k..n]$ está ordenado decrecientemente.

   (a) Escribí un algoritmo que determine si un arreglo dado tiene cima.

   (b) Escribí un algoritmo que encuentre la cima de un arreglo dado (asumiendo que efectivamente tiene una cima) utilizando una búsqueda secuencial, desde el comienzo del arreglo hacia el final.

   (c) Escribí un algoritmo que resuelva el mismo problema del inciso anterior utilizando la idea de *búsqueda binaria*.

   (d) Calculá y compará el orden de complejidad de ambos algoritmos.

- Aclaraciones para los incisos (b) y (c):
  - Se asume que el arreglo tiene cima.
  - Creciente y decreciente en sentido **estricto**!!
  - Puede suceder que la cima sea el primero o el último elemento (k = 1 o k = n resp.).

Ejemplos: Arreglos con cima

- a = [7]. Resultado: k = 1.
- a = [2, 7]. Resultado: k = 2.
- a = [9, 3, 0]. Resultado: k = 1.
- a = [-1, 0, 7, 2]. Resultado: k = 3.
- a = [-2, 8, 9, 5, 0]. Resultado: k = 3.

Ejercicio 2b: Búsqueda secuencial

- Idea:
  - Recorrer los elementos de izquierda a derecha.
  - Frenar cuando:
    - Encuentro un elemento que es mayor al que le sigue: es la cima.
    - Llego hasta el último elemento: es la cima.

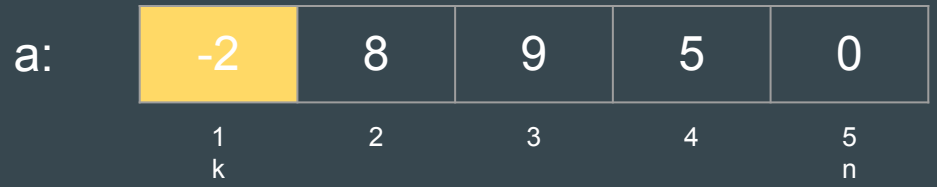
- Algoritmo:
  - Inicialización: `k := 1`
  - Invariante del ciclo: El arreglo es creciente hasta el k-ésimo elemento (inclusive).
  - Guarda del ciclo: `k < n and a[k] < a[k+1]`
    - ¿Estoy antes del último elemento? O sea, ¿ Es `k < n` ?
    - ¿Es el elemento actual menor al que le sigue? ¿ `a[k] < a[k+1]` ?
  - Cuerpo del ciclo: `k := k + 1`

Ejercicio 2b: Resultado Final

```
fun cima(a: array[1..n] of nat) ret k: nat
  k := 1
  do k < n and a[k] < a[k+1] ->
    k := k + 1
  od
end fun
```

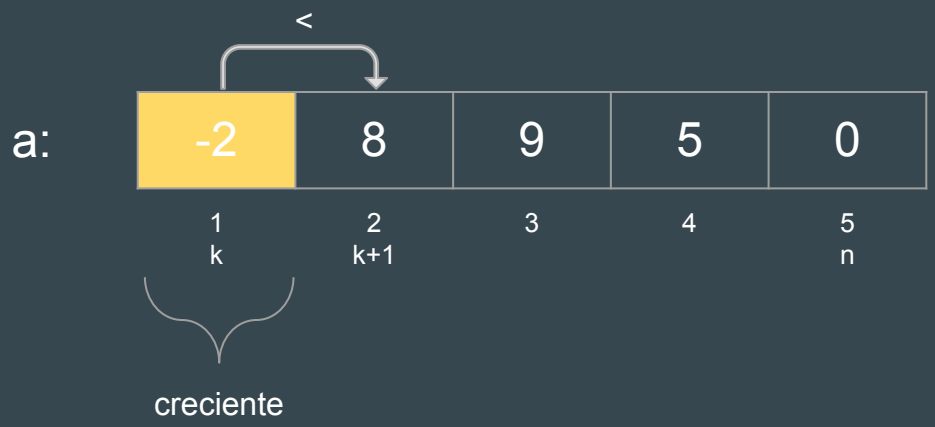
Ejercicio 2b: Ejemplo

```
fun cima(a: array[1..n] of nat) ret k: nat
  k := 1
  do k < n and a[k] < a[k+1] ->
    k := k + 1
  od
end fun
```



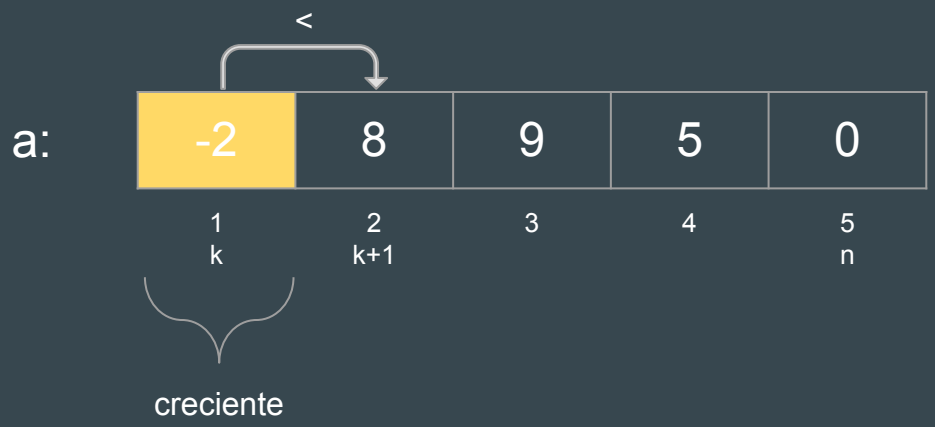
Ejercicio 2b: Ejemplo

```
fun cima(a: array[1..n] of nat) ret k: nat
  k := 1
  do k < n and a[k] < a[k+1] ->
    k := k + 1
  od
end fun
```



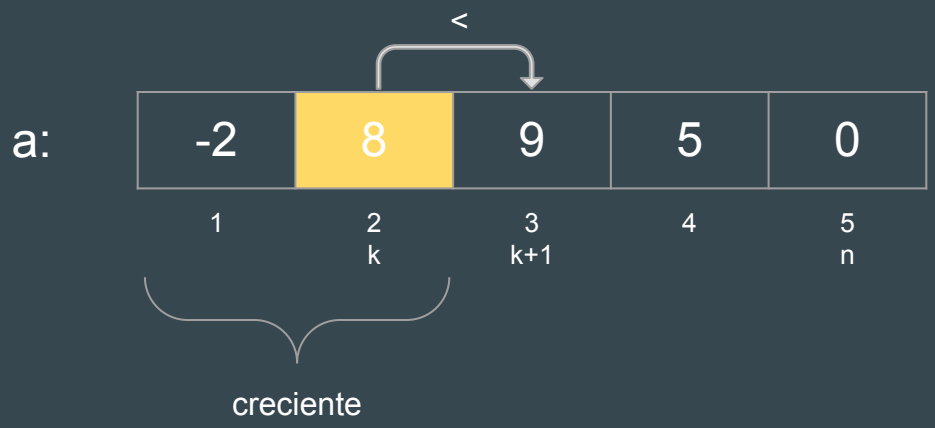
Ejercicio 2b: Ejemplo

```
fun cima(a: array[1..n] of nat) ret k: nat
  k := 1
  do k < n and a[k] < a[k+1] ->
    k := k + 1
  od
end fun
```



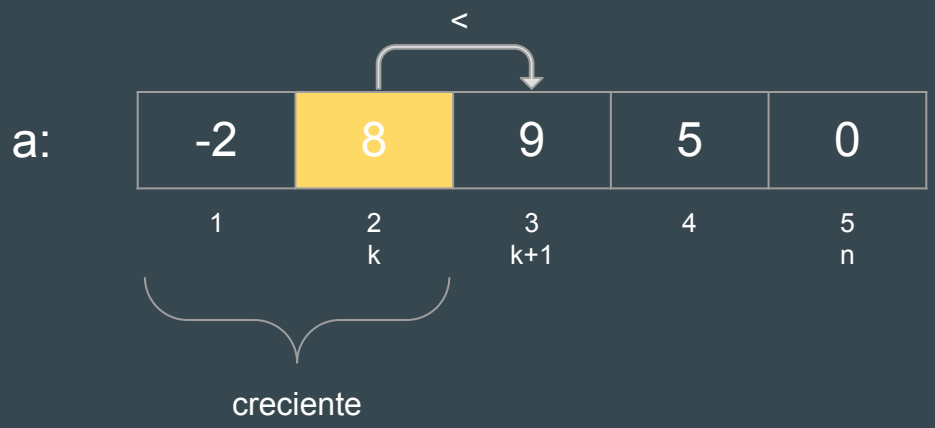
Ejercicio 2b: Ejemplo

```
fun cima(a: array[1..n] of nat) ret k: nat
  k := 1
  do k < n and a[k] < a[k+1] ->
    k := k + 1
  od
end fun
```



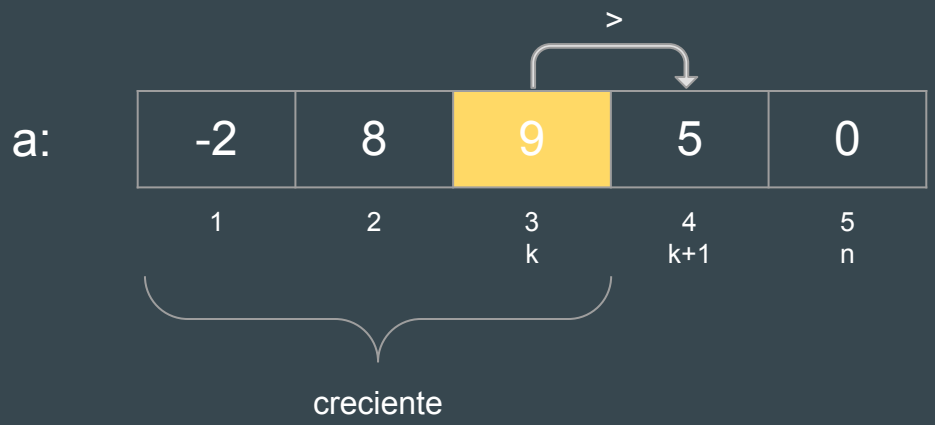
Ejercicio 2b: Ejemplo

```
fun cima(a: array[1..n] of nat) ret k: nat
  k := 1
  do k < n and a[k] < a[k+1] ->
    k := k + 1
  od
end fun
```



Ejercicio 2b: Ejemplo

```
fun cima(a: array[1..n] of nat) ret k: nat
  k := 1
  do k < n and a[k] < a[k+1] ->
    k := k + 1
  od
end fun
```



Ejercicio 2b: Ejemplo

```
fun cima(a: array[1..n] of nat) ret k: nat
  k := 1
  do k < n and a[k] < a[k+1] ->
    k := k + 1
  od
end fun
```

Ejercicio 2c: Repaso de búsqueda binaria

- Búsqueda: Dado un arreglo ordenado **a**, y un valor **x**, encontrar y devolver la posición de **x** en el arreglo (o **0** si no está).

- Búsqueda binaria: Aplicar la idea de **divide y vencerás**. Dado un segmento:
  - Si el segmento es vacío, **x** no está: devolver **0**.
  - Tomar el elemento del medio del segmento.
  - Si es **x**, ya lo encontramos.
  - Si es mayor que **x**, buscar en la primera mitad (llamada recursiva).
  - Si es menor que **x**, buscar en la segunda mitad (llamada recursiva).

- Comenzar con el segmento que comprende todo el arreglo (de **1** a **n**).

- Orden: $\log_2 n$ (o sea, muy rápido).

Ejercicio 2c: Repaso de búsqueda binaria

```
fun binary_search(a: array[1..n] of T, x: T) ret i: nat
  i := binary_search_rec(a, x, 1, n)
end fun


fun binary_search_rec(a: array[1..n] of T, x: T, lft, rgt: nat) ret i: nat
  var mid: nat
  if lft > rgt -> i := 0
     lft <= rgt -> mid := (lft+rgt) ÷ 2
         if x < a[mid] -> i := binary_search_rec(a, x, lft, mid-1)
            x = a[mid] -> i := mid
            x > a[mid] -> i := binary_search_rec(a, x, mid+1, rgt)
         fi
  fi
end fun
```

Ejercicio 2c: Repaso de búsqueda binaria

```
fun binary_search(a: array[1..n] of T, x: T) ret i: nat
  i := binary_search_rec(a, x, 1, n)   <- empezamos por todo el arreglo
end fun


fun binary_search_rec(a: array[1..n] of T, x: T, lft, rgt: nat) ret i: nat
  var mid: nat
  if lft > rgt -> i := 0
     lft <= rgt -> mid := (lft+rgt) ÷ 2  <- elemento del medio
         if x < a[mid] -> i := binary_search_rec(a, x, lft, mid-1)  <- izq
            x = a[mid] -> i := mid       <- lo encontré!
            x > a[mid] -> i := binary_search_rec(a, x, mid+1, rgt)  <- der
         fi
  fi
end fun
```

Ejercicio 2c: Cima con divide y vencerás

- Usemos la misma idea para encontrar la cima.

- Aplicar la idea de **divide y vencerás**. Dado un segmento:
  - Tomar el elemento del medio del segmento: `a[mid]`.
  - Si es la cima, ya está.
  - Si la cima está a la izquierda, buscar en la primera mitad (llamada recursiva).
  - Si la cima está a la derecha, buscar en la segunda mitad (llamada recursiva).

- Comenzar con el segmento que comprende todo el arreglo (de 1 a n).

- Diferencias con búsqueda binaria:
  - Asumimos que la cima siempre existe (nunca se devolverá 0).

Ejercicio 2c: Cima con divide y vencerás

```
fun cima(a: array[1..n] of nat) ret k: nat
  k := cima_rec(a, 1, n)
end fun


fun cima_rec(a: array[1..n] of nat, lft, rgt: nat) ret k: nat
  var mid: nat
  mid := (lft+rgt) ÷ 2
  if es_cima(a, mid)  -> k := mid
     izq_cima(a, mid) -> k := cima_rec(a, lft, mid-1)
     der_cima(a, mid) -> k := cima_rec(a, mid+1, rgt)
  fi
end fun
```

Ejercicio 2c: Cima con divide y vencerás

- Falta definir las tres funciones booleanas de las guardas (predicados):
  - La cima está en la i-ésima posición:

```
fun es_cima(a: array[1..n] of nat, i) ret b: bool
```

  - La cima está a la izquierda de la i-ésima posición:

```
fun izq_cima(a: array[1..n] of nat, i) ret b: bool
```

  - La cima está a la derecha de la i-ésima posición:

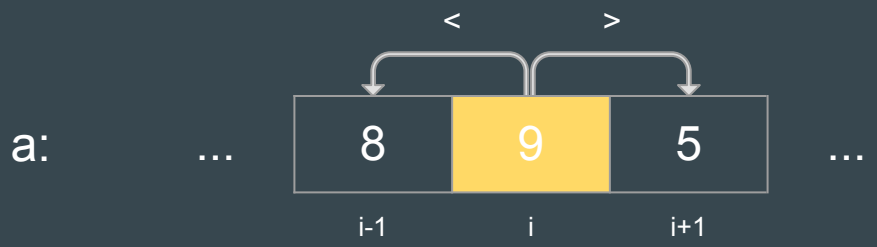
```
fun der_cima(a: array[1..n] of nat, i) ret b: bool
```

Ejercicio 2c: Cima con divide y vencerás

- `es_cima(a, i)`: la cima está en la i-ésima posición.
  - ¿ a[i] es mayor que el elemento anterior y que el siguiente ?

```
b := a[i] > a[i-1] and a[i] > a[i+1]
```

  - **¡Falla en los bordes!**

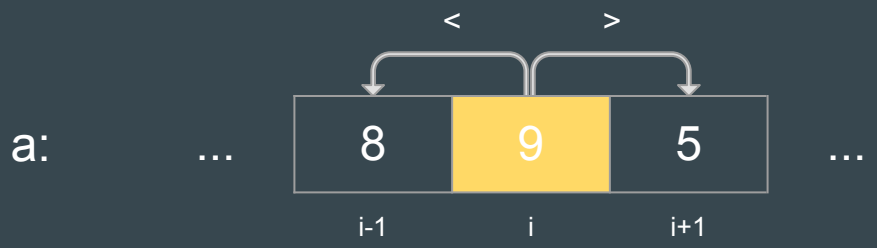


¡es la cima!

Ejercicio 2c: Cima con divide y vencerás

- `es_cima(a, i)`: la cima está en la i-ésima posición.
  - Comparar con anterior/siguiente sólo si existe (versión con if):

```
b := True
if i > 1 ->  b := a[i] > a[i-1]          fi
if i < n ->  b := b and a[i] < a[i+1]    fi
```

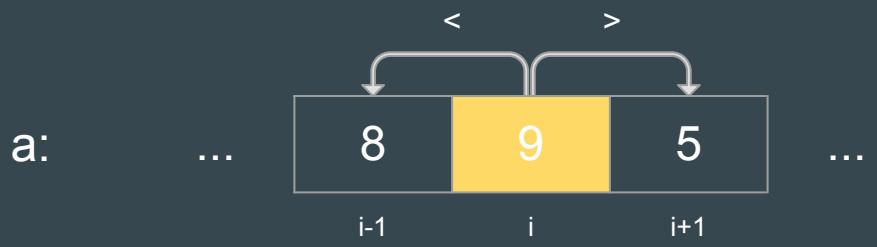


¡es la cima!

Ejercicio 2c: Cima con divide y vencerás

- `es_cima(a, i)`: la cima está en la i-ésima posición.
  - Comparar con anterior/siguiente sólo si existe (versión con or):

```
b := (i = 1 or a[i] > a[i-1]) and (i = n or a[i] > a[i+1])
```

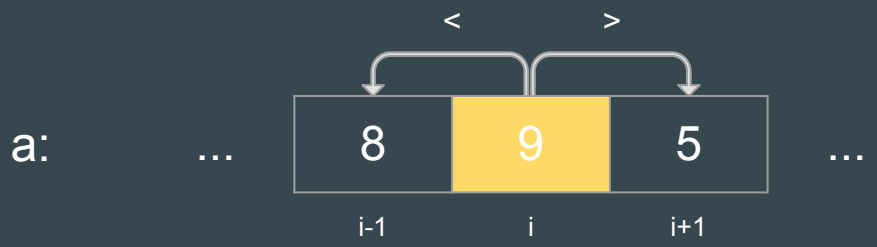


¡es la cima!

Ejercicio 2c: Cima con divide y vencerás

- `es_cima(a, i)`: la cima está en la i-ésima posición.
  - Comparar con anterior/siguiente sólo si existe (versión con or):

```
b := (i = 1 or a[i] > a[i-1]) and (i = n or a[i] > a[i+1])
       False          True           False          True
```



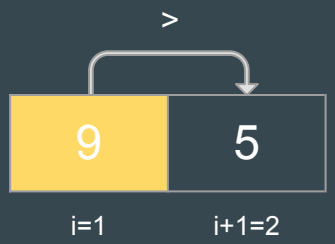
¡es la cima!

Ejercicio 2c: Cima con divide y vencerás

- `es_cima(a, i)`: la cima está en la i-ésima posición.
  - Comparar con anterior/siguiente sólo si existe (versión con or):

```
b := (i = 1 or a[i] > a[i-1]) and (i = n or a[i] > a[i+1])
       True        (no se evalúa)          False          True
```

a:



...

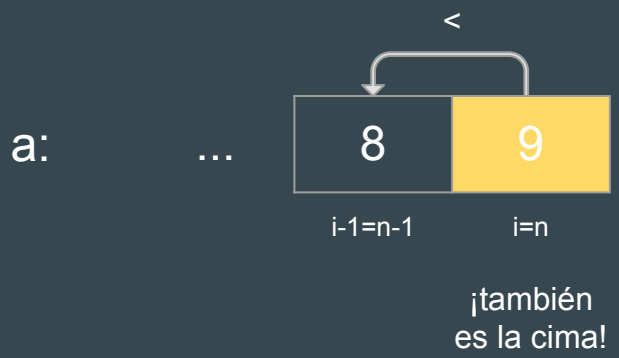
i=1 i+1=2

¡también
es la cima!

Ejercicio 2c: Cima con divide y vencerás

- `es_cima(a, i)`: la cima está en la i-ésima posición.
  - Comparar con anterior/siguiente sólo si existe (versión con or):

```
b := (i = 1 or a[i] > a[i-1]) and (i = n or a[i] > a[i+1])
       False          True          True       (no se evalúa)
```



Ejercicio 2c: Cima con divide y vencerás

- `es_cima(a, i)`: la cima está en la i-ésima posición.
  - Comparar con anterior/siguiente sólo si existe (versión con or):

```
b := (i = 1 or a[i] > a[i-1]) and (i = n or a[i] > a[i+1])
       True        (no se evalúa)          True        (no se evalúa)
```

a:

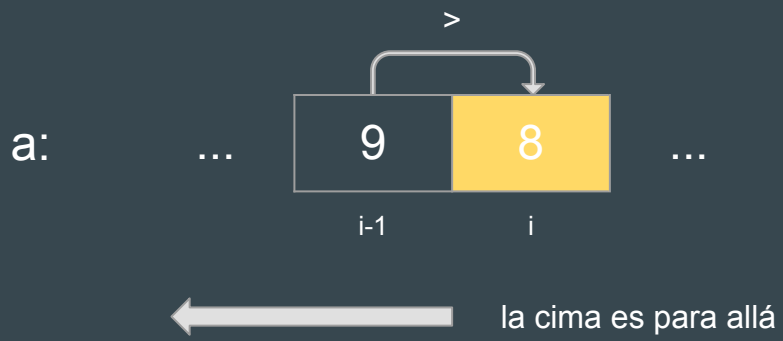
i=1=n

¡también
es la cima!

Ejercicio 2c: Cima con divide y vencerás

- `izq_cima(a, i)`: la cima está a la izquierda de la i-ésima posición.
  - Hay elementos a la izquierda, y el arreglo está decreciendo:

```
b := i > 1 and a[i-1] > a[i]
```



Ejercicio 2c: Cima con divide y vencerás

- `der_cima(a, i)`: la cima está a la derecha de la i-ésima posición.
  - Hay elementos a la derecha, y el arreglo está creciendo:

```
b := i < n and a[i] < a[i+1]
```

Ejercicio 2c: Resultado Final 1/2

```
fun cima(a: array[1..n] of nat) ret k: nat
  k := cima_rec(a, 1, n)
end fun


fun cima_rec(a: array[1..n] of nat, lft, rgt: nat) ret k: nat
  var mid: nat
  mid := (lft+rgt) ÷ 2
  if es_cima(a, mid)  -> k := mid
     izq_cima(a, mid) -> k := cima_rec(a, lft, mid-1)
     der_cima(a, mid) -> k := cima_rec(a, mid+1, rgt)
  fi
end fun
```

Ejercicio 2c: Resultado Final 2/2

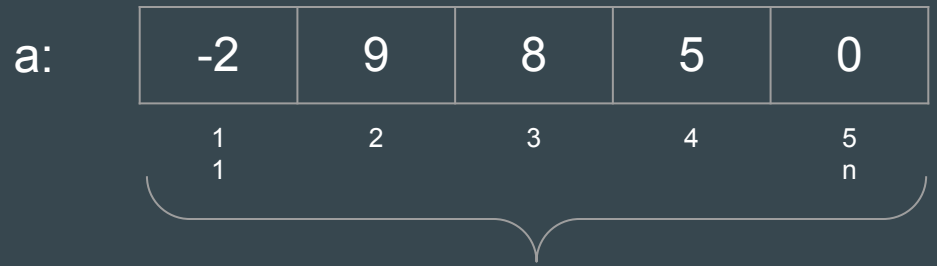
```
fun es_cima(a: array[1..n] of nat, i) ret b: bool
  b := (i = 1 or a[i] > a[i-1]) and (i = n or a[i] > a[i+1])
end fun


fun izq_cima(a: array[1..n] of nat, i) ret b: bool
  b := i > 1 and a[i-1] > a[i]
end fun


fun der_cima(a: array[1..n] of nat, i) ret b: bool
  b := i < n and a[i] < a[i+1]
end fun
```

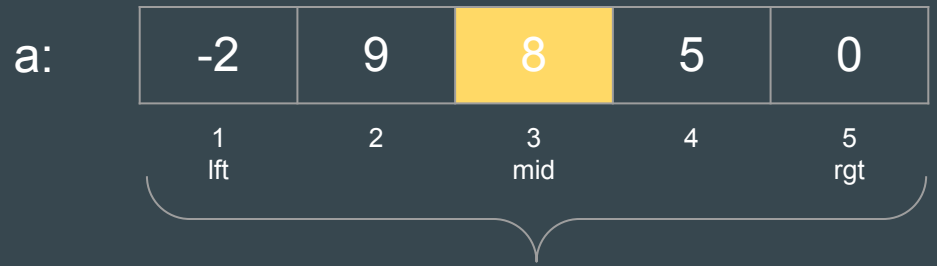
Ejercicio 2c: Ejemplo

```
fun cima(a: array[1..n] of nat) ret k: nat
  k := cima_rec(a, 1, n)
end fun
```



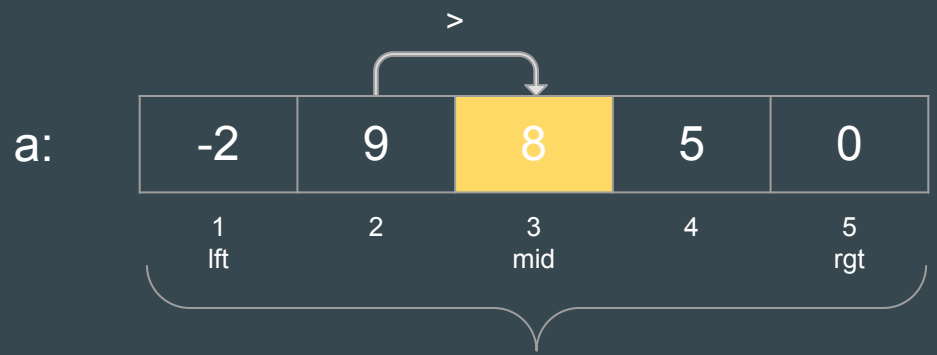
Ejercicio 2c: Ejemplo

```
fun cima_rec(a: array[1..n] of nat, lft, rgt: nat) ret k: nat
  var mid: nat
  mid := (lft+rgt) ÷ 2
  if es_cima(a, mid)  -> k := mid
     izq_cima(a, mid) -> k := cima_rec(a, lft, mid-1)
     der_cima(a, mid) -> k := cima_rec(a, mid+1, rgt)
  fi
end fun
```



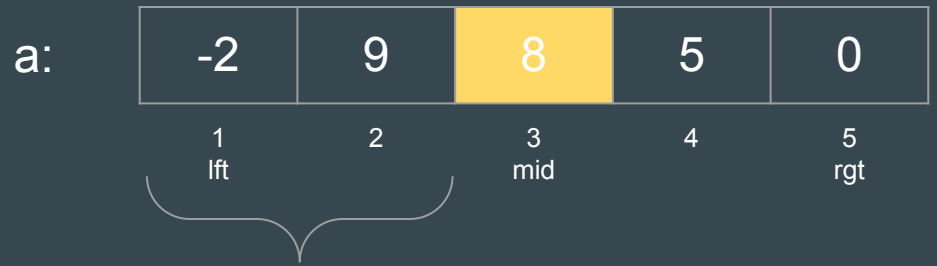
Ejercicio 2c: Ejemplo

```
fun cima_rec(a: array[1..n] of nat, lft, rgt: nat) ret k: nat
  var mid: nat
  mid := (lft+rgt) ÷ 2
  if es_cima(a, mid)  -> k := mid                          <- False
     izq_cima(a, mid) -> k := cima_rec(a, lft, mid-1)      <- True
     der_cima(a, mid) -> k := cima_rec(a, mid+1, rgt)      <- False
  fi
end fun
```



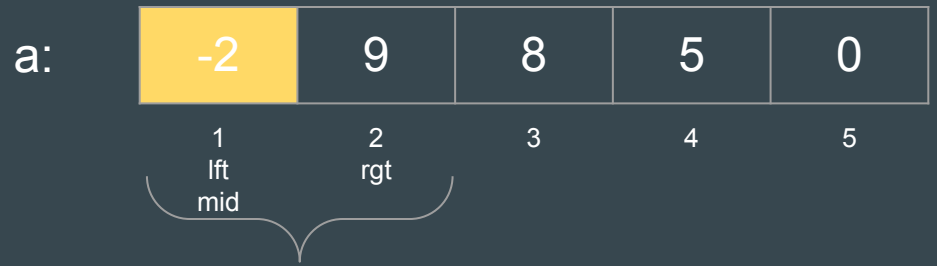
Ejercicio 2c: Ejemplo

```
fun cima_rec(a: array[1..n] of nat, lft, rgt: nat) ret k: nat
  var mid: nat
  mid := (lft+rgt) ÷ 2
  if es_cima(a, mid)  -> k := mid                          <- False
     izq_cima(a, mid) -> k := cima_rec(a, lft, mid-1)      <- True
     der_cima(a, mid) -> k := cima_rec(a, mid+1, rgt)      <- False
  fi
end fun
```



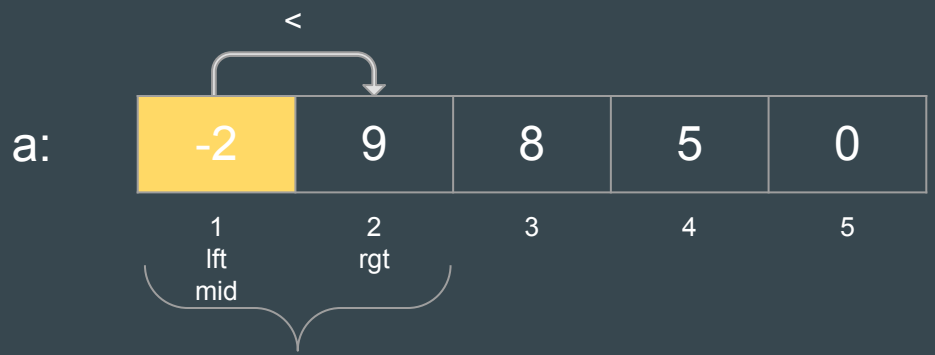
Ejercicio 2c: Ejemplo

```
fun cima_rec(a: array[1..n] of nat, lft, rgt: nat) ret k: nat
  var mid: nat
  mid := (lft+rgt) ÷ 2
  if es_cima(a, mid)  -> k := mid
     izq_cima(a, mid) -> k := cima_rec(a, lft, mid-1)
     der_cima(a, mid) -> k := cima_rec(a, mid+1, rgt)
  fi
end fun
```



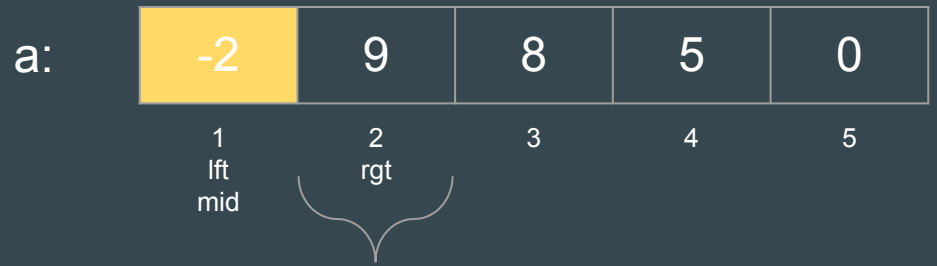
Ejercicio 2c: Ejemplo

```
fun cima_rec(a: array[1..n] of nat, lft, rgt: nat) ret k: nat
  var mid: nat
  mid := (lft+rgt) ÷ 2
  if es_cima(a, mid)  -> k := mid                          <- False
     izq_cima(a, mid) -> k := cima_rec(a, lft, mid-1)      <- False
     der_cima(a, mid) -> k := cima_rec(a, mid+1, rgt)      <- True
  fi
end fun
```



Ejercicio 2c: Ejemplo

```
fun cima_rec(a: array[1..n] of nat, lft, rgt: nat) ret k: nat
  var mid: nat
  mid := (lft+rgt) ÷ 2
  if es_cima(a, mid)  -> k := mid                          <- False
     izq_cima(a, mid) -> k := cima_rec(a, lft, mid-1)      <- False
     der_cima(a, mid) -> k := cima_rec(a, mid+1, rgt)      <- True
  fi
end fun
```



Ejercicio 2c: Ejemplo

```
fun cima_rec(a: array[1..n] of nat, lft, rgt: nat) ret k: nat
  var mid: nat
  mid := (lft+rgt) ÷ 2
  if es_cima(a, mid)   -> k := mid                          <- True
     izq_cima(a, mid) -> k := cima_rec(a, lft, mid-1)       <- False
     der_cima(a, mid) -> k := cima_rec(a, mid+1, rgt)       <- False
  fi
end fun
```

Ejercicio 2c: Otra versión con divide y vencerás

- Dado un segmento:
  - Si el segmento es de tamaño 1, el único elemento es la cima.
  - Si es más grande, tomar **dos elementos consecutivos** del medio: `a[mid]` y `a[mid+1]`.
    - Si `a[mid] > a[mid+1]`, buscar en la primera mitad del segmento (llamada recursiva).
    - Si `a[mid] < a[mid+1]`, buscar en la segunda mitad del segmento (llamada recursiva).

- Comenzar con el segmento que comprende todo el arreglo (de 1 a n).

- Diferencias con la versión anterior:
  - Predicados más sencillos en las guardas.
  - No termina hasta llegar a un segmento de largo 1.

Ejercicio 2c: Otra versión

```
fun cima(a: array[1..n] of nat) ret k: nat
  k := cima_rec(a, 1, n)
end fun

fun cima_rec(a: array[1..n] of nat, lft, rgt: nat) ret k: nat
  var mid: nat
  if lft = rgt -> k := lft
  else
    mid := (lft+rgt) ÷ 2
    if a[mid] < a[mid+1] -> k := cima_rec(a, mid+1, rgt)
    else                 -> k := cima_rec(a, lft, mid)    <- ojo acá
    fi
  fi
end fun
```

Ejercicio 2: Consideraciones finales

- Ejercicio 2b: Usa idea de la búsqueda secuencial.
  - Una iteración por elemento del arreglo en el peor caso: **`n`**.
    - (la mitad en caso medio: **`n / 2`**).
  - Orden lineal: proporcional a **`n`** .
  - Ejemplo: Para **500** elementos se realizarán **500** pasos como mucho.

- Ejercicio 2c: Usa idea de la búsqueda binaria.
  - Se divide por dos el tamaño del problema en cada recursión.
  - Orden logarítmico: proporcional a $\log_2 n$ (o sea, muy rápido).
  - Ejemplo: Para **500** elementos, se realizarán **9** pasos como mucho:
    - Segmentos de tamaño **500, 250, 125, 62, 31, 15, 7, 3, 1.**

¡Gracias!