TEMA 3

Secuencias

Emma Rollón
erollon@cs.upc.edu

Departamento de Ciencias de la Computación

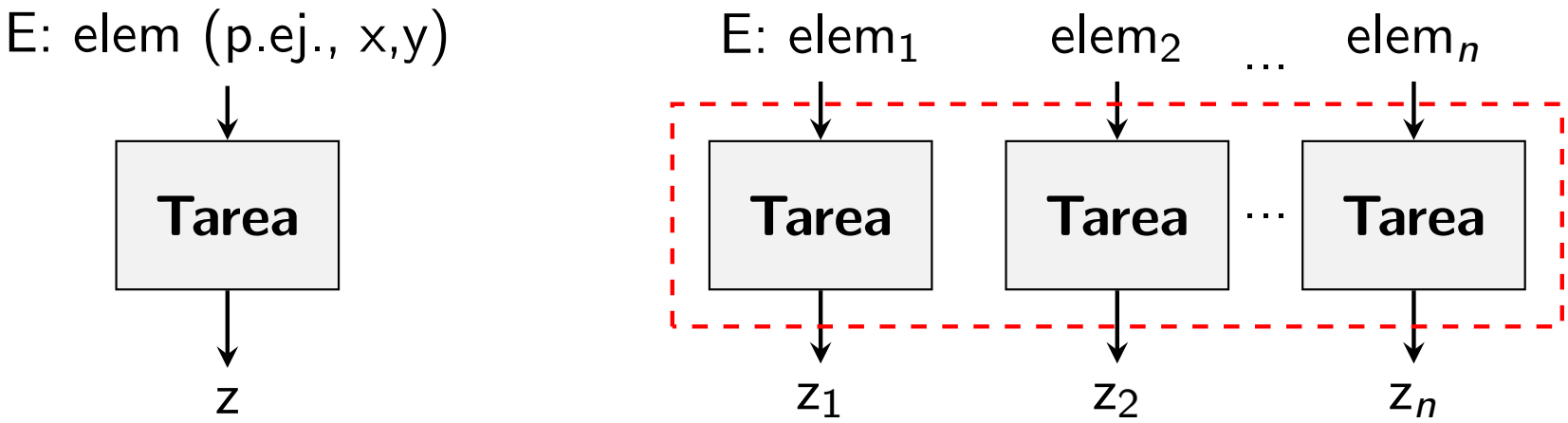
Índice

Introducción: Qué es una secuencia?

Una **secuencia** es un conjunto de elementos de un tipo de datos determinado.

Hasta ahora:

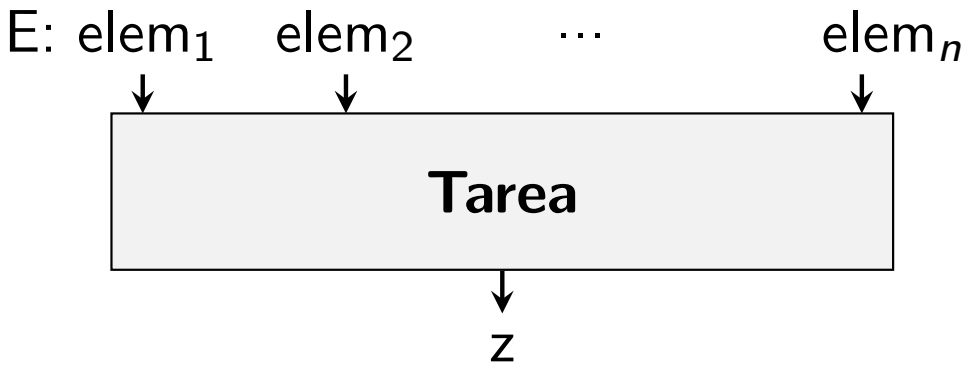
- Dado un elemento, realizar una tarea sobre él (fíjate que un elemento puede ser más de un valor).
- Dada una secuencia, realizar una tarea sobre cada uno de sus elementos (de forma independiente uno de otro).



Introducción: Qué es una secuencia?

A partir de ahora:

- Dada una secuencia, realizar una tarea sobre TODOS sus elementos.



Quiere decir esto que tenemos que almacenar todos los elementos de la secuencia? $\Rightarrow$ NO, sólo necesitaremos mantener cierta información.

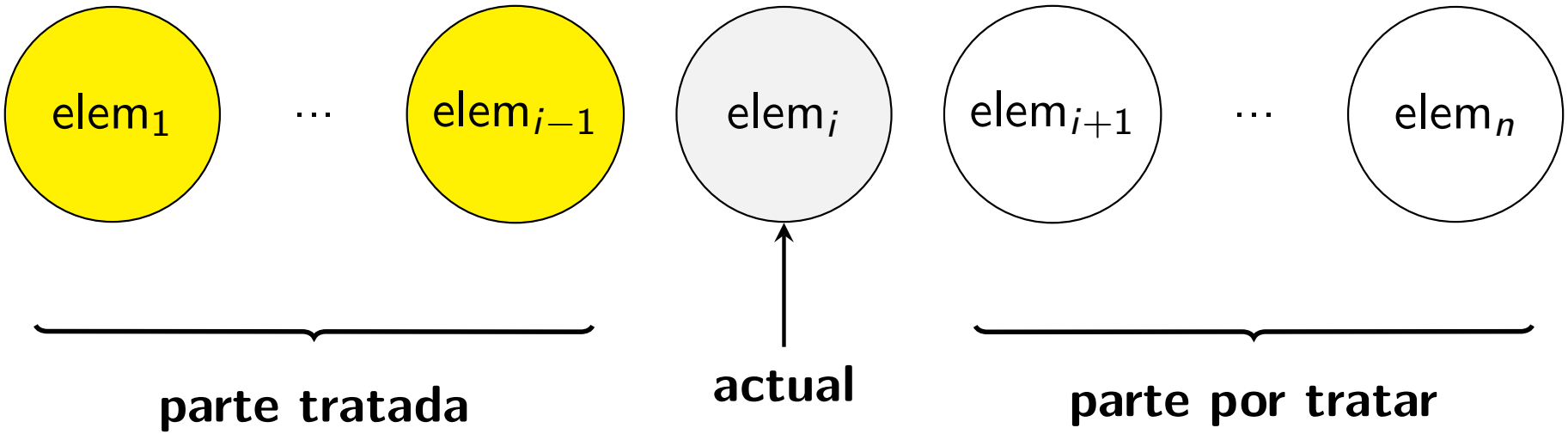
Determinar qué información (valores) necesitamos y, sobre todo, entender por qué los necesitamos y cómo los mantenemos es lo que veremos a partir de ahora.

Introducción: Elementos clave

Las secuencias las trataré con un bucle en el que:

- En cada iteración, trataré un nuevo elemento. Ese elemento es clave, y se denomina elemento actual.
- Todo lo que esté a la izquierda de ese elemento, será la parte tratada de la secuencia.
- Todo lo que esté a la derecha de ese elemento, será la parte por tratar de la secuencia.

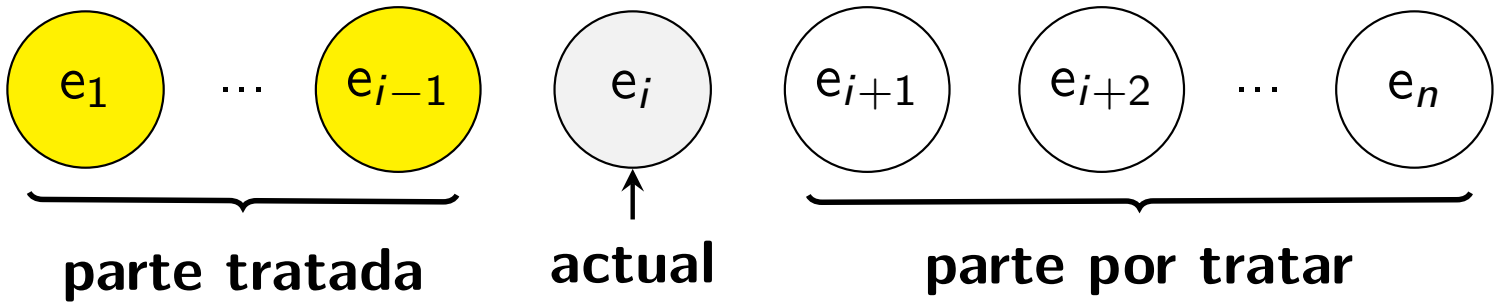
Visualmente:



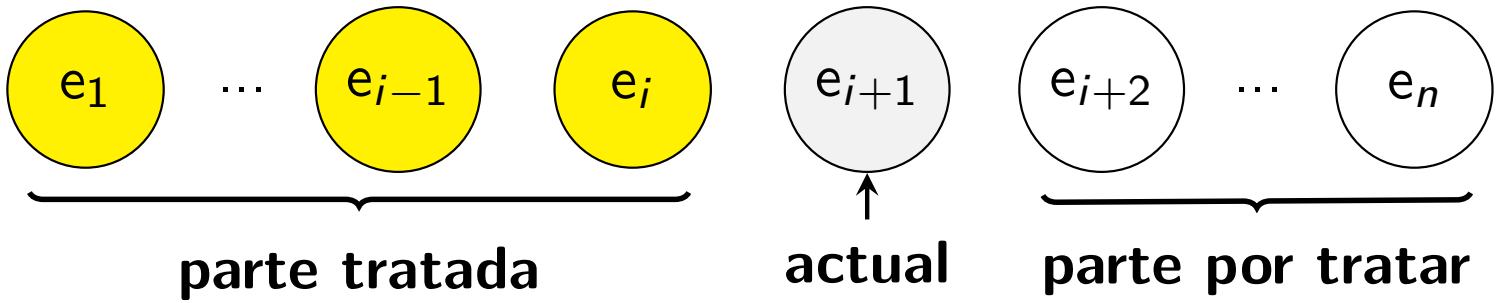
Introducción: Elementos clave

Cómo varían esos elementos de una iteración a la siguiente?

Iteración i:



Iteración i + 1:

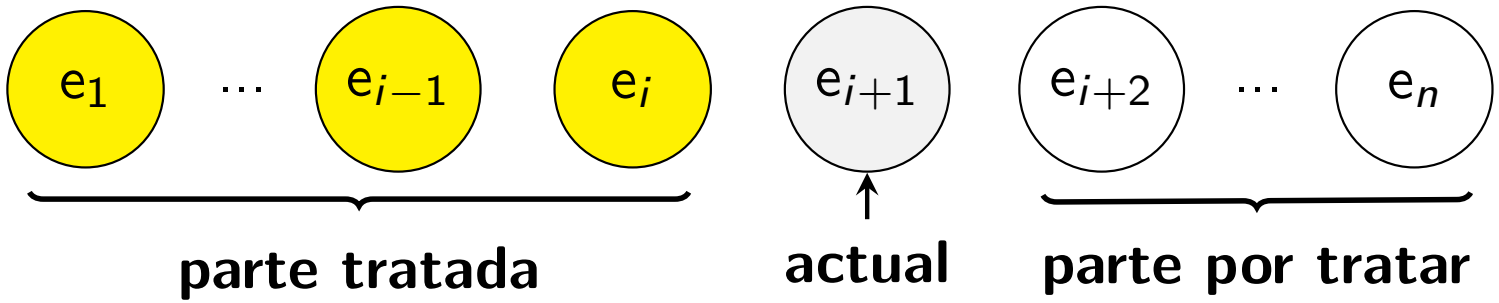


Es decir, el elemento actual en la iteración i pasa a estar incluído en la parte tratada en la iteración $i + 1$.

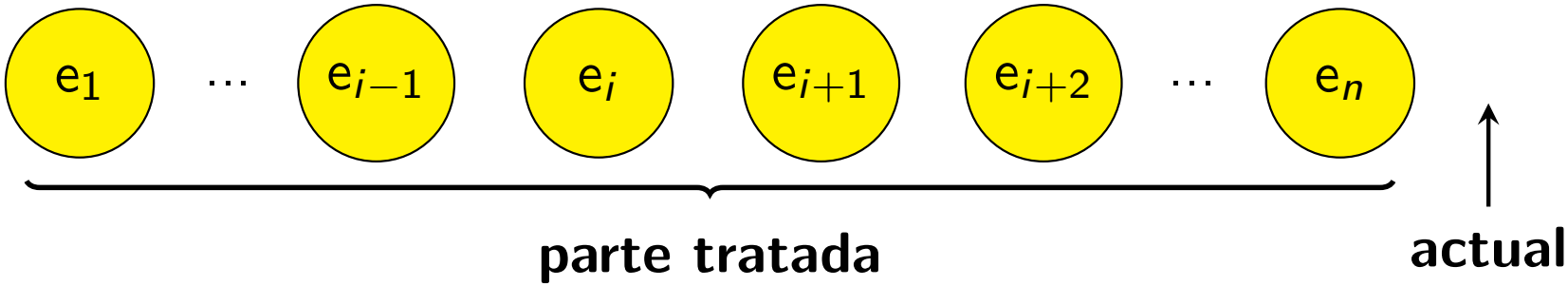
Introducción: Elementos clave

Por tanto, cuál será la imagen al salir del bucle?

Iteración i + 1:



Al salir del bucle:



Es decir, todos los elementos de la secuencia están incluídos en la parte tratada ⇒ Hemos tratado toda la secuencia.

Introducción: Razonamiento sobre elementos clave

Para resolver un problema tenemos que pensar:

A. Qué información necesito mantener de la parte tratada de la secuencia para que, un vez la he tratado *toda*, pueda resolver la tarea planteada?

B. Cómo actualizo esa información para que el elemento actual pase a estar incluído en la parte tratada? (esas instrucciones son el cuerpo del bucle)

C. Cómo detecto que se ha tratado toda la secuencia? (esa es la condición del bucle)

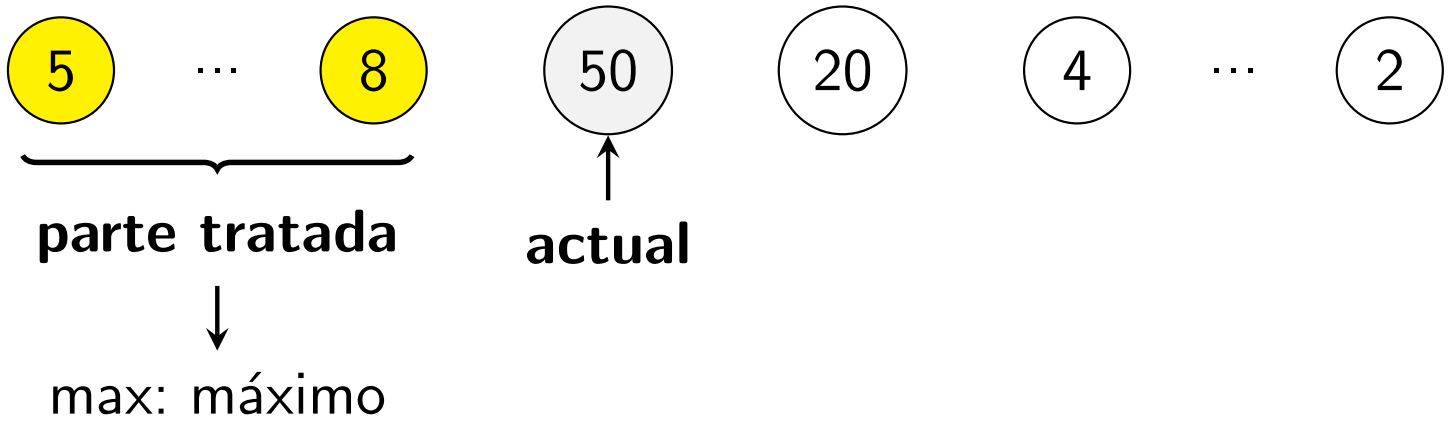
D. Cómo inicializo esa información manteniendo su significado? (instrucciones antes del bucle)

Introducción: Razonamiento sobre elementos clave

Ejemplo 1:

Dada una secuencia de naturales no vacía, escribir su valor máximo.

A. Qué información necesito mantener de la parte tratada?

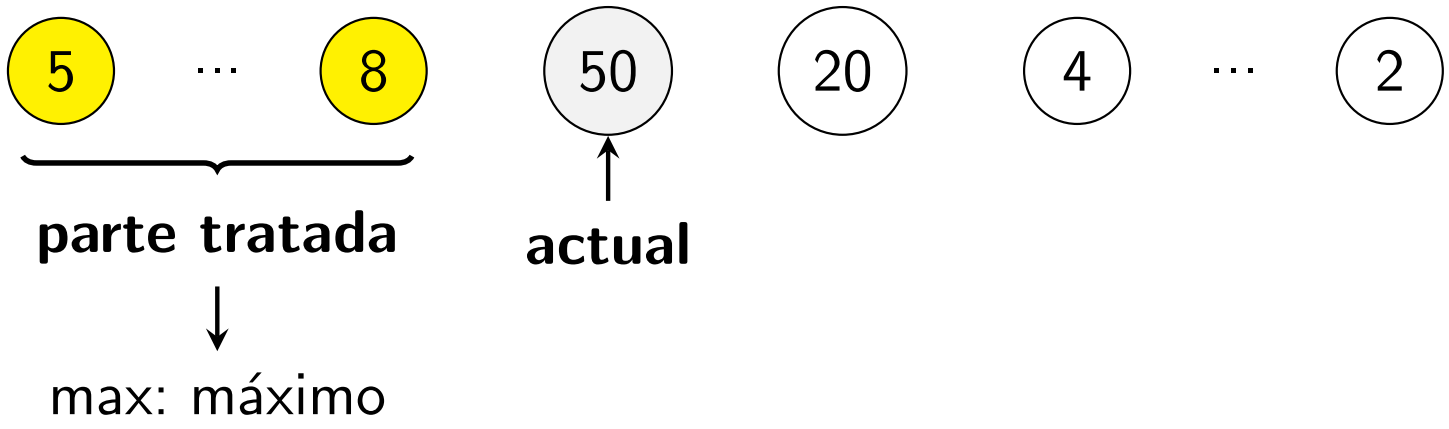


Si soy capaz de mantener esa variable max con el significado máximo de la parte tratada en todas y cada una de las iteraciones, cuando la parte tratada incluya toda la secuencia, ese valor max será "máximo de toda la secuencia", que será lo que tengo que escribir.

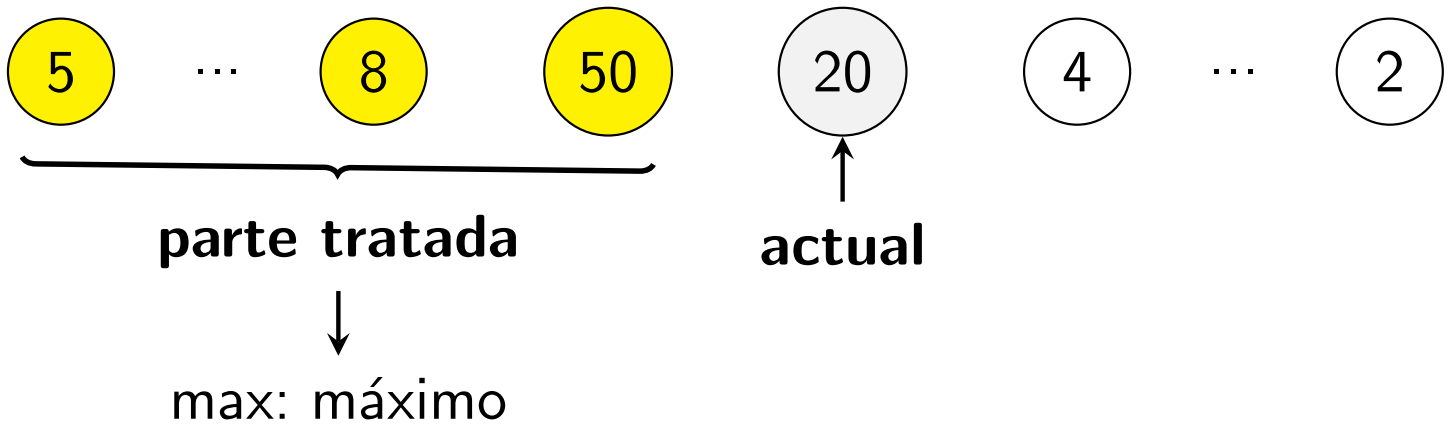
Introducción: Razonamiento sobre elementos clave

B. Cómo actualizo max manteniendo significado de una iter. a otra?

Iteración i:



Iteración $i + 1$:



Introducción: Razonamiento sobre elementos clave

```
int main() {
    // Inicialización
    ...;
    while (...) {
        if (actual > max) max = actual;
        cin >> actual;
    }
    // He tratado toda la secuencia
    cout << max << endl;
}
```

C. Cómo detecto que se ha tratado toda la secuencia?

La condición del bucle detectará si tengo más elementos a tratar o no. Fíjate que como en la entrada directamente me dan la secuencia, esa información me la da la instrucción cin que lee el nuevo elemento actual. Por tanto:

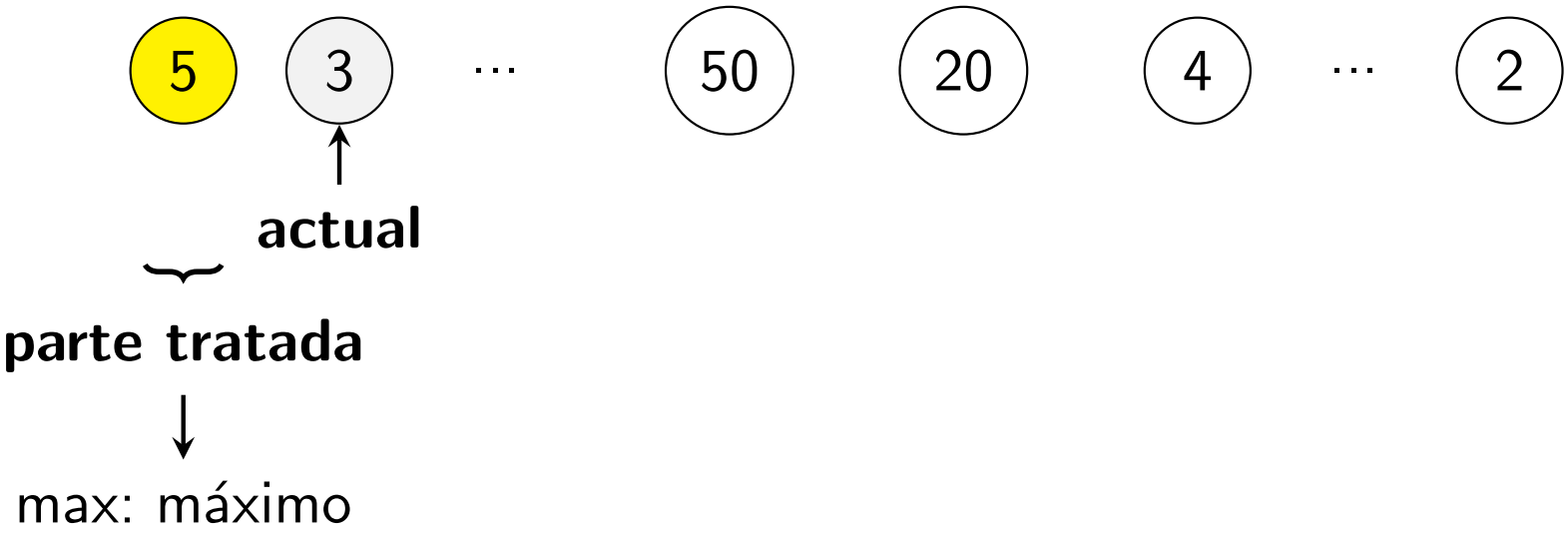
```
int main() {
    // Inicialización
    ...;
    while (cin >> actual) {
        if (actual > max) max = actual;
    }
    // He tratado toda la secuencia
    cout << max << endl;
}
```

Introducción: Razonamiento sobre elementos clave

D. Cómo inicializo max y actual?

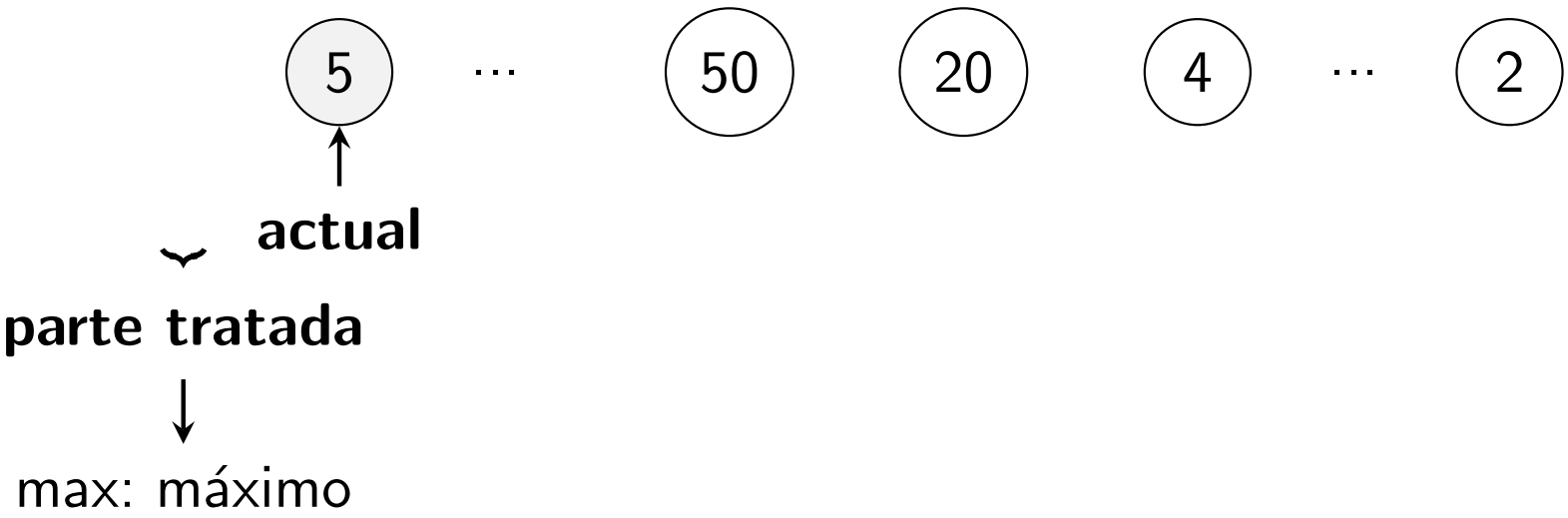
- La inicialización del actual me lo da la propia condición del bucle.
- Tengo dos opciones para inicializar correctamente max:

Opción 1: Como el enunciado dice que la secuencia no puede ser vacía, le daré a max el valor del primer elemento, y el bucle tratará a partir del segundo elemento de la secuencia.



Introducción: Razonamiento sobre elementos clave

Opción 2: Si quiero tratar el primer elemento de la secuencia en la primera iteración del bucle, la parte tratada hasta ese momento es una secuencia vacía.



max debe ser el máximo de una secuencia vacía (un valor que por definición signifique eso). Además, ese valor tiene que asegurar que en la primera iteración del bucle se produzca la asignación max = actual.

Introducción: Razonamiento sobre elementos clave

Opción 1:

```
int main() {
    // Inicialización
    int max;
    cin >> max;
    int actual;
    while (cin >> actual) {
        if (actual > max)
            max = actual;
    }
    // He tratado toda
    // la secuencia
    cout << max << endl;
}
```

Opción 2:

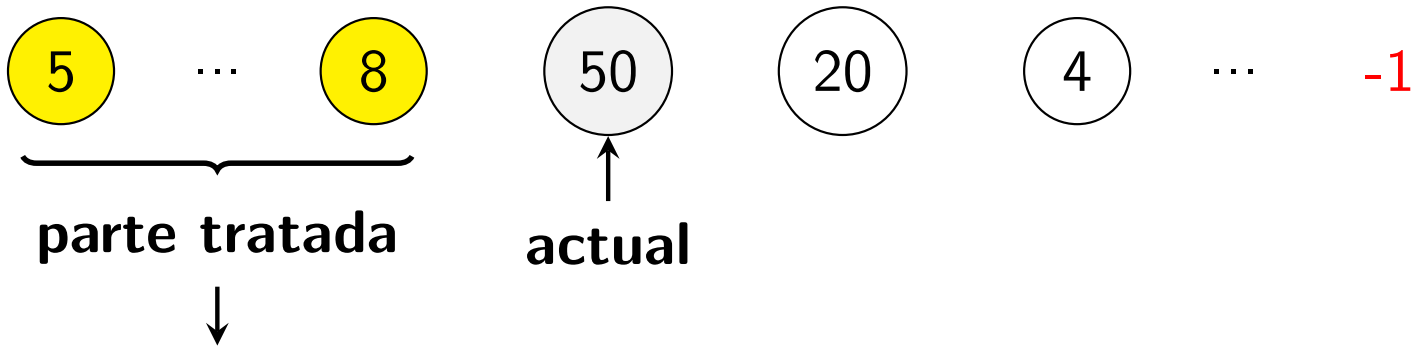
```
int main() {
    // Inicialización
    int max = -1;
    int actual;
    while (cin >> actual) {
        if (actual > max)
            max = actual;
    }
    // He tratado toda
    // la secuencia
    cout << max << endl;
}
```

Introducción: Razonamiento sobre elementos clave

Ejemplo 2:

Dada una secuencia de naturales acabada en -1, escribir cuántas veces un número es el doble del anterior.

A. Qué información necesito mantener de la parte tratada?

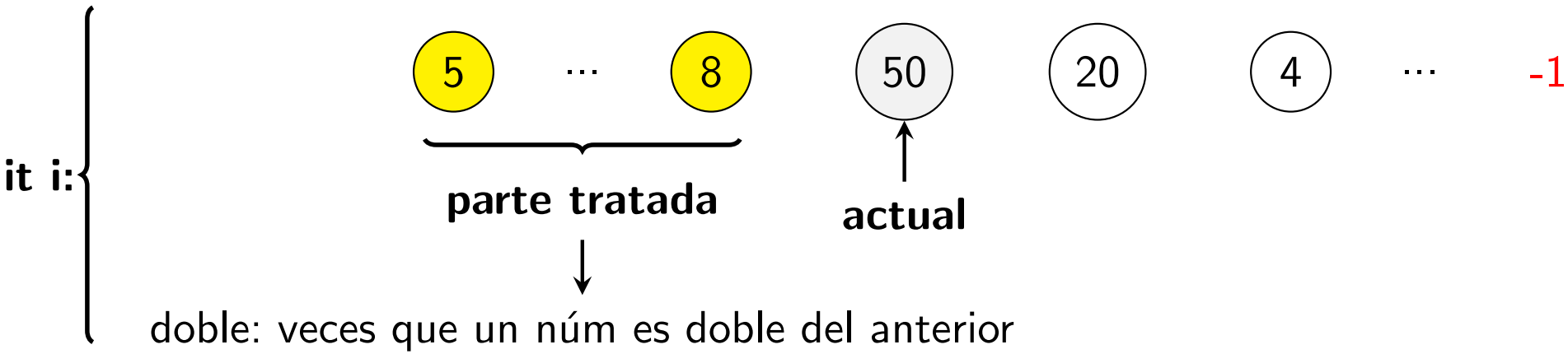


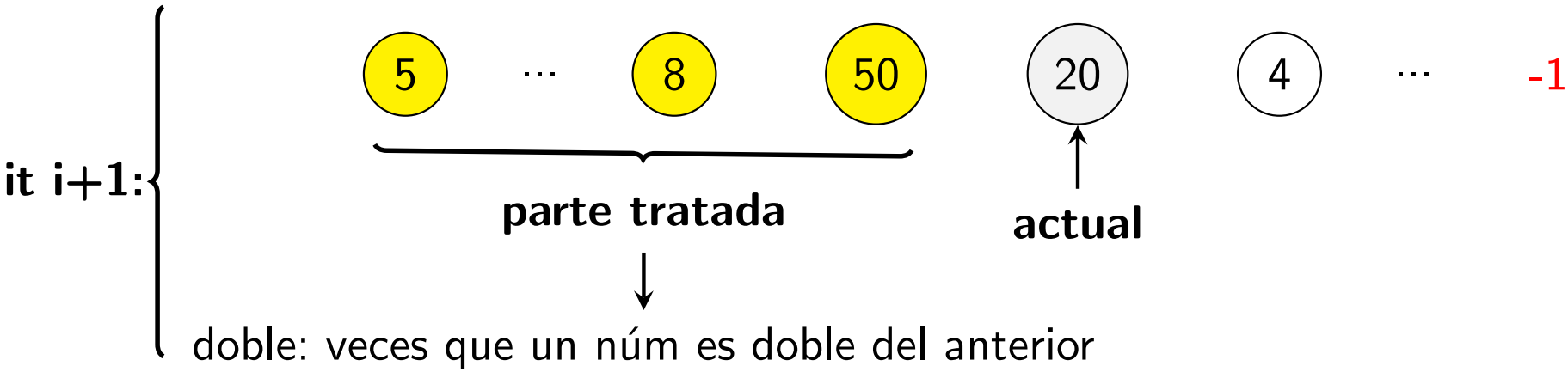
doble: veces que un núm es doble del anterior

Si soy capaz de mantener esa variable doble con el significado veces que un núm es el doble del anterior en todas y cada una de las iteraciones, cuando la parte tratada incluya toda la secuencia, ese valor será lo que tengo que escribir.

Introducción: Razonamiento sobre elementos clave

B. Actualización?

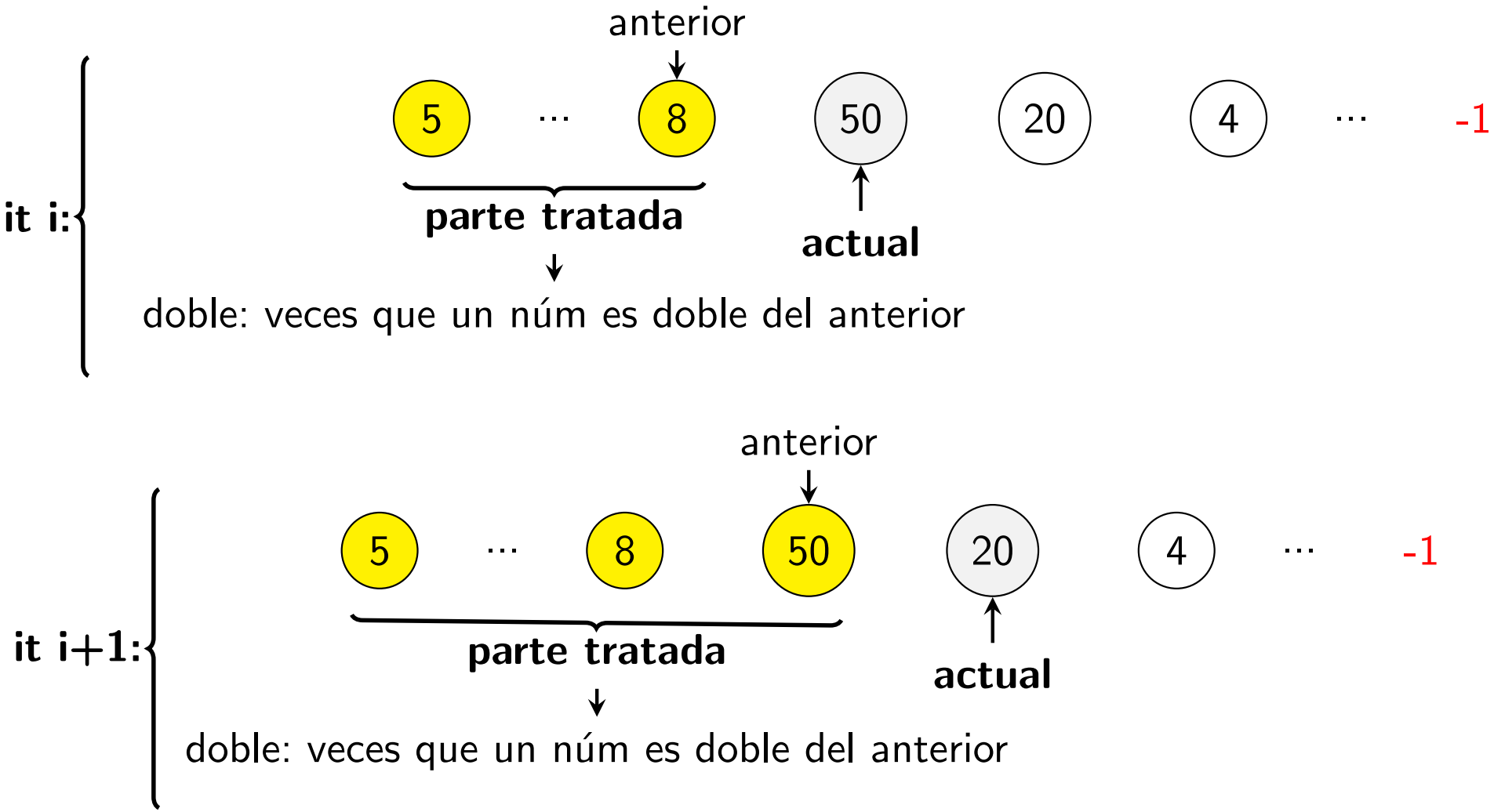




Para actualizar doble necesito saber si actual es el doble de su anterior!

Introducción: Razonamiento sobre elementos clave

B. Actualización?



Introducción: Razonamiento sobre elementos clave

```
int main() {
    // Inicialización
    ...;
    while (...) {
        if (actual == anterior*2) ++doble;
        anterior = actual;
        cin >> actual;
    }
    cout << doble << endl;
}
```

C. Condición del bucle?

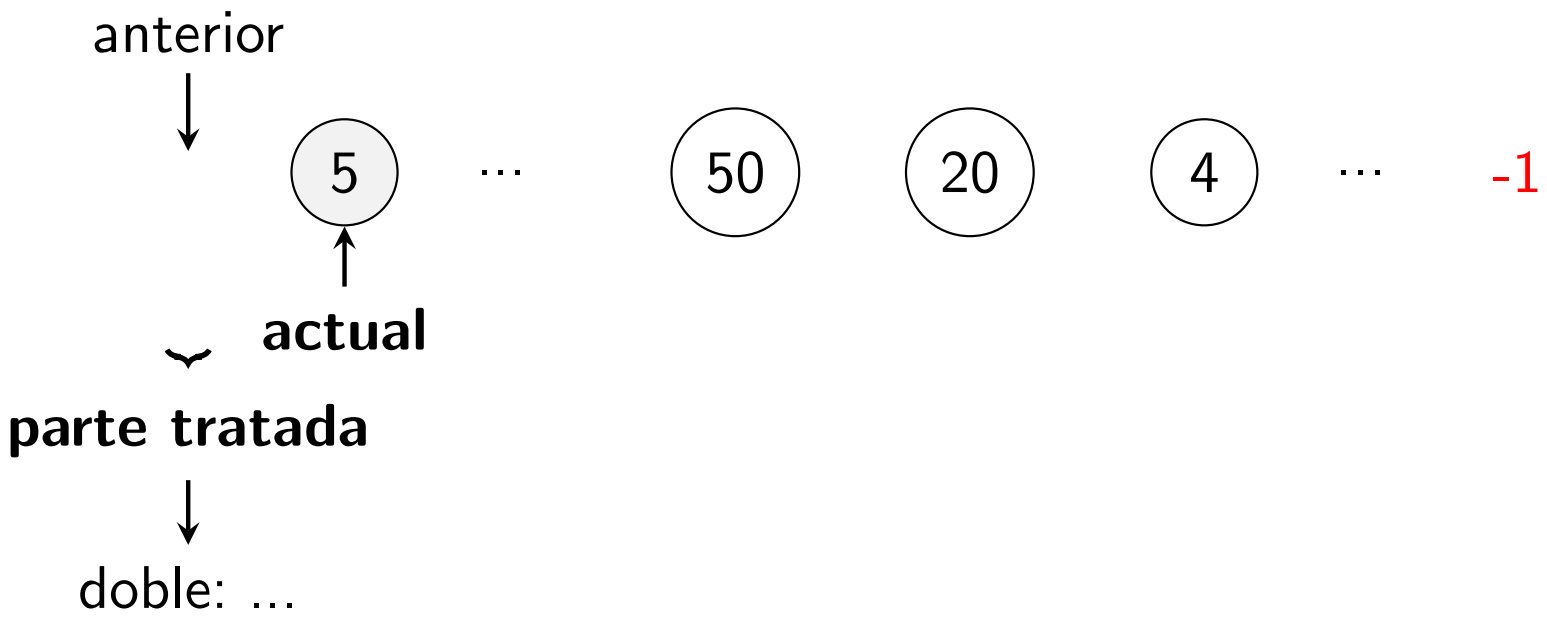
Como el final de la secuencia se indica con el centinela -1, sabré si tengo más elementos a tratar o no en función del valor de actual. Será final de secuencia cuando el actual valga -1.

```
int main() {
    // Inicialización
    ...;
    while (actual != -1) {
        if (actual == anterior*2) ++doble;
        anterior = actual;
        cin >> actual;
    }
    cout << doble << endl;
}
```

Introducción: Razonamiento sobre elementos clave

D. Inicialización?

El 1^r elem se ha de tratar en la 1^a iteración del bucle (sec. vacía válida):



- doble: número de veces que un número es el doble de su anterior en una secuencia vacía (la parte tratada en ese momento).
- anterior: un valor tal que no actualice doble en la primera iteración.
- actual: primer valor de la entrada (elemento o centinela).

Introducción: Razonamiento sobre elementos clave

```
int main() {
    // Inicialización
    int doble = 0;
    int anterior = -1;
    int actual;
    cin >> actual;
    while (actual != -1) {
        if (actual == anterior*2) ++doble;
        anterior = actual;
        cin >> actual;
    }
    cout << doble << endl;
}
```

Observación

Fíjate que el razonamiento concuerda con la forma que ya vimos en la que se hace la lectura cuando la secuencia acaba en centinela.

Esquemas algorítmicos sobre secuencias

Dada una secuencia de naturales:

- Tarea 1: contar el número de elementos pares
- Tarea 2: decir si contiene algún número par

Observación 1

Para solucionar la Tarea 1, tengo que visitar necesariamente TODOS los elementos de la secuencia. Si no lo hago, no podré solucionarla $\Rightarrow$ RECORRIDO.

Observación 2

Para solucionar la Tarea 2, puede que no necesite visitar todos los elementos: en el momento en que encuentro uno que es par, ya puedo solucionar el problema. Eso sí, para demostrar que todos son impares (no hay ninguno par) entonces sí que tendré que visitarlos todos $\Rightarrow$ BÚSQUEDA.

Esquemas algorítmicos sobre secuencias: Recorrido

El algoritmo será un recorrido cuando tiene que visitar necesariamente TODOS los elementos de la secuencia.

Ejemplos:

- Dada una secuencia de naturales, escribir su valor máximo.
- Dada una secuencia de naturales acabada en -1, escribir cuántas veces un número es el doble del anterior.
- Dada una secuencia de naturales, contar el número de pares.
- (...)
- Básicamente todos los ejemplos que hemos visto hasta ahora.

Esquema:

```
inicializar;
while (not fin_secuencia) {
    consultar_actual;
    tratar_actual;
    avanzar;
}
```

Esquemas algorítmicos sobre secuencias: Búsqueda

El algoritmo será una búsqueda cuando su objetivo sea buscar algo y, por tanto, puede parar de tratar elementos tan pronto como lo encuentra.

Ejemplos:

- Dada una secuencia de naturales, decir si contiene algún número par
- Dada una letra y una frase acabada en punto, decir si la frase contiene esa letra.
- Dado una palabra y un diccionario (secuencia de palabras ordenadas lexicográficamente), decir si la palabra está en el diccionario.
- (...)

Esquema:

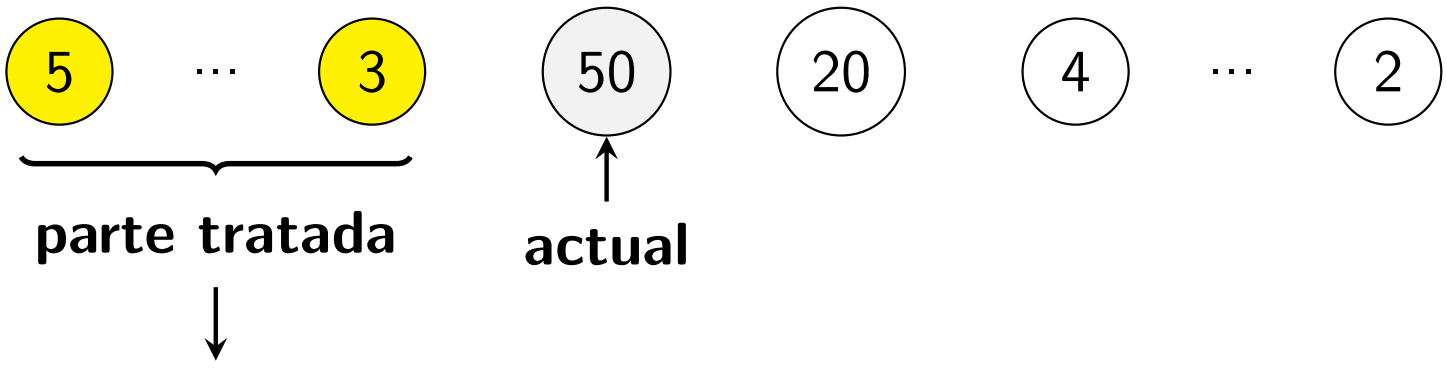
```
inicializar;
bool encontrado = false;
while (not encontrado and not fin_secuencia) {
    consultar_actual;
    if (propiedad(actual)) encontrado = true;
    avanzar;
}
```

Esquemas algorítmicos sobre secuencias: Búsqueda

Ejemplo 1:

Dada una secuencia de naturales, decir si contine algún número par.

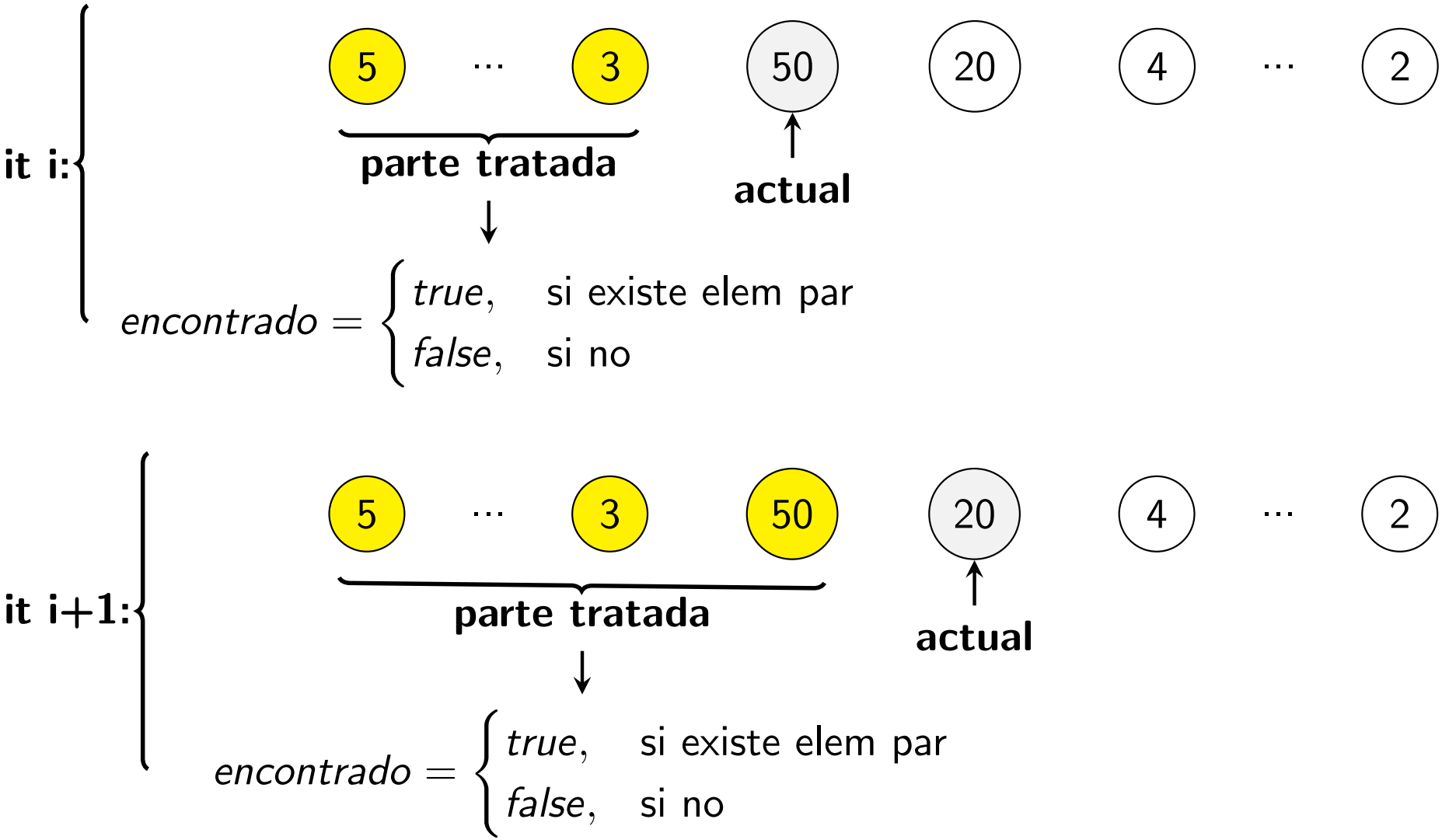
A. Información que necesito almacenar de la parte tratada?



$$encontrado = \begin{cases} true, & \text{si existe elem par} \\ false, & \text{si no} \end{cases}$$

Esquemas algorítmicos sobre secuencias: Búsqueda

B. Actualización?



Esquemas algorítimos sobre secuencias: Búsqueda

```
// Inicializar
...;
bool encontrado = false;
while (not encontrado and ...) {
    if (actual%2 == 0) encontrado = true;
    cin >> actual;
}
if (encontrado) cout << "SI" << endl;
else cout << "NO" << endl;
```

C. Condición del bucle?

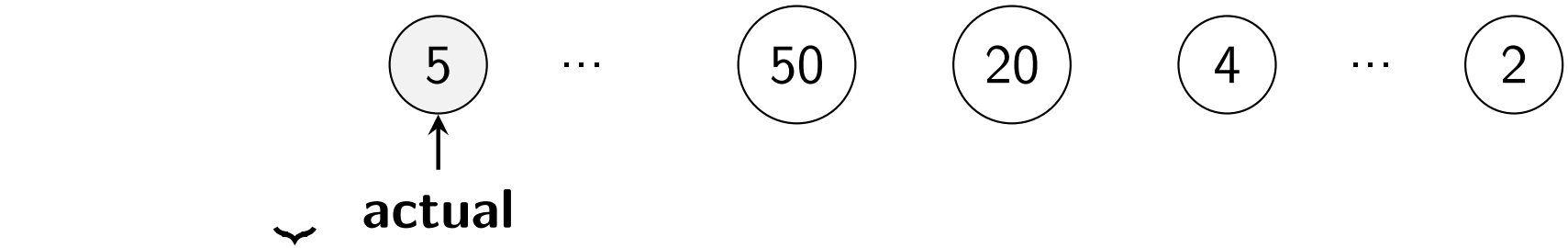
Cómo sabré que ya he tratado todos los elementos de la secuencia? Fíjate que como en la entrada directamente me dan la secuencia, el propio cin que avanza en la secuencia indica si existe un nuevo elemento o no. Por tanto:

```
// Inicializar
...;
bool encontrado = false;
while (not encontrado and cin >> actual) {
    if (actual%2 == 0) encontrado = true;
}
if (encontrado) cout << "SI" << endl;
else cout << "NO" << endl;
```

Esquemas algorítimos sobre secuencias: Búsqueda

D. Inicialización?

La secuencia puede ser vacía, así es que el primer elemento se ha de tratar en la primera iteración del bucle.



$$encontrado = \begin{cases} true, & \text{si existe elem par} \\ false, & \text{si no} \end{cases}$$

- encontrado: si la parte tratada es una secuencia vacía, existirá un elemento par en ella?
- actual : la propia condición del bucle se encarga de hacer esa lectura.

Esquemas algorítimos sobre secuencias: Búsqueda

```
// Inicializar
int actual;
bool encontrado = false;
while (not encontrado and cin >> actual) {
    if (actual % 2 == 0) encontrado = true;
}
if (encontrado) cout << "SI" << endl;
else cout << "NO" << endl;
```

Observación 1

Fíjate que si *encontrado* no forma parte de la condición del bucle, el resultado sería correcto. Sin embargo, estaría visitando TODOS los elementos de la secuencia: haría un recorrido. Sería ineficiente!! Mala idea!

Observación 2

```
while (cin >> actual and not encontrado)
```

Es correcto? Qué opción es mejor? (Nota: esta opción lee un elem más)

Esquemas algorítimos sobre secuencias: Búsqueda

Ejemplo 2:

Dado un dígito d y un natural n estictamente positivo, implementar una función que retorne cierto si n contine el dígito d, falso en caso contrario.

Con booleano:

```
// Pre: 0 <= d <= 9, n > 0
// Post: true si n contiene d
//       false en caso contrario
bool contiene(int d, int n) {
    bool found = false;
    while (not found and n != 0) {
        if (d == n%10) found = true;
        n = n/10;
    }
    return found;
}
```

Sin booleano:

```
// Pre: 0 <= d <= 9, n > 0
// Post: true si n contiene d
//       false en caso contrario
bool contiene(int d, int n) {
    while (n != 0) {
        if (d == n%10) return true;
        n = n/10;
    }
    return false;
}
```

Observaciones

- Las dos opciones implementan un esquema de búsqueda.
- Fíjate que si n pudiera ser 0, ninguno de los dos códigos sería correcto.

Invariantes

Un **invariante** es un predicado que siempre es cierto a lo largo de una secuencia específica de operaciones.

```
int x = 5;
int y = 10;
// Inv: x <= y
{
    x = x - 15;
    x = 2*x
}
// x <= y es cierto
```

Los utilizaremos para caracterizar el comportamiento de un bucle:

```
// Invariante
while (condición) {
    // Invariante ∧ condición
    ...;
    // Invariante
}
// Invariante ∧ ¬condición
```

Es decir ...

... el invariante es el significado de las variables que mantenemos en el cuerpo del bucle (parte tratada de la secuencia).

Invariantes

Ejemplo 1:

```
// Pre: n >= 0
// Post: retorna n!
int factorial(int n) {
    int f = 1;
    int i = 0;
    // Inv: f = i! ∧ i <= n
    while (i < n) {
        // f = i! ∧ i < n
        i = i + 1;
        f =  f*i;
        // f = i! ∧ i <= n
    }
    // f = i! ∧ i = n
    // f = n!
    return f;
}
```

i: último factor añadido.

```
// Pre: n >= 0
// Post: retorna n!
int factorial(int n) {
    int f = 1;
    // Inv: f = (i-1)! ∧ i <= n+1
    for (int i=1; i <= n; ++i) {
        // f = (i-1)! ∧ i <= n
        f =  f*i;
        // f = (i-1)! ∧ i <= n+1
    }
    // f = (i-1)! ∧ i = n+1
    // f = n!
    return f;
}
```

i: factor que añadiré en la próxima iteración.

Invariantes

Ejemplo 2:

```
// Pre: 0 <= d <= 9, n > 0
// Post: true si n contiene d
//       false en caso contrario
bool contiene(int d, int n) {
    bool found = false;
    // Inv: ningún dígito visitado es d (y found es falso)
    //   o d es el último dígito visitado (y found es cierto)
    while (not found and n != 0) {
        if (d == n%10) found = true;
        n = n/10;
    }
    return found;
}
```

Otros ejemplos

- Dada una palabra y un diccionario (que puede ser vacío), escribir "SÍ" si el diccionario contiene esa palabra y "NO" en caso contrario.

  Juegos de pruebas:

  E: casa asterisco mano voz $\Longrightarrow$ S: NO
  E: casa $\Longrightarrow$ S: NO
  E: casa casa mano voz $\Longrightarrow$ S: SÍ
  E: casa asterisco casa mano voz $\Longrightarrow$ S: SÍ
  E: casa asterisco barra casa $\Longrightarrow$ S: SÍ

- Implementar una función booleana que dado un natural n, retorne cierto si sus dígitos son crecientes de izquierda a derecha o falso en caso contrario.

  Juegos de pruebas:

  $n = 13456 \Longrightarrow$ true
  $n = 1354 \Longrightarrow$ false
  $n = 5 \Longrightarrow$ true

Otros ejemplos

- Dada una secuencia de enteros no vacía, decir cuál es la longitud máxima de sus llanuras. Una llanura es una subsecuencia del mismo elemento.

  Juegos de pruebas:

  E: 4 4 2 3 4 $\Longrightarrow$ S: 2
  E: 4 2 2 3 $\Longrightarrow$ S: 2
  E: 4 3 2 2 2 $\Longrightarrow$ S: 3

- Dada una secuencia de naturales con al menos dos elementos, decir el número de picos que hay en esa secuencia. Un pico es una terna de números tal que el central es mayor estricto o menor estricto que sus vecinos.

  Juegos de pruebas:

  E: 2 3 $\Longrightarrow$ 0
  E: 2 3 1 $\Longrightarrow$ 1
  E: 2 3 1 5 $\Longrightarrow$ 2
  E: 2 3 1 1 1 2 $\Longrightarrow$ 1