TEMA 2

Secuencias: fundamentos

Emma Rollón
erollon@cs.upc.edu

Departamento de Ciencias de la Computación

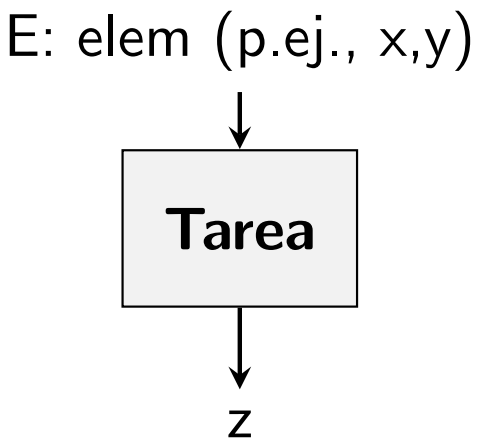
Índice

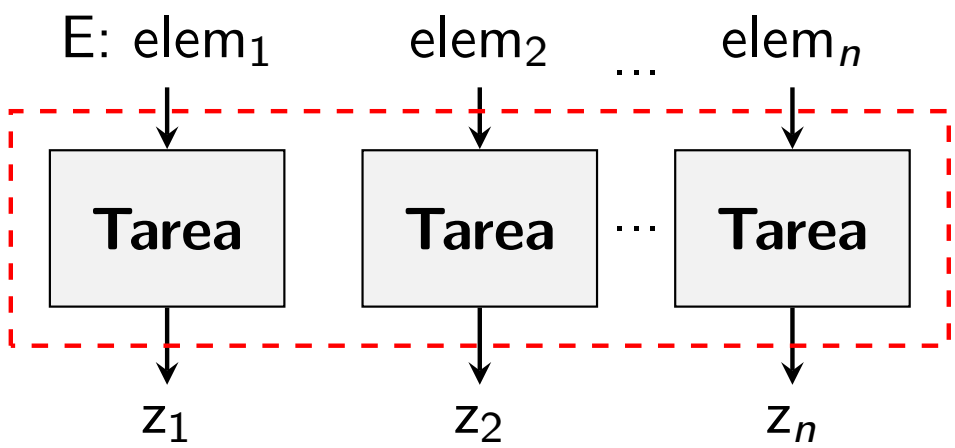
Introducción: Qué es una secuencia?

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).

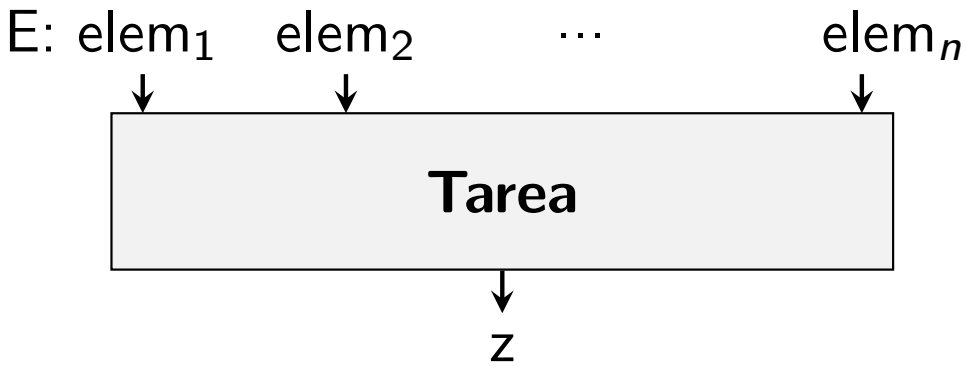




Introducción: Qué es una secuencia?

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.

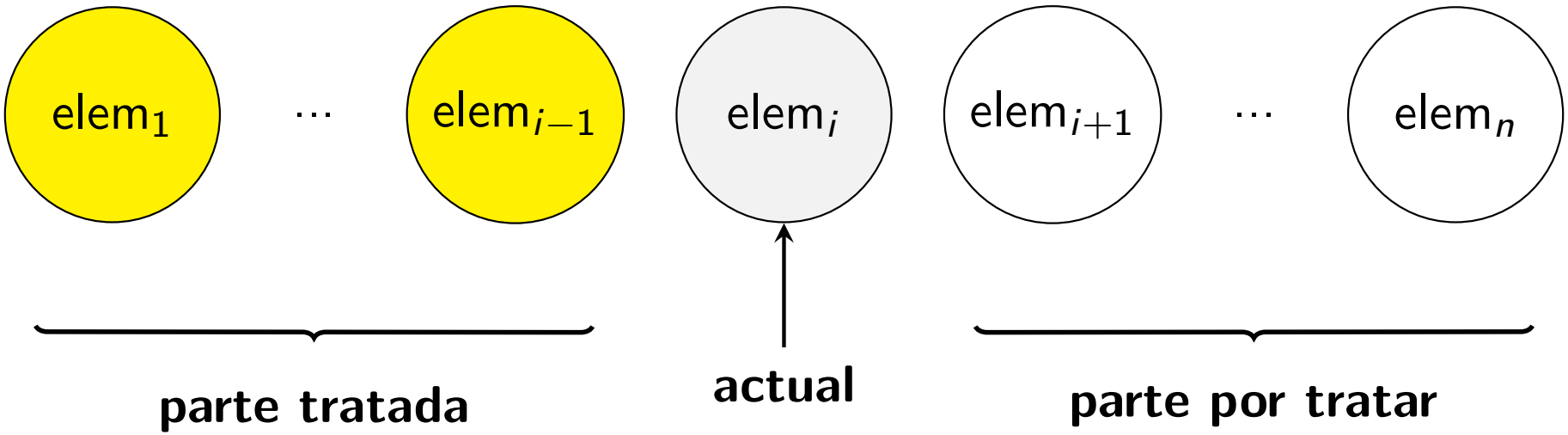
Determinar qué información (valores) necesitamos y, sobre todo, entender por qué los necesitamos y cómo los mantenemos es lo que veremos a partir de ahora.

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.

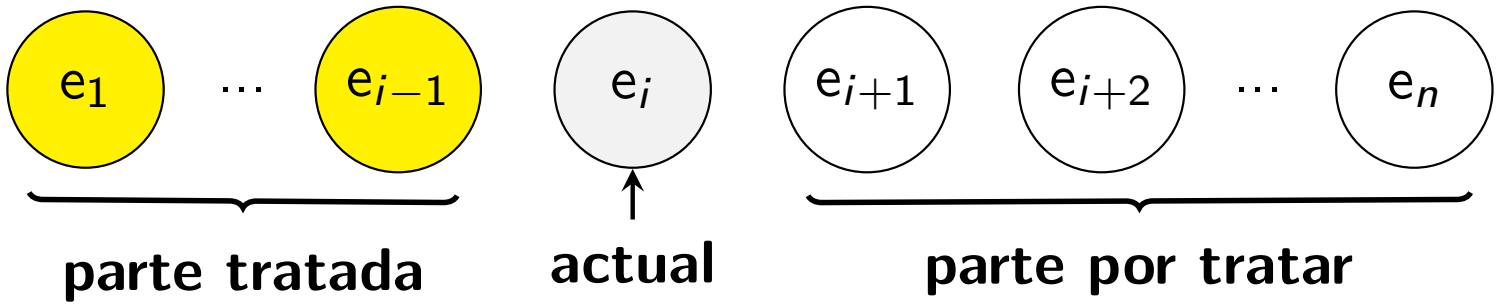
Visualmente:



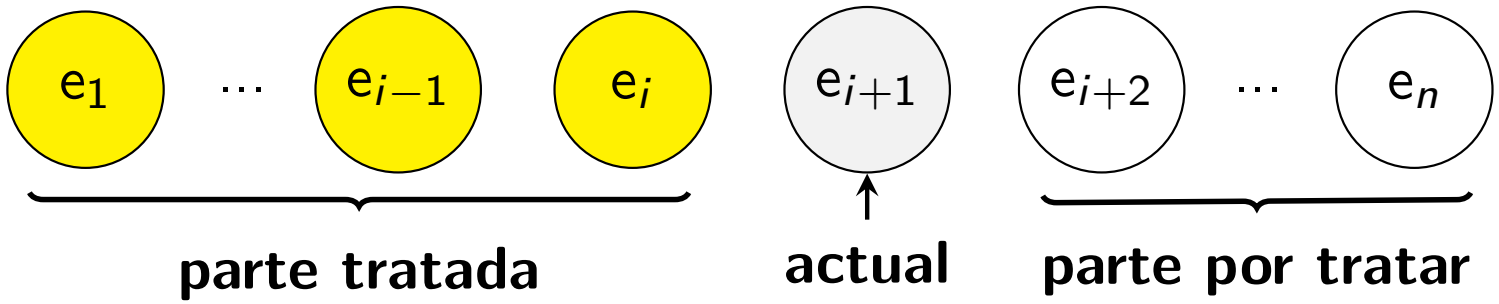
Introducción: Elementos clave

Cómo varían esos elementos de una iteración a la siguiente?

Iteración i:



Iteración i + 1:

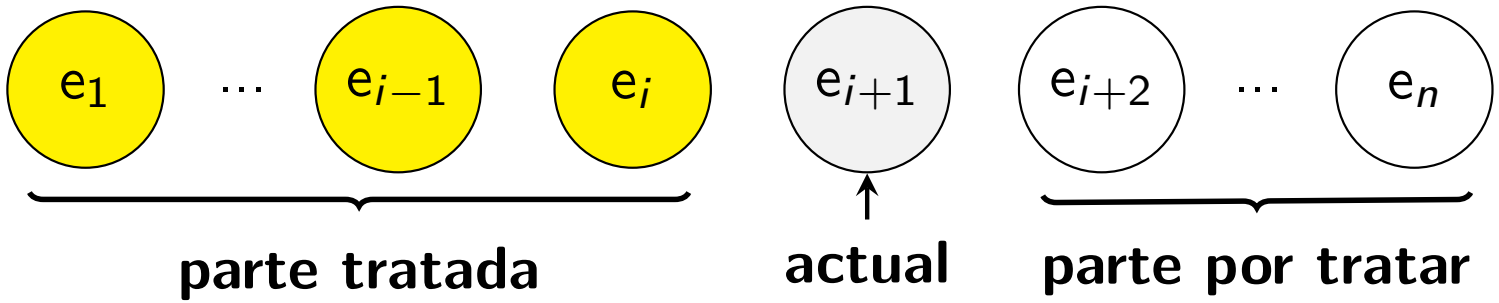


Es decir, el elemento actual en la iteración i pasa a estar incluído en la parte tratada en la iteración $i + 1$.

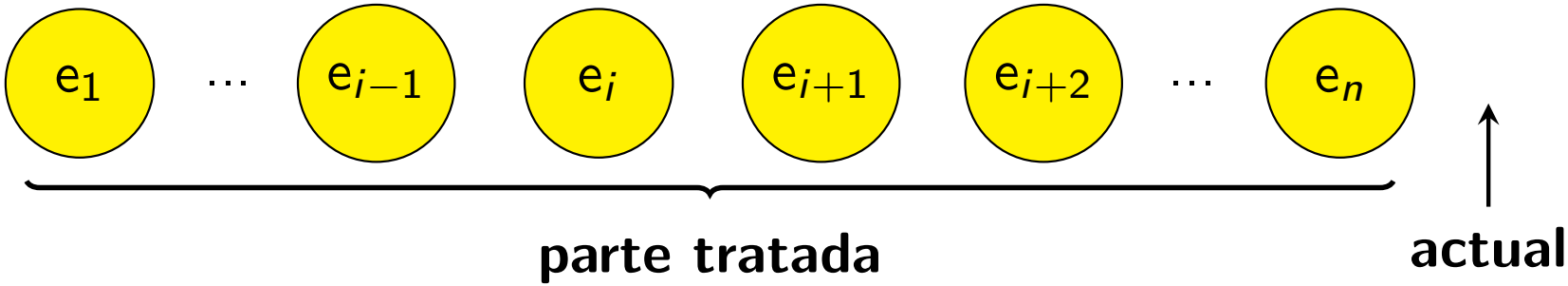
Introducción: Elementos clave

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 $\Rightarrow$ 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.

Juegos de prueba:

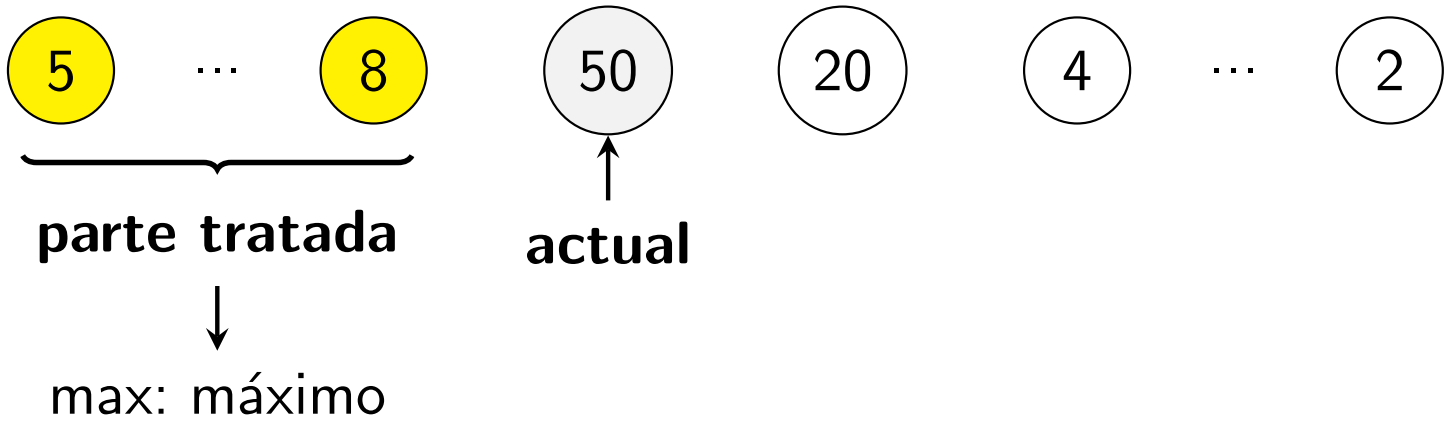
E: 2	S: 2
E: 10 2 5 9 1	S: 10
E: 1 4 10 5 7 9	S: 10
E: 1 4 9 10	S: 10

Observaciones

- La secuencia vacía no es una entrada válida.
- Piensa qué tipologías de juegos de prueba se muestran y si falta alguna.

Introducción: Razonamiento sobre elementos clave

A. ¿Qué información necesito mantener de la parte tratada?

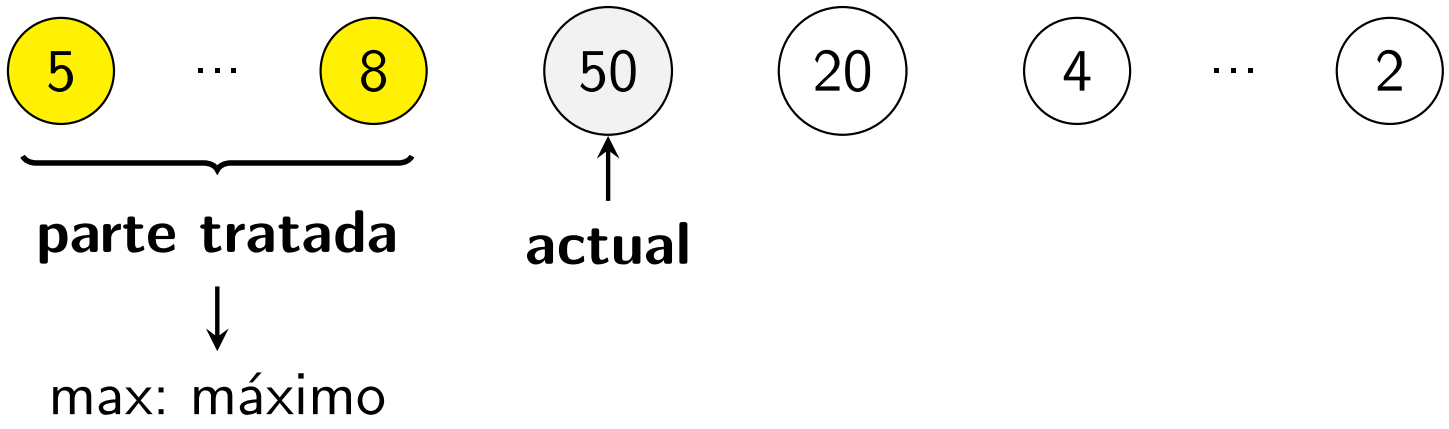


Si soy capaz de mantener esa variable max con el significado máximo de la parte tratada en todas y cada una de las iteraciones, cuando la parte tratada incluya toda la secuencia, ese valor max será "máximo de toda la secuencia", que será lo que tengo que escribir.

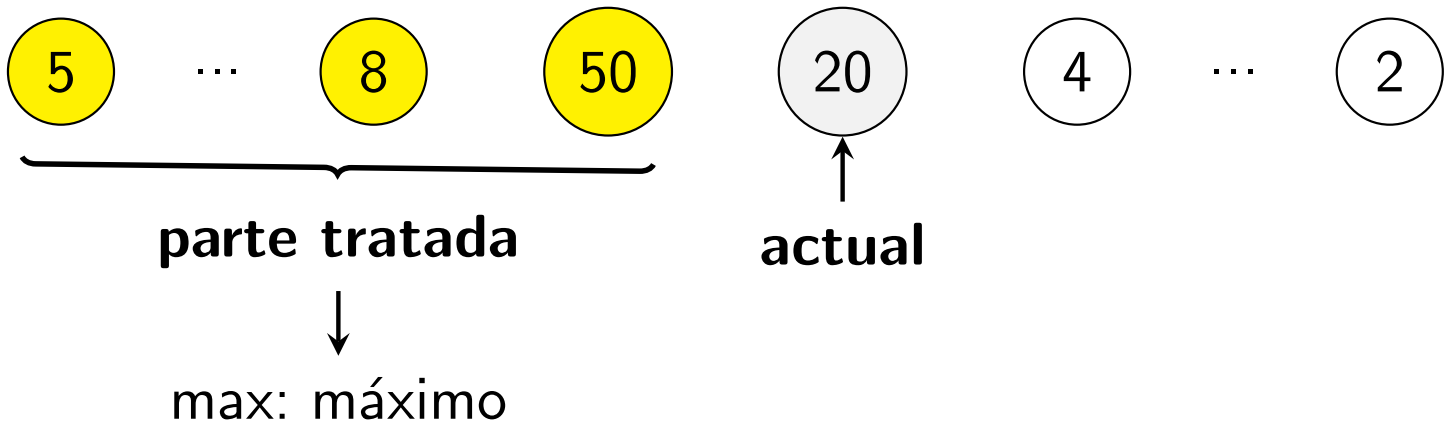
Introducción: Razonamiento sobre elementos clave

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;
}
```

Introducción: Razonamiento sobre elementos clave

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 las variables actual y max?

actual: me la da la propia condición del bucle.
otras variables de la parte tratada de la secuencia: depende de si la secuencia es o no vacía.

- secuencia no vacía:
  - puedo utilizar 1^{er} elemento para inicializar variable
  - entonces el bucle trata a partir del 2^o elemento de la secuencia
- secuencia vacía:
  - asignar un valor a la variable que mantenga su significado
  - entonces el bucle trata a partir del 1^{er} elemento de la secuencia

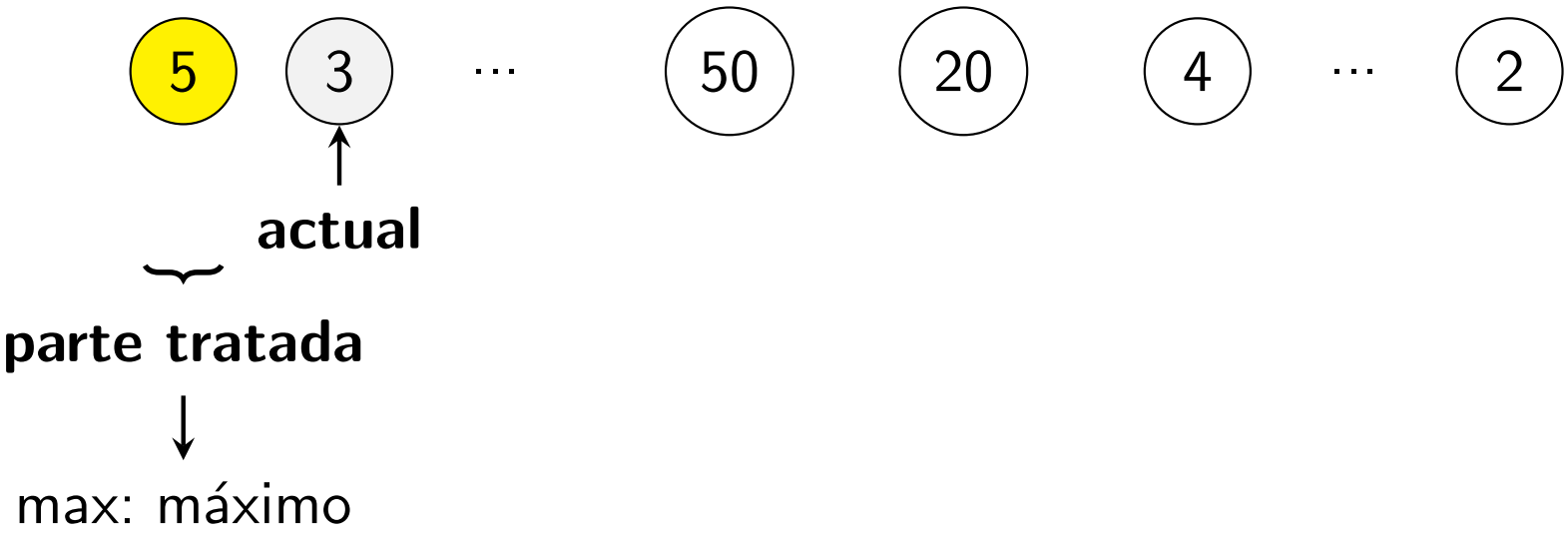
Fíjate que ...

... el esquema de secuencia vacía también es aplicable cuando la secuencia no es vacía.

Introducción: Razonamiento sobre elementos clave

Opción 1:

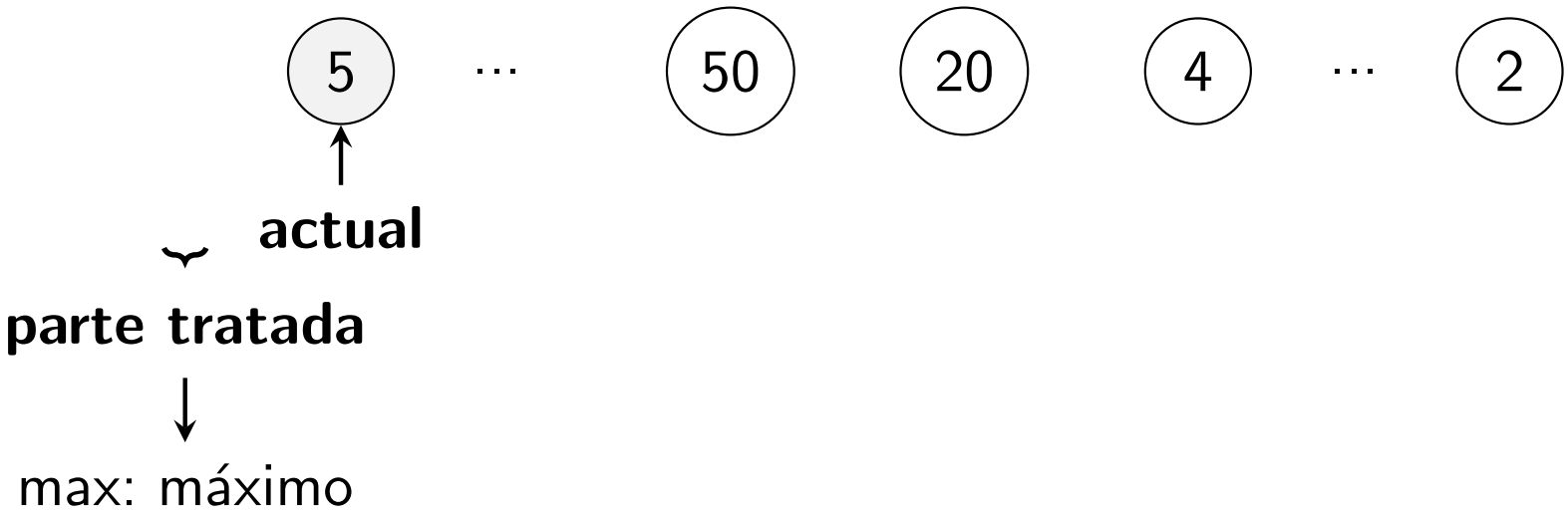
Como el enunciado dice que la secuencia no puede ser vacía, le daré a max el valor del primer elemento, y el bucle tratará a partir del segundo elemento de la secuencia.



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 no vacía de naturales acabada en -1, escribir cuántas veces un número es igual a la suma de todos los que están a su izquierda.

E: 0 0 1 1 3 2 7 2 -1	S: 3
E: 2 -1	S: 0
E: 2 3 1 4 -1	S: 0

Tenemos que responder a:

A. Qué información necesito mantener de la parte tratada?
B. Actualización de esa información?
C. Cómo detecto el final de la secuencia?
D. Inicialización?

¿Y si cambiamos las condiciones de la entrada y la secuencia pudiera ser vacía?

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, escribir SI si contine algún número par o NO en caso contrario.

Juegos de prueba:

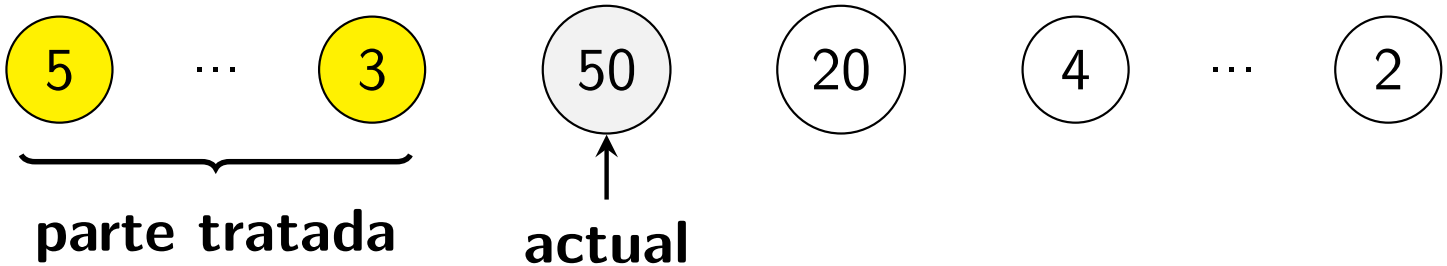
E: 3 5 11	S: NO
E: 2 5 7 9	S: SI
E: 5 2 7 9	S: SI
E: 5 9 7 2	S: SI
E:	S: NO

Observaciones

- La secuencia vacía es una entrada válida (e importante probarla).
- Piensa en qué tipología de juegos de prueba aparecen y si falta alguna.

Esquemas algorítmicos sobre secuencias: Búsqueda

A. Información que necesito almacenar de la parte tratada?

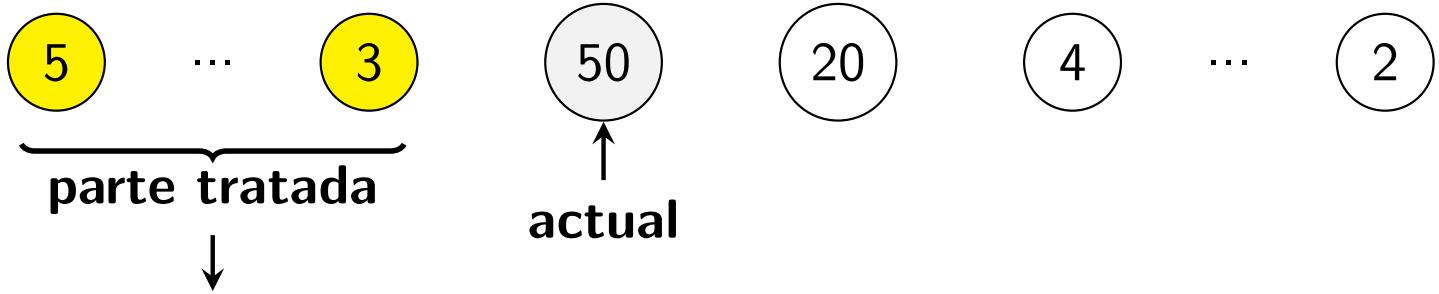


$$encontrado = \begin{cases} true, & \text{si existe elem par} \\ false, & \text{si no} \end{cases}$$

Esquemas algorítmicos sobre secuencias: Búsqueda

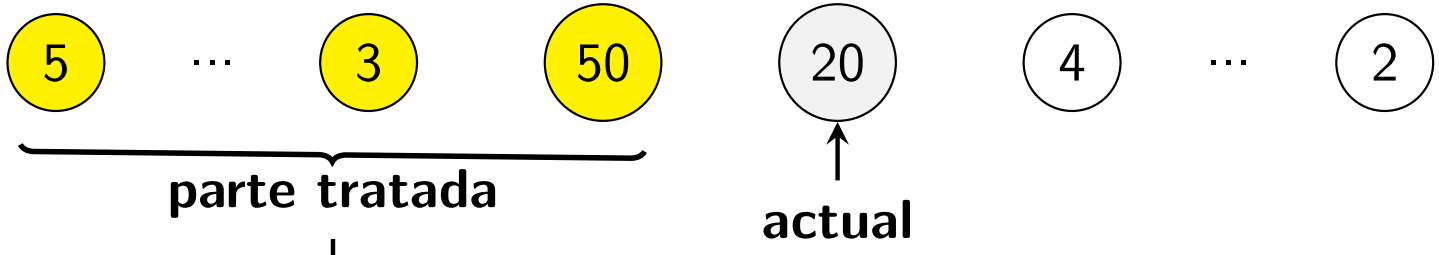
B. Actualización?

it i:



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

it i+1:



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

```
// 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;
```

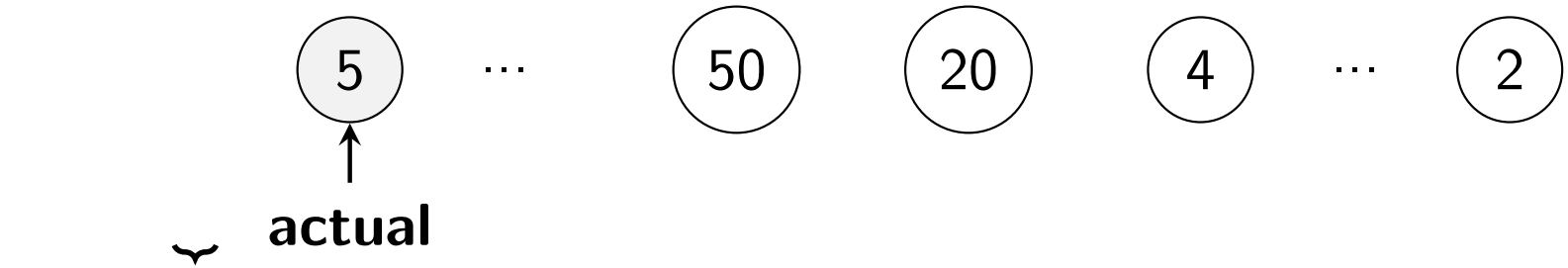
C. Condición del bucle?

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;
```

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 algorítmicos 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 algorítmicos sobre secuencias: Búsqueda

Ejemplo 2:

Dado un entero $0 \leq d \leq 9$ y un entero n estrictamente positivo, escribir SI si d es un dígito de n, o NO en caso contrario.

Juegos de prueba:

E: 1 234156	S: SI
E: 0 23	S: NO
E: 3 23	S: SI
E: 3 32	S: SI

Variaciones:

- ¿Y si n pudiera ser 0?
- Si existe el dígito hay que escribir la posición más pequeña en la que aparece (siendo la del dígito de menos peso la 1), -1 si el dígito no existe.

Esquemas algorítmicos sobre secuencias: Búsqueda

Observaciones:

1. Un entero es una secuencia de dígitos
2. La búsqueda se puede implementar explícitamente con el bool found y, a veces, directamente con la condición que éste representa (cuidado con los efectos colaterales que este cambio pueda tener)

Con booleano:

```
int d, n;
cin >> d >> n;
bool found = false;
while (not found and n != 0) {
    if (d == n%10) found = true;
    n = n/10;
}
if (found) cout << "SI" << endl;
else cout << "NO" << endl;
```

Sin booleano:

```
int d, n;
cin >> d >> n;

while (d != n%10 and n != 0) {
    n = n/10;
}

if (n != 0) cout << "SI" << endl;
else cout << "NO" << endl;
```

Fíjate que *found* será cierto cuando $d == n\%10$. Podemos sustituir en la condición del bucle *not found* por *not* $d == n\%10$ (es decir, $d! = n\%10$).

Esquemas algorítmicos sobre secuencias: Búsqueda

Ejemplo 3:

Dado un string p y una secuencia de strings ordenados lexicográficamente (es decir, representan un diccionario), escribir SI si p existe en el diccionario, NO en caso contrario.

Juegos de prueba:

E: ddd	aaa bbb ddd eee zzz	S: SI
E: ddd	aaa bbb eee zzz	S: NO
E: ddd		S: NO

Piensa ...

... qué tipologías de juegos de prueba faltan.

Esquemas algorítmicos sobre secuencias: Búsqueda

```
string p;
cin >> p;                    // palabra a buscar

bool all_smaller = true; // todos los tratados son menores estrictos que p

string s = "";               // importante! si diccionario vacío, cin >> s no lo inicializa
while (all_smaller and cin >> s) {
    if (p <= s) all_smaller = false;
}

// En este punto, es cierta una de las dos cosas:
// — el último tratado provocó asignación all_smaller = false
// — se ha tratado todo el diccionario (s es la última palabra del diccionario)
if (p == s) cout << "SI" << endl;
else cout << "NO" << endl;
```

Observación:

En el esquema genérico de lectura de "secuencia directamente en entrada":

```
while (cin >> s) {
    ...
}
```

el significado de la s es *último elemento tratado*.

Esquemas algorítmicos sobre secuencias: Mixto

Ejemplo 4:

Dado un entero p y una secuencia no vacía de enteros en orden ascendente, escribir la secuencia con el número p en la posición adecuada para que toda la salida esté ordenada.

Juegos de prueba:

E: 15	1 5 8 20 30
E: 15	1 5 8
E: 15	20 30

S:	1 5 8 15 20 30
S:	1 5 8 15
S:	15 20 30

Esquemas algorítmicos sobre secuencias: Mixto

Estrategia:

1. Buscar posición adecuada para p mientras se escriben los que van delante de él
2. Escribir p
3. Escribir todos los que faltan por leer/tratar

```
int p;
cin >> p;                    // entero a insertar

bool all_smaller = true; // todos los tratados son menores estrictos que p

int s;                       // importante! cin >> s siempre lo inicializa (sec. no vacía)
while (all_smaller and cin >> s) {
    if (p <= s) all_smaller = false;
    else cout << s << " ";
}

cout << p;

if (not all_smaller) { // s provocó asignación all_smaller = false
    cout << " " << s;
    while (cin >> s) cout << " " << s;
}
cout << endl;
```

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 main() {
    int n;
    cin >> n;
    int f = 1;
    int i = 0;
    // Inv: f = i! ∧ i <= n
    while (i < n) {
        // f = i! ∧ i < n
        ++i;
        f =  f*i;
        // f = i! ∧ i <= n
    }
    // f = i! ∧ i = n
    // f = n!
    cout << f << endl;
}
```

i: último factor añadido.

```
// Pre: n >= 0
// Post: retorna n!
int main() {
    int n;
    cin >> n;
    int f = 1;
    int i = 1;
    // Inv: f = (i−1)! ∧ i <= n+1
    while (i <= n) {
        // f = (i−1)! ∧ i <= n
        f =  f*i;
        ++i;
        // f = (i−1)! ∧ i <= n+1
    }
    // f = (i−1)! ∧ i = n+1
    // f = n!
    cout << f << endl;
}
```

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;
}
```