

**Universidad Nacional de Ingeniería
Facultad de Ciencias**

**Introducción a la Ciencia de la
Computación**

Algoritmos

**Prof: J. Solano
2011-I**

Objetivos

Después de estudiar este capítulo el estudiante sera capaz de:

- ☐ Definir un algoritmo y relacionarlo a la resolución de problemas.
- ☐ Definir tres constructores y describir su uso en algoritmos.
- ☐ Describir los diagramas UML y pseudocódigo y cómo se usan en los algoritmos.
- ☐ Listar algoritmos básicos y sus aplicaciones.
- ☐ Describir el concepto de ordenación y entender los mecanismos detrás de tres algoritmos de ordenación primitiva.
- ☐ Describir el concepto de búsqueda y entender los mecanismos detrás de dos algoritmos de búsqueda comunes.
- ☐ Definir subalgoritmos y sus relaciones con los algoritmos.
- ☐ Distinguir entre los algoritmos iterativos y recursivos.

Algoritmos para todas las eras

“Grandes algoritmos son la poesía de la computación”

Francis Sullivan

**Institute for Defense Analyses' Center for Computing Sciences Bowie,
Maryland**

La importancia de los Algoritmos

Análisis de tiempo de ejecución

Approximate completion time for algorithms, $N = 100$

$O(\text{Log}(N))$	10^{-7} seconds
$O(N)$	10^{-6} seconds
$O(N * \text{Log}(N))$	10^{-5} seconds
$O(N^2)$	10^{-4} seconds
$O(N^6)$	3 minutes
$O(2^N)$	10^{14} years.
$O(N!)$	10^{142} years.

La importancia de los Algoritmos

Ordenamiento – $O(N^2)$

Camino mas corto – $O(C^N)$ Ej: Algoritmo de Djikstra

Algoritmos aproximados

Algoritmos aleatorios – $O(N^2)$ - $O(N \cdot \log(N))$ Ej: Quicksort y median

Compresión

La importancia de conocer algoritmos

Mas ejemplos del mundo real

- Flujo Máximo
- Comparación de secuencias

Historia

Algos es la palabra griega para el dolor. Algor en latin, significa frío.

Ninguno es la raíz de algoritmo, que en su lugar tiene su origen de Al-Khwarizmi, erudito árabe del siglo nueve, cuyo libro **al-jabr wa'l muqabalah** derivó en los libros de texto de álgebra de hoy en las escuelas secundaria.

Al-Khwarizmi hizo hincapié en la importancia de procedimientos metódicos para la solución de problemas.

Si estuviera presente hoy, sin duda estaría impresionado por los avances en su enfoque del mismo nombre.

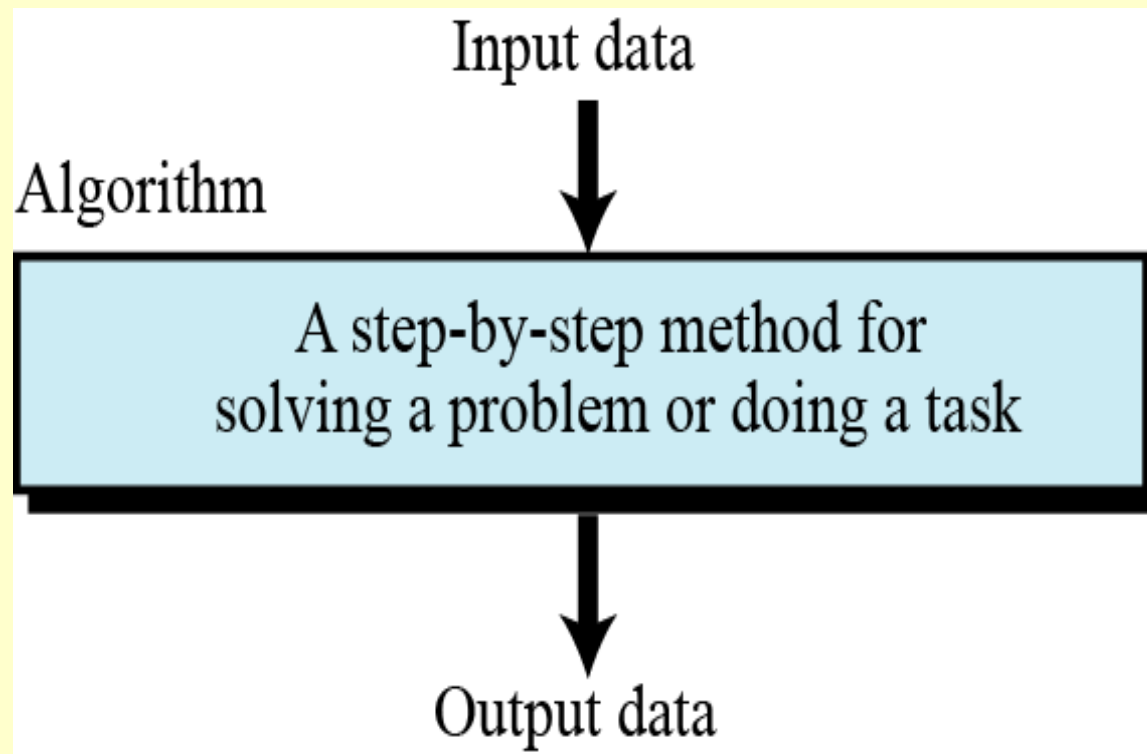
CONCEPTO

Aqui se define de manera informal un **algoritmo** y se desarrolla el concepto con un ejemplo.

Definición informal

Una definición informal de un algoritmo es:

Algoritmo: un método paso-a-paso para resolver un problema o hacer una tarea.

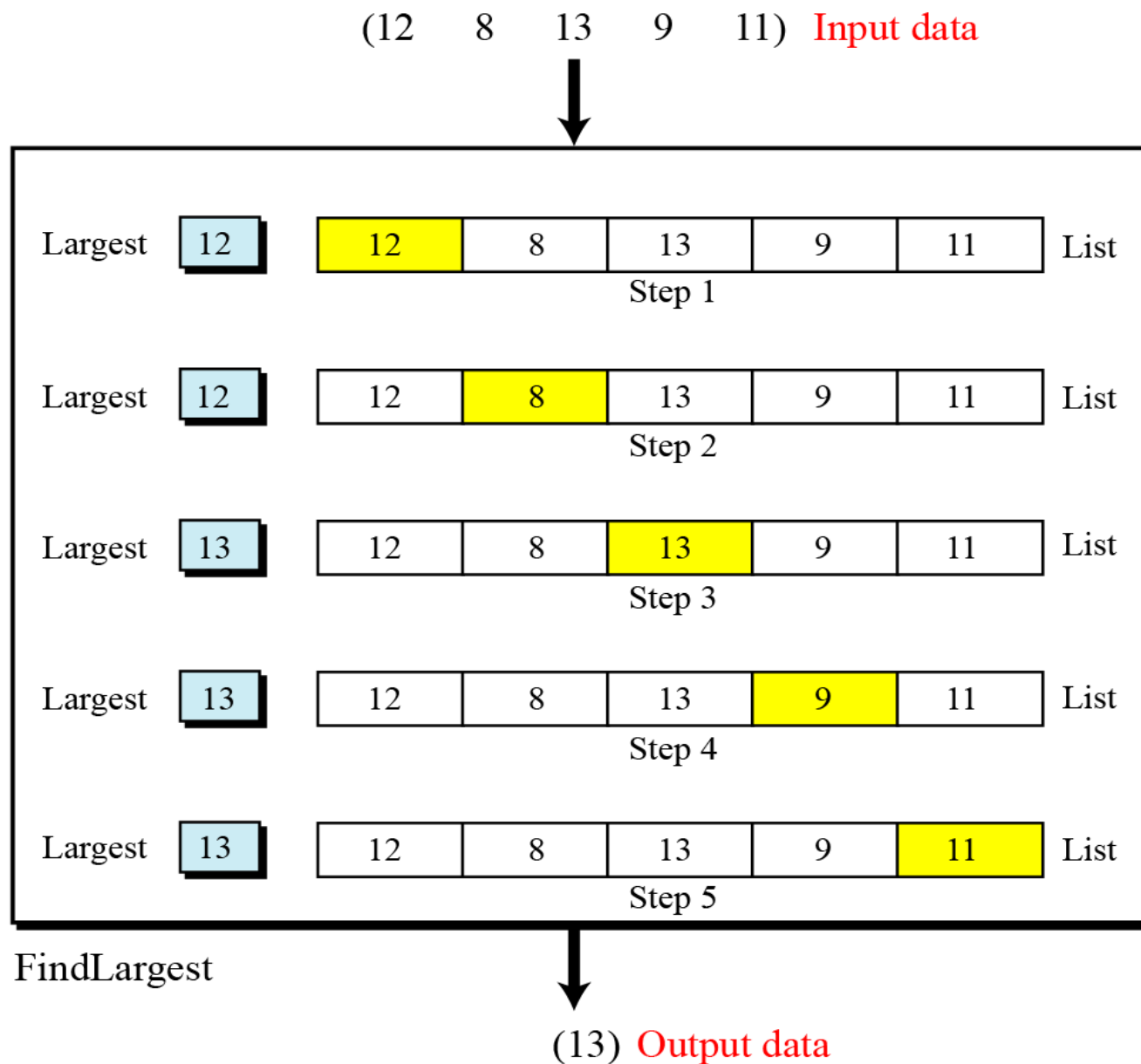


Ejemplo

Queremos desarrollar un algoritmo para encontrar el mayor entero entre una lista de números enteros positivos. El algoritmo debe encontrar el mayor entero entre una lista de valores (por ejemplo 5, 1000, 10000, 1000000). El algoritmo debe ser general y no depende del número de enteros.

Para resolver este problema, necesitamos un enfoque intuitivo. En primer lugar utilizar un pequeño número de enteros (por ejemplo, cinco), a continuación, extender la solución a cualquier número de enteros.

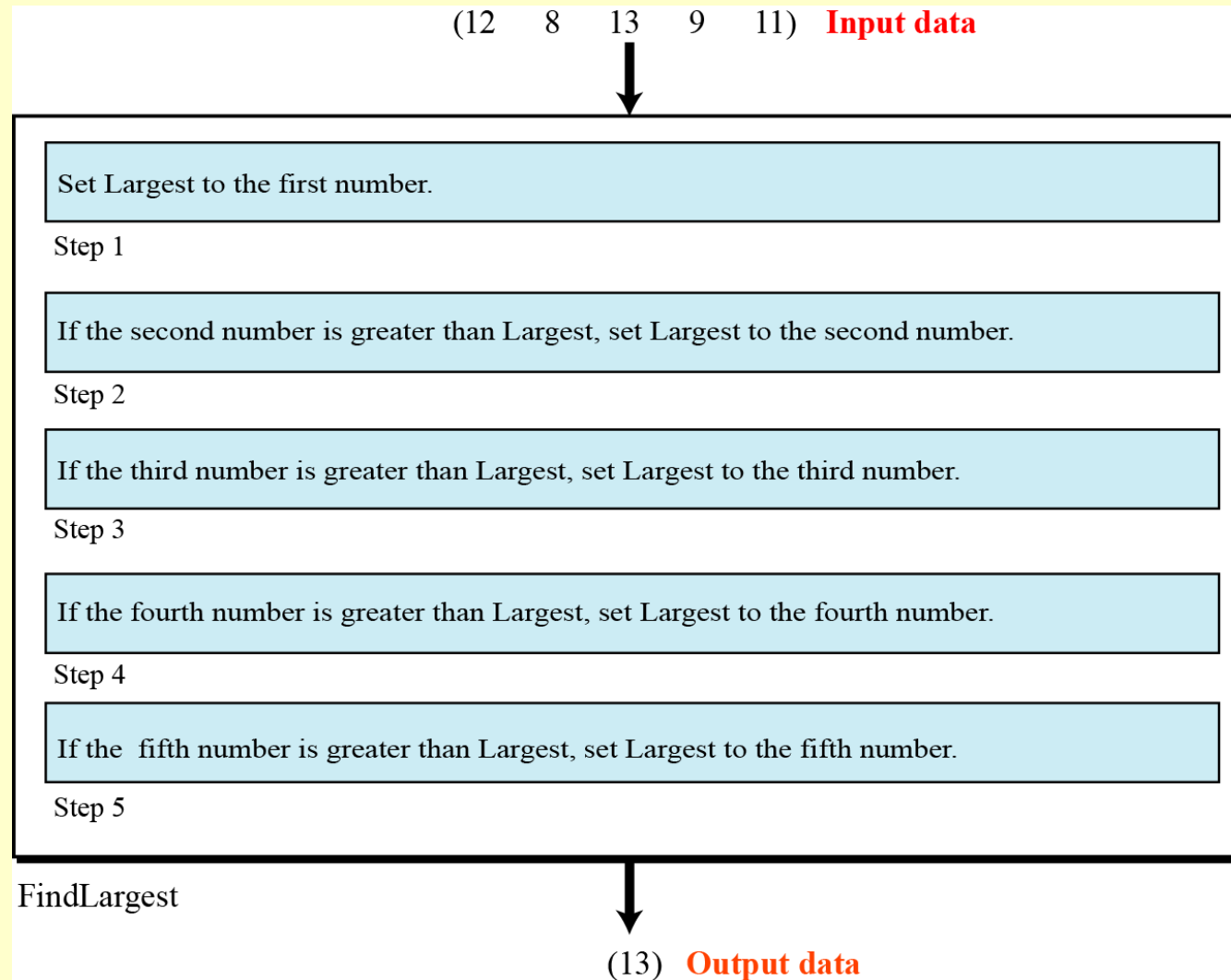
La siguiente figura muestra una manera de resolver este problema. Llamamos al algoritmo FindLargest. Cada algoritmo tiene un nombre para distinguirlo de otros algoritmos. El algoritmo recibe una lista de cinco números enteros como entrada y da el mayor entero como salida.



Hallar el mayor entero entre cinco enteros

Definiendo acciones

La figura anterior no muestra que debe hacerse en cada paso. Podemos modificar la figura para mostrar mas detalles.

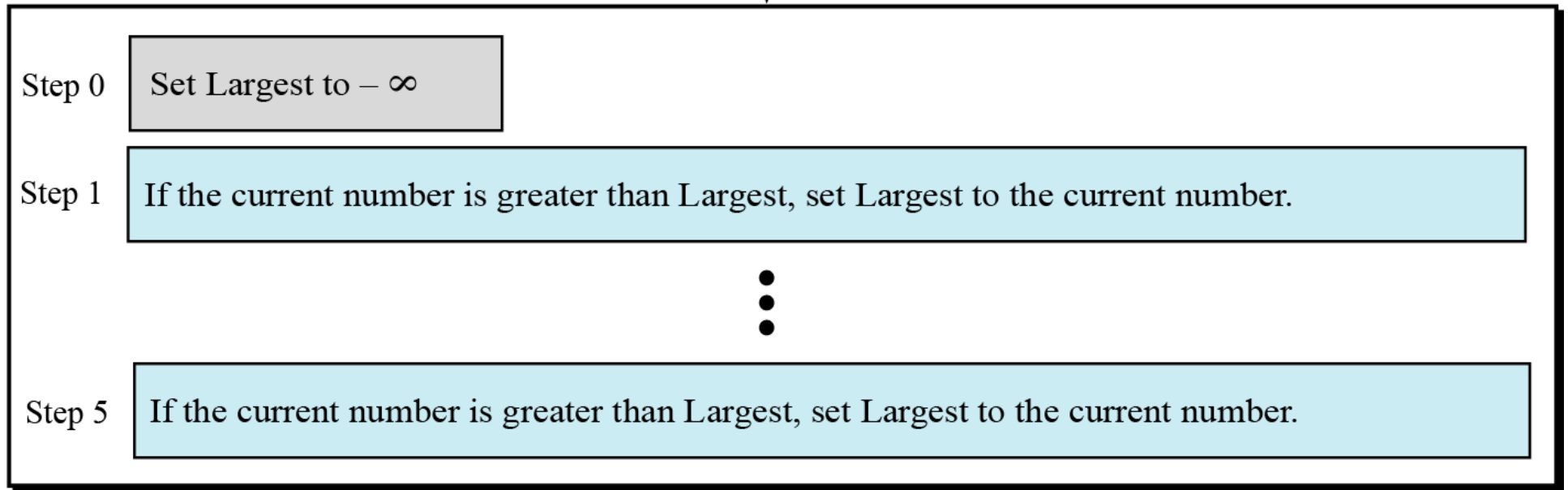


Definiendo acciones en algoritmo FindLargest

Refinamiento

Este algoritmo debe refinarse para ser aceptable para la comunidad de programadores. Hay dos problemas. En primer lugar, la acción en el primer paso es diferente a las de los otros pasos. En segundo lugar, la redacción no es la misma en los pasos 2 a 5. Fácilmente podemos redefinir el algoritmo para eliminar estos dos inconvenientes al cambiar la redacción en los pasos 2 a 5 "Si el entero actual es mayor que el más grande, llamar de mayor a el entero actual". La razón por la que el primer paso es diferente a los otros pasos se debe a que el Mayor no se ha inicializado. Si inicializamos mayor a $-\infty$ (menos infinito), entonces el primer paso puede ser el mismo que los otros pasos, por lo que añadimos un nuevo paso, llamándolo el paso 0 para indicar que se debe hacer antes de procesar cualquier enteros.

(12 8 13 9 11) **Input data**



FindLargest



(13) **Output data**

FindLargest refinado

Generalización

¿Es posible generalizar el algoritmo? Queremos encontrar el mayor de n enteros positivos, donde n puede ser 1000, 1.000.000, o más. Por supuesto, podemos seguir la figura anterior y repetir cada paso. Pero si cambiamos el algoritmo a un programa, entonces tenemos que escribir las acciones para los n pasos!

Hay una forma mejor de hacer esto. Podemos decirle a la computadora para repetir los pasos n veces.

A continuación se incluye esta función en nuestro algoritmo pictórica.

Input data (n integers)



Set Largest to $-\infty$

Repeat the following step n times:

If the current integer is greater than Largest, set Largest to the current integer.

FindLargest

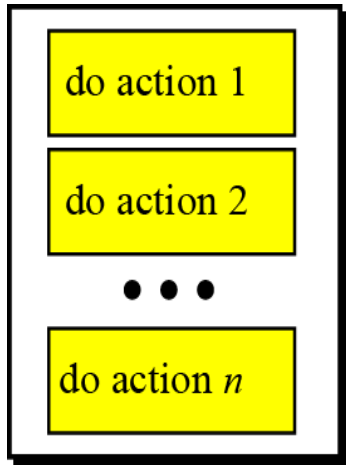


Largest

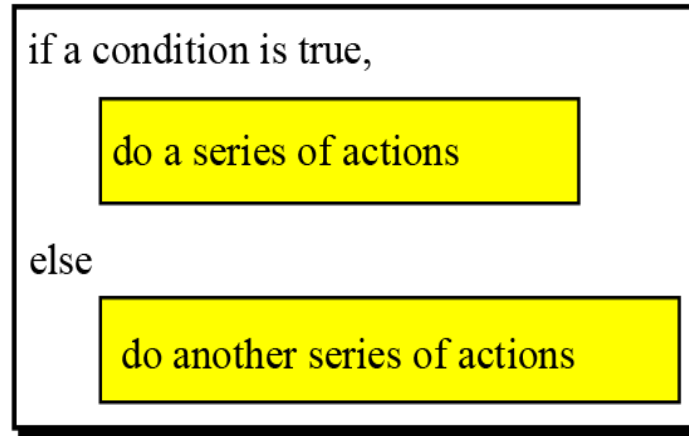
Generalización de FindLargest

TRES CONSTRUCTORES

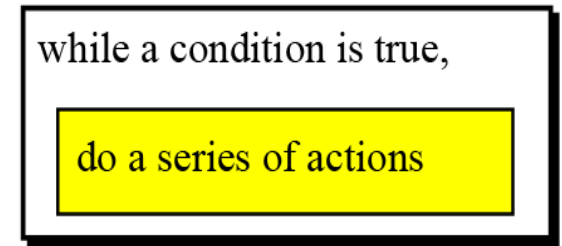
Los científicos en computación han definido **tres constructores** para un programa estructurado o algoritmo. La idea es que un programa debe estar hecho de una combinación de sólo estos tres constructores: **secuencia**, **decisión** (selección) y **repetición** (figura siguiente). Se ha demostrado que no hay necesidad de otros constructores. Utilizando sólo estos constructores hace que un programa o un algoritmo sea fácil de entender, depurar o cambiar.



a. Sequence



b. Decision



c. Repetition

Tres constructores

Secuencia

El primer constructor se llama secuencia. Un algoritmo, y, eventualmente un programa, es una secuencia de instrucciones, que pueden ser una simple instrucción o cualquiera de los otros dos constructores.

Decisión

Algunos problemas no pueden resolverse sólo con una secuencia de instrucciones simples. A veces tenemos que probar una condición. Si el resultado de las pruebas es verdadero, seguimos una secuencia de instrucciones: si es falso, seguimos una secuencia diferente de instrucciones. Esto se conoce como el constructor decisión (selección).

Repetición

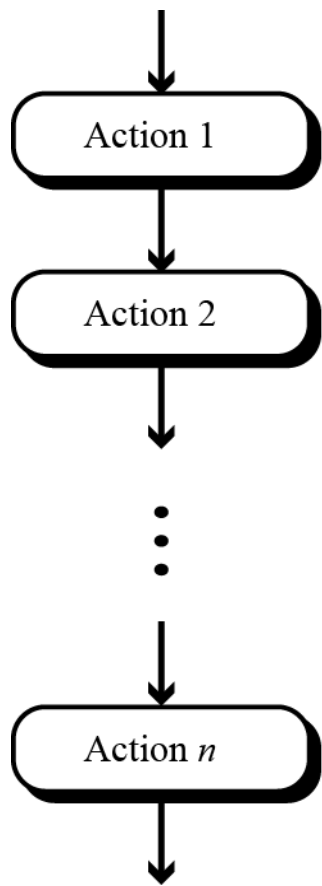
En algunos problemas, la misma secuencia de instrucciones que debe repetirse. Nosotros nos encargamos de esto con el constructor repetición o bucle (*loop*). Encontrar el número entero más grande entre un conjunto de números enteros puede utilizar una constructor de este tipo.

REPRESENTACIÓN DE ALGORITMOS

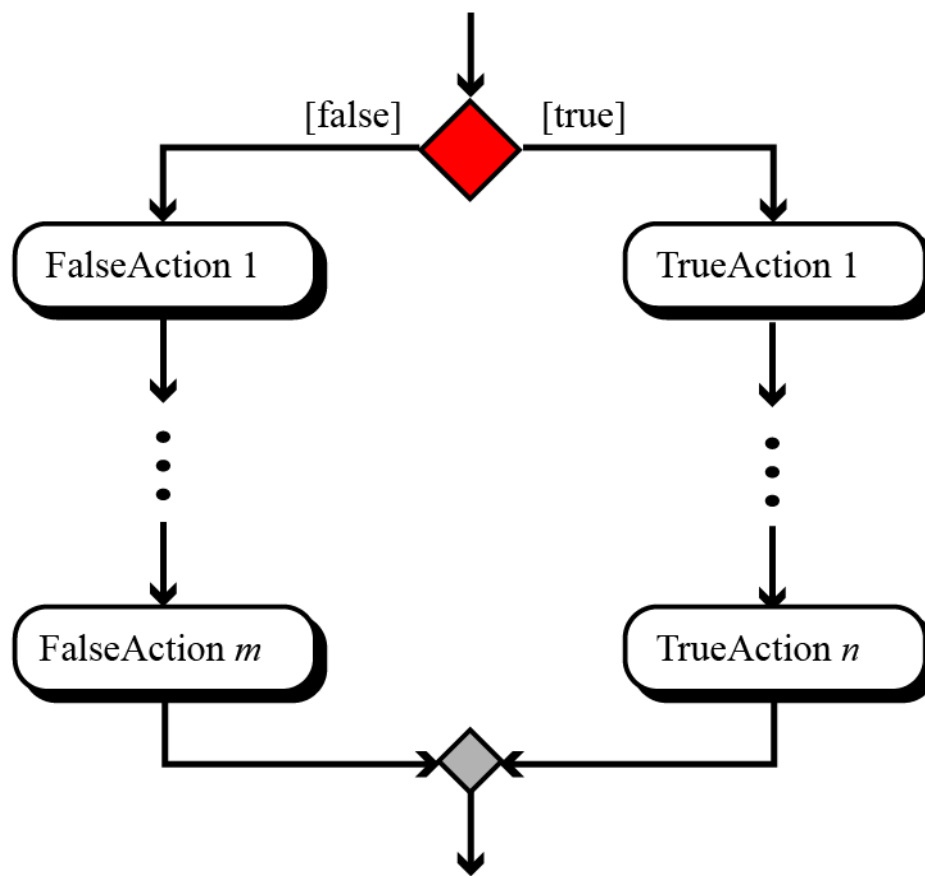
Hasta ahora, hemos utilizado figuras para transmitir el concepto de un algoritmo. Durante las últimas décadas, se han diseñado diferentes herramientas para este propósito. Dos de estas herramientas, **UML** y **pseudocódigo**, se presentan aquí.

UML

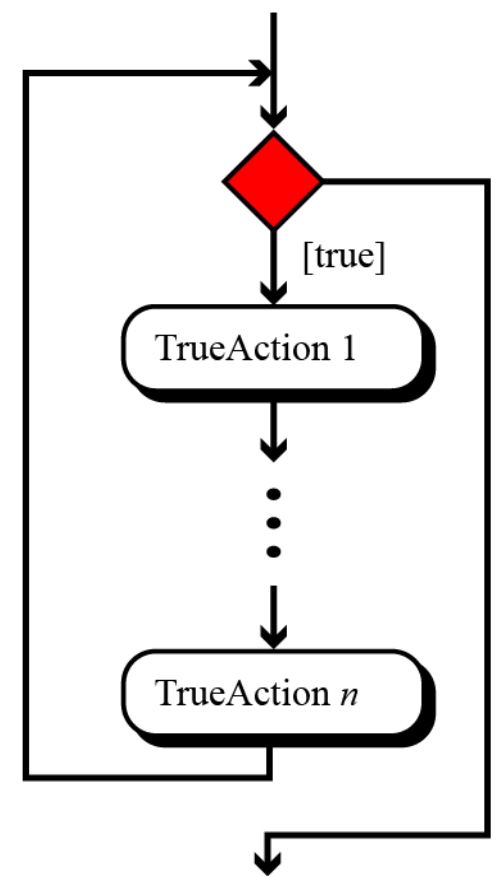
Unified Modeling Language (UML) es una representación pictórica de un algoritmo. Oculta todos los detalles de un algoritmo en un intento de dar el "cuadro grande " y mostrar cómo el algoritmo fluye de principio a fin. Hay mucha bibliografía sobre UML. Aquí sólo mostramos cómo los tres constructores son representados usando UML.



a. Sequence



b. Decision



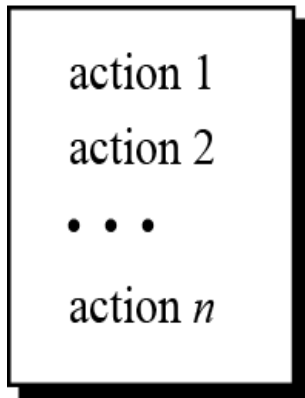
c. Repetition

UML para los tres constructores

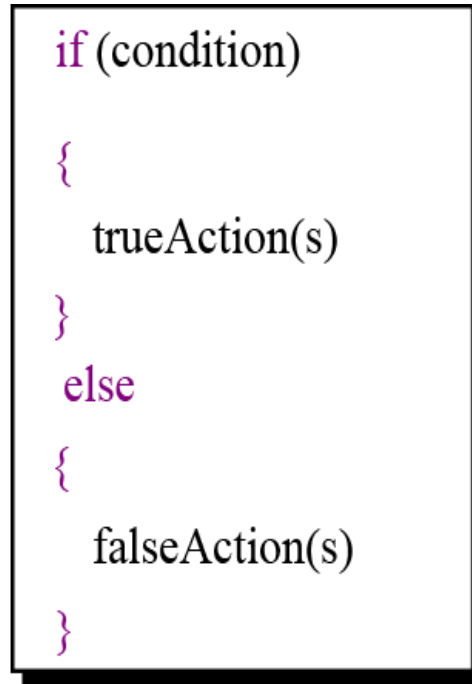
Pseudocódigo

Pseudocódigo es una representación como-idioma-Inglés de un algoritmo. No hay un estándar para pseudocódigo, algunas personas utilizan una gran cantidad de detalles, otros utilizan menos. Algunos utilizan un código que se encuentra cerca del Inglés, mientras que otros utilizan una sintaxis como el lenguaje de programación Pascal.

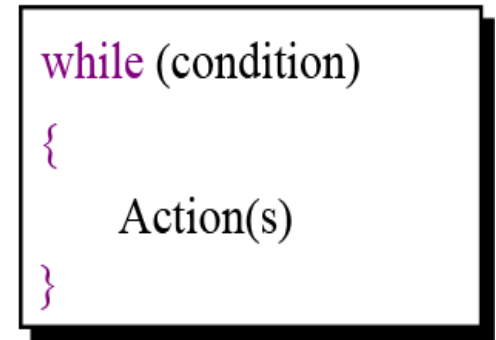
Aquí se muestra sólo cómo los tres constructores pueden ser representados por pseudocódigo.



a. Sequence



b. Decision



c. Repetition

Pseudocódigo para los tres constructores

Ejemplo 1

Escribir un algoritmo en pseudocódigo que halle la suma de dos enteros.

Calculating the sum of two integers

Algorithm: SumOfTwo (first, second)

Purpose: Find the sum of two integers

Pre: Given: two integers (first and second)

Post: None

Return: The sum value

```
{  
    sum ← first + second  
    return sum  
}
```

Ejemplo 2

Escribir un algoritmo para cambiar una nota numérica a pasó/no pasó (pass/no pass).

Assigning pass / no pass grade

Algorithm: Pass/NoPass (score)

Purpose: Creates a pass/no pass grade given the score

Pre: Given: the score to be changed to grade

Post: None

Return: The grade

```
{  
    if (score  $\geq$  70)        grade  $\leftarrow$  "pass"  
    else                    grade  $\leftarrow$  "nopass"  
    return grade  
}
```

Ejemplo 3

Escribir un algoritmo para cambiar una nota numérica (entero) a nota de letra.

Assigning a letter grade

Algorithm: LetterGrade (score)

Purpose: Find the letter grade corresponding to the given score

Pre: Given: a numeric score

Post: None

Return: A letter grade

```
{  
    if (100 ≥ score ≥ 90)    grade ← 'A'  
    if (80 ≥ score ≥ 89)    grade ← 'B'  
    if (70 ≥ score ≥ 79)    grade ← 'C'  
    if (60 ≥ score ≥ 69)    grade ← 'D'  
    if (0 ≥ score ≥ 59)     grade ← 'F'  
    return grade  
}
```

Ejemplo 4

Escribir un algoritmo para hallar el mayor número de un conjunto de enteros. No conocemos el número de enteros.

Finding the largest integer among a set of integers

Algorithm: FindLargest (list)

Purpose: Find the largest integer among a set of integers

Pre: Given: the set of integers

Post: None

Return: The largest integer

```
{  
    largest  $\leftarrow -\infty$   
    while (more integers to check)  
    {  
        current  $\leftarrow$  next integer  
        if (current > largest)           largest  $\leftarrow$  current  
    }  
    return largest  
}
```

Ejemplo 5

Escribir un algoritmo para hallar el mayor de los primeros 1000 enteros en un conjunto de enteros.

Finding the largest integer among the first 1000 integers

Algorithm: FindLargest2 (list)

Purpose: Find and return the largest integer among the first 1000 integers

Pre: Given: the set of integers with more than 1000 integers

Post: None

Return: The largest integer

```
{  
    largest  $\leftarrow -\infty$   
    counter  $\leftarrow 1$   
    while (counter  $\leq 1000$ )  
    {  
        current  $\leftarrow$  next integer  
        if (current  $>$  largest)           largest  $\leftarrow$  current  
        counter  $\leftarrow$  counter + 1  
    }  
    return largest  
}
```

UNA DEFINICION MAS FORMAL

Ahora que hemos discutido el concepto de algoritmo y mostrado su representación, aquí tenemos una definición más formal.

Algoritmo:

Un conjunto ordenado de pasos sin ambigüedades que produce un resultado y termina en un tiempo finito.

Conjunto ordenado

Un algoritmo debe ser un conjunto ordenado, y bien definido, de instrucciones.

Pasos no ambiguos

Cada paso en un algoritmo debe ser claro y sin ambigüedades. Si un paso es sumar dos números enteros, debemos definir ambos "enteros", así como la operación "add" (sumar): no podemos, por ejemplo, usar el mismo símbolo para representar adición en un lugar y multiplicación en otro lugar.

Produce un resultado

Un algoritmo debe producir un resultado, de lo contrario es inútil. El resultado puede ser datos entregados al algoritmo de llamada, o algún otro efecto (por ejemplo, la impresión).

Termina en un tiempo finito

Un algoritmo debe terminar (halt). Si no es así (es decir, tiene un bucle infinito), no hemos creado un algoritmo. Existe toda una area para discutir problemas solubles e insolubles. Veremos que un problema soluble tiene una solución en la forma de un algoritmo que termina.

ALGORITMOS BÁSICOS

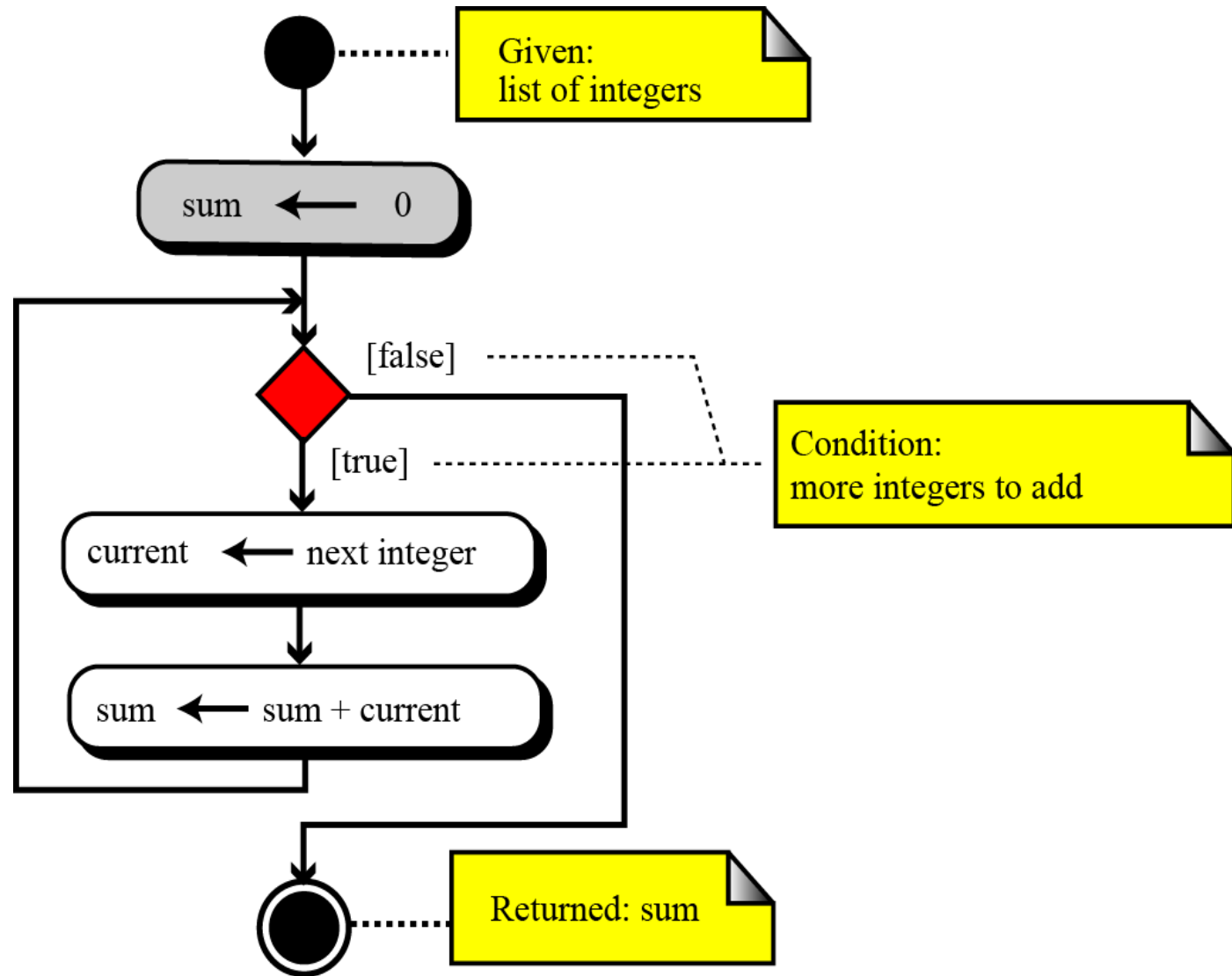
Varios algoritmos que se utilizan prevalentemente en informática se consideran "básicos". Discutiremos los más comunes. Esta discusión es muy general: la implementación depende del lenguaje.

Adición (resumen)

Un algoritmo de uso general en ciencia de la computación es la **adición**. Podemos sumar dos o tres números enteros muy fácilmente, pero ¿cómo podemos sumar muchos enteros? La solución es simple: utilizar el operador de añadir (add) en un bucle.

Un algoritmo de adición tiene tres partes lógicas:

1. Inicialización de la suma al principio.
2. El bucle, que en cada iteración añade un nuevo entero a la suma.
3. Retorno del resultado después de salir del bucle.



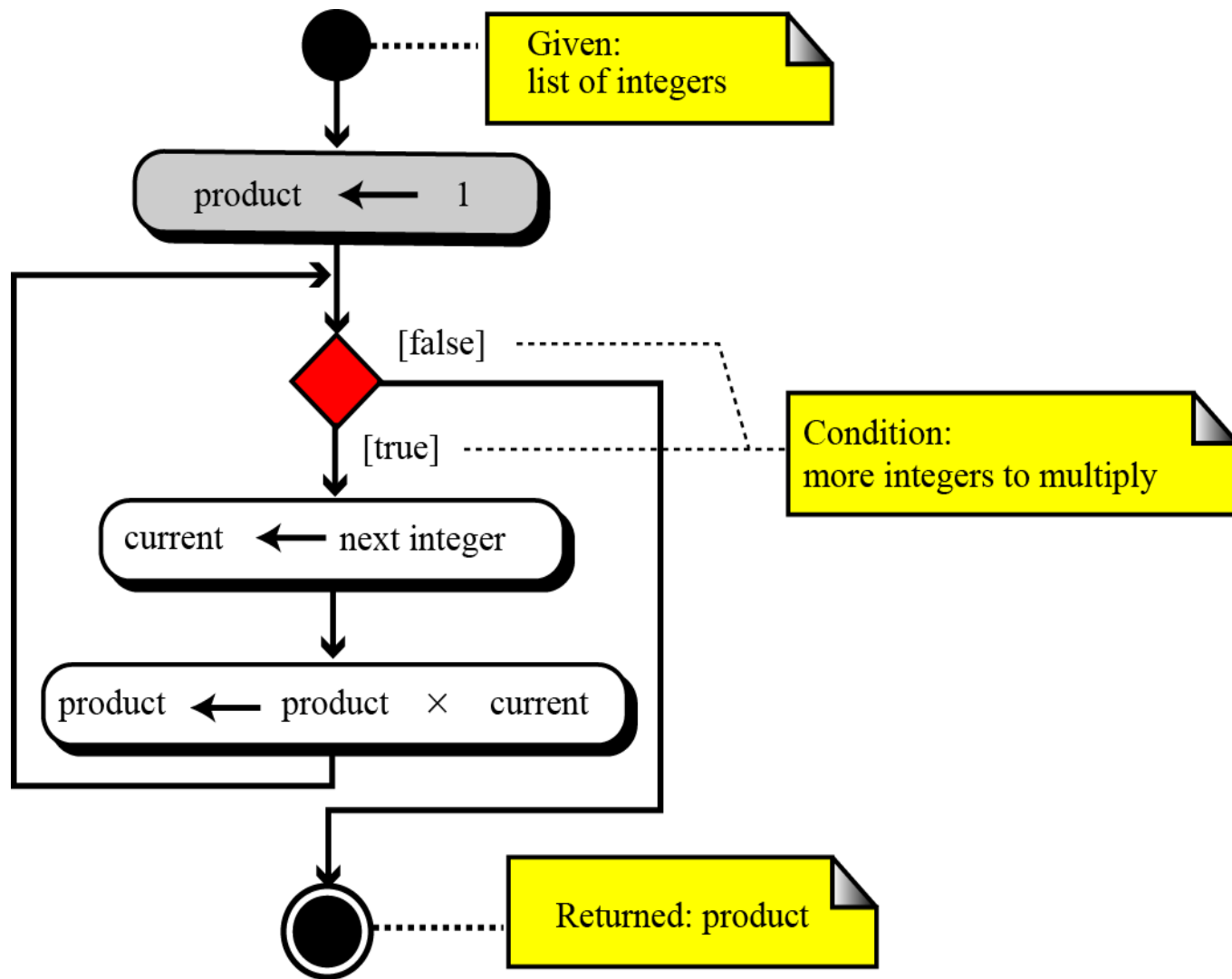
Algoritmo de adición

Producto

Otro algoritmo común es encontrar el **producto** de una lista de números enteros. La solución es simple: utilizar el operador de multiplicación en un bucle.

Un algoritmo de producto tiene tres partes lógicas:

1. Inicialización del producto al principio.
2. El bucle, que en cada iteración multiplica un nuevo entero con el producto.
3. Retorno del resultado después de salir del bucle.



Algoritmo de producto

El menor y el mayor

Hemos discutido el algoritmo para encontrar el mayor entre una lista de enteros en el comienzo de este capítulo. La idea era escribir un constructor decisión para encontrar el mayor de dos números enteros. Si ponemos este constructor en un bucle, podemos encontrar el mayor de una lista de números enteros.

Encontrar el menor entero entre una lista de números enteros es similar, con dos pequeñas diferencias. En primer lugar, utilizamos un constructor decisión para encontrar el menor de dos números enteros. En segundo lugar, inicializar con un entero muy grande en lugar de uno muy pequeño.

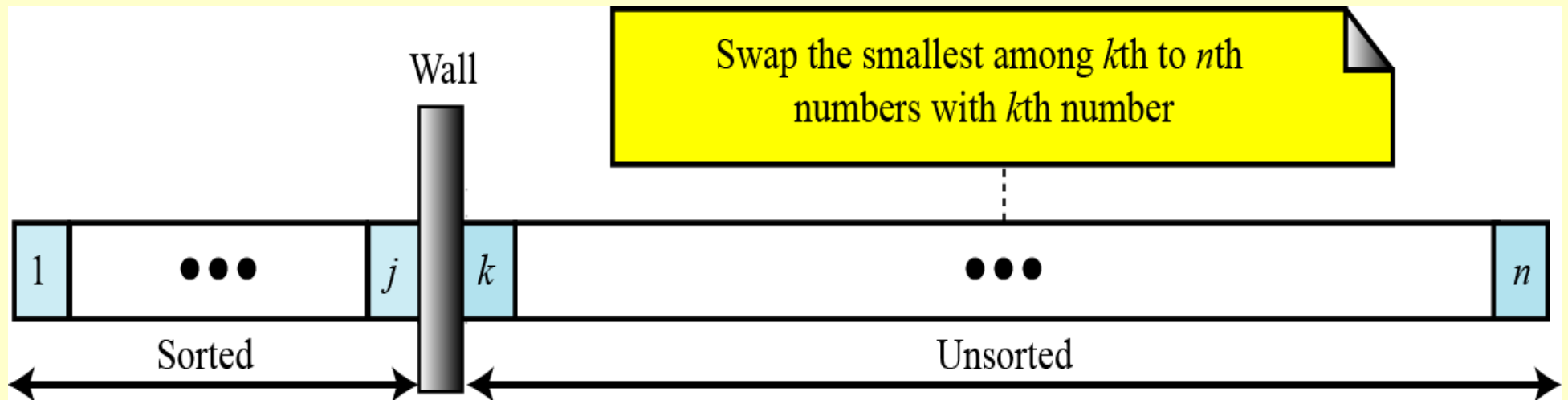
Algoritmos de ordenación

Una de las aplicaciones más comunes en ciencia de la computación es la ordenación, que es el proceso mediante el cual los datos se organizan de acuerdo a sus valores. La gente está rodeada de datos. Si los datos no estuvieran ordenados, se tardaría horas y horas para encontrar una sola pieza de información. Imagine la dificultad de encontrar el número de teléfono de alguien en un directorio telefónico que no esta ordenado.

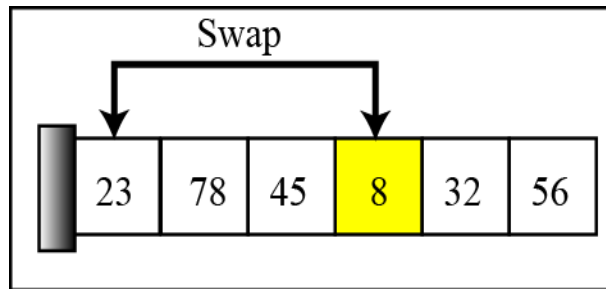
En esta sección, presentamos tres algoritmos de ordenación: **ordenamiento de selección**, **ordenamiento de burbuja** y la **ordenación por inserción**. Estos tres algoritmos de ordenación son la base de los algoritmos de ordenación más rápidos utilizados en la informática de hoy.

Ordenamiento de selección

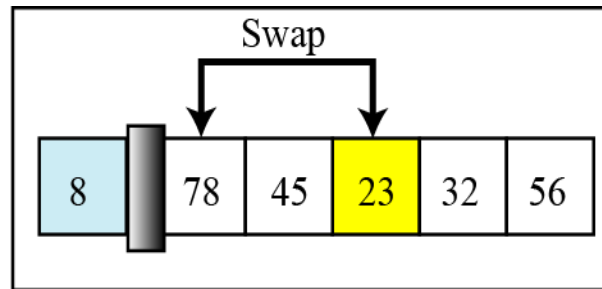
En un ordenamiento de selección, la lista a ser ordenada se divide en dos sublistas, clasificados y no clasificados, que están separadas por una pared imaginaria. Encontramos el elemento más pequeño en la lista secundaria sin clasificar y lo intercambiamos con el elemento al principio de la lista secundaria sin clasificar. Después de cada selección e intercambio, la pared imaginaria entre los dos sublistas se mueve un elemento para adelante.



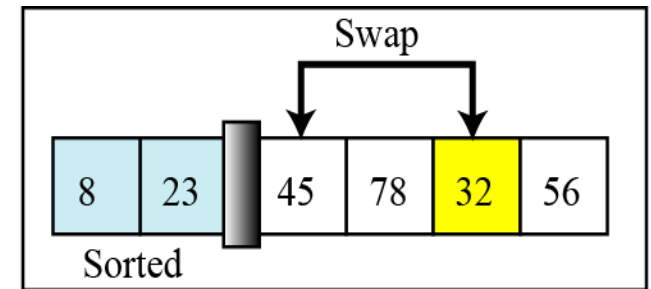
Ordenamiento de selección



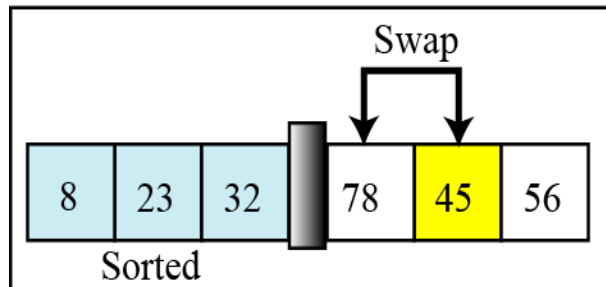
Original list



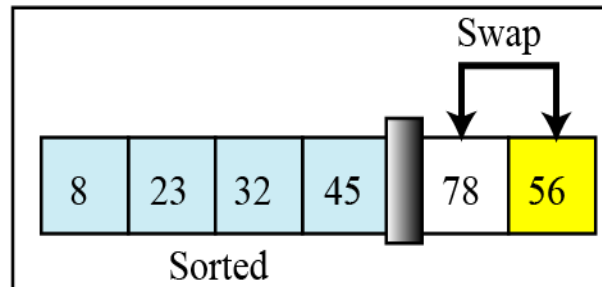
After pass 1



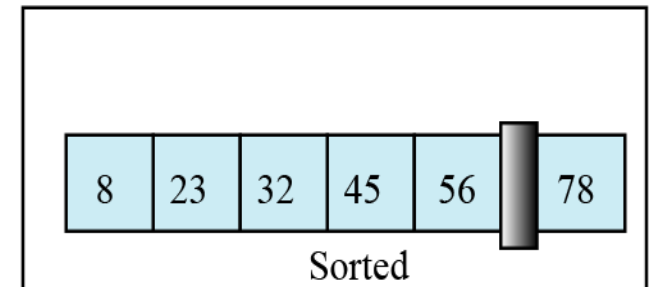
After pass 2



After pass 3

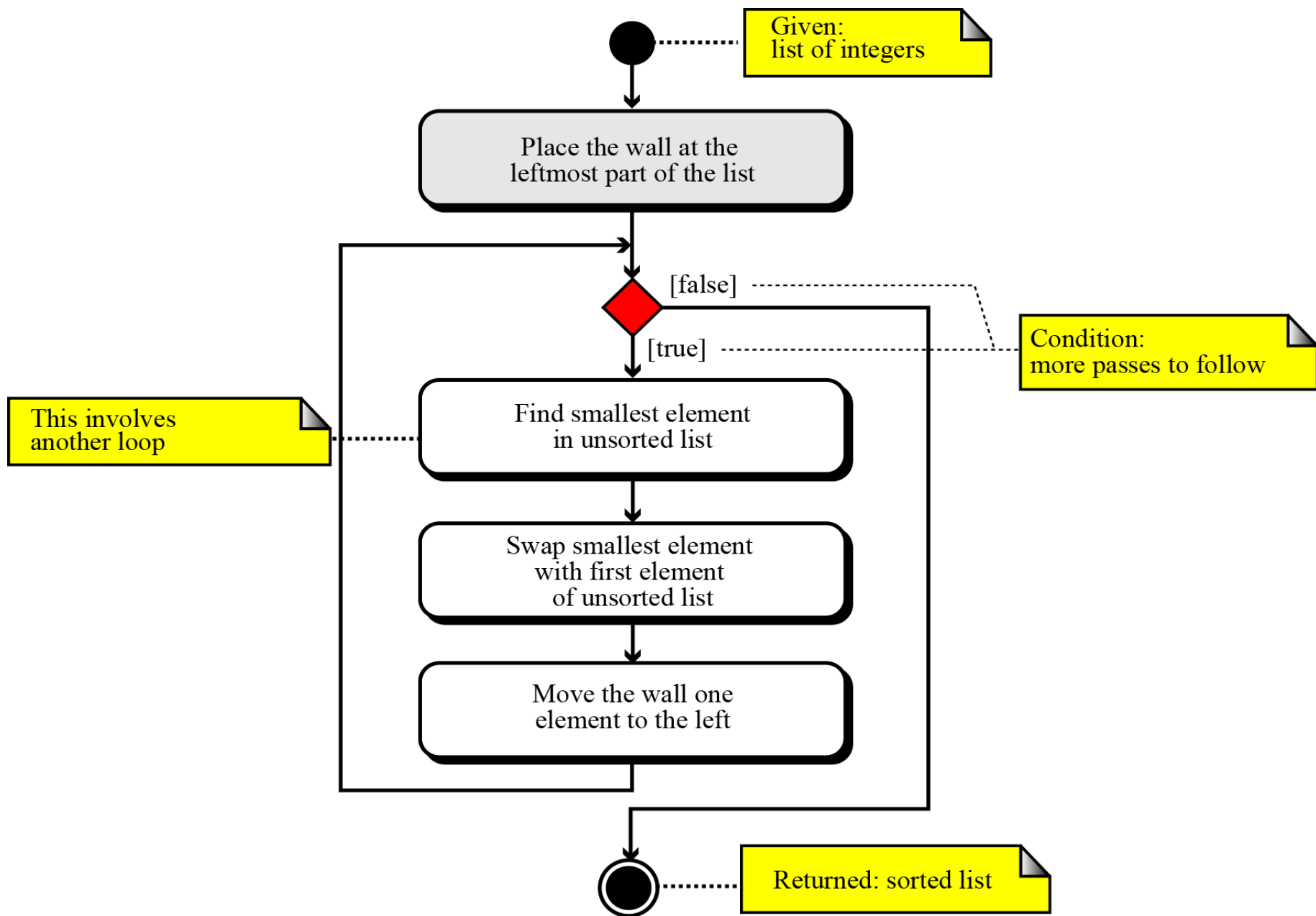


After pass 4



After pass 5

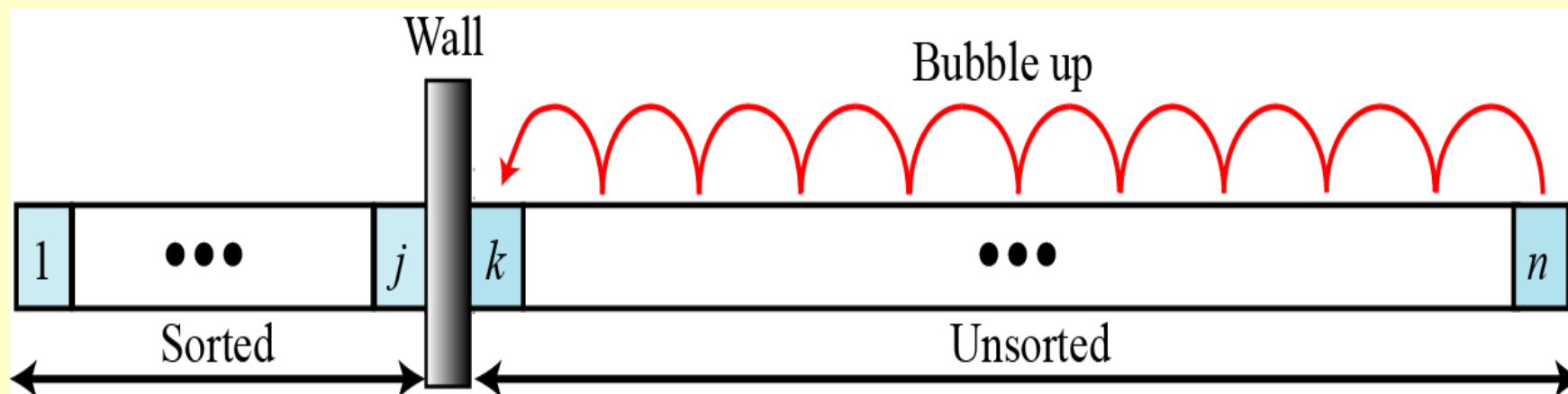
Ejemplo de ordenamiento de selección



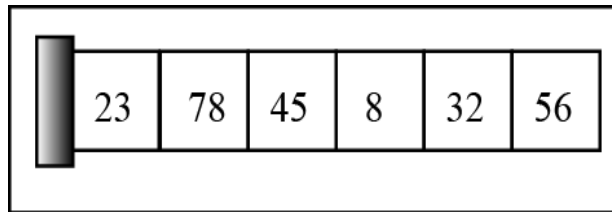
Algoritmo de ordenamiento de selección

Ordenamiento de burbuja

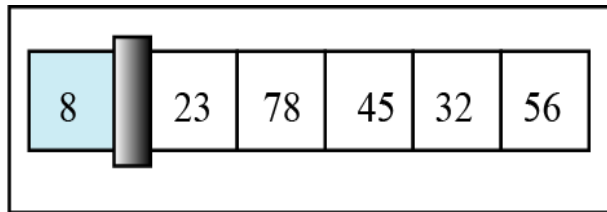
En el método de ordenamiento de burbuja, la lista a ser ordenada también se divide en dos sublistas, clasificados y sin clasificar. El elemento más pequeño se “burbujea” desde la lista secundaria sin clasificar y se traslada a la sublista ordenada. Después de que el elemento más pequeño ha sido trasladado a la lista ordenada, la pared se mueve un elemento por delante.



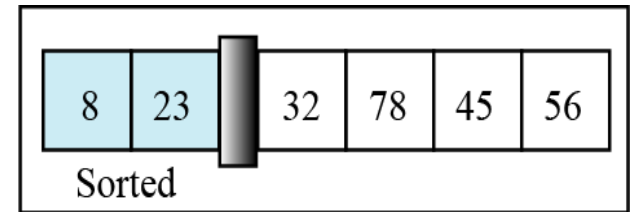
Ordenamiento de burbuja



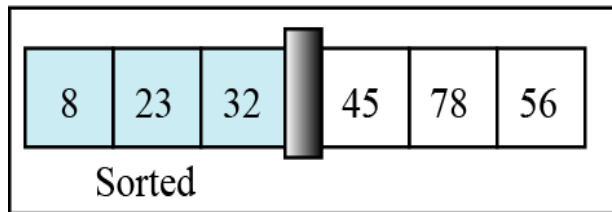
Original list



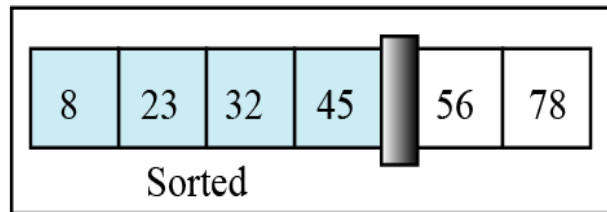
After pass 1



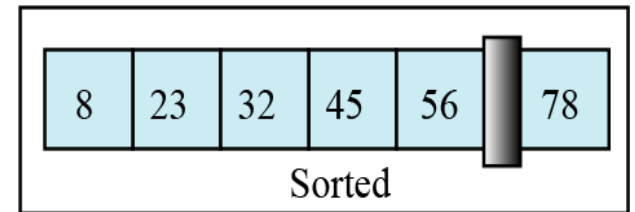
After pass 2



After pass 3



After pass 4

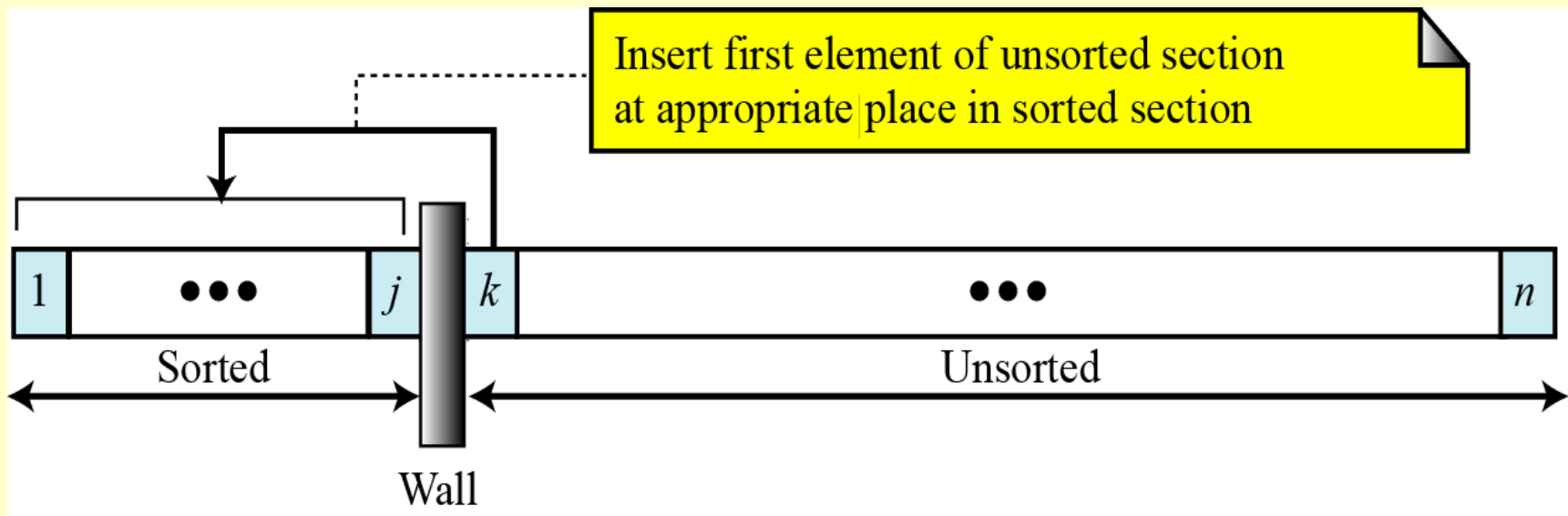


After pass 5

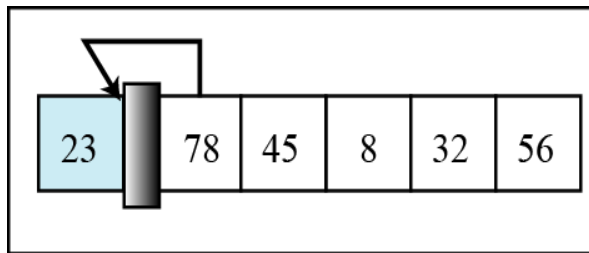
Ejemplo de ordenamiento de burbuja

Ordenamiento por inserción

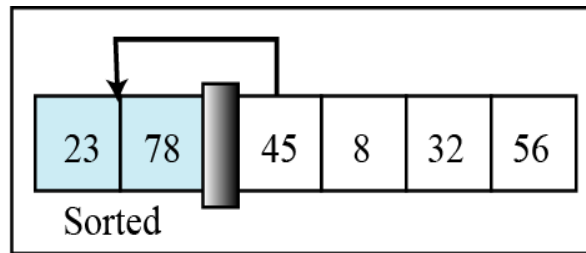
El algoritmo de ordenamiento por inserción es una de las técnicas de ordenamiento más comunes, y es a menudo utilizado por los jugadores de cartas. Cada tarjeta que un jugador coge la inserta en el lugar que le corresponde en su mano de cartas para mantener una secuencia particular.



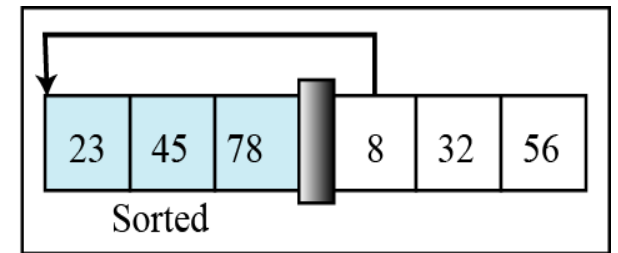
Ordenamiento por inserción



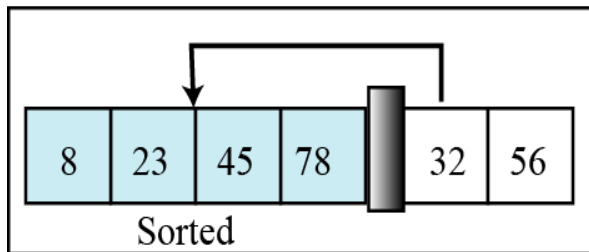
Original list



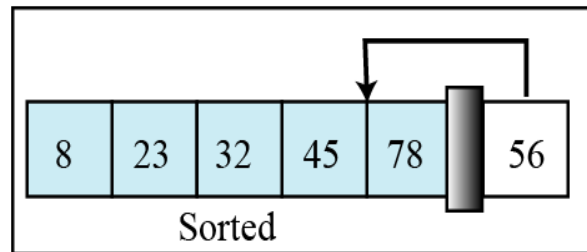
After pass 1



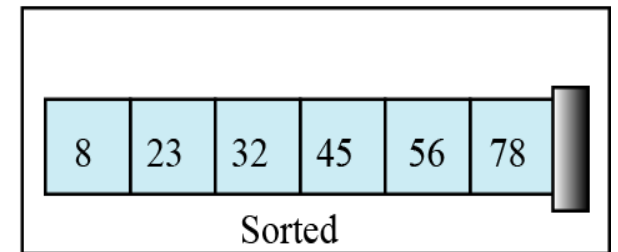
After pass 2



After pass 3



After pass 4



After pass 5

Ejemplo de ordenamiento por inserción

Ordenamiento rápido (Quicksort)

Quicksort es un algoritmo de divide y vencerás. Quicksort primero divide una larga lista en dos pequeñas sub-listas: los elementos menores y los elementos mayores. Quicksort puede ordenar a continuación las sub-listas de forma recursiva.

```
funcion quicksort(conjunto)
```

```
    var list less, greater
```

```
    if longitud(conjunto)  $\leq$  1
```

```
        return conjunto //conj. de 0-1 elementos ya esta ordenado
```

```
    Seleccione y remueva un valor pivot del conjunto
```

```
    Para cada x en conjunto
```

```
        if  $x \leq$  pivot then colocar x en less
```

```
        else colocar x en greater
```

```
    return concatenate(quicksort(less), pivot, quicksort(greater)).
```



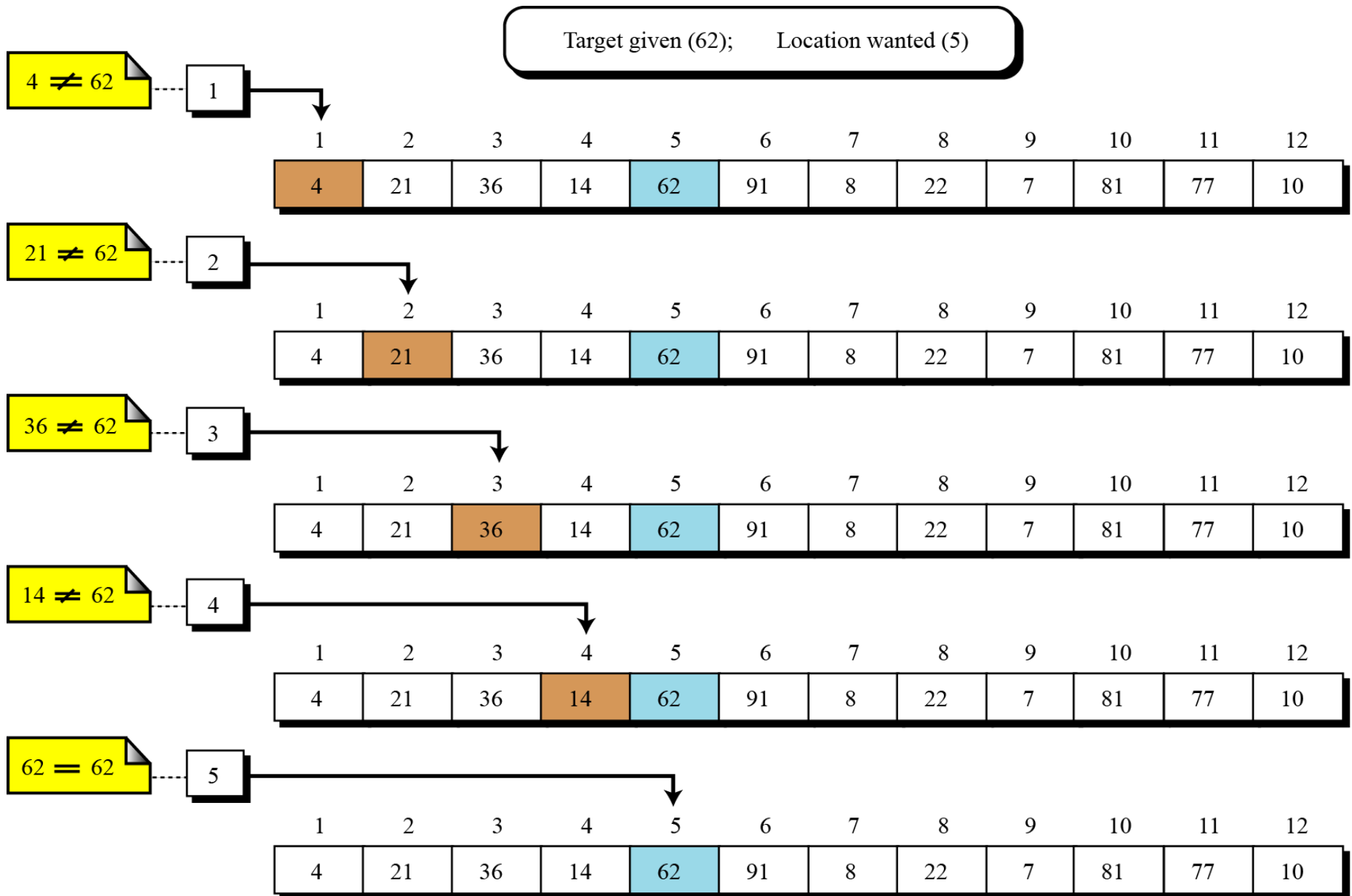
Búsqueda

Otro algoritmo común en ciencias de la computación está buscando, que es el proceso de encontrar la ubicación de un objetivo de entre una lista de objetos. En el caso de una lista, buscando los medios que le da un valor, queremos encontrar la ubicación del primer elemento de la lista que contiene ese valor. Hay dos registros de base para las listas: la **búsqueda secuencial** y **búsqueda binaria**. Búsqueda secuencial puede ser utilizado para localizar un elemento en cualquier lista, mientras que la búsqueda binaria requiere que la este primero ordenada.

Búsqueda secuencial

Búsqueda secuencial se utiliza si la lista en la que se debe buscar no está ordenada. En general, utilizamos esta técnica sólo para pequeñas listas o listas en las que no se busca a menudo. En otros casos, lo mejor es primero ordenar la lista y, a continuación buscar usando la búsqueda binaria discutida antes.

En una búsqueda secuencial, se empieza buscando el objetivo desde el principio de la lista. Continuamos hasta que encontremos el objetivo o alcanzemos el fin de la lista.



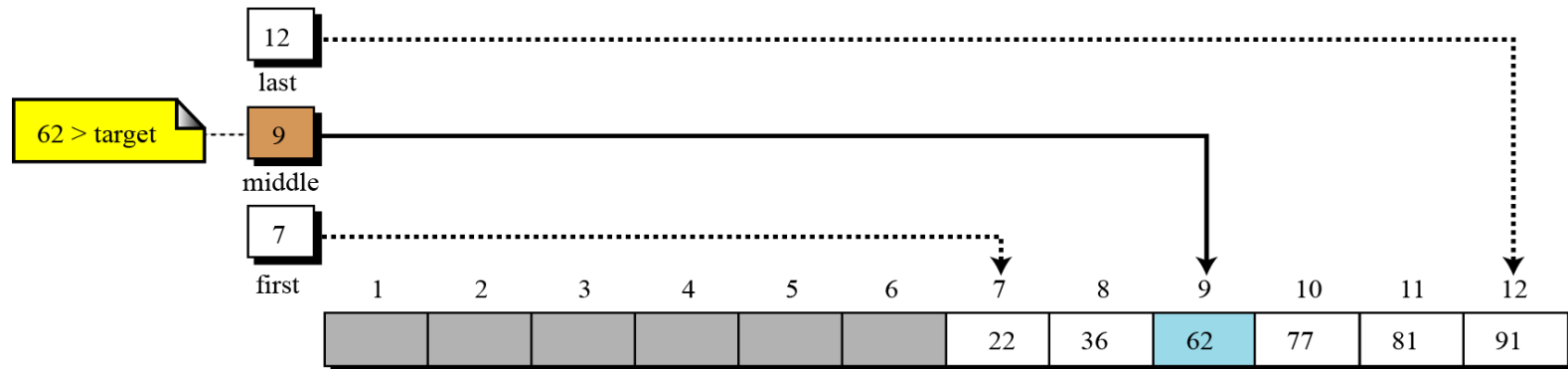
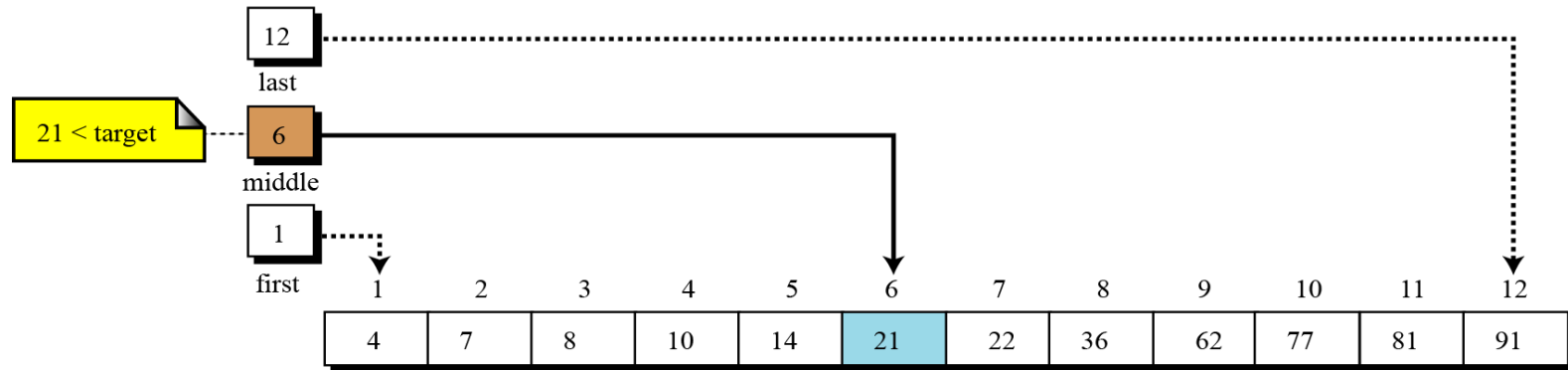
Ejemplo de búsqueda secuencial

Búsqueda binaria

El algoritmo de búsqueda secuencial es muy lento. Si tenemos una lista de un millón de elementos, tenemos que hacer un millón de comparaciones en el peor de los casos. Si la lista no está ordenada, esta es la única solución. Si la lista está ordenada, sin embargo, podemos usar un algoritmo más eficiente llamada búsqueda binaria. En general, los programadores utilizan una búsqueda binaria cuando la lista es grande.

Una búsqueda binaria comienza probando los datos en el elemento en la mitad de la lista. Esto determina si el objetivo está en la primera mitad o la segunda mitad de la lista. Si está en la primera mitad, no hay necesidad de controlar más el segundo segmento. Si está en la segunda mitad, no hay necesidad de controlar más el primer segmento. En otras palabras, eliminamos la mitad de la lista de cualquier consideración posterior.

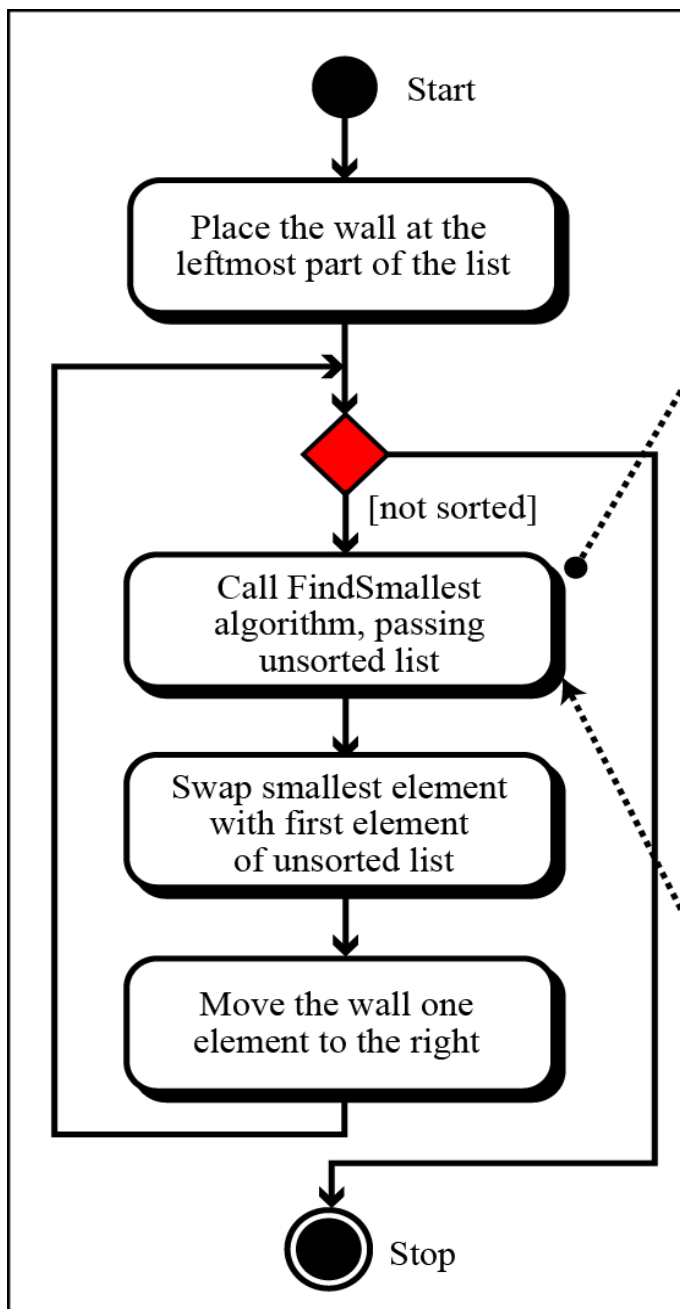
Target given (22); Location wanted (7)



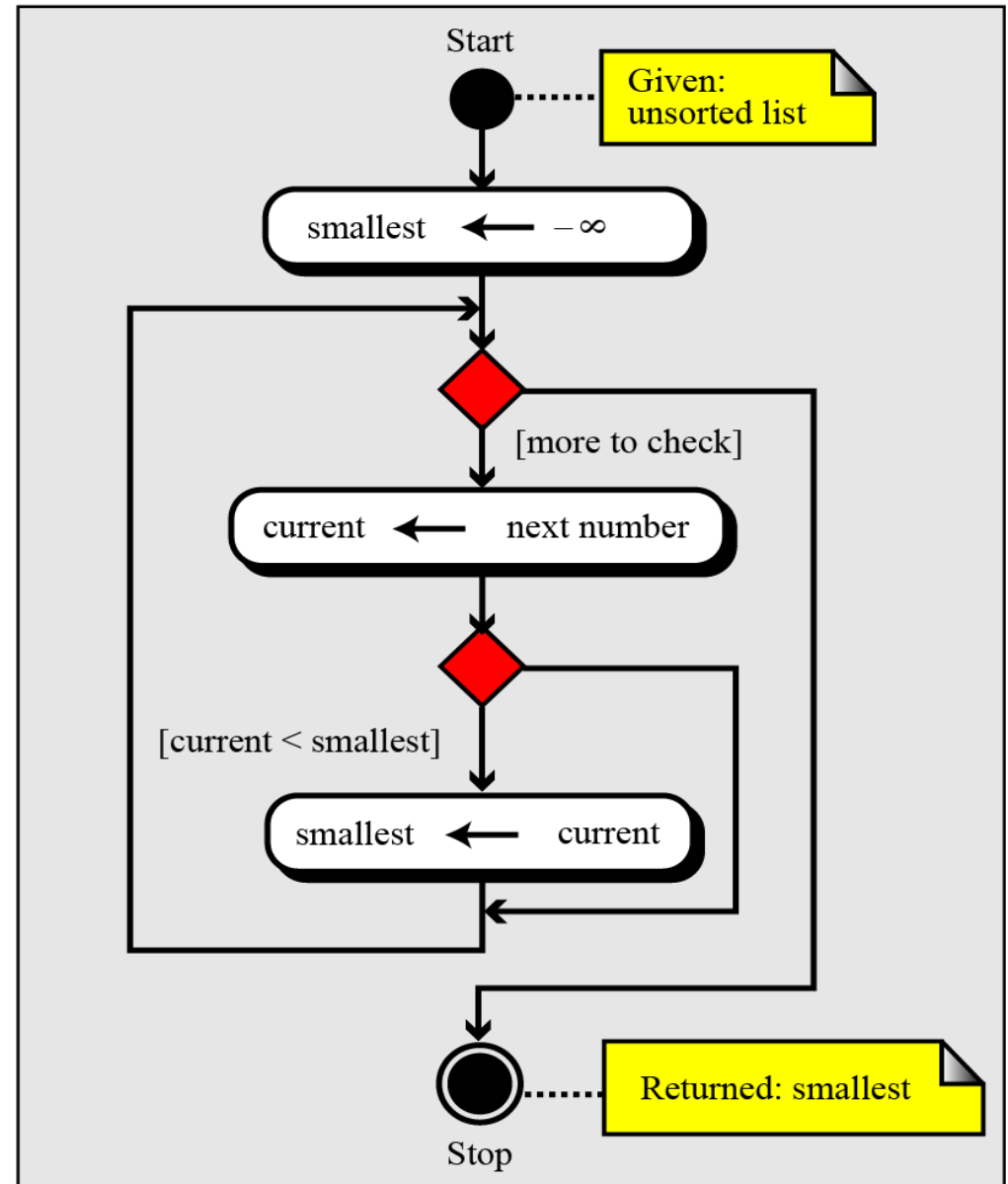
Ejemplo de búsqueda binaria

SUBALGORITMOS

Los tres constructores de programación descritos anteriormente nos permite crear un algoritmo para cualquier problema soluble. Los principios de la programación estructurada, sin embargo, requieren que un algoritmo se divida en pequeñas unidades llamadas subalgorithms. Cada subalgoritmo a su vez se divide en pequeños **subalgoritmos**. Un buen ejemplo es el algoritmo para ordenamiento por selección en la siguiente figura.



SelectionSort algorithm



FindSmallest algorithm

Concepto de subalgoritmo

Structure chart (mapa estructurado)

Otra herramienta que los programadores usan es el mapa estructurado. Un mapa estructurado es una herramienta de diseño de alto nivel que muestra la relación entre los algoritmos y subalgoritmos. Se utiliza principalmente a nivel de diseño en lugar de a nivel de programación.

RECURSIVIDAD

En general, hay dos enfoques para escribir algoritmos para resolver un problema. Uno de ellos utiliza la **iteración**, el otro usa la **recursividad**. La recursividad es un proceso en el que un algoritmo se llama a si mismo.

Definición iterativa

Para estudiar un ejemplo sencillo, consideremos el cálculo de un factorial. El factorial de un entero es el producto de los valores enteros entre 1 y el entero. La definición es iterativa. Un algoritmo es iterativo cada vez que la definición no implica el propio algoritmo.

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \cdots 3 \times 2 \times 1 & \text{if } n > 0 \end{cases}$$

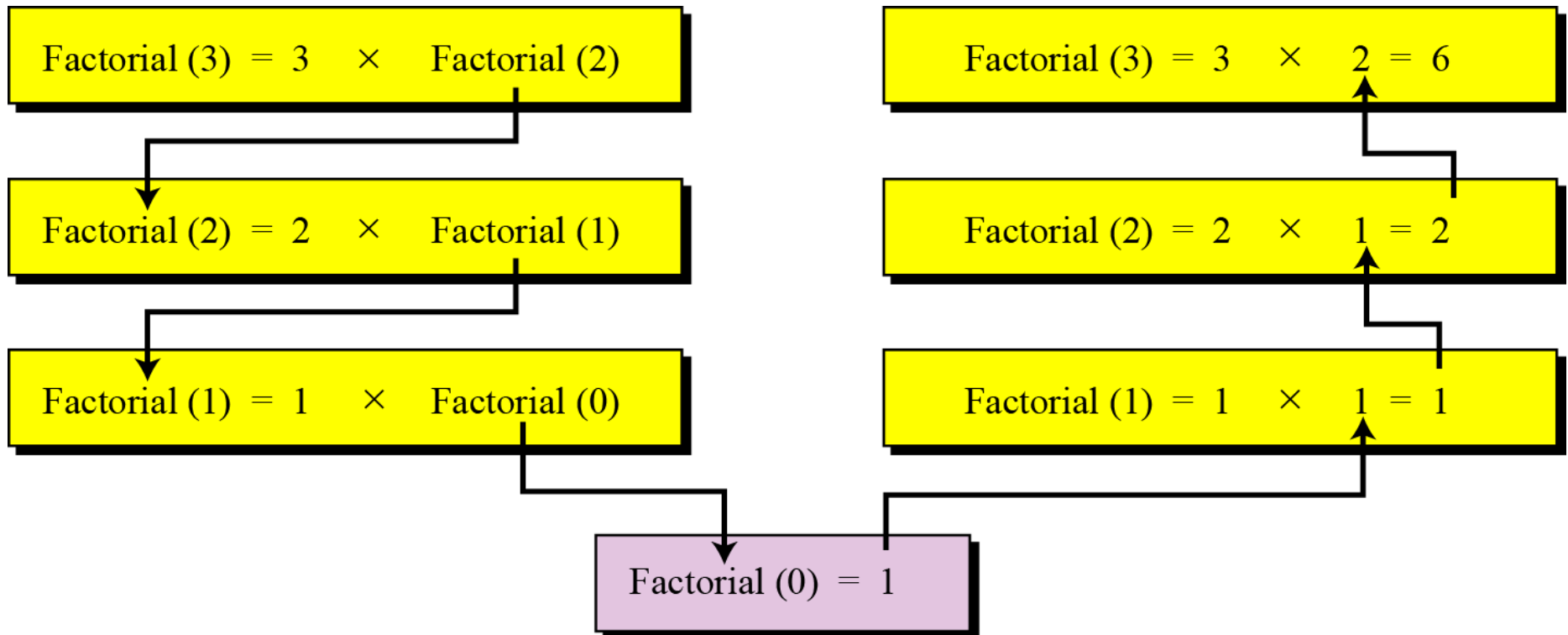
Definición iterativa de factorial

Definición recursiva

Un algoritmo se define de forma recursiva cada vez que el algoritmo aparece dentro de la propia definición. Por ejemplo, la función factorial se puede definir de forma recursiva, como se muestra en la Figura.

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times \text{Factorial}(n - 1) & \text{if } n > 0 \end{cases}$$

Definición recursiva de factorial



Determinación de la solución recursiva al problema del factorial

Solución iterativa

Esta solución implica generalmente un bucle.

An iterative solution to the factorial problem

Algorithm: Factorial (n)

Purpose: Find the factorial of a number using a loop

Pre: Given: n

Post: None

Return: $n!$

```
{  
     $F \leftarrow 1$   
     $i \leftarrow 1$   
    while ( $i \leq n$ )  
    {  
         $F \leftarrow F \times i$   
         $i \leftarrow i + 1$   
    }  
    return  $F$   
}
```

Solución recursiva

La solución no necesita un bucle, ya que el concepto de recursividad envuelve repetición.

Pseudocode for recursive solution of factorial problem

Algorithm: Factorial (n)

Purpose: Find the factorial of a number using recursion

Pre: Given: n

Post: None

Return: $n!$

{

if ($n = 0$) **return** 1

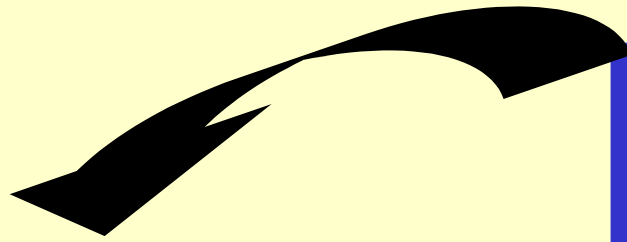
else **return** $n \times \text{Factorial}(n - 1)$

}

Una serie simple: dos soluciones

- $f(1) = 0$
- $f(2) = 1$
- ...
- $f(n) = f(n-1) + f(n-2)$

Una serie simple: solucion 1 (iterativa)

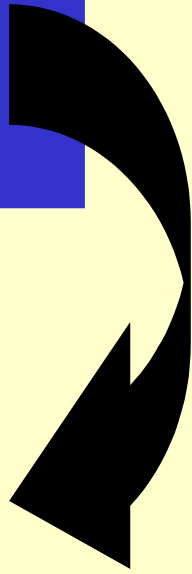


- $f(1) = 0$
- $f(2) = 1$
- ...
- $f(n) = f(n-1) + f(n-2)$

```
long fibo(int n)
{  si(n<= 2) retornar n-1;
   a = 0, b = 1, c, i;
   para( i = 3; i < n ; i++ )
   {
       c = a + b;
       a = b; b = c;
   }
   retornar c;
}
```


Una serie simple: solucion 2 (recursiva)

- $f(1) = 0$
- $f(2) = 1$
- ...
- $f(n) = f(n-1) + f(n-2)$



```
long f(int n)  
{ si(n <= 2)  
    retornar n-1;  
    retornar f(n-1)+f(n-2);  
}
```

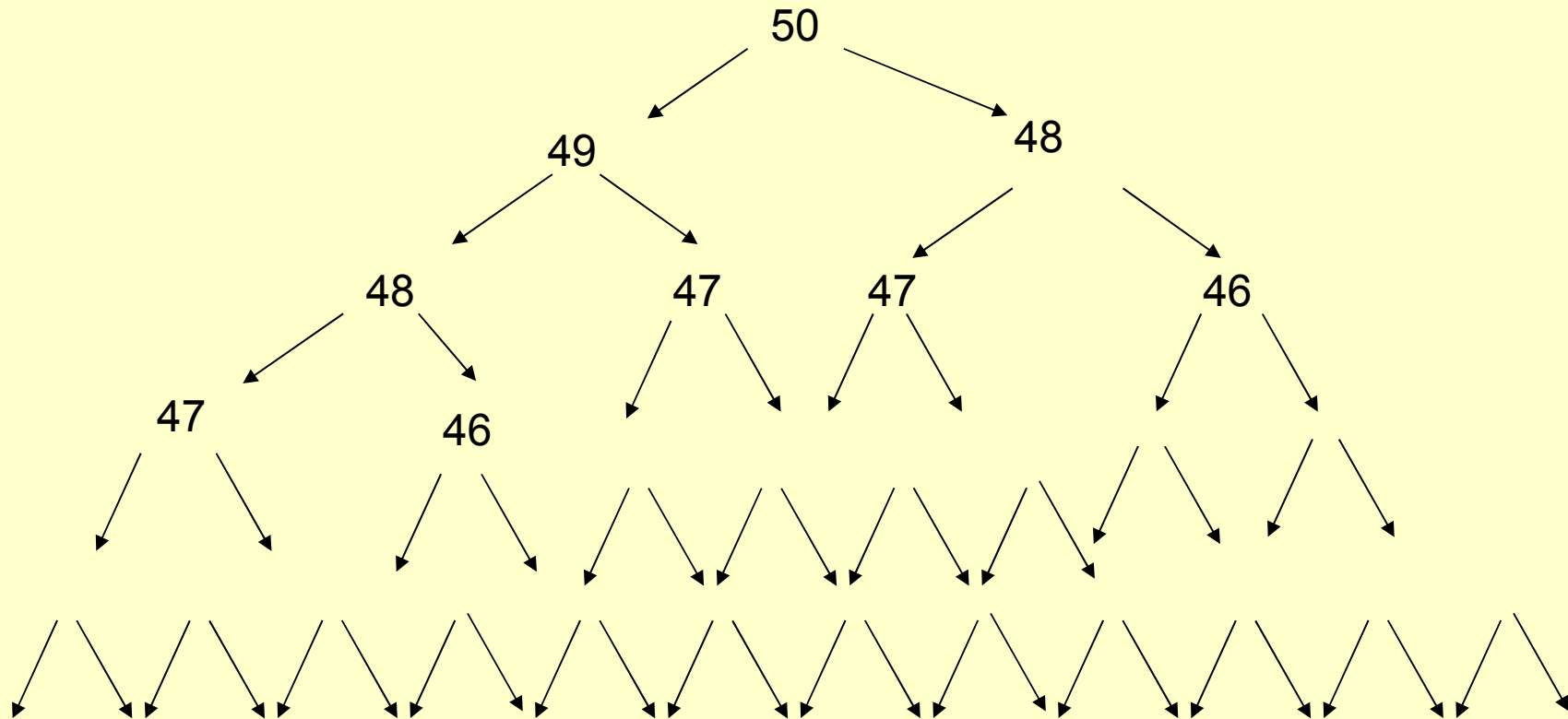
Una serie simple: dos soluciones

```
long fibo(int n)
{  si(n<= 2) retornar n-1;
   a = 0, b = 1, c, i;
   para( i = 3; i < n ; i++ )
   {
       c = a + b;
       a = b; b = c;
   }
   retornar c;
}
```

- $f(1) = 0$
- $f(2) = 1$
- ...
- $f(n) = f(n-1) + f(n-2)$

```
long f(int n)
{  si(n<= 2)
    retornar n-1;
    retornar f(n-1)+f(n-2);
}
```

Una serie simple: solucion 2 (recursiva)



$n \rightarrow 2^n$ operaciones !!!

Análisis de tiempo de ejecución

Approximate completion time for algorithms, $N = 100$

$O(\text{Log}(N))$	10^{-7} seconds
$O(N)$	10^{-6} seconds
$O(N * \text{Log}(N))$	10^{-5} seconds
$O(N^2)$	10^{-4} seconds
$O(N^6)$	3 minutes
$O(2^N)$	10^{14} years.
$O(N!)$	10^{142} years.

Lo mejor del siglo XX: Top 10 algoritmos

1. **1946: El algoritmo de Metropolis de Monte Carlo.** Mediante el uso de procesos aleatorios, este algoritmo ofrece una manera eficiente para conseguir respuestas a problemas que son demasiado complicados de resolver en forma exacta.
2. **1947: Método Simplex para programación lineal.** Una solución elegante a un problema común en planificación y toma de decisiones.
3. **1950: Método de iteración de subespacios de Krylov.** Una técnica para solución rápida de ecuaciones lineales que abundan en computación científica.
4. **1951: Enfoque descomposicional para cómputo de matrices.** Un conjunto de técnicas para el álgebra lineal numérica.
5. **1957: El compilador Fortran de optimización.** Cambia código de alto nivel en código informático eficiente y legible.

Lo mejor del siglo XX: Top 10 algoritmos

6. **1959: Algoritmo QR para cálculo de valores propios.** Otra operación crucial de matrices hecha rápida y práctica.
7. **1962: Algoritmos de ordenamiento rápido (Quicksort).** Para el manejo eficiente de grandes bases de datos.
8. **1965: Transformada Rápida de Fourier (FFT).** Tal vez el algoritmo de mayor alcance en matemáticas aplicadas en uso hoy en día,. Descompone formas de onda en componentes periódicos.
9. **1977: Detección de relación entre enteros.** Un método rápido para detectar ecuaciones simples satisfechas por colecciones de números aparentemente inconexos.
10. **1987: Método rápido multipolar.** Un gran avance en el tratamiento de la complejidad de los cálculos de n-cuerpos, aplicado en problemas que van desde la mecánica celeste al plegamiento de proteínas.