UNIVERSIDAD SAN SEBASTIAN

VOCACIÓN POR LA EXCELENCIA

Paradigmas de Programación

UNIVERSIDAD
SAN SEBASTIAN
VOCACIÓN POR LA EXCELENCIA

Recursión

Recursión

- Existen en matemáticas, computación y otras áreas definiciones que se sustentan en sí mismas. Por ejemplo:
  - Factorial de un número
  - Sucesión de Fibonacci
- Es frecuente en el reconocimiento de lenguajes:
  - Expresiones aritméticas
  - Estructura de frases y comandos.
- En programación se manifiesta por la invocación de un método a sí mismo, en forma directa o indirecta.

Ejemplo: Factorial

- El factorial de un número n, denotado por $n!$ se define como

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{en caso contrario} \end{cases}$$

- Se puede apreciar que la secuencia de llamadas será:

$$f(n), f(n-1), \ldots f(1), f(0).$$

- Todas estas llamadas quedarán suspendidas hasta llegar a $f(0)$ (*caso base*).
- El valor final se obtiene recorriendo la secuencia de llamadas en forma reversa recuperando los valores obtenidos.

Ejemplo: Factorial

- Un algoritmo para $n!$, en seudocódigo:

```
funcion factorial(n)
   si n = 0
      devolver 1
   sino
      devolver n * factorial(n – 1)
   fin si
fin función
```

  - *condición de salida* o *caso base* evita que la función se llame a sí misma indefinidamente.

- Todo algoritmo recursivo debe incluir al menos un caso base y garantizar que se ejecuta en algún momento para evitar la recursión infinita.

Como Construir una Función Recursiva

- Los procedimientos o funciones recursivas se componen de:
  - **Caso base**
    - Una solución simple para un caso particular (puede haber más de un caso base). ($n = 0$, en este ejemplo)
  - **Regla recursiva**
    - La regla recursiva se utiliza como una forma de acercarse al caso base en forma controlada. De esta manera
      - El problema se resuelve, recurriendo a otro idéntico pero de tamaño menor.
    - La manera en la cual el tamaño del problema disminuye asegura que el caso base eventualmente se alcanzará.

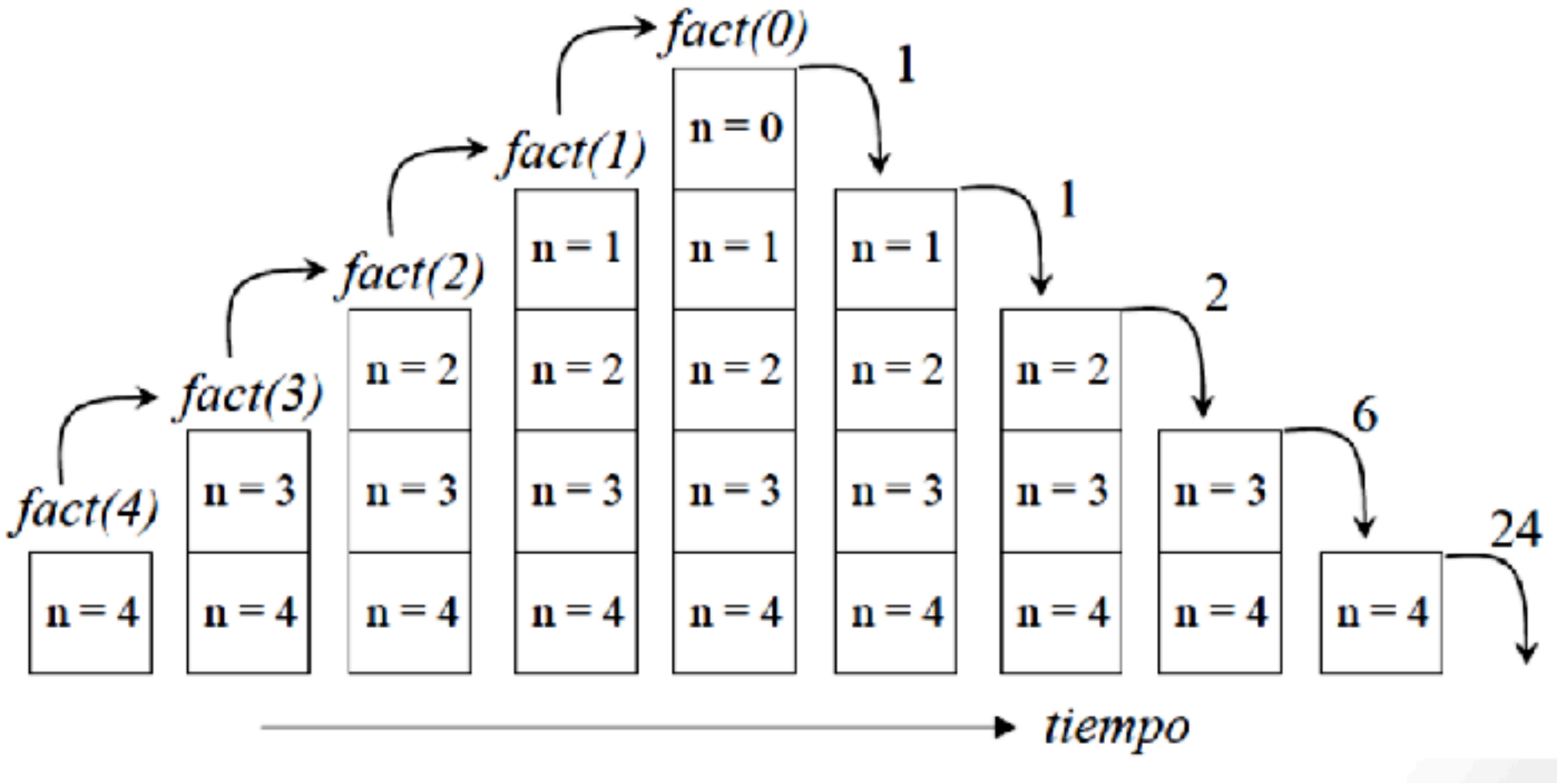
Ejemplo: Factorial (recursivo)

- La función en Java que implementa el algoritmo:

```
public static int factorial(int n){
        if (n == 0){
            return 1;
        }
        else
            return n * factorial(n - 1);
    }
```

- Problema propuesto: completar el programa y ejecutarlo con valores de n entre 1 y 20. ¿Que pasa con valores superiores?

Pila de Ejecución



UNIVERSIDAD SAN SEBASTIAN
VOCACIÓN POR LA EXCELENCIA

Complementos

- En el caso de los programas, es el *runtime* del lenguaje el que se encarga de administrar:
  - El espacio necesario para mantener “vivas” las activaciones de la función,
  - El espacio necesario para la transmisión de los parámetros,
  - El retorno de la ejecución al lugar en que fue suspendida.

- **Runtime** o entorno tiempo de ejecución es software que provee servicios para un programa en ejecución pero no es considerado en sí mismo como parte del sistema operativo.

UNIVERSIDAD SAN SEBASTIAN
VOCACIÓN POR LA EXCELENCIA

Complementos

- Cuando un procedimiento recursivo se llama recursivamente a sí mismo varias veces, para cada llamada se crean copias independientes de las variables declaradas en el procedimiento.

- Definiciones:
  - **Recursión directa**: se denomina de esta manera al caso en que un procedimiento incluye una llamada a sí mismo.
  - **Recursión indirecta**: en este caso, una cadena de llamadas a procedimientos termina con el último procedimiento llamando al primero.

Ejemplo: Factorial

- Problemas con la solución propuesta:
  - Esta propuesta funciona sólo con valores pequeños para n. Esto se debe al crecimiento explosivo del $n!$.
  - Aunque compacta y elegante, se puede encontrar mejores soluciones para el factorial.

- Problema propuesto: implemente esta misma versión recursiva en *C* o *C++* y compare los resultados, llevando n al punto en que reciban una respuesta errónea.

UNIVERSIDAD SAN SEBASTIAN
VOCACIÓN POR LA EXCELENCIA

¿Por qué programas recursivos?

- **Generalmente son cercanos a la descripción matemática**:
  - En muchos casos, la recursión es evidente
- **Suelen ser fáciles de analizar**:
  - La estructura de los algoritmos y del código resultante suele ser clara y sin la necesidad de “trucos” de programación.
- **Se adaptan bien a las *estructuras de datos recursivas***:
  - Así como existen algoritmos recursivos también existen estructuras de datos que son recursivas en su definición (árboles, por ejemplo).
- Los algoritmos recursivos ofrecen, en general, soluciones estructuradas, modulares y elegantemente simples.

Formato de una Función Recursiva

- Una función recursiva tiene los componentes:

```
<tipo_de_regreso><nom_fnc> (<param>) {
      [declaración de variables propias]
      [condición de salida]
      [instrucciones]
      [llamada a <nom_fnc> (<param>)]
   return <resultado>
 }
```

Cuándo Usar Recursión

- La recursión es una herramienta potente cuando:

a) Se utiliza con contextos de diseño de algoritmos con estrategias del tipo **divide y vencerás**.
   - Esta estrategia consiste en dividir un problema en 2 o más partes y, luego, resolver cada parte resultante por separado. Un ejemplo clásico es el *Quicksort*.

b) El problema a resolver tiene una estructura apropiada para ser resuelto recursivamente.
   - La estructura de datos más conocida en este caso es el *árbol*.

Cuándo no usar recursión

- La recursión consume recursos de computador. Por ello es conveniente descartarla si:
  - Se puede elaborar un algoritmo alternativo que ocupe iteraciones.
    - Por ejemplo, el propio factorial tiene una solución alternativa.
  - Cuando las estructuras de datos de los parámetros sean voluminosas.
    - No es lo mismo un entero como parámetro que un objeto complejo, un grafo, por ejemplo.

UNIVERSIDAD SAN SEBASTIAN
VOCACIÓN POR LA EXCELENCIA

Iteración vs recursión

ITEM	ITERACION	RECURSION
Repetición	Ciclo explícito (`for`, `while`, ...)	Llamada recursiva (directa o indirecta)
Término de la repetición	El criterio de término está asociado a la condición del ciclo	Se llega a examinar el caso base
Ciclo infinito	Posible	Posible

Ejemplo: Sucesión de Fibonacci

- La sucesión fue descrita y dada a conocer en Occidente por Fibonacci (Leonardo de Pisa) como la solución a un problema de la cría de conejos.

- La sucesión queda definida por la función:

$$f(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ f(n-1) + f(n-2) & \text{si } x \geq 2 \end{cases}$$

Ejemplo: Sucesión de Fibonacci

- Los primeros valores de la sucesión son:

$$f(2) = 1, f(3) = 2, f(4) = 3, \ldots, f(8) = 21, \ldots$$

- Se puede reconocer:
  - Caso base: $f(0) = 0,\ f(1) = 1$
  - Caso recursivo: $f(n) = f(n-1) + f(n-2)$

Algoritmo de Fibonacci

- Formulación en seudocódigo:

```
entero f(entero n)
inicio
     si n ≤ 1 entonces   //condición base
          retornar n;
     sino                //condición recursiva
          retornar f(n − 1) + f(n − 2);
     fin si
fin
```

Algoritmo de Fibonacci

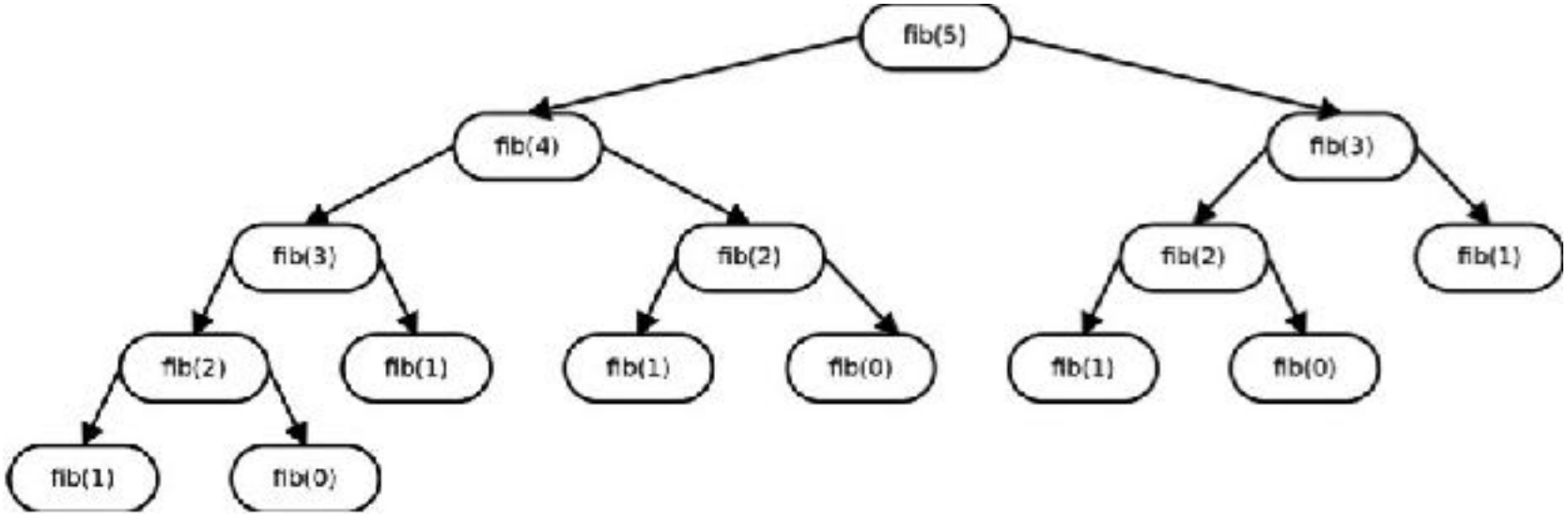
- El algorimo se traduce fácilmente a varios lenguajes. En particular, en Java, se puede escribir:

```
private static int f(int n) {
        if(num == 0 || num==1)
            return n;
        else
            return f(n-1) + f(n-2);
}
```

- Usualmente, en Java u otro lenguaje, los nombres serán más explícitos.

Algoritmo de Fibonacci

- La ejecución del programa producirá un estructura de llamadas, puede apreciarse que muchas éstas son redundantes.
- ¿Es una buena solución?



Algoritmo de Fibonacci

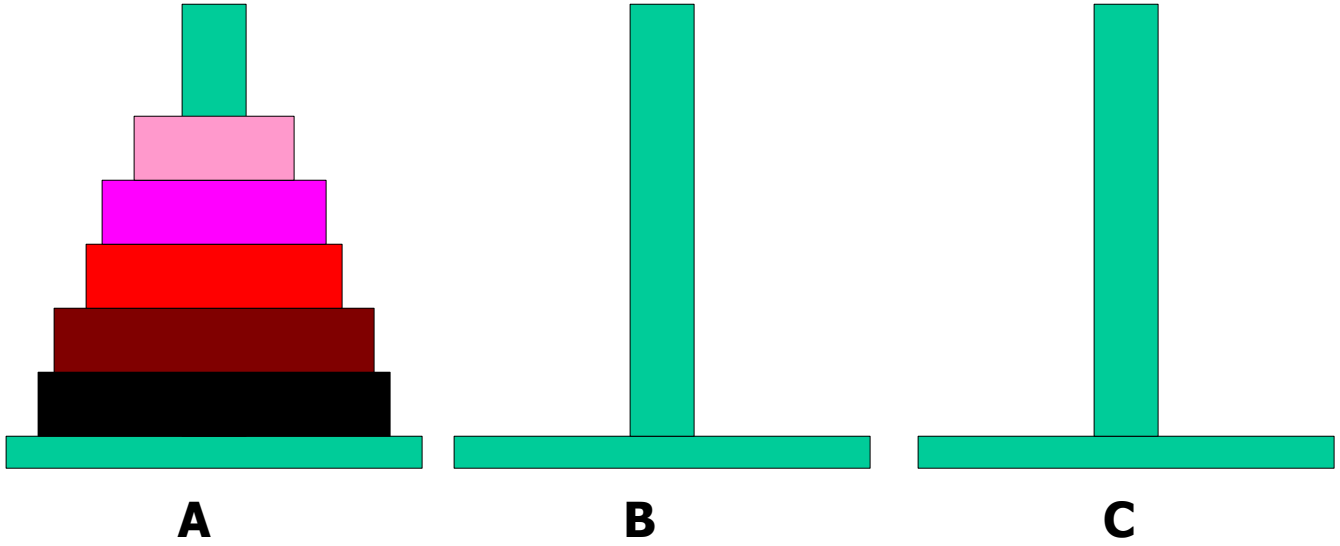
- Es necesario dejar en claro que este desarrollo tiene sólo fines académicos, hay formas mucho más eficientes de resolver este problema.
  - Iterativas y por aproximación.
- Problema propuesto:
  - Escribir programas en algún lenguaje de programación que implementen esta función, tanto recursivamente como iterativamente..
  - Verificar la corrección de la implementación.
  - Modificar el programa para desplegar cuál es la activación que se está ejecutando en ese momento.

UNIVERSIDAD SAN SEBASTIAN
VOCACIÓN POR LA EXCELENCIA

Ejemplo: Torres de Hanoi

- Se dispone de tres pértigas A, B y C y un conjunto de *n* discos, todos de distintos tamaños.
- El problema comienza con todos los discos colocados en la pértiga A de tal forma que ninguno de ellos puede estar sobre uno más pequeño a él.
- El propósito del problema es lograr apilar los *n* discos, en el mismo orden, pero en la pértiga C.

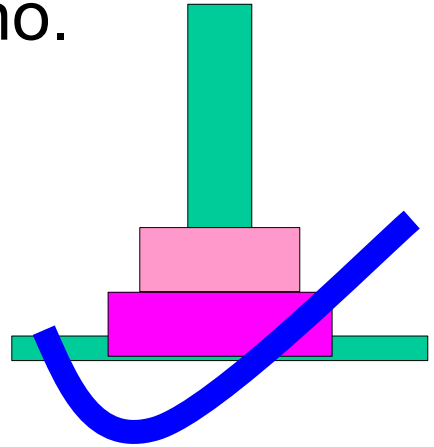
Ejemplo: Torres de Hanoi



- Situación inicial, con $n = 5$.

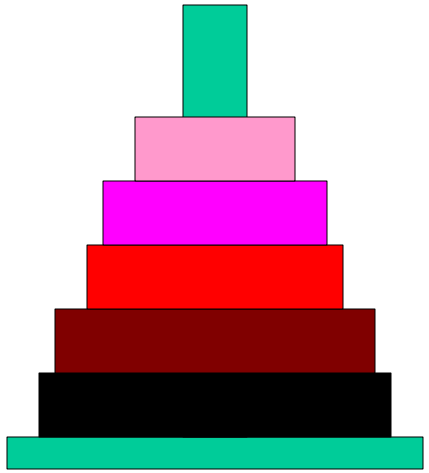
Ejemplo: Torres de Hanoi

- Durante el proceso se puede colocar los discos en cualquier pértiga, pero debe respetarse las siguientes reglas:
  - Solo se puede mover el disco superior de cualquiera de las pértigas.
  - Un disco más grande nunca puede estar encima de uno más pequeño.



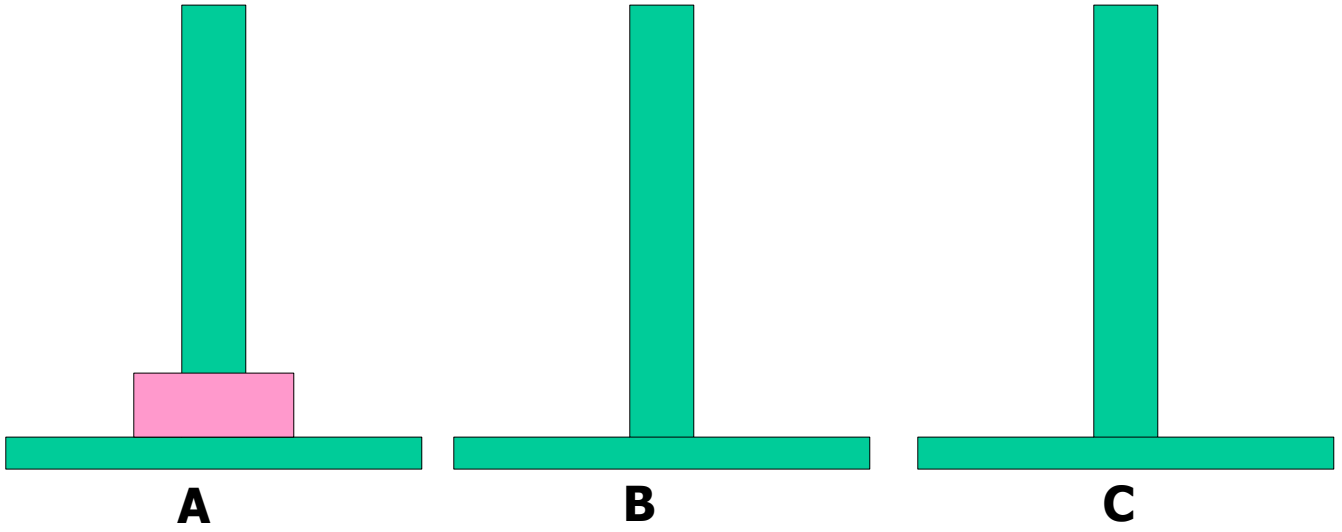
Cómo resolver el problema

- Estudiando el problema, se encuentra se puede plantear distintas situaciones.



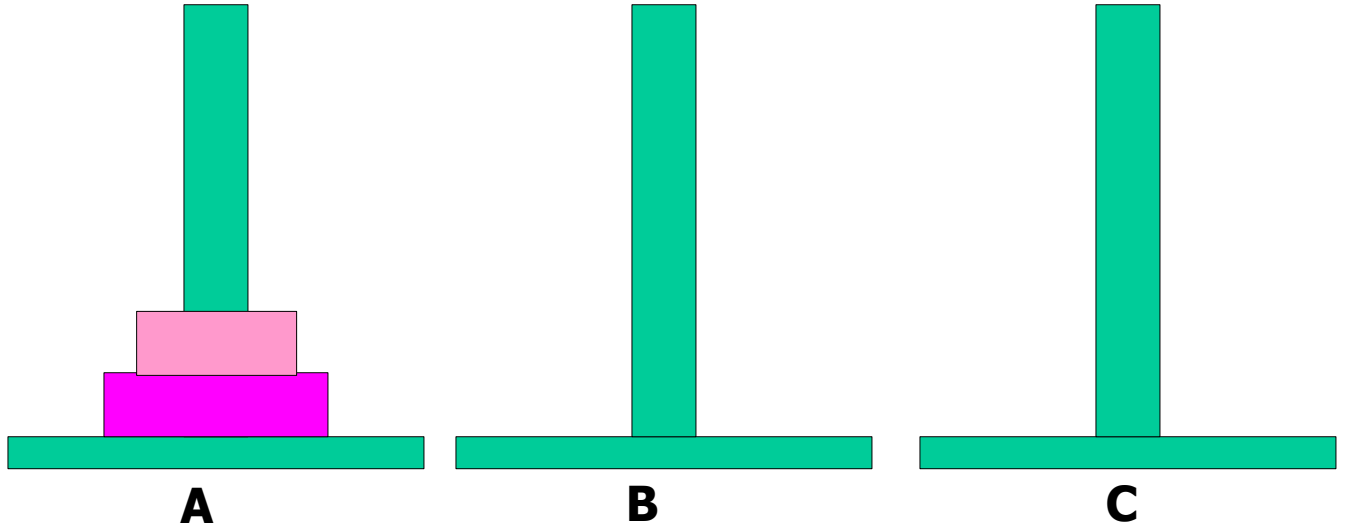
Caso con un solo disco

Situación muy simple, basta con desplazar el (único) disco al destino

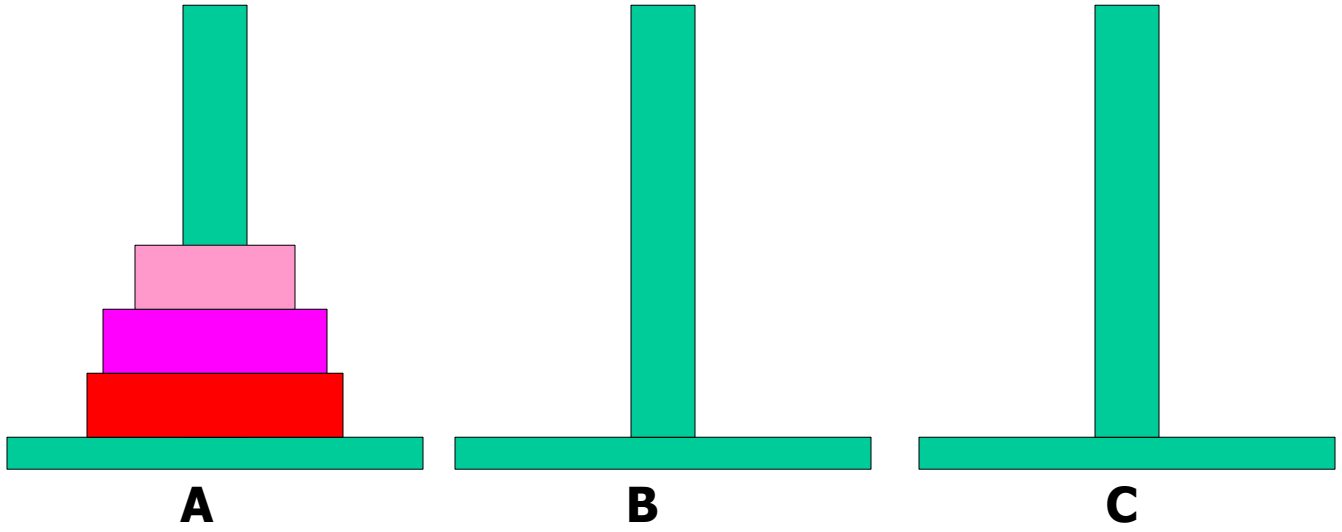


Caso con 2 discos

Colocando el disco más pequeño en la pértiga **B**, pasando el grande a la pértiga **C** y después moviendo el que está en **B** a **C**.



¿Cómo se resolvería el caso de 3 discos? ¿Y con *n* discos?



- En https://www.youtube.com/watch?v=Q3U6PRZDjTA puede verse una animación con 5 discos.

UNIVERSIDAD SAN SEBASTIAN
VOCACIÓN POR LA EXCELENCIA

El problema de las Torres de Hanoi

- Entonces, por las razones expuestas, el algoritmo puede definirse de la siguiente manera:
  - Si es un solo disco, se mueve éste de A a C.
  - En otro caso, suponiendo que *n* es la cantidad de discos que hay que mover.
    - Se mueve los *n-1* discos superiores —es decir, sin contar el más grande— de A a B (utilizando a C como auxiliar).
    - Se mueve el último disco (el más grande) de A a C.
    - Se mueve los discos que quedaron en B a C (utilizando a A como auxiliar).

UNIVERSIDAD SAN SEBASTIAN
VOCACIÓN POR LA EXCELENCIA

Implementación en C++

```
//
//   Version en C++ para las torres de Hanoi
//
# include <iostream>
using namespace std;
typedef char pole ;
void hanoi ( int N , pole org , pole aux , pole dst ) {
    if ( N ==1)
        cout << " Mueve un disco de " << org << " a " << dst << endl ;
    else {
        hanoi ( N - 1 , org , dst , aux );
        cout << " Mueve un disco de " << org << " a " << dst << endl ;
    hanoi ( N - 1 , aux , org , dst );
    }
}

int main() {
    int n;
    cout << "Numero de discos: "; cin >> n;
    cout << "\n";
    hanoi(n, 'A', 'B', 'C');
}
```

UNIVERSIDAD SAN SEBASTIAN
VOCACIÓN POR LA EXCELENCIA

Corolario

- Muchas veces es posible dividir un problema en subproblemas más pequeños, generalmente del mismo o menor tamaño, resolver los subproblemas y entonces combinar sus soluciones para obtener la solución del problema original.

- **Dividir para vencer** es una técnica natural para las estructuras de datos, ya que por definición están compuestas por piezas. Cuando una estructura de tamaño finito se divide, las últimas piezas ya no podrán ser divididas.

UNIVERSIDAD
SAN SEBASTIAN
VOCACIÓN POR LA EXCELENCIA

FIN