Recursividad, excepciones, testeo Estrategia y ordenamientos

- Materia Algoritmos y Estructuras de Datos
- Cátedra Schmidt
- Recursividad, manejo de errores con excepciones y la estrategia divide y reinaras

Recursividad

Una función recursiva es aquella que se llama a si misma para resolverse.

Una función recursiva tiene al menos una llamada a si misma, con el argumento modificado.
El proceso de llamadas recursivas siempre tiene que acabar en al menos una llamada a la función que se resuelve de manera directa, sin necesidad de invocar de nuevo la función (condición de corte). La modificación en el argumento debe asegurar el acercamiento, a lo largo de las sucesivas invocaciones, a la condición de base o de corte, para la cual el resultado se obtiene de modo inmediato, o al menos no recursivo.

Esto siempre es así para que en determinado momento se corten las llamadas reiterativas a la función y no haya un bucle infinito de invocaciones.

Ejemplo de factorial

La recursión es un medio particularmente poderoso en las definiciones matemáticas. Para apreciar mejor cómo es una llamada recursiva, estudiemos la descripción matemática de factorial de un número entero no negativo:

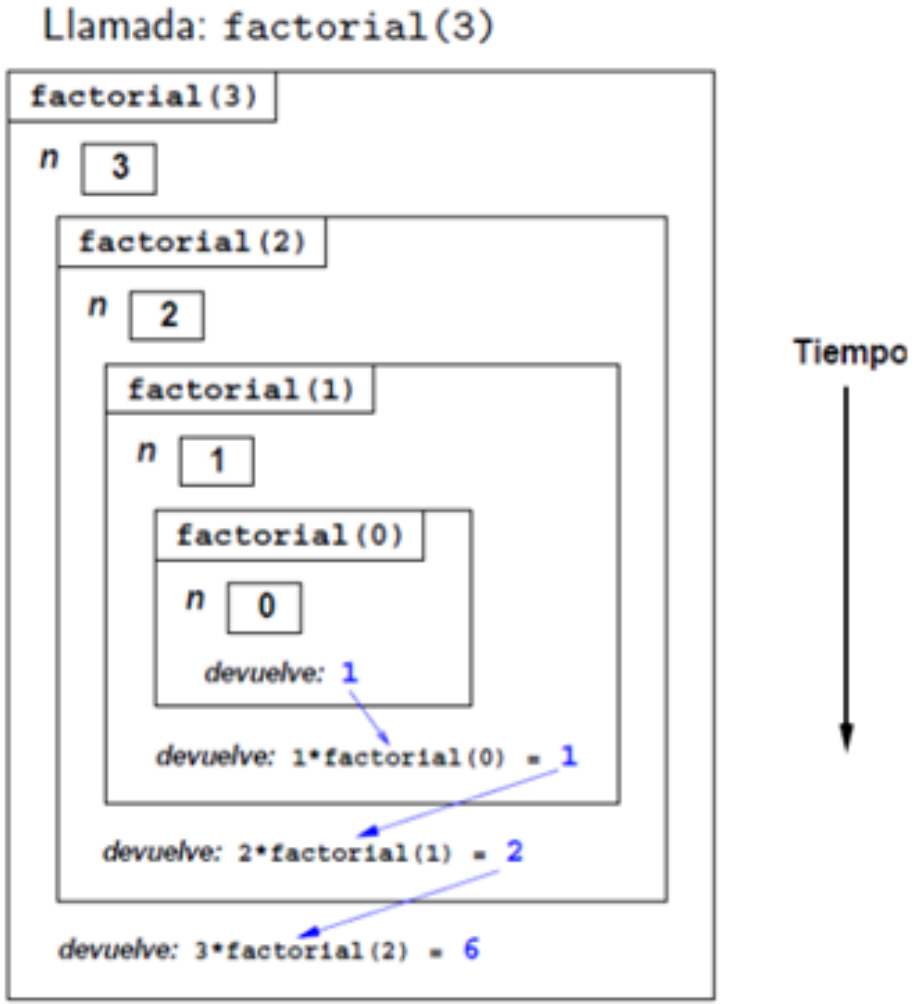
$$n! = \begin{cases} 1, & \text{si } n = 0 \\ n * (n - 1) * (n - 2) * .. * 1 = n * (n - 1)! & \text{si } n > 0 \end{cases}$$

Esta definición es recursiva ya que expresamos la función factorial en términos de sí misma.
Ejemplo: 4! = 4 * 3!
Por supuesto, no podemos hacer la multiplicación aún, puesto que no sabemos el valor de 3!
3! = 3 * 2!
2! = 2 * 1!
1! = 1 * 0!
0! = 1
Podemos ahora completar los cálculos
1! = 1 * 1 = 1
2! = 2 * 1 = 2
3! = 3 * 2 = 6
4! = 4 * 6 = 24

$$N! = \begin{cases} 1 & \text{si } N = 0 \text{ (base)} \\ N * (N - 1)! & \text{si } N > 0 \text{ (recursión)} \end{cases}$$

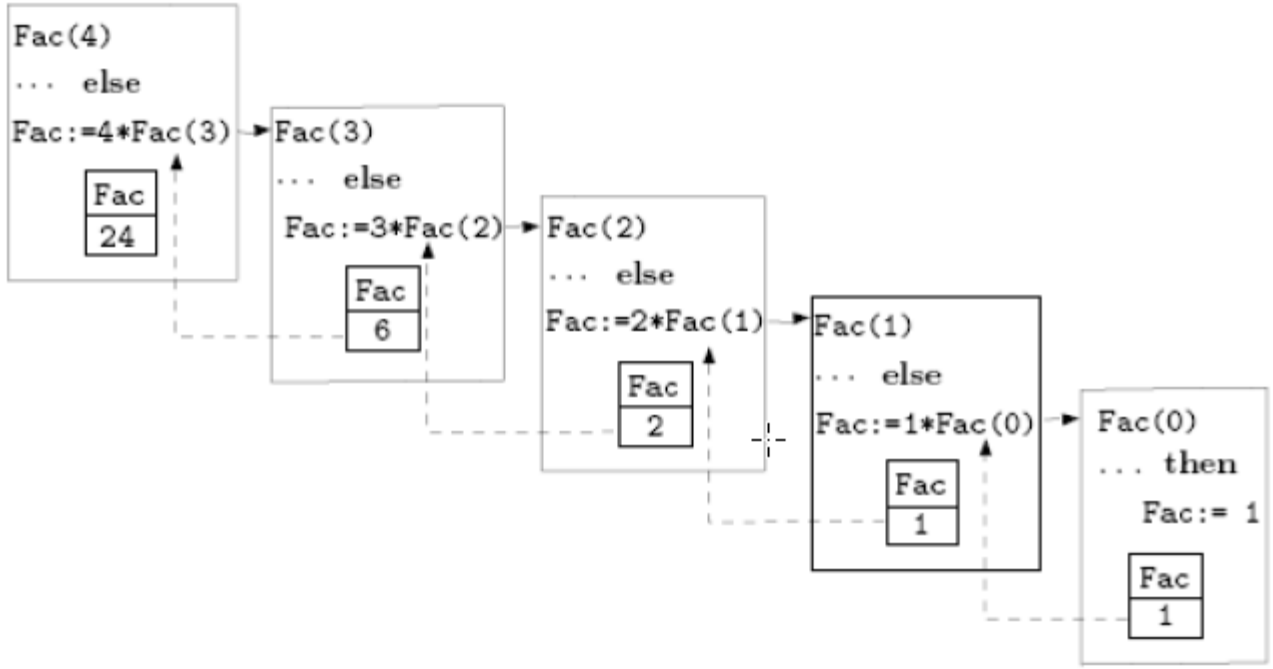
```
int factorial(unsigned int x) {
      if((x==0) ||
         (x==1)) {
            return 1;
      }
      return x*(factorial(x-1));
}
```

Retomando el ejemplo 1 Cómo funciona la pila o stack en la recursividad



Ahora desapilamos y vamos reemplazando el resultado obtenido en la instancia posterior:

1- Desapilo Instancia 6	~~0~~	~~factorial(0)~~	1
2- Desapilo Instancia 5	~~1~~	~~1 * factorial(0)~~	1 * 1 = 1
3- Desapilo Instancia 4	~~2~~	~~2 * factorial(1)~~	2 * 1 = 2
4- Desapilo Instancia 3	~~3~~	~~3 * factorial(2)~~	3 * 2 = 6
5- Desapilo Instancia 2	~~4~~	~~4 * factorial(3)~~	4 * 6 = 24
6- Desapilo Instancia 1	~~5~~	~~5 * factorial(4)~~	5 * 24 = **120**
	N	n* factorial(n-1)	**factorial**



Tipos de Recursividad:

•**Simple** o **múltiple**
En la recursividad simple hay un solo llamado recursivo en cada rama del algoritmo (es decir, puede haber más de un llamado en el código, pero en cada nivel de ejecución sólo se ejecutará uno). (Ej Factorial)
En la recursividad múltiple hay más de un llamado al menos en una rama y se opera luego con los resultados obtenidos. (ej. Fibonacci con el código habitual). La recursividad simple puede ser directa, indirecta o línea.

•**De cola (“tail recursion”) o no**
En la recursividad de cola lo último que se hace es el llamado recursivo. La función retorna lo que recibe de la invocación recursiva realizada, sin ninguna operación adicional sobre ese resultado. Si el lenguaje lo habilita (por ejemplo, se puede en Java), el sistema puede optimizar el uso del stack de la siguiente manera: al detectar la recursividad de cola, no apila un stack frame sobre otro, sino que reutiliza el mismo. (Ej MCD por Euclides)
Algunas funciones con recursividad no de cola pueden reescribirse para que tengan esta característica, lo cual es útil si el lenguaje permite la optimización mencionada

•**Anidada o no**
En la recursividad anidada al menos un llamado recursivo tiene como argumento el valor retornado por la misma función**.** (Ej: Función de Ackermann)

•**Mutua o no**
En la recursividad mutua se tienen al menos dos funciones que trabajan de la siguiente manera: para cada una si se cumple la condición de base, se retorna un valor sin invocación recursiva. De otro modo se invoca a la otra (o a alguna de las otras) funciones luego de haber modificado el argumento de modo que se vaya aproximando a la condición de corte respectiva. (Ej Fibonacci calculado con dos funciones con recursividad mutua)

Recursividad simple. Potencia

Recursividad Directa

•Ocurre cuando una función se llama a sí misma directamente.

•Ejemplo: Un método para calcular el factorial de un número por ejemplo.

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \cdot x^{n-1} & \text{si } n > 0 \end{cases}$$

```
int potencia(int base, int expo){
  if (expo==0)
    return 1;
  else
    return base * potencia(base,expo-1);
}
```

Recursividad simple. Producto recursivo.

$$producto(a,b) = \begin{cases} 0 & \text{si } b = 0 \\ a + producto(a, b-1) & \text{si } b > 0 \end{cases}$$

```
int producto(int a, int b){
  if (b==0)
    return 0;
  else
    return a+producto(a,b-1);
}
```

Recursividad Indirecta

Sucede cuando una función no se llama a sí misma directamente, sino que lo hace a través de otra función. Es decir, hay una cadena de llamadas recursivas entre varias funciones.

```
public void funcionA() {
    funcionB();
}

public void funcionB() {
    funcionA();
}
```

Recursividad Lineal

En este tipo de recursividad, una función se llama a sí misma una sola vez en cada nivel de recursión.

```java
public int suma(int n) {
    if (n == 0) {
        return 0;
    }
    return n + suma(n - 1);
}
```

Recursividad multiple. Fibonacci

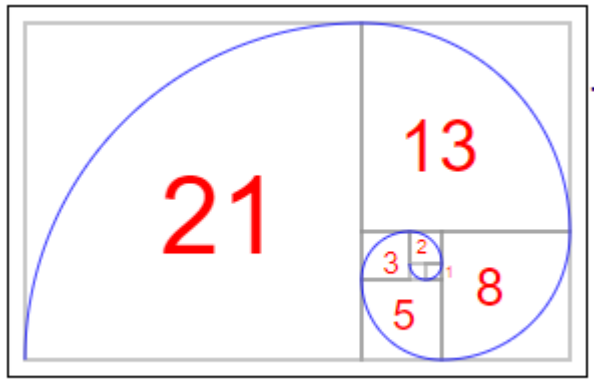


Figura 1. Espiral de Fibonacci

La función recursiva de **Fibonacci** se define como

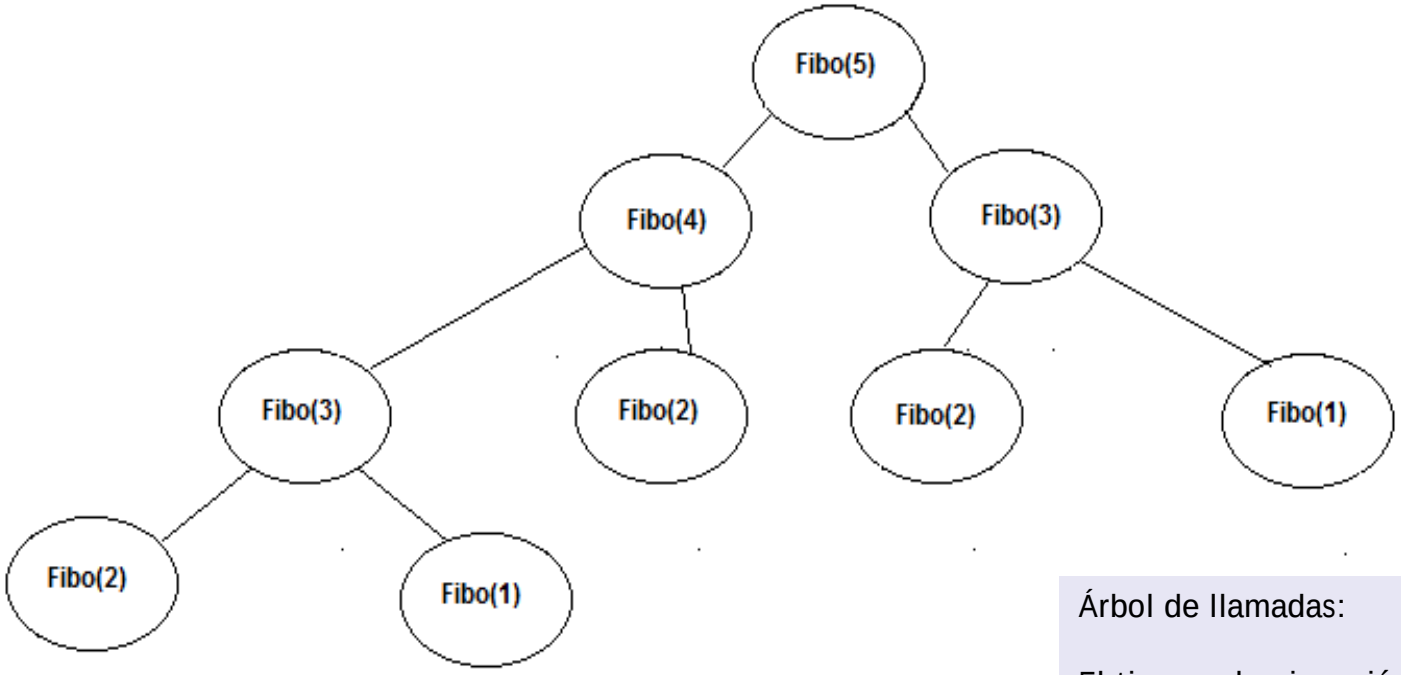
$f(0)=0$, $f(1)=1$ y

$f(n) = f(n-1) + f(n-2)$ para $n \geq 2$.

Los primeros téminos de la sucesión son $0, 1, 1, 2, 3, 5, 8, 13, 21$,... La **espiral de Fibonacci** de la *Figura* representa esta sucesión como arcos de cuadrante inscritos en rectángulos cuyos lados siguen esta sucesión.

```
function fibonacci1(n) {
    if (n<2){
        return n;
    } else {
        return fibonacci1(n-1) + fibonacci1(n-2);
    }
}
```

Árbol de llamadas



Árbol de llamadas:

El tiempo de ejecución está en función del número de nodos del mismo.

El espacio requerido del stack está en función del número de niveles del árbol de llamadas.

Graficar el árbol de llamadas de los ejemplos anteriores para un determinado valor de entrada

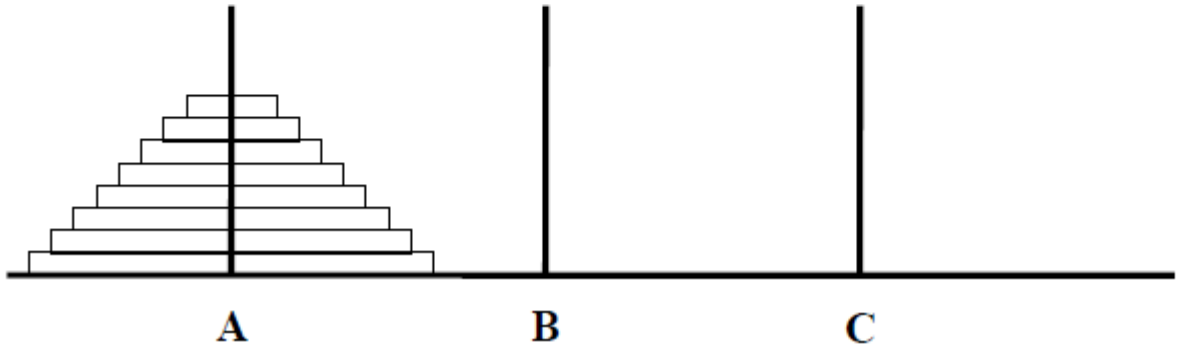
Recursividad múltiple. Torres de Hanoi

Torres de Hanoi

Se dan tres barras verticales y *n* discos de diferentes tamaños. Los discos pueden apilarse en las barras formando "torres". Los discos aparecen inicialmente en la primer barra en orden decreciente y la tarea es mover los **n** discos de la primer barra a la tercera de manera que queden ordenados de la misma forma inicial. Las reglas a cumplir son:

En cada paso se mueve exactamente un disco desde una barra a otra.

En ningún momento pueden situarse un disco encima de otro más pequeño.



Este ejemplo de problema ilustra la situación (no son muchas) en que un algoritmo recursivo permite resolver la situación de forma más "directa" que uno iterativo.

Vamos a intentar resolverlo probando con 1 ó 2 discos:

1 disco
Mover el disco 1 de A a C

2 discos
Mover el disco 1 de A a B
Mover el disco 2 de A a C
Mover el disco 1 de B a C

Veamos qué pasa con 3 y 4 discos:

3 discos
Mover el disco 1 de A a C
Mover el disco 2 de A a B
Mover el disco 1 de C a B
Mover el disco 3 de A a C
Mover el disco 1 de B a A
Mover el disco 2 de B a C
Mover el disco 1 de A a C

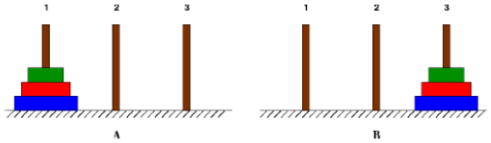
- Si es un solo disco, lo movemos de A a C.
- En otro caso, suponiendo que n es la cantidad de aros que hay que mover
  - Movemos los n-1 aros superiores - es decir, sin contar el más grande- de A a B (utilizando a C como auxiliar).
  - Movemos el último aro (el más grande) de A a C.
  - Movemos los aros que quedaron en B a C (utilizando a A como auxiliar).

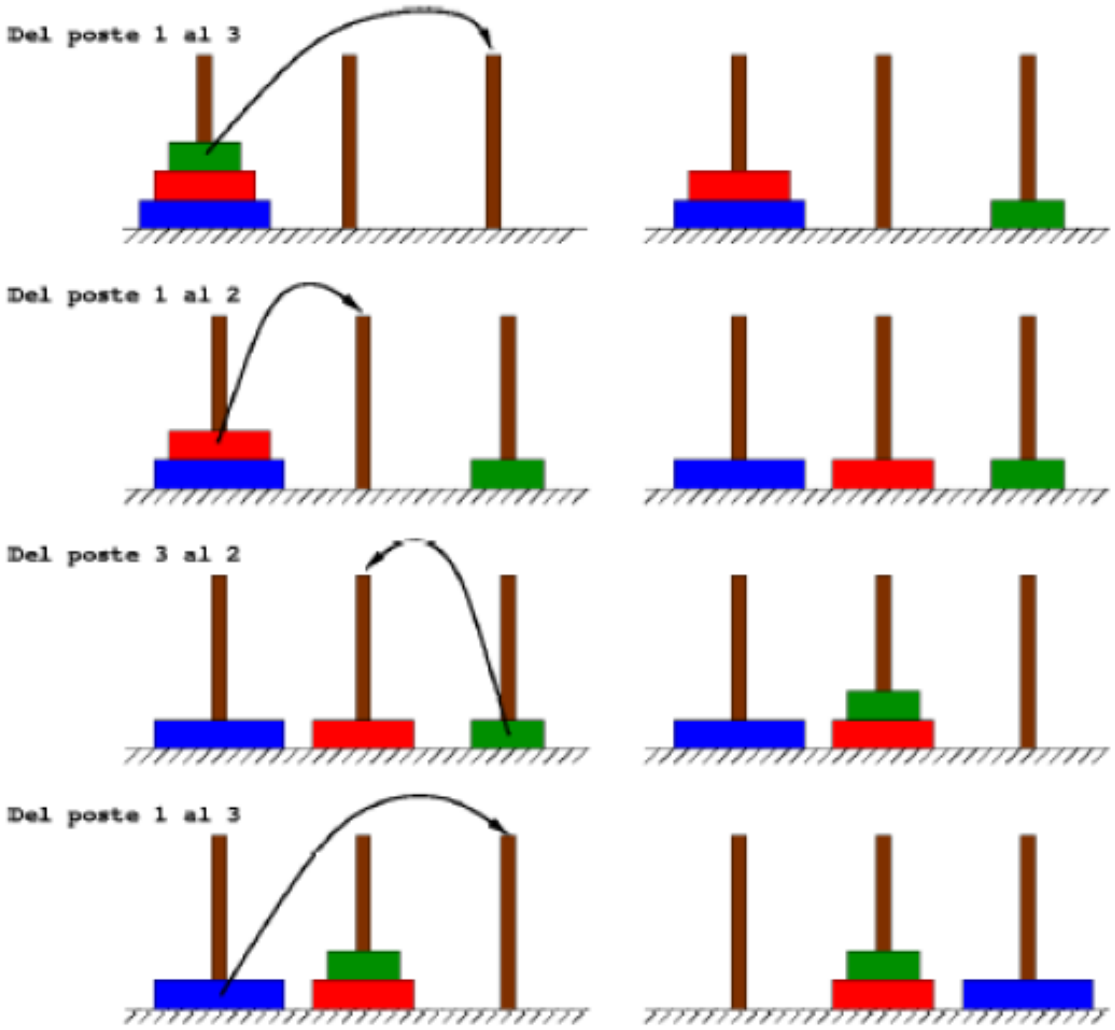
Considerando A origen, C destino, B auxiliar

```
void Hanoi   (int N, char ori, char dest, char aux)
{
if (N>0)
 {
   Hanoi (N-1, ori, aux, dest);
   std :: cout <<"Mover disco desde"<<ori <<"hasta" <<dest;
   Hanoi (N-1, aux, dest, ori);
 }
}
```

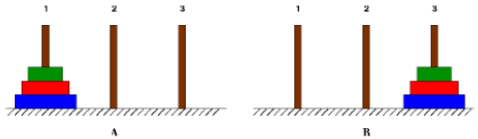
¿Cómo queda el árbol de llamadas del algoritmo que resuelve las Torres de Hanoi?

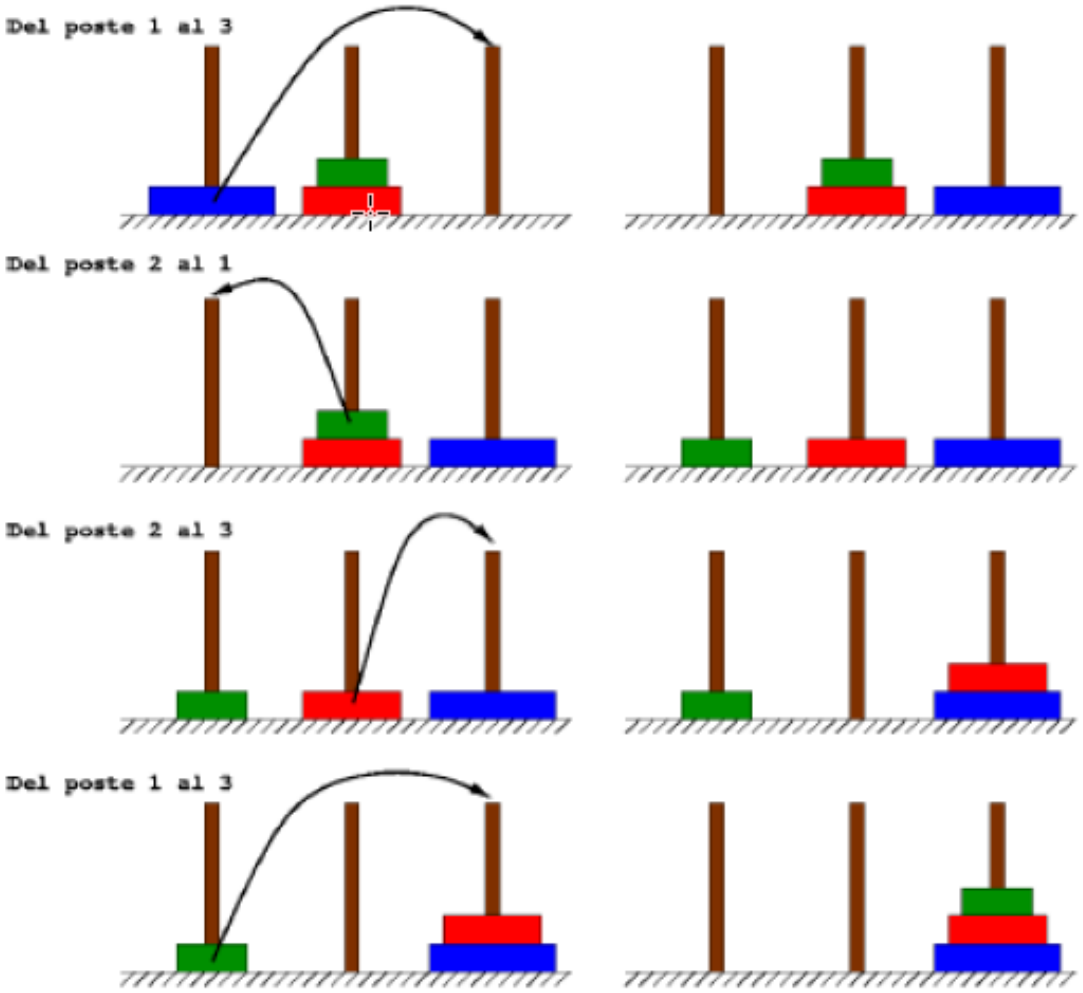
Solución de Hanoi para tres discos - 1





Solución de Hanoi para tres discos - 2





Recursividad de cola. Máximo común divisor

MÁXIMO COMÚN DIVISOR (M.C.D)

El máximo común divisor (M.C.D.) de dos o más números es el mayor de los números que pueden dividirlos de manera exacta.

20 1 2 4 5 10 20

35 1 5 7 35

Recuerda las tablas de multiplicar
1x20=20
2x10=20
4x5= 20

Los divisores que tienen en común son 1 y 5
El mayor o el máximo es 5

MÁXIMO COMÚN DIVISOR (M.C.D)

El máximo común divisor (M.C.D.) de dos o más números es el mayor de los números que pueden dividirlos de manera exacta.

20	2
10	2
5	5
1	

Sus factores primos son:
(2)(2)(5)
$(2^2)(5)$

35	5
7	7
1	

(5)(7)

M.C.D.

Busca los factores que tengan en común el 20 y el 35 y multiplicalos

El Máximo Común Divisor entre dos enteros A y B se define formalmente así:

$$MCD(M,N) \quad N \begin{cases} MCD(N,M) & \text{si } M < N \\ & \text{si M es divisible por N: M mod N = 0} \\ MCD(N,\ M \bmod N) & \text{en cualquier otro caso} \end{cases}$$

```
int mcd(int a, int b)
{
 if (a<b)
      {return mcd(b,a);}
       else
            if(a%b==0)
              {return b;}
            else
              { return mcd(b,a%b); }
}
```

El **algoritmo de Euclides** es un método antiguo y eficiente para calcular el máximo común divisor (**MCD**). Fue originalmente descrito por Euclides en su obra *Elementos* (Siglo V AC)

Recursividad de cola. Factorial

```
int factorial(int x)
{
return factrecu (x, 1);
}




int  factrecu(int xx, int acu)
{
if (xx==0)
    {return acu;}
else
   {return factrecu(xx-1, acu*xx);}
}
```

Recursividad anidada. Función de Ackermann

$$\text{Ackerman}(m, n) = \begin{cases} n + 1 \text{ si } m = 0 \\ \text{Ackerman}(m - 1, 1) \quad \text{si } m > 0 \wedge n = 0 \\ \text{Ackerman}(m - 1, \text{Ackerman}(m, n - 1)) \quad \text{si } m > 0 \wedge n > 0 \end{cases}$$

Explicación intuitiva

La primera fila de la función de Ackerman contiene los enteros positivos, dado que $A(0, n)$) consiste en sumar uno a n. El resto de las filas se pueden ver como indirecciones hacia la primera. En el caso de $m = 1$, se redirige hacia $A(0, n + 1)$; sin embargo, la simplificación es algo complicada:

$$\begin{aligned} A(1,2) &= A(0, A(1,1)) \\ &= A(0, A(0, A(1,0))) \\ &= A(0, A(0,2)) \\ &= A(0,3) \\ &= 4 \end{aligned}$$

Números de (m,n)

*m**n*	0	1	2	3	4	*n*
0	1	2	3	4	5	*n* + 1
1	2	3	4	5	6	*n* + 2
2	3	5	7	9	11	2*n* + 3
3	5	13	29	61	125	$8 \cdot 2^n - 3$
4	13	65533	$2^{65536} - 3 \approx 2 \cdot 10^{19728}$	A(3,2^{65536}-3)	A(3,A(4,3))	$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} - 3$ ($n+3$ términos)
5	65533	A(4,65533)	A(4,A(5,1))	A(4,A(5,2))	A(4,A(5,3))	
6	A(5,1)	A(5,A(5,1))	A(5,A(6,1))	A(5,A(6,2))	A(5,A(6,3))	

La recursividad anidada es un tipo particular de recursividad en la que una función no solo se llama a sí misma, sino que lo hace varias veces en un mismo nivel de la recursión y además dentro de las propias llamadas recursivas. Es decir, la función se invoca con los resultados de otras llamadas recursivas anidadas dentro de sí misma.

Características de la Recursividad Anidada:

Las llamadas recursivas están "anidadas" dentro de otras llamadas recursivas.

Generalmente, se utiliza cuando se necesita resolver subproblemas dentro de otros subproblemas, y los resultados de esas subproblemas se combinan en cada nivel.

Ejemplo de Recursividad Anidada: El ejemplo más típico es la función de Ackermann, que es una función matemática famosa por su naturaleza altamente recursiva.

```java
public class Ackermann {
    public static int ackermann(int m, int n) {
        if (m == 0) {
            return n + 1;
        } else if (m > 0 && n == 0) {
            return ackermann(m - 1, 1);
        } else {
            return ackermann(m - 1, ackermann(m, n - 1));
        }
    }

    public static void main(String[] args) {
        System.out.println(ackermann(2, 3)); // Resultado: 9
    }
}
```

Recursividad mutua. Función par

```java
public class ParImpar {

    // Función recursiva para verificar si un número es par
    public static boolean esPar(int n) {
        if (n == 0) {
            return true; // 0 es par
        } else if (n == 1) {
            return false; // 1 es impar
        } else {
            return esPar(n - 2); // Reduce el número en 2 y verifica
        }
    }

    // Función recursiva para verificar si un número es impar
    public static boolean esImpar(int n) {
        return !esPar(n); // Un número es impar si no es par
    }

    public static void main(String[] args) {
        int numero = 7;

        if (esPar(numero)) {
            System.out.println(numero + " es par.");
        } else {
            System.out.println(numero + " es impar.");
        }
    }
}
```

Excepciones

Las excepciones son situaciones anómalas que requieren un tratamiento especial.

¡¡No tienen por qué ser errores!!

Si se consigue dominar su programación, la calidad de las aplicaciones que se desarrollen aumentará considerablemente.

Por norma general, los comandos try y catch conforman bloques de código.

Cada uno de estos bloques se recomienda, aunque sea de una única línea, envolverlos en llaves, como muestra el siguiente ejemplo:

```
// ...código previo...
try {
  // bloque de código a comprobar
} catch( tipo )  // Ha ocurrido un suceso en el try que se ha terminado
//la ejecución del bloque y catch recoge y analiza lo sucedido
{
  // bloque de código que analiza lo que ha lanzado el try
}
// ...código posterior...
```

Generalmente entre el try y el catch no se suele insertar código, pero se insta al lector a que lo intente con su compilador habitual y que compruebe si hay errores o no de compilación.

Una excepción es un error que puede ocurrir debido a una mala entrada por parte del usuario, un mal funcionamiento en el hardware, un argumento inválido para un cálculo matemático, etc. Para remediar esto, el programador debe estar atento y escribir los algoritmos necesarios para evitar a toda costa que un error de excepción pueda hacer que el programa se interrumpa de manera inesperada. Java soporta una forma más directa y fácil de ver tanto para el programador como para los revisores del código en el manejo de excepciones que su símil en el C++ estándar y esta consiste, tratándose del lenguaje Java, en el mecanismo try, throw y catch.

La lógica del mecanismo mencionado consiste en:
Dentro de un bloque try se pretende evaluar una o más expresiones y si dentro de dicho bloque se produce un algo que no se espera se lanza por medio de throw una excepción, la misma que deberá ser capturada por un catch específico.
Puesto que desde un bloque try pueden ser lanzados diferentes tipos de errores de excepción es que puede haber más de un catch para capturar a cada uno de los mismos.
Si desde un try se lanza una excepción y no existe el mecanismo catch para tratar dicha excepción el programa se interrumpirá abruptamente después de haber pasado por todos los catchs que se hayan definido y de no haber encontrado el adecuado.
Los tipos de excepciones lazados pueden ser de un tipo primitivo tal como: int, float, char, etc. aunque normalmente las excepciones son lanzadas por alguna clase escrita por el usuario o por una clase de las que vienen incluidas con el compilador.
En el programa que se listará a continuación muestra un ejemplo de como lanzar una excepción de tipo int dentro del bloque try, y cómo capturar la excepción por medio de catch.

Ejemplo básico de excepción

```
3  public class TesteoDeExcepcion {
4
5      public static void main(String[] args) {
6          int i = 0;
7          try {
8              System.out.println("La division es: " + 10 / i);
9          } catch (Exception e) {
10             System.out.println("Hay una excepcion " + e.getMessage());
11         }
12     }
13 }
14
```

Excepciones genéricas [editar]

Como ya se ha mencionado, los errores pueden deberse a una multitud de situaciones muchas veces inesperadas, por tal motivo, en C++ existe una forma de manejar excepciones desconocidas (genéricas) y es buena idea que si se está escribiendo un controlador de excepciones incluya un **catch** para capturar excepciones inesperadas. Por ejemplo, en el siguiente progama se escribe un **catch** que tratará de capturar cualquier excepción inesperada.

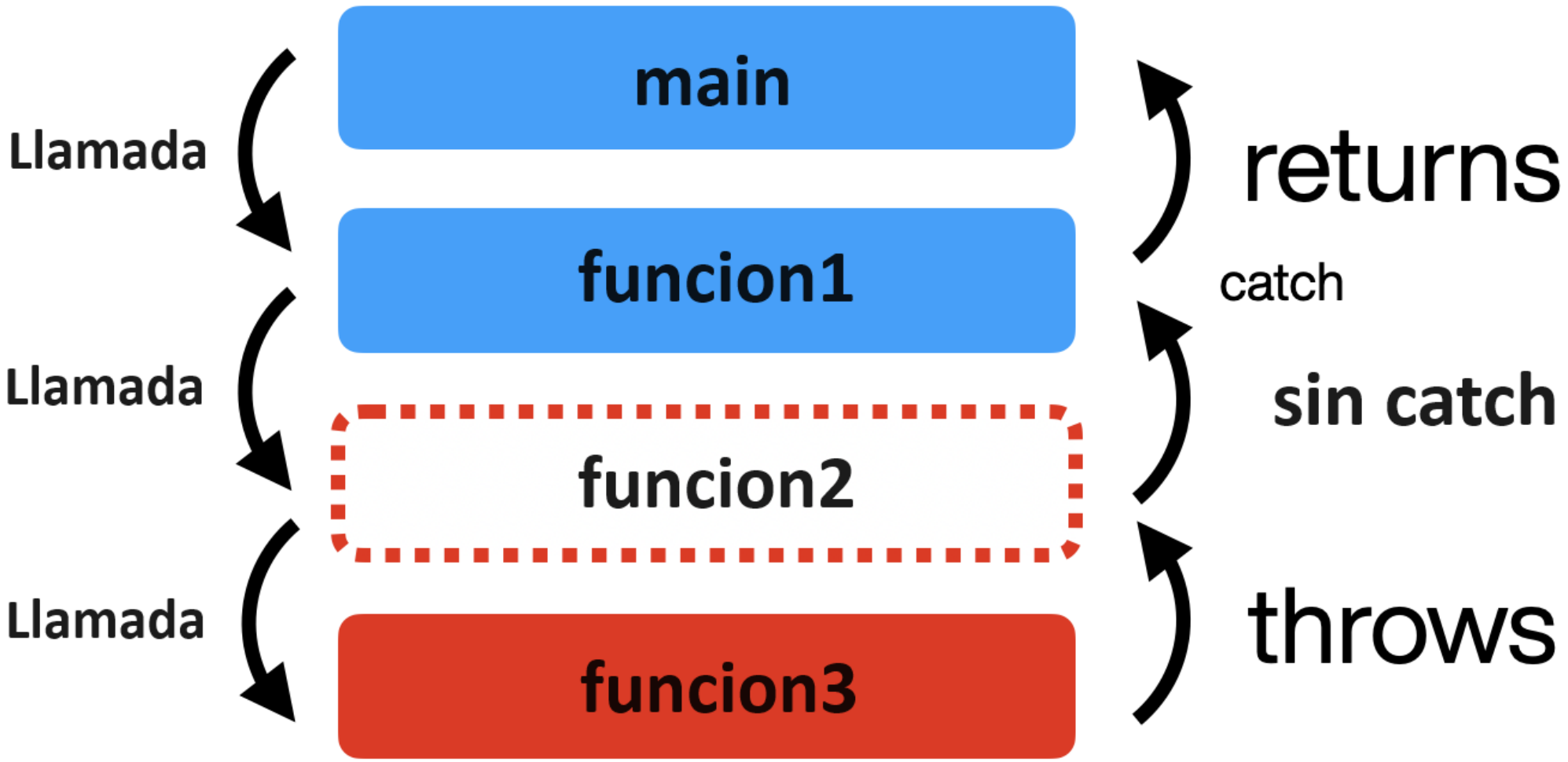
```
// Demostración: try, throw y catch
#include <iostream>

using namespace std;

int main()
{
    try {
        throw 125;
        }
    catch(...) {
        cout << "Ha ocurrido un error inesperado..." << endl;
        }
    cin.get();
    return 0;
}
```

Excepciones y el stack

```
3  public class TesteoDeExcepcion {
4
5      public static double funcion3(int i) {
6          return 10 / i;
7      }
8
9      public static double funcion2(int i) {
10         return funcion3(i);
11     }
12
13     public static double funcion1(int i) {
14         return funcion2(i);
15     }
16
17     public static void main(String[] args) {
18         int i = 0;
19         try {
20             System.out.println("La division es: " + funcion1(i));
21         } catch (Exception e) {
22             System.out.println("Hay una excepcion " + e.getMessage());
23         }
24     }
25 }
26
```


main
funcion1
funcion2
funcion3
Llamada
Llamada
Llamada
returns
catch
sin catch
throws

Excepciones anidadas

```
5      public static double funcion3(int i) {
6          return 10 / i;
7      }
8
9      public static double funcion2(int i) {
10         return funcion3(i);
11     }
12
13     public static double funcion1(int i) {
14         return funcion2(i);
15     }
16
17     public static void main(String[] args) {
18         int i = 0;
19         try {
20             try {
21                 System.out.println("La division es: " + funcion1(i));
22             } catch (Exception e) {
23                 throw new Exception(e);
24             }
25             try {
26                 System.out.println("La division es: " + funcion1(10));
27             } catch (Exception e) {
28                 throw new Exception(e);
29             }
30         } catch (Exception e) {
31             System.out.println("Hay una excepcion " + e.getMessage());
32         }
33     }
34 }
```

Excepciones por tipo

```
9    public static double funcion2(int i) {
10       return funcion3(i);
11   }
12
13   public static double funcion1(int i) {
14       return funcion2(i);
15   }
16
17   public static void main(String[] args) {
18       int i = 0;
19       try {
20           try {
21               System.out.println("La division es: " + funcion1(i));
22           } catch (Exception e) {
23               throw new RuntimeException(e);
24           }
25           try {
26               System.out.println("La division es: " + funcion1(10));
27           } catch (Exception e) {
28               throw new Exception(e);
29           }
30       } catch (RuntimeException e) {
31           System.out.println("Hay una excepcion " + e.getMessage());
32       } catch (Exception e) {
33           System.out.println("Hay una excepcion " + e.getMessage());
34       }
35   }
```

Cambiar tipo de excepción

```
try {
    System.out.println("La division es: " + funcion1(i));
} catch (Exception e) {
    throw new RuntimeException(e);
}
```

Excepciones de TDA

Se pueden hace un TDA de una excepción y luego lanzarlo con throw, pero para eso es necesario aprender TDA, así que retomaremos luego este tema.

```java
public class EdadInvalidaException extends Exception {
    public EdadInvalidaException(String mensaje) {
        super(mensaje);
    }
}

public class ValidadorDeEdad {

    // Método que valida la edad
    public void validarEdad(int edad) throws EdadInvalidaException {
        if (edad < 18) {
            throw new EdadInvalidaException("La edad debe ser mayor o igual a 18.");
        }
    }
}

public class TestExcepcion {
    public static void main(String[] args) {
        ValidadorDeEdad validador = new ValidadorDeEdad();

        try {
            validador.validarEdad(16);  // Esto va a lanzar la excepción
        } catch (EdadInvalidaException e) {
            System.out.println("Excepción capturada: " + e.getMessage());
        }
    }
}
```

Testeos

JUnit es una herramienta esencial para hacer pruebas unitarias en Java. Las pruebas unitarias se utilizan para verificar que las distintas partes del código funcionen correctamente de manera aislada. Este marco de trabajo es sencillo de aprender y extremadamente útil para garantizar la calidad y la fiabilidad del código.

¿Qué es una prueba unitaria?

Una **prueba unitaria** es un pequeño fragmento de código que verifica el comportamiento de una función o método. El objetivo es validar que las salidas (outputs) sean correctas para un conjunto determinado de entradas (inputs).

¿Por qué usar JUnit?

1. **Facilita la identificación de errores**: Permite encontrar errores en el código antes de que este crezca demasiado o sea desplegado.
2. **Automatización**: Las pruebas pueden ejecutarse de manera automática con cada cambio en el código.
3. **Aumenta la confianza**: Un código bien probado genera confianza tanto en el desarrollador como en los usuarios.
4. **Fomenta el diseño modular**: Al dividir el código en partes pequeñas y probables, se mejora la calidad del diseño.
5. **Facilita la refactorización**: Permite modificar el código con mayor seguridad, sabiendo que las pruebas te avisarán si algo se rompe.

Primeros pasos con Junit

Agregar la dependencia de JUnit: añade la dependencia de JUnit 5 al proyecto y al path

Anatomía básica de una prueba: Aquí un ejemplo básico de cómo sería una prueba usando JUnit.

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class CalculadoraTest {

    @Test
    public void testSuma() {
        Calculadora calculadora = new Calculadora();
        int resultado = calculadora.sumar(2, 3);
        assertEquals(5, resultado, "La suma de 2 + 3 debería ser 5");
    }
}
```

Componentes clave de Junit

Anotaciones:

@Test: Marca un método como una prueba.

@BeforeEach: Método que se ejecuta antes de cada prueba, ideal para inicialización.

@AfterEach: Método que se ejecuta después de cada prueba.

@BeforeAll y @AfterAll: Ejecutados una vez antes/después de todas las pruebas.

Métodos de aserción: Las aserciones son clave para verificar si el código cumple con los resultados esperados.

assertEquals(expected, actual): Compara si dos valores son iguales.

assertTrue(condition): Verifica que una condición sea verdadera.

assertThrows(): Verifica que se lance una excepción esperada.

Beneficios de Junit

Retroalimentación rápida: Permite conocer rápidamente si el código funciona como se espera.

Documentación implícita: Las pruebas sirven como documentación, mostrando cómo debería comportarse el código.

Integración continua: Funciona bien con herramientas de integración continua como Jenkins o Travis, para que las pruebas se ejecuten automáticamente con cada cambio.

Mejores prácticas al usar Junit

Escribir pruebas pequeñas e independientes: Cada prueba debe enfocarse en una funcionalidad específica, lo que facilita la identificación de errores.

Nombrar claramente las pruebas: Los nombres deben describir lo que hacen las pruebas. Por ejemplo, deberiaSumarDosNumeros() es más descriptivo que testSuma().

Usar assertThrows() para probar excepciones: Asegúrate de probar los casos de error, no solo los casos exitosos.

Mantener el código de prueba separado del código de producción: Los archivos de prueba deben estar en un paquete separado, por ejemplo src/test/java.

Hacer pruebas continuamente: No esperes a que el código esté completo para empezar a probar. Hazlo continuamente conforme avanzas en el desarrollo.

La estrategia “Divide y vencerás”

Dado un problema de tamaño N ,

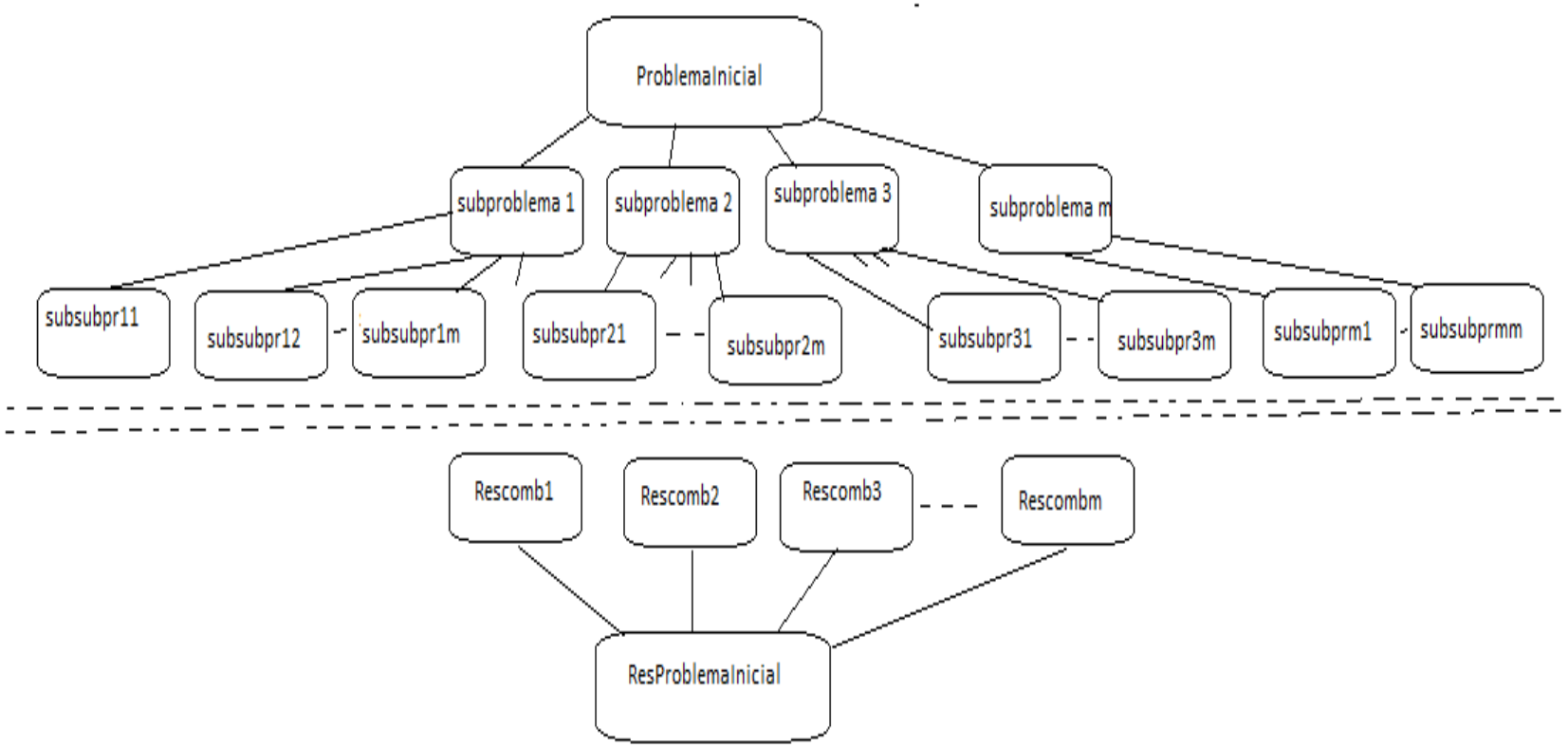
Si N > L , siendo L un valor arbitrario, dividir el problema en m subproblemas independientes (es decir, no solapados), de la misma estructura que el problema original (es decir que estaremos resolviendo, en cada uno de esos m subproblemas, lo mismo que en el problema original). Si los subproblemas tienen tamaño similar entre si, es más conveniente.

Para cada uno de los m subproblemas,

Si el subproblema tiene un tamaño mayor que L, proceder del mismo modo que con el problema original, dividiéndolo nuevamente en m subproblemas independientes.

Si, en cambio, el tamaño del problema no supera el valor de L, resolverlo de otra forma, utilizando generalmente un camino más directo.

La solución del problema original se obtiene por combinación de las m soluciones obtenidas de los subproblemas en que se lo había dividido.


ProblemaInicial
subproblema 1
subproblema 2
subproblema 3
subproblema m
subsubpr11
subsubpr12
subsubpr1m
subsubpr21
subsubpr2m
subsubpr31
subsubpr3m
subsubprm1
subsubprmm
Rescomb1
Rescomb2
Rescomb3
Rescombm
ResProblemaInicial

Tamaño del problema

Cuando resolvemos un problema computacional generalmente podemos indicar fácilmente qué elemento o elementos ingresados tienen responsabilidad directa en el consumo de recursos del algoritmo, en particular del recurso espacio (de memoria) y el tiempo (de ejecución). Al variar, varía el consumo de tiempo, o de espacio. Esos elementos constituyen el tamaño de la entrada del problema (o tamaño del problema)

El tamaño de la entrada de un problema se puede definir de forma más precisa como el número de símbolos necesarios (de un cierto alfabeto, finito, obviamente) para poder codificar todos los datos de un problema.

Ejemplos

¿De qué depende el tiempo consumido para ordenar un vector?

¿De qué depende el tiempo consumido para obtener un valor de Fibonacci?

¿De qué depende el tiempo consumido para multiplicar dos matrices?

Ejemplos de algoritmos que aplican Divide y Venceras

la búsqueda binaria (o dicotómica) en un array ordenado

los ordenamientos internos Mergesort,

Quicksort,

Radixsort,

la resolución de las Torres de Hanoi,

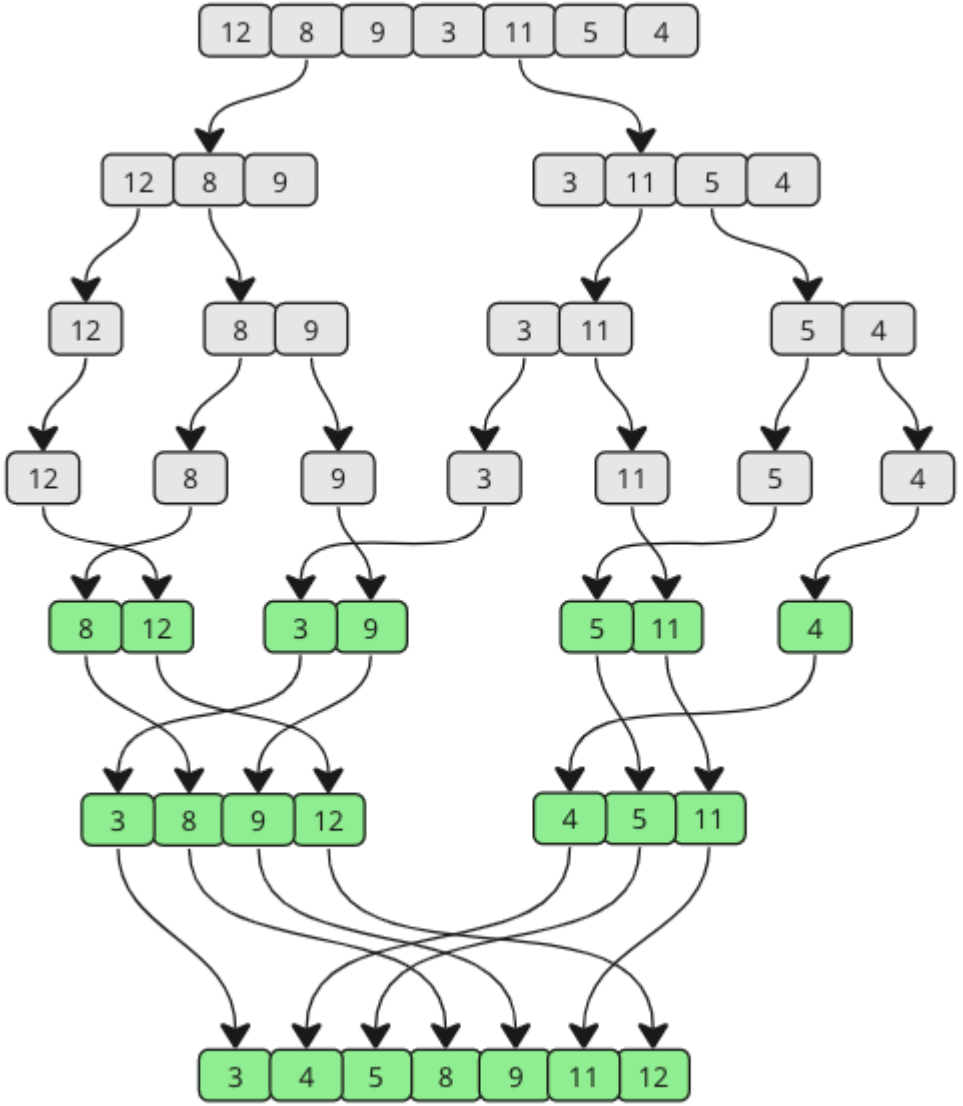
la multiplicación de enteros grandes de Karatsuba y Ofman,

la multiplicación de matrices de Strassen,

el cálculo de la Transformada rápida de Fourier,

la determinación del par de puntos más cercanos en un plano.

Mergesort


12 8 9 3 11 5 4
12 8 9
3 11 5 4
12
8 9
3 11
5 4
12
8
9
3
11
5
4
8 12
3 9
5 11
4
3 8 9 12
4 5 11
3 4 5 8 9 11 12

Para ordenar un array de N elementos (tamaño N) proceder así: si N es mayor que 1, dividir el array a la mitad,ordenar cada mitad e intercalar ordenadamente las mitades ordenadas. Si no, el array ya está ordenado

```
void mergesort (double v[], int pr, int ul ) {
 if (pr < ul) {
     int me= (pr+ul)/2;
     mergesort (v, pr, me);
     mergesort(v, me+1, ul);
     intercalar (v, pr, me, ul);      // este proceso realiza la intercalación ordenada de las mitades ordenadas
   }
}
```

N (dado por ul – pr +1) es el tamaño del problema

El valor arbitrario con el cual se compara ese tamaño es 1

Análisis complejidad temporal:

$T(n) = 2 * T(n/2) + n$

$T(1) = 1$

Por aplicación del Teorema Maestro de Reducción por División resulta T(n) pertenece a O (n * log n)

Determinación del umbral más adecuado

¿cual será el umbral (valor de L) más adecuado?

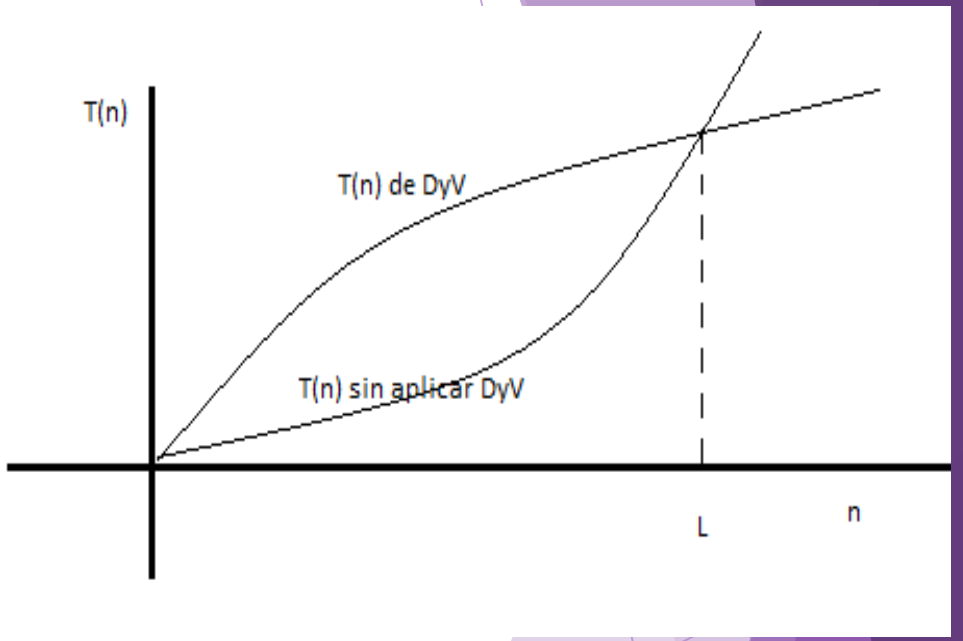
¿Cual será el que nos permita obtener los resultados con la mayor rapidez?

La determinación del mejor L no es un problema simple.

Hay dos formas básicas:

Empíricamente: construyendo tablas para ambos algoritmos (aplicando DyV y sin aplicarlo) para valores de n crecientes y estableciendo con ellas el tamaño de la entrada para el cual se igualan los valores en ambas tablas.

Teóricamente: si tenemos las expresiones formales correspondientes a T(n) para cada algoritmo, matemáticamente se calcula el n para el cual la solución aplicando D y V y la solución que no aplica esa estrategia tienen igual coste.



Quicksort

```
//versión recursiva del ordenamiento rápido.
void quicksort (double v[], int pr, int ul) {
  if ( ul >pr) {
      int pospiv = Ubicarpivote (v, pr, ul);
      quicksort (v, pr, pospiv-1);
      quicksort(v, pospiv+1, ul );
    }
}
int Ubicarpivote(int v[] int pr, int ul) {
    int ppiv= pr;
    int k= pr;
    int j= k+1;
    while (j<=ul) {
      if (a[j]<a[ppiv] ) {
        k= k+1;
        intercambiar(a, k, j);
      }
      j= j+1;
    }
    intercambiar(a, k, ppiv);
    return k;
  }
```

Análisis de complejidad temporal:

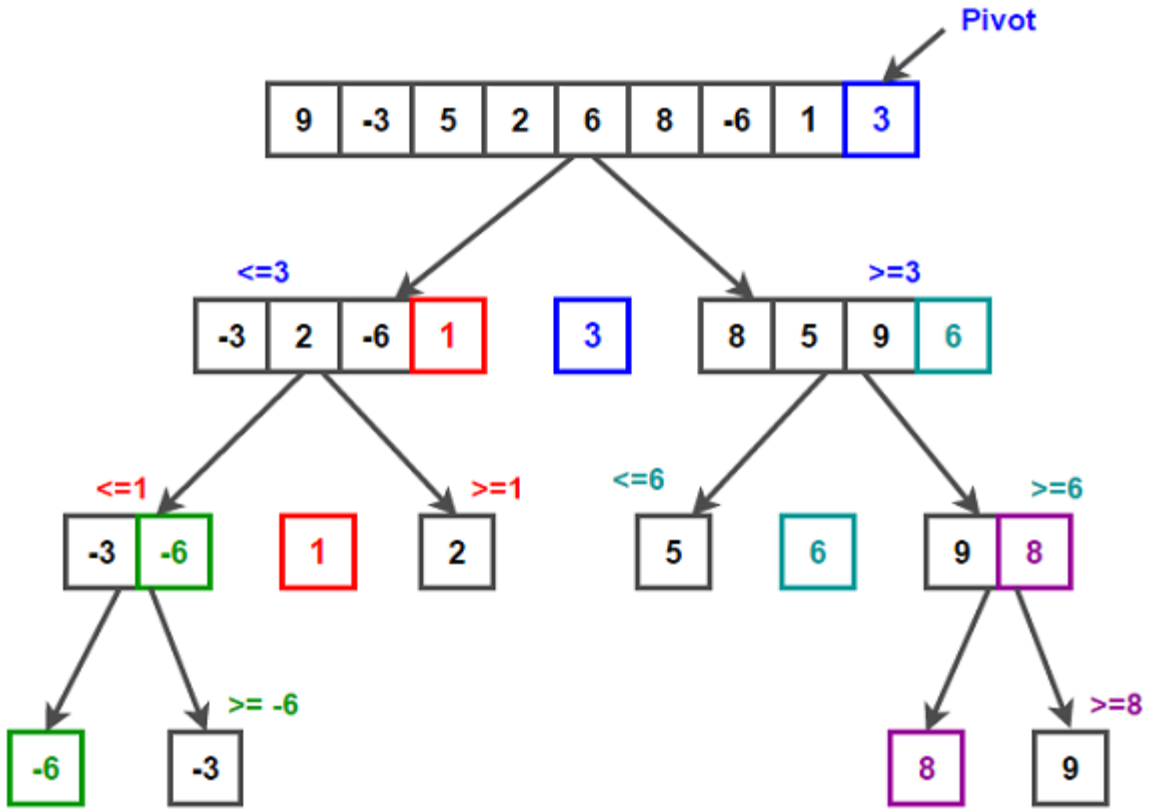
Al ubicar el pivote, los subvectores que quedan al lado izquierdo y derecho del pivote pueden tener tamaños parecidos o muy disímiles => hay dos ‘situaciones límite’:

•que cada vez que ubiquemos el pivote el subvector lado izquierdo del pivote tenga el mismo tamaño (o casi)
que el subvector del lado derecho del pivote

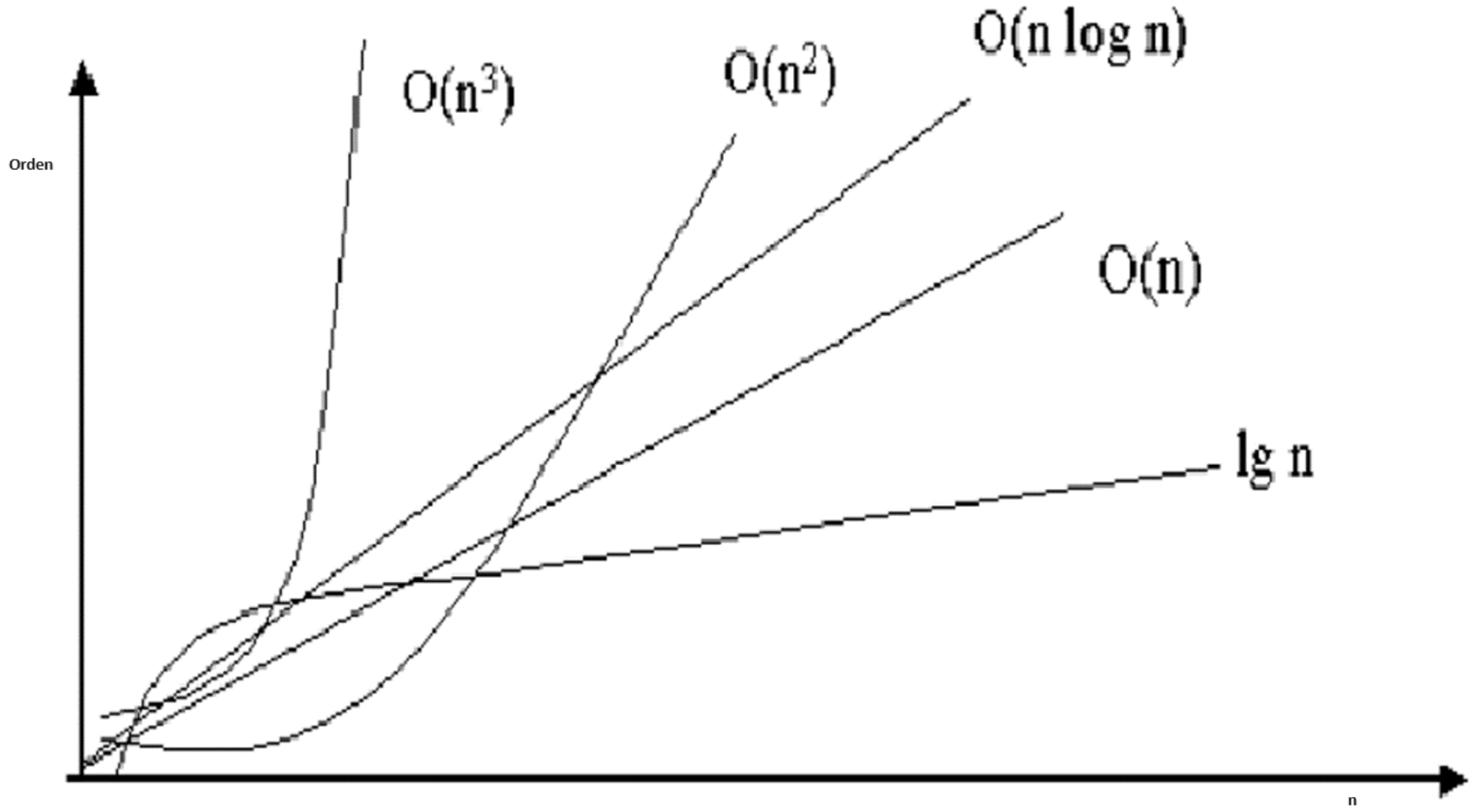
•que cada vez que ubiquemos el pivote, uno de los dos subvectores quede sin elementos y el otro tienga n-1 elementos

Si en cada etapa se da la situación 1, entonces las ecuaciones que resultan son:
$T(n) = 2 * T(n/2) + n$
$T(1) = 1$
$\Rightarrow T(n)$ pertenece a $O(n * \log n)$.

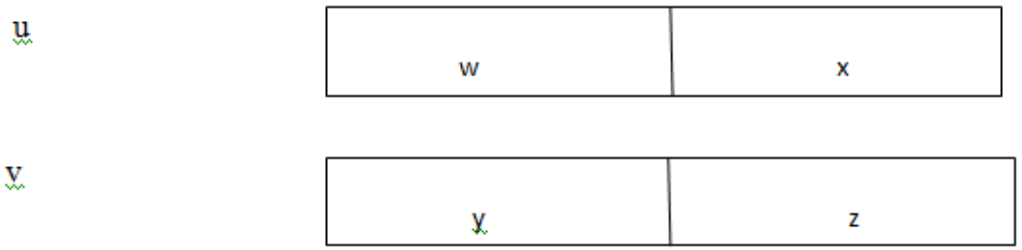
Si, en cambio, en cada etapa se da la situación 2, entonces las ecuaciones que resultan son:
$T(n) = T(n - 1) + n$
$T(0) = 1$
$\Rightarrow T(n)$ pertenece a $O(n^2)$


Pivot
9 -3 5 2 6 8 -6 1 3
<=3
-3 2 -6 1
3
>=3
8 5 9 6
<=1
-3 -6
1
2
>=1
<=6
5
6
9 8
>=6
>= -6
-6
-3
8
9
>=8

Comparación de ordenes computacionales



Multiplicación de Enteros Grandes (A. Karatsuba e Y. Ofman)



En símbolos,

$u = 10^s w + x$
$v = 10^s y + z$
con $0 <= x < 10^s$ $0 <= z < 10^s$ $s = piso(n/2)$ (s es **piso** de n/2)

Resulta:
$u * v = 10^{2s} w y + 10^s (wz+xy) + xz$

Se puede plantear

```
Enterogrande mul (enterogrande u, enterogrande v)
{
  int  n = max(u.tamanyo(), v.tamanyo();   //como medimos los tamaños??
  if (n >L)        // en que valor convendrá fijar L??
    {
       int  s= n/2 ;
       Enterogrande w= u/ 10^s ;
       Enterogrande x=u % 10^s ;
       Enterogrande y= v/10^s ;
       Enterogrande z=v%10^s ;
       return mul (w,y ) * 10^2s + (mul (w,z) + mul (x,y)) * 10^s + mul (x, z);
    }
 else return u * v;
}
```

La multiplicación de dos enteros de tamaño n tiene un coste $O(n^2)$. Y la suma $O(n)$

Entonces, si contamos cuántas operaciones y de que tipo se realizan, y cuántas invocaciones recursivas hay en mul, se tiene:

$T(n) = 4 * T(n/2) + O(n)$ y $T(1)=1$

Enterogrande mul tiene un coste $O(n^2)$.

No hemos ganado nada con sólo aplicar la estrategia DyV

Karatsuba propuso calcular wy, wz+xy, y xz con menos de cuatro multiplicaciones:

Dado que r = (w + x)(y + z) = wy + (wz + xy) + xz

Se puede desarrollar esta función:

```
Enterogrande mul2 (enterogrande u, enterogrande v)
{
  int  n = max(u.tamanyo(), v.tamanyo();   //como medimos los tamaños??
  if (n >L)        // en que valor convendrá fijar L??
    {
       int  s= n/2 ;
       Enterogrande w= u/ 10^s ;
       Enterogrande x=u % 10^s ;
       Enterogrande y= v/10^s ;
       Enterogrande z=v%10^s ;
       Enterogrande r= mul2 (w+x, y +z);
       Enterogrande p= mul2 (w , y);
       Enterogrande q = mul2 (x, z);
       return p * 10^2s  + (r-p-q) * 10^s + q;
    }
 else return u * v;
}
```

$T(n) = 3\,T(n/2) + n \quad T(1) = 1$

Y resulta entonces que

$T(n)$ pertenece a $O(n^{\log_2 3})$

lo cual muestra la conveniencia del algoritmo para n grande.

Algoritmo de Strassen

para la multiplicación de matrices

El objetivo de este algoritmo es obtener el producto de matrices cuadradas, de n x n.

La suma de matrices de n x n pertenece a $O(n^2)$,

El producto de matrices de n x n pertenece a $O(n^3)$.

A ›

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} * \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Siendo , al aplicar D y V

$C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$
$C_{12} = A_{11} * B_{12} + A_{12} * B_{22}$
$C_{21} = A_{21} * B_{11} + A_{22} * B_{21}$
$C_{22} = A_{21} * B_{12} + A_{22} * B_{22}$

y así se continúa dividiendo las submatrices mientras n sea mayor que 1. Son 8 multiplicaciones y 4 sumas .
Las ecuaciones asociadas son
$T(n) = 8\ T(n/2) + n^2$
$T(1) = 1$
Lo cual nos lleva a que T (n) pertenece a $O(n^3)$.

Strassen propuso estos cálculos:

m1 = (a21 + a22 - a11)(b22 - b12 + b11)

m2 = a11 . b11

m3 = a12 . b21

m4 = (a11 - a21)(b22 - b12)

m5 = (a21 + a22)(b12 - b11)

m6 = (a12 - a21 + a11 - a22)b22

m7 = a22(b11 + b22 - b12 - b21)

Resultando así,

c11=m1+m4-m5+m7;

c12=m3+m5;

c21=m2+m4;

c22=m1-m2+m3+m6;

Ahora son 24 sumas y restas, pero sólo 7 multiplicaciones (antes eran 8). Reescribiendo el algoritmo, es

$T(n) = 7\ T\ (n/2) + n^2 \quad T(1) = 1$

Y resulta entonces que T(n) pertenece a O ($n^{\log_2 7}$),

Algoritmos de ordenamiento no comparativos

Los algoritmos de ordenamiento no comparativos son aquellos que no se basan en la comparación directa de los elementos a ordenar. Aquí hay algunos ejemplos:

1. **Radix Sort**: Clasifica los elementos procesándolos por sus dígitos individuales. Puede ser implementado utilizando counting sort como un subproceso.
2. **Counting Sort**: Este algoritmo cuenta el número de elementos que tienen valores distintos y utiliza esa información para colocar cada elemento en su posición correcta en el arreglo de salida.
3. **Bucket Sort**: Distribuye los elementos en "cubetas" (buckets) basándose en su valor y luego aplica un algoritmo de ordenación (puede ser otro bucket sort recursivamente o un algoritmo de ordenación más simple) dentro de cada cubeta.

Estos algoritmos son eficientes en ciertos contextos, pero tienen sus limitaciones. Por ejemplo, Counting Sort y Radix Sort funcionan mejor para datos que tienen un rango limitado de valores, mientras que Bucket Sort puede ser menos eficiente si las cubetas no están distribuidas uniformemente.

Ordenamiento Radix

La mayor parte de los ordenadores digitales representan internamente todos sus datos como representaciones electrónicas de números binarios, por lo que procesar los dígitos de las representaciones de enteros por representaciones de grupos de dígitos binarios es lo más conveniente. Existen dos clasificaciones de radix sort: el de dígito menos significativo (LSD) y el de dígito más significativo (MSD). Radix sort LSD procesa las representaciones de enteros empezando por el dígito menos significativo y moviéndose hacia el dígito más significativo. Radix sort MSD trabaja en sentido contrario.

Vector original:

```
25 57 48 37 12 92 86 33
```

Asignamos los elementos en colas basadas en el dígito menos significativo de cada uno de ellos.

0:
1:
2:12 92
3:33
4:
5:25
6:86
7:57 37
8:48
9:

Después de la primera pasada, la ordenación queda:

```
12 92 33 25 86 57 37 48
```

Colas basadas en el dígito más significativo.

0:

1:12

2:25

3:33 37

4:48

5:57

6:

7:

8:86

9:92

Lista ordenada:

```
12 25 33 37 48 57 86 92
```

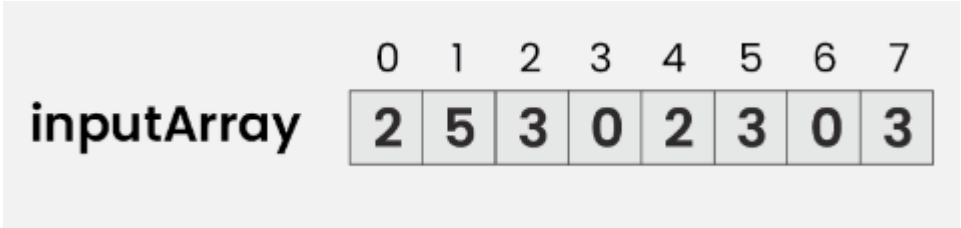
Ordenamiento Counting sort

Paso 1 :

- Descubra el elemento **máximo** de la matriz dada.

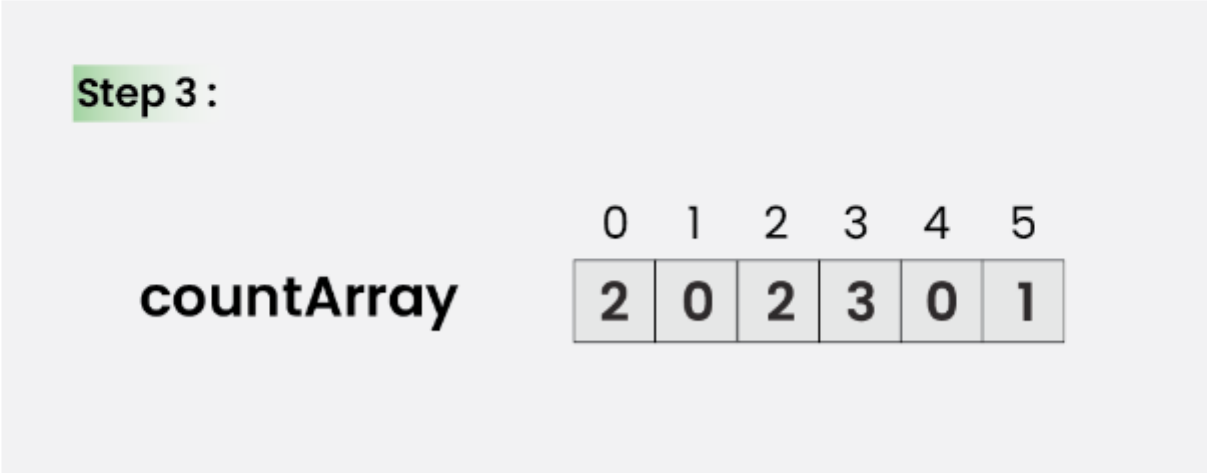
Paso 2:

- Inicialice un **countArray[]** de longitud **max+1** con todos los elementos como **0** . Esta matriz se utilizará para almacenar las apariciones de los elementos de la matriz de entrada.



Paso 3:

- En **countArray[]** , almacene el recuento de cada elemento único de la matriz de entrada en sus respectivos índices.
- **Por ejemplo:** el recuento del elemento **2** en la matriz de entrada es **2.** Entonces, almacene **2** en el índice **2** en **countArray[]** . De manera similar, el recuento del elemento **5** en la matriz de entrada es **1** , por lo tanto, almacene **1** en el índice **5** en **countArray[]** .



countArray

0	1	2	3	4	5
2	0	2	3	0	1

Etapa 4:

- Almacene la **suma acumulativa** o **suma de prefijo** de los elementos de **countArray[]** haciendo **countArray[i] = countArray[i – 1] + countArray[i].** Esto ayudará a colocar los elementos de la matriz de entrada en el índice correcto en la matriz de salida.

Step 4 :

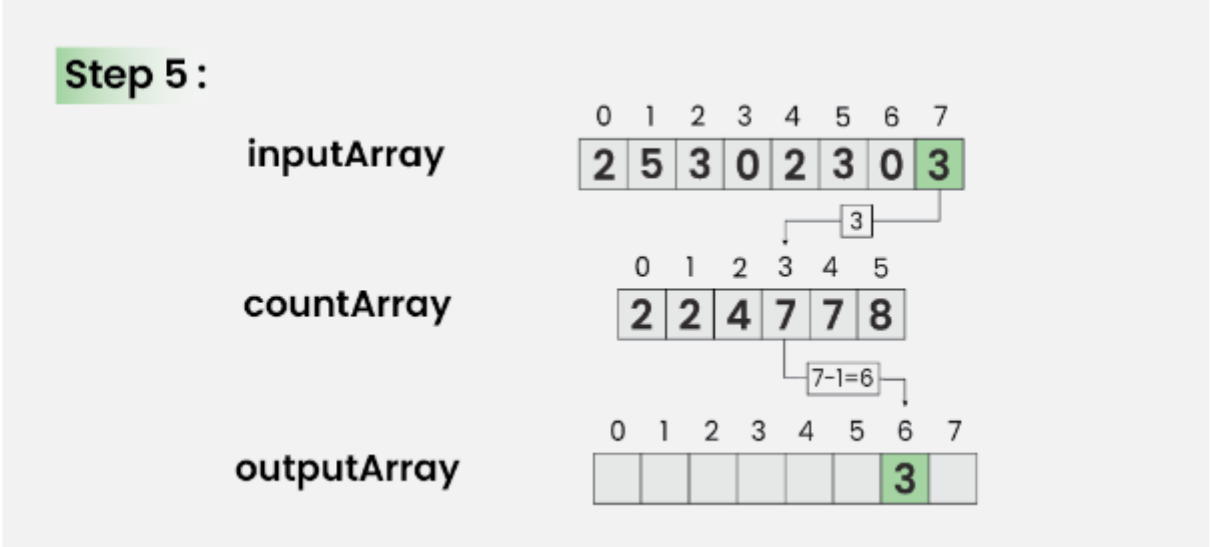
countArray

0	1	2	3	4	5
2	2	4	7	7	8

Paso 5:

- Iterar desde el final de la matriz de entrada y porque atravesar la matriz de entrada desde el final preserva el orden de los elementos iguales, lo que eventualmente hace que este algoritmo de clasificación sea **estable** .

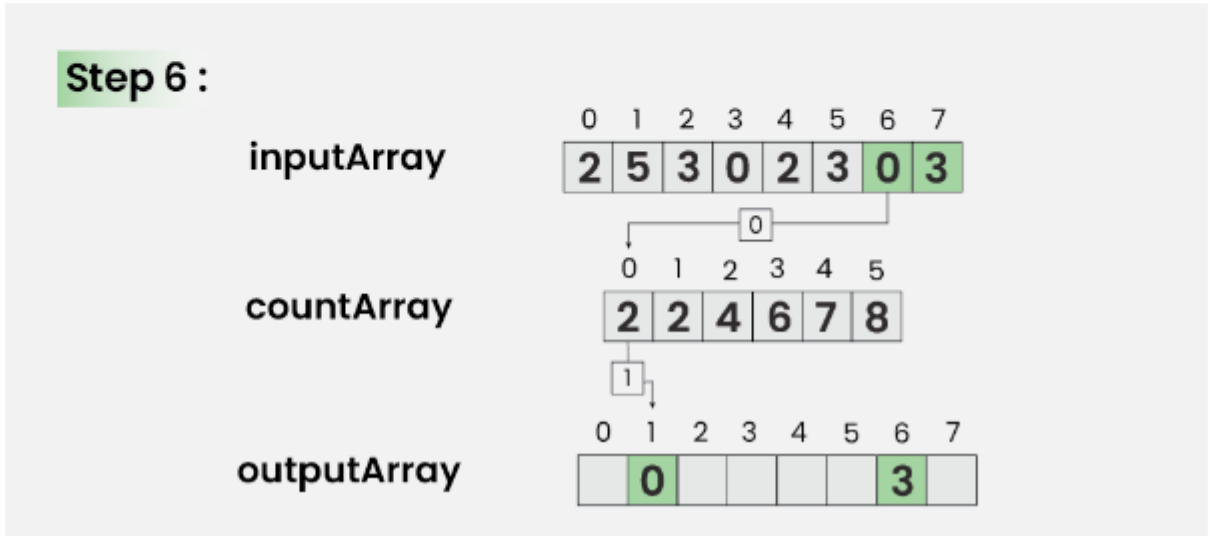
- *Actualice **OutputArray[countArray[inputArray[i]] – 1] = inputArray[i]** .*
- *Además, actualice **countArray[inputArray[i]] = countArray[inputArray[i]] – -.***



Paso 6: Para i = 6 ,

Actualice ***OutputArray[countArray[inputArray[6]] – 1] = inputArray[6]***

Además, actualice ***countArray[inputArray[6]] = countArray[inputArray[6]]- –***



Paso 12: Para i = 0,

Actualice ***OutputArray[countArray[inputArray[0]] – 1] = inputArray[0]***

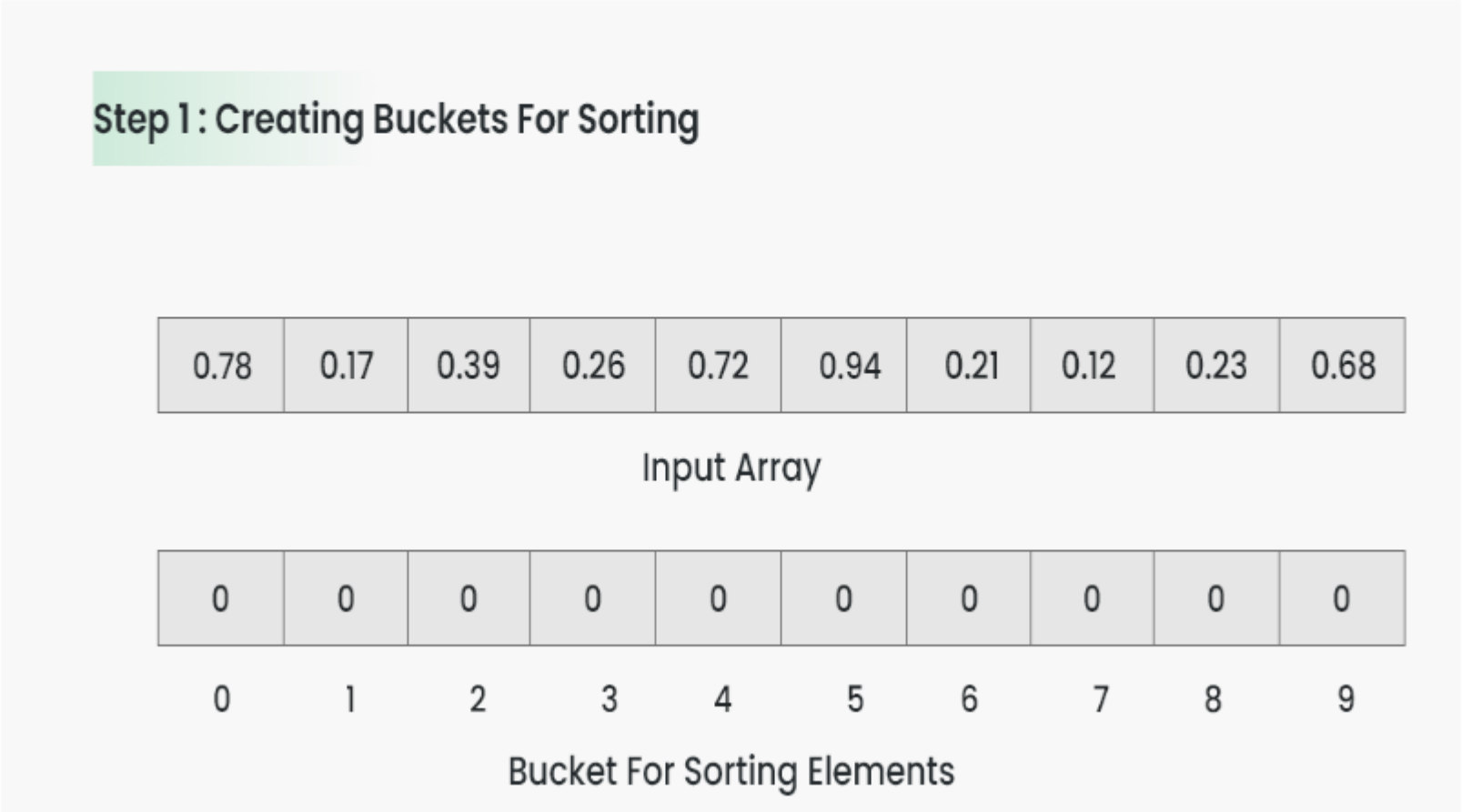
Además, actualice ***countArray[inputArray[0]] = countArray[inputArray[0]]- –***

Ordenamiento Bucket sort

¿Cómo funciona la clasificación de cubos?

Para aplicar la clasificación por depósitos en la matriz de entrada **[0,78, 0,17, 0,39, 0,26, 0,72, 0,94, 0,21, 0,12, 0,23, 0,68]** , seguimos estos pasos:

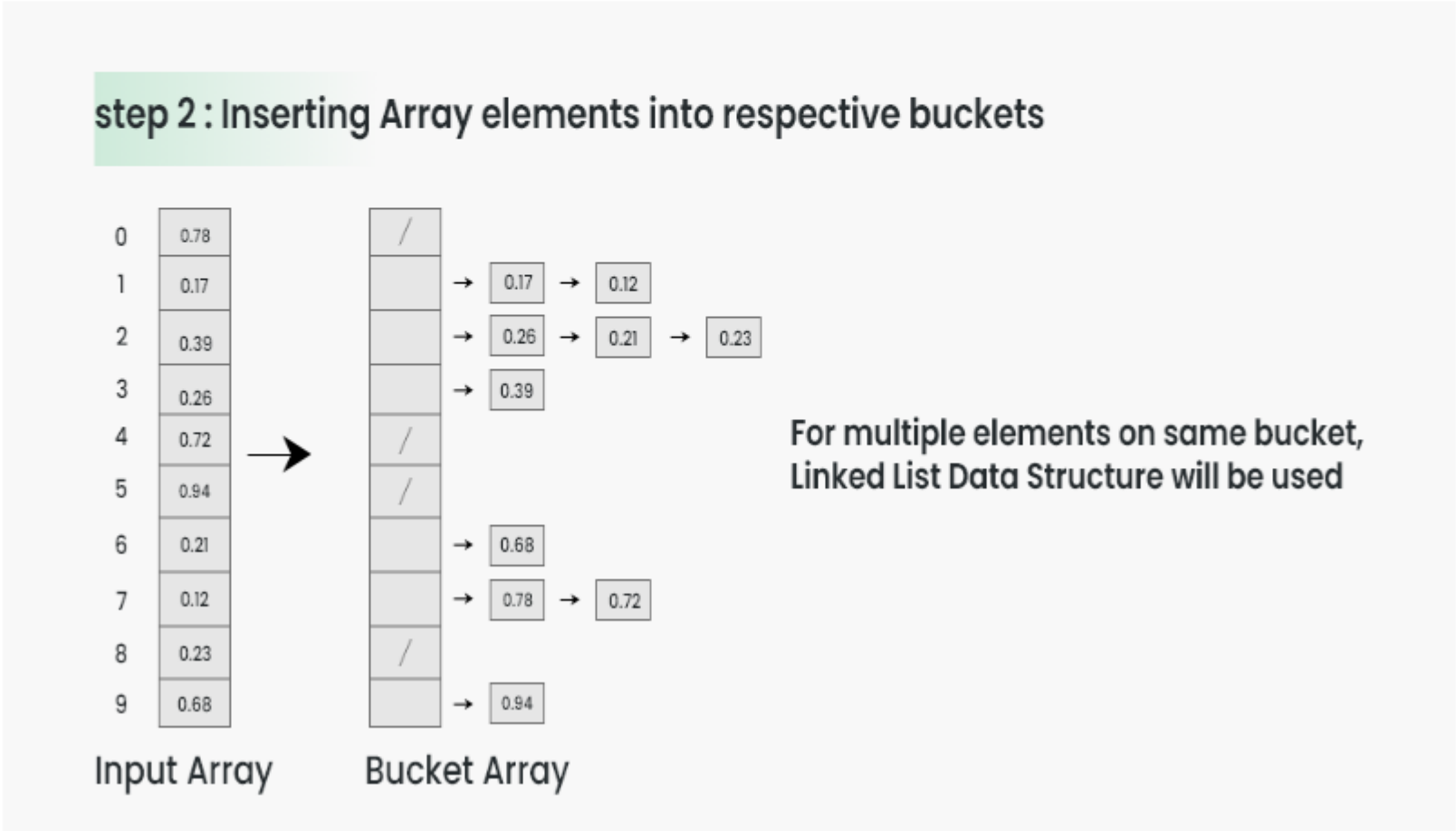
Paso 1: cree una matriz de tamaño 10, donde cada ranura represente un depósito.



Paso 2: inserte elementos en los depósitos de la matriz de entrada según su rango.

Insertar elementos en los cubos:

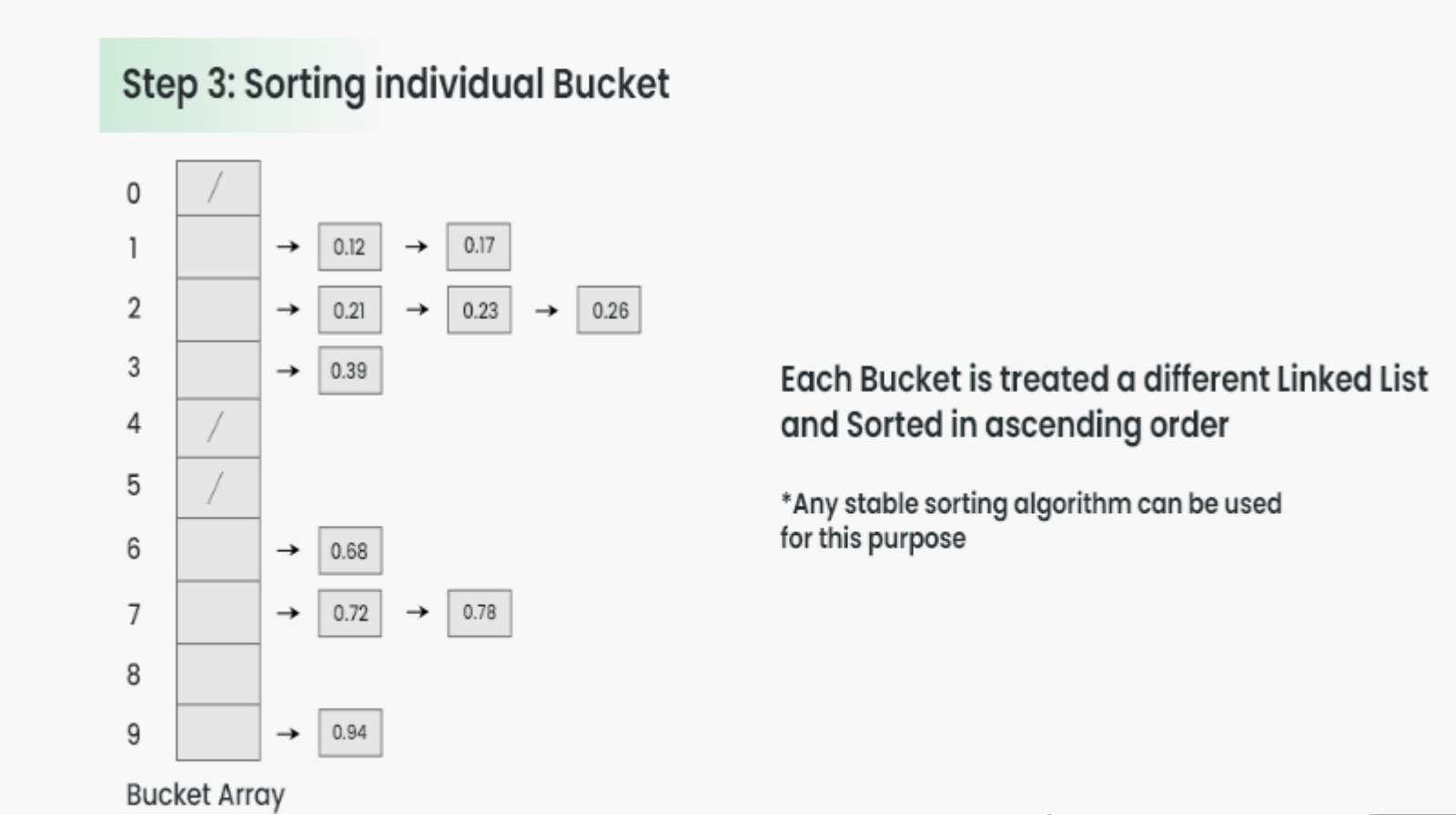
- Tome cada elemento de la matriz de entrada.
- Multiplique el elemento por el tamaño de la matriz del depósito (10 en este caso). Por ejemplo, para el elemento 0,23, obtenemos 0,23 * 10 = 2,3.
- Convierta el resultado a un número entero, lo que nos da el índice del depósito. En este caso, 2,3 se convierte al número entero 2.
- Inserte el elemento en el depósito correspondiente al índice calculado.
- Repita estos pasos para todos los elementos de la matriz de entrada.



Paso 3: Ordena los elementos dentro de cada cubo. En este ejemplo, utilizamos la clasificación rápida (o cualquier algoritmo de clasificación estable) para ordenar los elementos dentro de cada depósito.

Ordenar los elementos dentro de cada depósito:

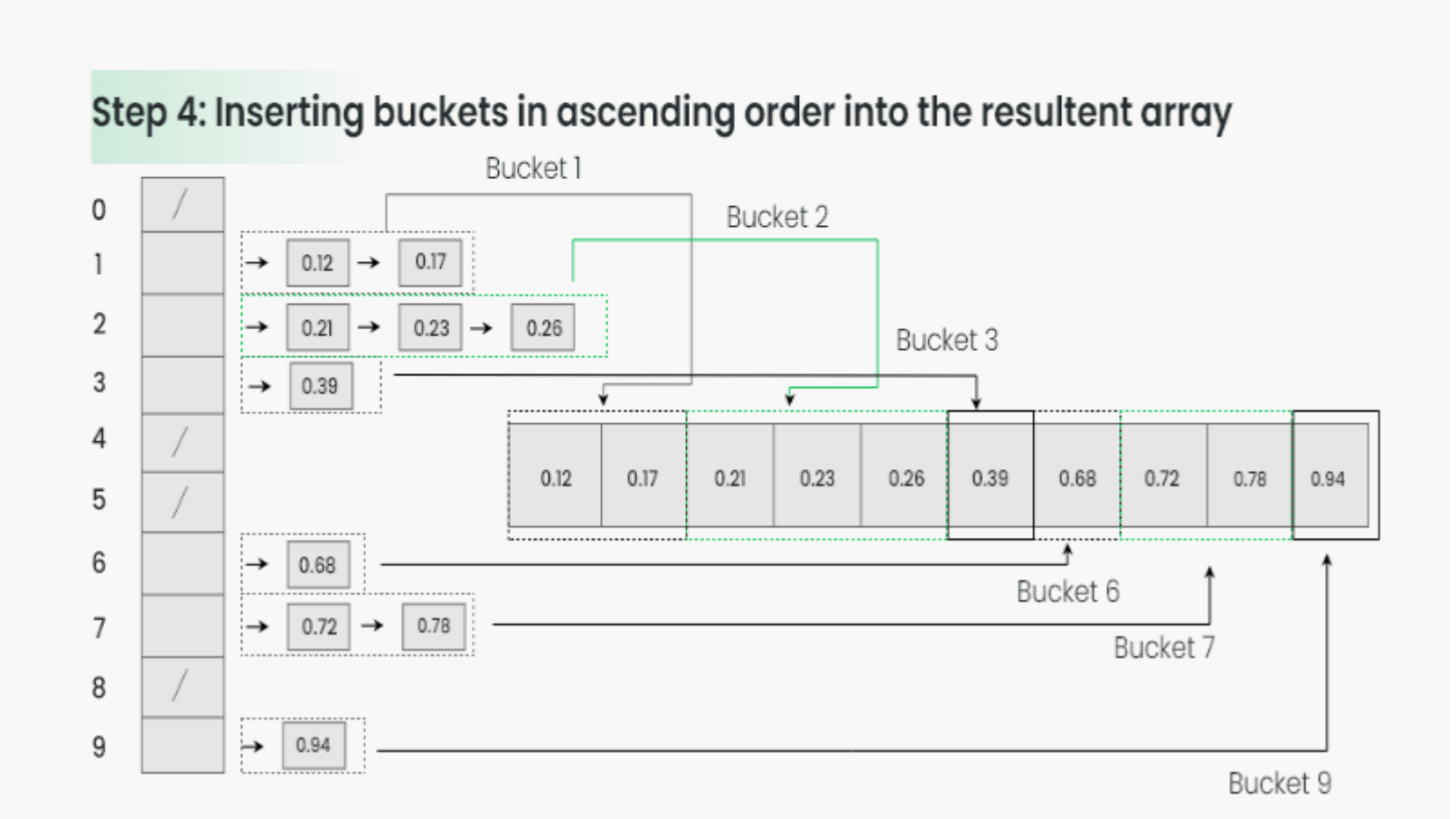
- Aplique un algoritmo de clasificación estable (por ejemplo, Bubble Sort, Merge Sort) para ordenar los elementos dentro de cada depósito.
- Los elementos dentro de cada depósito ahora están ordenados.



Paso 4: Reúna los elementos de cada depósito y vuelva a colocarlos en la matriz original.

Reuniendo elementos de cada cubo:

- Repita cada depósito en orden.
- Inserte cada elemento individual del depósito en la matriz original.
- Una vez que se copia un elemento, se elimina del depósito.
- Repita este proceso para todos los depósitos hasta que se hayan reunido todos los elementos.



Paso 5: la matriz original ahora contiene los elementos ordenados.

La matriz ordenada final que utiliza la clasificación por cubos para la entrada dada es [0,12, 0,17, 0,21, 0,23, 0,26, 0,39, 0,68, 0,72, 0,78, 0,94].

Step 5 : Return the Sorted Array

0.12	0.17	0.21	0.23	0.26	0.39	0.68	0.72	0.78	0.94

Fin